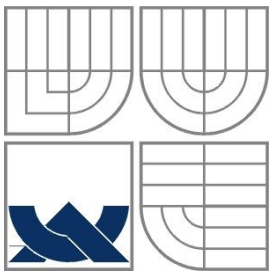
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

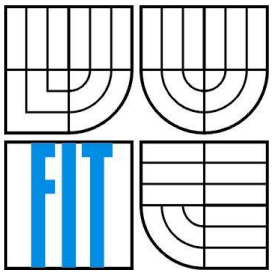
Fakulta informačních technologií
Faculty of Information Technology

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

Brno, 2016 Lenka Mazancová



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PODPORA VÝVOJE A TESTOVÁNÍ INTERPRETŮ JEDNODUCHÝCH JAZYKŮ

DEVELOPMENT AND TESTING SUPPORT FOR INTERPRETERS OF SIMPLE LANGUAGES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — Lenka Mazancová
AUTHOR

VEDOUCÍ PRÁCE — Ing. Zbyněk Křivka, Ph.D.
SUPERVISOR

BRNO 2016

Abstrakt

Tento práce je zaměřena na analýzu, návrh a implementaci referenčního interpretu, který má sloužit jako pomůcka při vypracovávání a opravování projektů do předmětu Formální jazyky a překladače. Řešení tohoto problému se skládá z vytvoření instrukční sady, knihovny na čtení a zápis navržené instrukční sady a interpretu navržené instrukční sady. Zvláštní pozornost je věnována samotnému interpretu, jehož důležitou vlastností se rozšiřitelnost pomocí konfiguračních XML souborů a zdrojových souborů v jazyce C# nebo Visual Basic. Na závěr je popsáno testování interpretu navržené instrukční sady včetně možností přizpůsobení na různé typy víceúrovňových jazyků.

Abstract

This document is focused on analysis, design and implementation of a reference interpreter that can be used as a tool for fulfilling and evaluating the team project of Formal Languages and Compiler course. The solution of this task includes the design a new instruction set, a library for reading and writing for this instruction set and an interpreter of this instruction set. Special attention is payed to the interpreter itself that can be extended to accept modifed instruction set. Such extension is described using XML configuration files and extern source files in C# or Visual Basic.NET. In the end the text describes the testing of the interpreter of the design instruction set including the possibilities to adjust the instruction set to different types of high-level languages.

Klíčová slova

Intermediální reprezentace, instrukční sada, knihovna, C++, interpret, C#

Keywords

Intermediate representation, instruction set, library, C++, interpreter, C#

Citace

MAZANCOVÁ, Lenka. *Podpora vývoje a testování interpretů jednoduchých jazyků.* Brno, 2016. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Zbyněk Křivka.

Podpora vývoje a testování interpretů jednoduchých jazyků

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracovala samostatně pod vedením Ing. Zbynka Křivky, PhD.
Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

……
Lenka Mazancová
18. 5. 2016

Poděkování

Moje poděkování patří nejprve Prof. RNDr. Alexandru Medunovi, CSc. a předmětu Formální jazyky a překladače, bez kterých by myšlenka tohoto projektu nevznikla.
Dále bych ráda poděkovala Ing. Zbyňku Křivkovi, PhD. za vedení, trpělivost a poskytnutí užitečných rad.
Nesmím zapomenout na svého otce, Ing. Jaroslava Mazance, který mi ve formě konzultací poskytl velké množství odborných znalostí a vlastních zkušeností ještě před samotným zadáním bakalářské práce a který mi je v profesním životě obrovským vzorem.
Také chci poděkovat svým spolužákům, kteří mě byli vždy ochotni vyslechnout a poskytnout nové vhledy při řešení problémů.

Obsah

1 Úvod

Cílem této práce je vytvořit nástroj, který bude nápomocný jak lektorům předmětu IFJ (Formální jazyky a překladače), tak studentům, kteří by rádi úspěšně absolvovali tento předmět. Lektorům by měl především usnadnit vytváření zadání projektu do IFJ (především díky možnosti si zadání vyzkoušet), lépe odhadnout náročnost projektu (získáním lepší představy o výsledném projektu) a následné opravování (referenční interpret pro konkrétní zadání). Studentům by měl pomoci lépe pochopit funkci překladačů, umožnit včasné a spravedlivější rozdělení práce v týmu a vidět na vlastní oči, co se od nich vlastně očekává.

V následující kapitole se budeme zabývat nejprve analýzou a výzkumem, ve které se budeme snažit přijít na co nejvhodnější způsob řešení dané problematiky. Další kapitola bude už řešit konkrétní návrh způsobu řešení, který vyplyne z druhé kapitoly. V kapitole Implementace bude popsána reálná implementace tohoto řešení. Potom budeme pokračovat kapitolou věnující se možnostem rozšíření a na závěr se podíváme na testování na existujících zadáních projektu IFJ.

Nebudeme se zabývat samotnou teorií probíranou v IFJ [1], [2], při čtení tohoto dokumentu se očekává alespoň základní znalost probírané problematiky. Mnohé pojmy však převezmeme a budeme je dále rozvádět.

2 Výzkum, analýza

Tato kapitola se bude zabývat výzkumem a analýzou samotné problematiky. Nejprve se podíváme na samotný projekt v předmětu IFJ, důkladně si prohlédneme zadání z předešlých let a shrneme požadavky na výslednou bakalářskou práci.

2.1 Projekt předmětu Formální jazyky a překladače

Tento projekt je skupinový a jeho cílem je naimplementovat v jazyce C interpret zadaného formálně definovatelného programovacího jazyka. Většinou se jedná o zjednodušené podmnožiny již existujících jazyků. Určit vhodný zdrojový jazyk a jaké jeho funkce má interpret podporovat, aby projekt splňoval jistou míru náročnosti, není zrovna jednoduché. Jedním z problémů je samotná existence velké množiny těchto jazyků. Samozřejmě by se neměla zadání příliš opakovat, aby studenti nepodléhali pokušení projekty kopírovat z minulých let. Z tohoto hlediska je obrovské množství existujících jazyků vlastně výhodou.

Z toho nám vyplývá první požadavek ze strany lektorů: **Najít vhodné sjednocení již existujících zadání pro zjednodušení tvorby budoucích.**

Splnit takový požadavek však vůbec není jednoduché. Každý jazyk má úplně jinou syntax a sémantiku, různou míru abstrakce, jinou vnitřní práci s proměnnými apod. Jednou obrovskou výhodou je, že nám jde pouze o jistý druh „vnějšího chování“, tzn.: nezáleží na použitém překladači. Jinak řečeno jde o vykonání vstupu ve formě zdrojového kódu v definovaném pořadí akcí jak jazykem, tak kódem samotným. Jak a co se děje mezi tím, nás už ale tolik nezajímá. Dalo by se říct, že přesně na této nedefinované části je založen celý projekt IFJ. Je to část, která každý překladač činí osobitým, vhodným pro jisté množiny jazyků kvůli rychlosti zpracování, práci s pamětí, optimalizace výsledného kódu atd. A přesně tuto část by bylo vhodné nějak zobecnit pro různá zadání.

Cílem je tedy vytvořit nástroj, který bude podporovat co největší škálu jazyků a omezovat tak budoucí zadání co nejméně.

2.1.1 Rozbor zadání z minulých let

Pro jednodušší rozhodování slouží Tabulka 2.1-1. V ní vidíme, že všechny jazyky mají nějaké interní typy, které podporují sadu interních funkcí, konstrukce řízení toku programu s cykly i bez. Statické a dynamické typování se mění, stejně tak se mění platnost proměnných v programu. Jazyková syntaxe i sémantika je každý rok jiná, avšak celkem jednoduše popsatelná formálními definicemi. Student by měl být schopen sestavit tyto formální definice (konečný automat pro lexikální analýzu, LL gramatiku pro syntaktickou analýzu, atd.). Z těchto definic lze jako referenční řešení pro tvorbu syntaktického stromu nebo mezikódu využít již existující nástroj, např. ANTLR [3].

Tabulka 2.1-1: Přehled zadání projektu IFJ z minulých let

Zadání rok:	**2010**	**2011**	**2012**	**2013**	**2014**
Nadmnožina:	**FreeBASIC**	**Lua**	**Falcon**	**PHP**	**Pascal**
Typování:	statické	dynamické	dynamické	dynamické	statické
Deklarace, **Platnost proměnné:**	ano, lokální	ano, lokalní	ne, lokální	ne, lokální, define() globální	ano, lokální i globální
Podporované typy:					
- **celočíselný**	ano	ne	ne	ano	ano
- **desetinný**	ano	ano	ano	ano	ano
- **řetězcový**	ano	ano	ano	ano	ano
- **logický**	ne	ano	ano	ano	ano
- **nil/null typ**	ne	ano	ano	ano	ne
- **pole**	rozšíření	rozšíření	rozšíření	rozšíření	rozšíření
Funkce:	ano	ano	ano	ano	ano
Návratová hodnota:	ano	ano	ano	ano	ano
Vestavěné funkce:	ano	ano	ano	ano	ano
- **typ hodnoty**	ne	ano	ano	ne	ne
- **převod typu**	ne	ne	numeric()	ano	ne
- **podřetězec**	ne	ano	[od,do]	ano	ano
- **find, sort**	ano	ano	ano	ano	ano
- **délka řetězce**	ne	rozšíření	ano	ano	ano
Čtení vstupu:	ano	ano	ano	ano	ano
Zápis do výstupu:	ano	ano	ano	ano	ano
Binární operátory:	ano	ano	ano	ano	ano
Unární operátory:	rozšíření	rozšíření	rozšíření	rozšíření	rozšíření
Podmíněný příkaz: **if-elseif-else, ...**	ano	ano	ano	ano	ano
Cykly: **for, while, do-while, ...**	ano	ano	ano	ano	ano

Výstupem zmíněných nástrojů většinou bývá AST (abstraktní syntaktický strom), který lze jednoduše zkonvertovat na 3AC (tří adresný kód) nebo na jinou IR (intermediální reprezentace) [4], které lze potom jednoduše interpretovat. Z toho vyplývá, že obecné nástroje, které mohou pomoci při tvorbě interpretu především po fázi generování mezikódu, již existují.

To je jeden z důvodů, proč se tato bakalářská práce zabývá IR a jeho reprezentací. Zvolením nebo vytvořením vhodného IR by se totiž dalo jednak studentům vytvořit záchytný bod na cestě k úspěšnému dokončení projektu do IFJ (lepší možnost rozdělení práce na tvorbu IR a interpretaci IR), ale také poskytnout jeden nástroj, který umožní se správnou vstupní definicí vytvořit referenční řešení.

Existujících IR existuje mnoho. Na některé z nich existují interprety. Jejich vhodnosti se budeme zabývat dále.

2.2 Existující intermediální reprezentace

Jak již bylo řečeno, IR existuje několik druhů, nejčastější podobou je IL (intermediální jazyk - *intermediate language*), nebo p-code (přenositelný jazyk – *portable code*) nebo-li bajt (*byte*) kód (*bytecode*). Existují v různých úrovních abstrakce [5]. Mezi rozšířené patří CIL (*common intermediate language*) vytvořen firmou Microsoft a využívaný v prostředí .NET [6], nebo Java byte kód využívaný v překladačích Javy [7].

Ačkoliv tyto příklady jsou pro účely projektů IFJ nevhodné[1], je možné je použít aspoň jako příklad toho, jak fungují IR v překladačích nejrozšířenějších prostředí. Jazyky, které jsou při překladu reprezentovány v prostředí .NET pomocí CIL je hned několik. Díky tomu optimalizace a interpretace nebo převody do strojové podoby jsou pro všechny tyto jazyky jednotné. Tzn.: není nutná jejích další implementace při změně vstupního programovacího jazyka. Postačí vytvořit instrukční sekvenci v CIL.

Jednou z možností je také méně známé C--, které jak již z názvu můžeme usoudit je jednodušší formou existujícího C. Tento jazyk je ale stále dost abstraktní a vyžaduje vcelku složitou režii kolem syntaxe a sémantiky, než je vhodně interpretován nebo převeden do strojového kódu.

Nejvhodnější IR pro potřeby této bakalářské práce by mělo představovat jednotlivé instrukce k vykonání. Instrukce by měli být natolik elementární, aby se jimi dal interpretovat jakýkoliv jazyk, a zároveň aby byly srozumitelné a snadno čitelné.

2.3 Implementační prostředí

Zde jsou shrnuty požadavky na výsledný bakalářský projekt a možnosti vyplívající z předchozích kapitol.

2.3.1 Požadavky

Jelikož součástí zadání projektu IFJ je, že projekt musí být přeložitelný a spustitelný v prostředí na školních serverech, bylo nutné zvolit takové prostředí, které bude na stejné platformě spustitelné. Přestože školní servery pro tyto účely jsou Linuxové a Unixové, existují studenti, kteří programují v MSVS (*Microsoft Visual Studio*) na prostředí Windows a následně kód upravují, aby byl přeložitelný pod GCC (*GNU Compiler Collection*) na školních serverech. Z tohoto důvodu by měl být výsledný projekt nezávislý na platformě, aby do posledních úprav mohli pracovat ve svém oblíbeném prostředí.

Další podmínkou je, aby projekt pro studenty fungoval jako černá skříňka, nebo alespoň byl napsaný takovou formou, aby nebylo možné kopírovat zdrojový text tohoto projektu do studentských projektů (tj. vytvářet částečný plagiát).

Důležitým aspektem výsledného programu je snadná a opakovatelná rozšiřitelnost výsledného programu pomocí konfiguračních souborů ve snadno čitelných formátech pro serializaci dat, např. XML/JSON.

Neopomenutelným požadavkem je také již zmíněná srozumitelnost a dobrá čitelnost samotného navrženého referenčního IR.

[1] kvůli své složitosti a podpoře objektů, která se zatím v minulých zadáních nevyskytla a jejich podpora by značně přesahovala náročnost tohoto předmětu

2.3.2 Možnosti

Pro vytvoření syntaktického stromu lze použít ANTLRv4 [3], které kromě verze v Javě existuje i pro C#. Implementace interpretu jednoduchého IR v jednom z těchto prostředí by umožnila vytvoření celého referenčního interpretu. Vzhledem ke zkušenostem autorky se zaměříme spíše na zhodnocení možnosti implementace v C#.

Programovací jazyk C# verze 6 je podporován především ve svém původním prostředí .NET. Existuje na platformě nezávislá verze Mono C# [8], která nepodporuje jenom knihovny závislé na platformě Windows. Díky zpřístupnění .NET kompilačního jádra známého jako Microsoft Roslyn [9] je množství na Windows platformě závislých knihoven nižší.

Při použití tohoto programovacího jazyka není třeba hledat nebo implementovat knihovny pro načítání formátů pro serializaci dat.

Další variantou je jazyk C++, který je přeložitelný jak pomocí MSVS, tak na Unixových systémech v g++. Oba překladače umí vytvářet knihovní soubory pro dané prostředí[2], které lze importovat i do projektu v jazyce C (ve kterém je psán projekt do IFJ). Zde lze pro načítání formátů pro serializaci dat využít existující knihovny (např. pro XML je dobrá multiplatformní knihovna *boost* [10]). Avšak napsání celého projektu v tomto prostředí by nebylo příliš vhodné, protože toto prostředí neumožňuje kompilaci externích zdrojových souborů za běhu, tzn. při každé změně IR nebo jiné dílčí části by se projekt musel znovu celý zkompilovat.

Tady se opět vrátíme k prostředí C#, které běží na kompilačním jádru Microsoft Roslyn (které je volně dostupné pro běžné Unixové i iOS platformy), umožňuje kompilaci externích zdrojových kódů za běhu (pomocí *Systém.CodeDom.Compiler* [11]). Přistupovat k metodám či objektům, které již v programu existují, lze také pomocí obyčejného textového řetězce. Tato vlastnost se může hodit při implementaci instrukční sady, která díky tomuto bude snadno rozšiřitelná.

[2] .so pro Unix a .dll pro Windows

3 Návrh

Následující text se bude věnovat návrhu samotného IR, knihovny pro jeho načítání a referenčního interpretu. Přiblíží konkrétní vlastnosti jednotlivých částí a načrtne vnitřní architekturu.

3.1 Návrh vlastní intermediální reprezentace

Na vytvoření 3AC nebo syntaktického stromu je možno užít existujících nástrojů jako je ANTLRv4. Výstupy z těchto nástrojů jsou snadno převeditelné do textové podoby při vhodně zvolené instrukční sadě IR.

Samotná instrukční sada intermediální reprezentace by měla být snadno pozměnitelná. Nynější návrh je koncipován především s ohledem na minulá zadání projektu IFJ a může opomenout nějaké funkční prvky potřebné v budoucích projektů IFJ. Také může obsahovat chyby, nebo dokonce postrádat důležité funkční prvky, které bude žádoucí vykonávat v jedné instrukci (např. kvůli skrytí implementace). Přesto se snaží pokrýt všechny existující a možné budoucí konstrukční prvky.

Díky snadné možnosti úpravy by případné nedostatky neměly být překážkou v užívání výsledného projektu.

3.1.1 Textová reprezentace instrukce a její definice

Jelikož má výsledné IR být co nejjednodušeji převeditelné z 3AC nebo syntaktického stromu do textu, a zároveň snadno čitelná, byla zvolena obdoba assemblerovského instrukčního kódu.

Tzn.:

- název instrukce (který bude zároveň jedinečným identifikátorem operace),
- až tři volitelné operandy / argumenty.

Každá instrukce by měla být logicky i viditelně oddělená (třeba znakem středník nebo novým řádkem). Pro snadnější čtení textu v kódu je třeba oddělit i jednotlivé argumenty instrukce. Definice instrukce by měla mít být v jednoduše čitelné a jednoznačné formě. U každé instrukce je důležitý název instrukce, počet argumentů, a pro interpretaci jejich rozdělení na vstupní a výstupní. Každá instrukce by měla odpovídat právě jedné funkci napsané v externím souboru, která bude přijímat stejný počet argumentů stejného typu, jako je v definici.

3.1.2 Neopomenutelné instrukce a jejich vlastnosti

Dle minulých zadání shrnutých v kapitole 2.1.1 je vidět které konstrukce pro řízení toku patří mezi nejčastější zadané. Jejich sekvenční vykonání obvykle zahrnuje skoky a podmíněné skoky.

Zvláštní pozornost z hlediska interpretace je třeba věnovat i funkcím, které obyčejnými skoky příliš snadno popsat nelze. Mnohé jazyky mají ve funkcích či blocích lokální tabulku symbolů. Také je potřeba nějakým způsobem předat argumenty a návratovou hodnotu a zajistit správný návrat z funkce na správnou instrukci.

Co se týče tabulky symbolů, s tímto tématem souvisí i platnost proměnných. Proměnná v jednom okamžiku vznikne, ale také může v kterémkoliv dalším momentě přestat existovat. Proměnné mohou být staticky nebo dynamicky typované. Tohle je záležitost, kterou je možné podchytit ještě před vytvořením instrukční sady, ale nemalé procento hotových implementací studentů sémantickou

typovou kontrolu provádí až těsně před interpretací konkrétní operace. Nebylo by vhodné je v tomto omezovat, pokud by chtěli použít tuto intermediální reprezentaci.

Další možností je přistupovat k proměnným přes zásobník, nebo tabulku symbolů. Intermediální reprezentace by v tomto ohledu měla být nezávislá. Každá instrukce, kterou lze interpretovat pomocí zásobníku nebo tabulky symbolů, by měla existovat v obou variantách.

V tomto případě se nabízí navrhnout dvě instrukční sady, jedna zaměřena čistě na zásobníkové funkce, druhá zaměřená na tabulku symbolů.

Nesmí se zapomenout ani na vestavěné funkce jako tisk na standardní výstup, čtení ze standardního vstupu, kontrola typu, třídící a vyhledávácí funkce, aritmetické a logické operace apod. Neimplementované instrukce v koncové textové IR půjde snadno rozšířit.

3.2 Návrh knihovny pro načítání vlastní intermediální reprezentace

Tato část projektu by měla být schopná samostatně fungovat. Aby byla vhodná jako pomůcka pro studenty k usnadnění práce na projektu, je nutné, aby byla snadno zahrnutelná do jejich implementace projektu IFJ. Tzn.: musí být volatelná z prostředí jazyka C.

Programování knihovny v jazyce C je nevhodné kvůli náročnosti a taky možnému kopírování použitých algoritmů. Nejvhodnější variantou je vytvoření knihovny v jazyce C++, která bude na platformě nezávislá. Tuto knihovnu bude možné volat přímo v C, nebo bude možné si ji lehce upravit a použít, pokud se celý projekt bude kompilovat v g++.

Instrukční sada se bude načítat z textového souboru, akceptované instrukce z externího souboru s definicí instrukční sady ve formátu XML. Knihovna bude analyzovat vstupní textový soubor a za každým voláním bude vracet strukturu načtené instrukce:

- Identifikátor instrukce
- Operand 1
- Operand 2
- Operand 3

Pro jednoduchost budou operandy textové řetězce.

3.3 Návrh referenčního interpretu

Referenční interpret by měl využívat knihovnu k načítání instrukční sady a následně ji nějakým způsobem analyzovat a vykonat.

3.3.1 Průchod instrukční páskou

Aby nebylo nutné pozice pro skoky a definice funkcí definovat před začátkem vykonavatelné instrukce, bude první průchod instrukční páskou sloužit k analýze a vytvoření potřebných proměnných pro rychlejší procházení a vykonávání instrukční pásky.

Zde je možnost dvou přístupů:

- instrukční pásku uložit do paměti při prvním průchodu (rychlejší interpretace, potenciálně náročnější na paměť při dlouhé instrukční pásce), nebo
- instrukce načítat pokaždé znovu ze souboru (nutnost některé instrukce přeskakovat, vícenásobné čtení, potenciálně náročnější na procesorový čas, méně náročné na paměť).

Každá z těchto variant má své výhody i nevýhody. V ideálním případě by tato volba byla volitelná přepínačem. Protože se očekává, že studenti budou interpretovat spíš menší a ne příliš složité programy, lze předpokládat, že instrukční páska nikdy nebude natolik dlouhá, aby došlo k přehlcení operační paměti. Z toho důvodu je první varianta prioritní. Druhou lze případně implementovat jako rozšíření.

3.3.2 Definice instrukcí

Pro vykonávání instrukcí je třeba definice instrukcí, která se však bude od definice pro načítací knihovnu lišit, protože pro interpretaci bude třeba přesnější údaje, jako:

- jak se má instrukce interpretovat,
- počet vstupních operandů,
- počet výstupních operandů,
- zdroj argumentů pro vykonávání instrukce (zda vzít operandy ze zásobníku nebo z tabulky symbolů),
- kam uložit výstupní hodnotu (zda vzít operandy ze zásobníku nebo z tabulky symbolů).

Tyto informace by měli být uloženy ve zvláštním souboru odděleně od definice pro knihovnu, aby nebyly zavádějící pro studenty, kteří se rozhodnou knihovnu ve svém projektu využít. Zároveň se tím usnadní načítání definičních souborů.

3.3.3 Vykonávání instrukcí

Lze předpokládat, že interpretace každé instrukce bude jiná. Aby byly instrukce opravdu snadno rozšiřitelné, jejich implementace by měla být definovaná v externím souboru a jádro programu by mělo být na změnách v těchto dílčích implementacích nezávislé. Při použití prostředí jazyka C# a jeho knihovny `System.CodeDom.Compiler` je možné snadno implementovat další funkčnost nebo pomocí `System.Reflection` volat integrované funkční prvky které pracují s vnitřními strukturami v externím souboru. Jediné co je potřeba pro zavolání metody `Invoke()` je název souboru, název existujícího objektu (nebo předem zkonstruovaného) a název metody, vše ve formě textového řetězce.

Pokud se metoda vyskytuje ve více podobách s různými typy a počtem argumentů, je možné vyhledání metody upřesnit podle dostupných vstupních argumentů. Jestliže se metoda pro v instrukci zadané argumenty nenajde, je možné instrukci považovat za sémanticky neplatnou. Neplatné instrukce lze jednoduše ignorovat, protože každá instrukce by měla představovat jednu atomickou operaci.

Pokud měla instrukce vliv na další vykonávání (např.: inicializovat nějakou proměnnou), přijde se na tento problém během vykonávání dalších instrukcí a bude možné program ukončit s chybou. Tento výrok není pravdivý v případě instrukcí pro řízení toku. Předpokládejme však, že instrukční páska je výstupem nějakého jiného nástroje, který převádí nějaký kód na zmíněnou instrukční pásku. Syntaktické a sémantické chyby při čtení samotného textového IR by neměly nastat, pokud tento proces proběhl v pořádku podle platných definic. Kontrola pásky tedy nebude předmětem tohoto projektu, avšak je možné ji zmínit v rozšířeních.

3.3.4 Návrh architektury

Interpret určitě bude muset načítat, zpracovávat a uchovávat definici instrukční sady. O seznam platných instrukcí vyplývající z definice by se měla starat jedna třída, která bude umožňovat i základní operace nad tímto seznamem (např. vyhledávání). O této části budeme mluvit jako o **definici instrukční sady**.

Protože volání externí knihovny není zrovna syntakticky elegantní, je vhodné vytvořit třídu, která bude poskytovat volání knihovních funkcí pomocí metod. Tuto část lze definovat jako **knihovní obal**.

Každá načtená instrukce nejprve bude potřebovat přiřadit nějakou definici, podle které ji bude možné vykonat. K tomu bude nutné najít vhodnou definici a tu definici následně dekódovat do interně použitelné podoby. Toto bude mít na starost **dekodér instrukčních definic**.

Správně dekódovaná instrukce může být interpretována. Toto bude vykonávat **interpret instrukce**.

Některé instrukce však budou potřebovat přístup k tabulce symbolů nebo zásobníku, případně tabulce funkcí a podobně. Tyto věci by měli být ve zvláštní třídě **informace doby běhu programu**, která se bude také zajímat o aktuální pozici v instrukční pásce a bude určovat další instrukci k vykonání.

4 Implementace

Tato kapitola obsahuje podrobný popis implementace jak instrukční sady textového IR, knihovny pro načítání textového IR, tak implementaci referenčního interpretu.

4.1 Instrukční sada

Referenční interpret by měl využívat knihovnu k načítání instrukční sady a následně ji analyzovat a vykonat. Celá instrukční sada je dostupná v příloze, zde bude vypsáno pouze pár instrukcí, kterým je potřeba věnovat více pozornosti.

4.1.1 Formát instrukce a instrukční pásky

Formát instrukce je následovný:

```
KLIC [operand1[, operand2[, operand3]]]
```

Každá instrukce je oddělená alespoň jedním znakem konce řádku. Celý program je v jednom souboru, který je ukončen znakem EOF (konec souboru – *End Of File*).

4.1.2 Definice v XML

Pro správnou funkci knihovny pro načítání / zápis IR a interpret IR musí být každá definovaná instrukce s kódovým jménem a svým identifikačním číslem jedinečná. Při mnohonásobné definici se bude brát pouze ta první.

4.1.2.1 Pro knihovnu na načítání a zápis

Vzorový kód 1 zobrazuje strukturu definičního XML souboru pro knihovnu načítání a zápis vlastního IR. Je v něm vidět kořenový element `IRdefinition` obsahující element `instructions`, který obaluje pole elementů `instruction`. Každá instrukce obsahuje klíčové slovo instrukce `codename`, identifikační číslo instrukce `ID` které je možné použít pro přímé mapování na výčet (*enum*) a počet operandů `Operands`.

Vzorový kód 1 - Struktura definičního souboru ve XML formátu pro knihovnu na načítání a zápis vlastního IR

```
<?xml version="1.0" encoding="utf-8"?>
<IRdefinition>
  <instructions>   <!-- začátek pole pro definice instrukcí -->
    <instruction>  <!-- začátek definice instrukce KLIC --
      <codename>KLIC</codename>
      <ID>4</ID>
      <Operands>2</Operands>
    </instruction> <!-- konec definice instrukce KLIC -->

    … definice dalších instrukcí …

  </instructions>  <!-- konec pole pro definice instrukcí -->
</IRdefinition>
```

4.1.2.2 Pro interpret intermediální reprezentace

Vzorový kód 2 zobrazuje strukturu definičního souboru pro interpret vlastního IR.

Vzorový kód 2 - Struktura definičního souboru ve XML formátu pro interpret vlastního IR

```
<?xml version="1.0" encoding="utf-8"?>
<ArrayOfInstructionDefinition xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
                                <!-- začátek pole pro definice instrukcí -->
  <InstructionDefinition>       <!-- začátek definice instrukce KLIC -->
    <Codename>KLIC</Codename>
    <Precompile>Soubor.cs@Trida.Metoda</Precompile>
    <Operation>Soubor.cs@Trida.Metoda</Operation>
    <NumOfInputOperands>2</NumOfInputOperands>
    <OperationDestination>null</OperationDestination>
    <SourceOfOperands>raw</SourceOfOperands>
  </InstructionDefinition>      <!-- konec definice instrukce KLIC -->

  … definice dalších instrukcí …

</ArrayOfInstructionDefinition> <!-- konec pole pro definice instrukcí -->
```

Kořenový element `ArrayOfInstructionDefinition` je složen z několika po sobě jdoucích elementů `InstructionDefinition`, kde každý je jedinečný. Tento element popisuje instrukci pomocí:

- `Codename`, které se musí shodovat s `Codename` v předešlém definičním souboru,
- `Precompile`, kde je uvedena operace, kterou je třeba zavolat při prvním průchodu instrukční páskou,
- `Operation`, kde je uvedena operace pro vykonání instrukce,
- `NumOfInputOperands`, kde je uveden počet vstupních operandů 0-2, i pokud jsou skryté (vzaty z vrcholu zásobníku),
- `OperationDestination`, který říká co udělat s návratovou hodnotou operace (ignorovat – `null`, uložit ji na vrchol zásobníku – `stack`, nebo do tabulky symbolů na pozici uvedenou v dodatečném operandu – `code`),
- `SourceOfOperands`, který upřesňuje zdroj argumentů pro operaci (žádné operandy – `null`, čistý text z instrukční pásky – `raw`, vrchol zásobníku – `stack` nebo `stacktop`, hodnota buď přímo, nebo z pozice v tabulce symbolů – `code`).

Platné vstupní hodnoty elementů `Precompile`, `Operation`, `NumOfInputOperands`, `OperationDestination` a `SourceOfOperands` budou popsány v kapitole o interpretu.

4.2 Knihovna pro načítání instrukční pásky

Knihovna slouží jednak k načtení instrukce z instrukční pásky, tak k jejímu zapsání. Referenční interpret by měl využívat knihovnu k načítání instrukční pásky a následně ji nějakým způsobem analyzovat a vykonat.

4.2.1 Prostředí knihovny

Knihovna je napsána v jazyce C++, přeložitelné v prostředí Unixu i Windows. Knihovna využívá části multiplatformní knihovny *boost*, především:

- `boost/tokenizer.hpp` pro snadnější zpracování vstupní pásky a dělení vstupních řetězců,
- `boost/foreach.hpp` pro využití algoritmu *foreach*,
- `boost/property_tree/ptree.hpp` pro procházení stromu, který je výstupem z knihovny
- `boost/property_tree/xml_parser.hpp` pro načítání XML souboru s definicí instrukční pásky.

K implementaci knihovny bylo využito MSVS.

Knihovna existuje i jako `IR_Reader_DLL.dll` a v této podobě je využita v interpretu, který je popsán níže. Knihovnu lze zahrnout do C/C++ projektů jako zdrojové soubory.

4.2.2 Vstupy knihovny

Základním vstupem knihovny je definice interpretovaného instrukčního jazyka a jeho jednotlivých instrukcí. Tento vstup uložen v souboru ve formátu XML.

Dalším vstupem knihovny je zdrojový soubor se samotnou instrukční páskou, ze které instrukce čerpány, nebo naopak do které chceme zapisovat. Pokud soubor neexistuje, je vytvořen.

Oba tyto parametry se předávají při volání konstruktoru ve formě odkazu na počátek textového řetězce obsahujícího cestu k souboru.

V případě, že instrukce zapisujeme, je nutné vyplnit jméno nebo id instrukce a volitelné parametry při volání zapisovací funkce.

4.2.3 Výstupy knihovny

Pokud knihovnu používáme pro čtení, je návratová hodnota volané funkce struktura `TInstruction` (Vzorový kód 3), jejíž ukazatel se předává při každém volání načítací funkce.

Vzorový kód 3 - Implementace struktury TInstruction

```
struct TInstruction
{
	int id;			// číselný identifikátor pro propojení s enumerací
	char* codename;		// kódové jméno užité v instrukční pásce
	char* op1;		// operátor 1
	char* op2;		// operátor 2
	char* op3;		// operátor 3
};
```

Pokud se knihovna užívá pro zápis instrukce, je výstupem připsání instrukce na konec instrukční pásky v zadaném souboru.

4.2.4 Rozhraní knihovny

Knihovna má definované čtyři vstupní funkce, jejichž hlavičky vidíme ve níže. (Vzorový kód 4)

Vzorový kód 4 - Rozhraní knihovny pro čtení a načítání vlastního IR

```
IR_Reader_Obj* IR_Reader_Construct(char* def, char* ir);

void IR_Reader_GetInstruction(IR_Reader_Obj* reader_ptr, TInstruction* instr);

void IR_Reader_SetInstruction(IR_Reader_Obj* reader_ptr, int id, char* codename,
char* op1, char* op2, char* op3);

void IR_Reader_Destroy(IR_Reader_Obj* reader_ptr);
```

`IR_Reader_Construct()` přijímá odkaz na řetězec `char* def` obsahující jméno definičního XML souboru pro IR a odkaz na řetězec `char* ir` obsahující jméno textového souboru s instrukční páskou. Její návratová hodnota je ukazatel `void*` na vytvořený objekt (instance třídy `IR_Reader`), který je nutným parametrem ostatních funkcí pro správnou funkci knihovny.

`IR_Reader_GetInstruction()` je funkce přijímající jako první parametr ukazatel `IR_Reader_Obj*` na instanci třídy `IR_Reader`, se kterým pracuje a který již obsahuje platné načtené definice. Druhý parametr je ukazatel `TInstruction*` na existující prázdnou alokovanou strukturu `TInstruction`. Tato struktura je ve volané funkci naplněna buď platnými hodnotami načtené instrukce z instrukční pásky, jejichž platnost trvá do konce programu nebo do jejich uvolnění uživatelem, nebo jsou naplněny nulou. Při každém zavolání této funkce je načtena další instrukce v pořadí, prázdné řádky se vynechávají.

Pro zápis do instrukční pásky je možné použít funkci `IR_Reader_SetInstruction()`, která pro správnou práci opět potřebuje ukazatel `IR_Reader_Obj*` na instanci třídy `IR_Reader`. Pokud máme instrukce propojené přes výčet, vyplníme mezi parametry funkce pouze výčtovou hodnotu převedenou na číslo `int id` shodné s definicí instrukce a `char* codename` nastavíme na *NULL*. Volitelné parametry instrukce `char* op1, char* op2, char* op3` nastavíme na platné hodnoty nebo v případě jejich absence opět na *NULL*.

Každá instance třídy se uvolní z paměti pomocí `IR_Reader_Destroy()`. Po zavolání této funkce nad objektem definovaným ukazatelem `void* reader_ptr` již nelze s tímto objektem dále pracovat.

4.2.5 Algoritmus knihovny

Před použitím knihovny je nutné inicializovat všechny potřebné části. K tomu slouží funkce `IR_Reader_Construct()`, která vytvoří instanci třídy `Reader` a vrátí její ukazatel. Při vytváření tohoto objektu je otevřen soubor pro zápis / čtení instrukční pásky a načten definiční soubor instrukcí z XML do paměti pomocí metody `IRDef::Load()`. Tato metoda přímo využívá `boost::foreach()` a `boost::ptree`.

Po vytvoření této instance je možné zapisovat do souboru pomocí funkce `IR_Reader_SetInstruction()` a číst pomocí `IR_Reader_GetInstruction()`. Jednu vytvořenou instanci třídy `Reader` je doporučeno používat jen k jedné z těchto činností.

Pokud používáme knihovnu k zapisování, každým zavoláním k tomu určené funkce je do souboru zapsaná na nový řádek definovaná instrukce. Pokud se volající rozhodne nezadávat jméno

instrukce, musí zadat alespoň platné identifikační číslo (třeba z výčtu) definované v XML definičním souboru. V tomto případě je nutné argument `codename` nastavit na *NULL*.

Při volání funkce pro čtení dojde k zavolání metody `Reader::GetInstruction()`. Tato funkce:

1. Přečte jeden řádek ze vstupního souboru.
2. Zkontroluje, zda není načtený řádek prázdný. Pokud je, vrátí se na bod 1. Zkontroluje, zda řádek neobsahuje znak uvozovky. Pokud ano, pomocná funkce textový řetězec nahradí identifikátorem #číslo.
3. Vytvoří instanci třídy `boost::tokenizer` a jako parametr konstruktoru použije přečtený řádek.
4. Vezme první prvek ze zpracovaného řádku, který by měl představovat klíčové slovo instrukce a vyhledá jeho definici pomocí `IRDef::Find()`.
5. Pokud je hledání neúspěšné, vrátí ukazatel na strukturu instrukce s klíčovým slovem `codename` řetězec „UNKNOWN“.
6. Jestliže je hledání úspěšné a vrátí existující definici funkce, je na jejím základě vytvořena struktura instrukce a vrácen její ukazatel.

Po ukončení práce nad souborem by mělo následovat zavolání `IR_Reader_Destroy()`, aby došlo ke správnému zavření souboru a uvolnění použitých zdrojů.

4.2.6 Export knihovny

Export knihovny do DLL (dynamicky linkované knihovny – *dynamic-link library*) je využito především pro použití knihovny v interpretu. Všechny funkce popsané v kapitole 4.2.4 jsou v tomto formátu přístupné s příponou E_ (`E_IR_Reader_...`).

4.3 Interpret instrukční sady

Referenční interpret by měl využívat knihovnu k načítání instrukční sady a následně ji analyzovat a vykonat.

4.3.1 Prostředí interpretu

Celý program interpretu je napsán v jazyce C#. Kód je psaný v MSVS, ale lze jej zkompilovat a spustit i na jiných platformách než Windows pomocí kompilační platformy Mono a Microsoft Roslyn.

4.3.2 Vstupy interpretu

Vstupem interpretu je samotná instrukce k interpretování, která je získána z knihovny pro načítání.

Tato knihovna se musí nacházet ve stejném adresáři jako zkompilovaný projekt s názvem IR_Reader.DLL, jinak program hlásí chybu a nelze interpretovat vstup. Pro správné používání knihovny je nutné při spouštění interpretu definovat také vstupní hodnoty pro knihovnu: definiční soubor XML s definicí instrukční sady pro knihovnu a zdrojový soubor instrukční pásky.

Dalším důležitým vstupem je definiční soubor v XML se všemi potřebnými údaji pro správnou interpretaci. V tomto souboru je u každé instrukce údaj, kterou metodu má interpret k interpretaci použít. Tyto metody mohou existovat v externích zdrojových souborech a mohou být napsané v jazyce C# nebo v jazyce Visual Basic s příslušnými příponami.

Možným rozšířením interpretu může být umožnění psát operace pro instrukce v jiných jazycích, jako je například JavaScript.

4.3.2.1 Vstupní parametry

Interpret přijímá tyto povinné vstupní parametry:

- `--InputFile=filename.txt` - parametr je cesta k souboru s instrukční páskou,
- `--IRDefDLL=filename1.xml` - parametr je cesta k souboru ve formátu XML s definicí IR pro knihovnu,
- `--IRDefInterpret=filename2.xml` - parametr je cesta k souboru ve formátu XML s definicí IR pro interpret.

Je možné zadat následující volitelné vstupní parametry, které upravují způsob zpracování programu:

- `-v` nebo `--UseLocalVariables` – umožní práci s lokání tabulkou symbolů nebo lokálního zásobníku ve funkcích.
- `-l` nebo `--LogAll` – zapne pomocné výpisy na standartní výstup.
- `-s` nebo `--UseStack` – umožní užívání zásobníku. Pokud tento parametr není zapnut, zásobníkové funkce se neprovádějí korektně.
- `-t` nebo `--StaticTypeSymbols` – zapne kontrolu typu, tzn.: při každém pokusu o změnu typu symbolu interpret vypíše chybové hlášení a změna hodnoty se neuskuteční

4.3.3 Výstupy interpretu

Výstupem interpretu je vykovávání samotných instrukcí v instrukční pásce a případně vystup těchto operací. Při zadání přepínače pro povolení logování se na standardní výstup vypisují informativní hlášky.

4.3.4 Architektura interpretu

Následující text doplňuje kapitolu 3.3.4, kde je popsán návrh architektury.

4.3.4.1 Základní implementované třídy

Třídy, které představují funkční moduly:

- `Program` obsahuje vstupní funkci `Main()`,
- `Parameter` parsuje a ukládá parametry celého programu,
- `IR_Reader` poskytuje **knihovní obal**,
- `InstructionDecoder` implementuje **dekodér instrukčních definic** a obsahuje **definici instrukční sady**,
- `InstructionInterpreter` funguje jako **interpret instrukce**,
- `RuntimeInfo` udržující **informace doby běhu programu**,
- `IRType` nabízí podporu typu IR a základní převodní funkce pro tyto typy.

Třídy, které představují především objekty, se kterými se pracuje:

- `Instruction` je načtenou instrukcí s pořadovým číslem,
- `InstructionDefinition` je definice jedné instrukce podle XML definice,
- `DecodedInstructionDefinition` představuje definici jedné instrukce rozšířenou o dekódované informace a funkce pro vytváření vstupních parametrů pro interpretaci,
- `Symbol` s hodnotou, jménem a typem,
- `SymbolTable` seznam (list) symbolů,
- `Function`, obsahuje lokální tabulky symbolů a zásobník, funkce pro začátek a konec, informaci o funkci kterou byla volaná a kterou volala (pokud existuje vnoření).

4.3.5 Algoritmus interpretu

Program si v první řadě přečte a inicializuje vstupní parametry pomocí `Parameter.Get()`. Dalším krokem je inicializace třídy `RuntimeInfo` a zavolání metody `Precompile()`, která představuje první průchod instrukční páskou.

V této metodě dojde k vytvoření objektu `IR_Reader`, který je přímo napojen na knihovnu pro čtení z instrukční pásky. Po vytvoření seznamu pro instrukční pásku se postupně v cyklu načte a zpracovává každá načtená instrukce ve vstupním souboru. Pokud má načtená instrukce v definici v XML platné údaje elementu `Precompile`, dojde k zavolání zde definované metody.

`RuntimeInfo` si uchovává informace potřebné pro správné vykonávání programu. Mezi tyto informace patří např.: tabulka symbolů, zásobník, funkce. K inicializaci některých z těchto informací dojde během volání funkcí uložených v XML elementu `Precompile`.

Další průchod zajišťuje metoda `RuntimeInfo.Run()`. V rámci této metody se nejprve nastaví jako aktivní funkce počáteční funkce „main“, která představuje vstupní bod instrukční pásky. Nedochází k načítání instrukcí z pásky, ale z vytvořeného seznamu při prvním průchodu. Tento seznam se prochází v cyklu a každá následující instrukce je okamžitě vykonána. Pozice v instrukční pásce nebo aktuální funkce může být ale při tomto průchodu změněna operací instrukce definovanou v XML elementu `Operation`.

K ukončení vykonávání dojde při:

- dosažení konce instrukční pásky,
- nastavení hodnoty aktuální funkce na *null* (tzn. funkce main i všechna zanoření skončila),
- dosažení k tomu určené instrukce (například EXIT).

Při ukončení vykonávání se program ukončí.

4.3.6 Vykonání instrukce z instrukční pásky

Vykonání jedné instrukce z instrukční pásky probíhá pomocí třídy `InstructionInterpreter` a metody `Interpret()` (nebo `Precompile()` při prvním průchodu). Tato třída v sobě uchovává volatelné zdroje vestavěné v programu a všechny již přeložené zdrojové soubory. K přeložení zdrojového souboru dojde pouze jednou a to při jeho prvním použití.

Před vykonáním instrukce se program nejprve podívá na jeho definici a najde údaj načtený z elementu `Operation` (`Precompile`) pro nalezení metody, která má představovat operaci pro vykonání instrukce.

Definice této operace se očekává ve formátu: `soubor.cs@trida.metoda`, kde:

- `soubor` je název zdrojového souboru, který se nachází ve stejném adresáři jako interpret,
- `.cs` je koncovka (.cs pro C#, .vb pro visual basic),
- `trida` je jméno třídy, nad kterou se má metoda volat,
- `metoda` je jméno metody, která se má volat.

V případě, že soubor s koncovkou není obsažen, dochází k hledání třídy v seznamu registrovaných interních tříd uvnitř `InstructionInterpreter`. Aktuálně se registruje pouze třída RuntimeInfo při inicializaci.

4.3.6.1 Výpočet parametrů metody

Parametry hledané metody se vypočítávají pomocí dalších elementů obsažených v definici instrukce, které upravují chování před a po zavolání metody. `NumOfInputOperands` určuje počet vstupních argumentů, nebo-li kolik vstupních parametrů je potřeba k vykonání operace. `SourceOfOperands` definuje zdroj těchto vstupních argumentů.

Zde je několik předem definovaných možností:

- `null` – nemá žádné vstupní parametry
- `raw` – vezme vstupní parametry z prvních operandů vykonávané instrukce jako textové řetězce (string),
- `code` – vezme vstupní parametry z prvních operandů vykonávané instrukce a určí jejich typ (existující proměnná, hodnota) pomocí funkce `Symbol.DetermineSymbol()` a pomocí typu hledá metodu,
- `stack` – stejně jako `code`, s tím rozdílem, že jako parametry použije hodnoty z vrcholu zásobníku,
- `stacktop` – stejně jako `stack`, s tím rozdílem, že hodnota z vrcholu zásobníku nezmizí (reálně lze takto přečíst tedy pouze jednu hodnotu).

Při volání funkce při prvním průchodu (`Precompile`) jsou vždy užity `raw` parametry.

V případě, že není operace definovaná, se jako návratová hodnota počítá první vstupní hodnota. Díky tomuto je snadná definice operací pro např. zásobníkové funkce.

4.3.6.2 Návratová hodnota metody

Co se má vykonat s návratovou hodnotou metody určuje element `OperationDestination`, který může nabývat následujících hodnot:

- `null` – neočekává se návratová hodnota nebo se má ignorovat,
- `code` – uložit na pozici definovanou posledním operandem instrukce (očekává se jméno proměnné z tabulky symbolů),
- `stack` – uložit návratovou hodnotu na vrchol zásobníku.

4.3.6.3 Příklad vykonání instrukce

Mějme instrukci `KLIC`. Její XML definici pro knihovnu na načítání a zápis vidíme ve vzorových kódech níže. (Vzorový kód 5 a Vzorový kód 6)

Vzorový kód 5 - Definice instrukce KLIC pro knihovnu pro načítání a zápis vlastního IR

```
<instruction>
  <codename>KLIC</codename>
  <ID>56</ID>
  <Operands>2</Operands>
</instruction>
```

Vzorový kód 6 - Definice instrukce KLIC pro interpret vlastního IR

```
<InstructionDefinition> <!-- začátek definice instrukce KLIC -->
  <Codename>KLIC</Codename>
  <Operation>Soubor.cs@Trida.DoKlic</Operation>
  <NumOfInputOperands>2</NumOfInputOperands>
  <OperationDestination>stack</OperationDestination>
  <SourceOfOperands>code</SourceOfOperands>
</InstructionDefinition> <!-- konec definice instrukce KLIC -->
```

Vykonáváme instrukci: `KLIC var, 4`

`InstructionInterpreter` dostane zpracovanou instrukci z knihovny ve struktuře `Instruction` a předá její kódové jméno (`KLIC`) metodě `InstructionDecoder.Decode()`, která vrátí definici hledané instrukce jako objekt typu `DecodedInstructionDefinition`. Pokud byla definice instrukce nalezena (tzn.: není *null*), vytvoří na jejím základě ze zadané instrukce pole typů argumentů a pole argumentů pomocí funkce `DecodedInstructionDefinition.GetArguments()`.

V tomto příkladu máme zadaný počet vstupních argumentů 2 a jejich zdrojem jsou operandy instrukce. Výsledná pole budou obsahovat dva prvky. Řekněme, že var je platná hodnota v tabulce symbolů typu textový řetězec s hodnotou „hello world“. Pole typů bude {textový řetězec, celé číslo} a pole argumentů bude {„hello world“, 4}. Hledanou operací bude tedy metoda `DoKlic` přijmajici dva argumenty typu {`string`, `int`} ve třídě `Trida` v souboru `Soubor.cs`.

Řekněme, že `Soubor.cs` je uložen ve stejné složce jako spuštěný interpret a Vzorový kód 7 znázorňuje jeho strukturu.

Vzorový kód 7 - Vzor platné struktury zdrojového souboru v jazyce C#

```
class Trida
{
	public static string DoKlic()
	{
		…
	}
	public static string DoKlic(string s, int i)
	{
		…
	}
}
```

Interpret zavolá pomocnou funkci `CompileSourceFile()` kde se nejprve podívá, zda už soubor `Source.cs` není v paměti zkompilovaný. Pokud je hledání neúspěšné, otevře si soubor `Soubor.cs` a pomocí `CodeDomProvider` vytvoří příslušný objekt třídy `CompilerResults` obsahující sestavu zdrojového souboru `Assembly`.

V případě, že by soubor nebyl nalezen v adresáři spuštěného programu nebo nešel zkompilovat, `CompileSourceFile` vrátí `false` a nelze pokračovat ve vykonávání této instrukce. Při zapnutí přepínače `--LogAll` vypíše chyby kompilace.

Předpokládejme, že `Soubor.cs` je v pořádku, `CompileSourceFile()` tedy vrátí `true` (ať už proto, že zkompilovaný soubor našel, nebo jeho sestavu teprve vytvářel). Dalším krokem je kontrola, zda třída `Trida` u nalezené sestavy již není vytvořená. Pokud není, je zavolán konstruktor této třídy bez vstupních parametrů. Nad touto třídou je potom vyhledána metoda `DoKlic(string, int)` (metoda bez parametru je ignorována, protože neodpovídá vyhledávacím kritériím definovaným polem typů argumentů) a pomocí `Invoke()` zavolána s polem argumentů, tedy hodnotami „hello world“ a 4.

Element `OperationDestination` je nastaveno na `stack`. Z toho vyplývá, že je očekávána návratová hodnota, která má být vložena na vrchol zásobníku. Návratová hodnota vyvolané metody je analyzována, je z ní vytvořen objekt třídy `Symbol` a je uložen na vrchol zásobníku. Pokud výstupem metody `DoKlic` bude vzít textový řetězec a vrátit prvních x znaků, bude po vykonání zadané instrukce na vrcholu zásobníku symbol typu textový řetězec s hodnotou „hell“.

4.3.7 Objekt Symbol a práce s typem

Objekt `Symbol` je definován trojicí:

- číselný identifikátor typu,
- textový řetězec hodnota a
- řetězec jméno.

Všechny hodnoty v programu, se kterými pracuje instrukční sada, jsou definované tímto objektem. Díky tomu je možné mít pod kontrolou práci s proměnnými. Číselné identifikátory typu a převodní funkce jsou definované ve třídě `IRType`, která v rámci odevzdané verze není v externím souboru, proto je nutné při změně této třídy vždy překompilovat celý interpret. Oddělení této části do externího souboru by bylo vhodným rozšířením.

Možnost změny práce s typem symbolu je ale vlastnost, která se hodí u jazyků, které například nepodporují celé číslo, ale pouze reálná čísla.

Aktuálně je třída pro práci s typem implementována pro podporu typů:

- nula (nill, psáno hodnotou „null“) s číselným označením 0,
- logický literál (bool, „true“ / „false“) s číselným označením 1,
- celé číslo (int) s číselným označením 2,
- desetinné číslo (double) s číselným označením 3,
- textový řetězec (string) s číselným označením 4.

U textového řetězce je nutné poznamenat, že po celou dobu jeho existence v symbolu má počáteční i koncové uvozovky. To znamená, že při práci s ním (tisk, počítání znaků apod.) je nutné uvozovky nejprve odstranit. Pokud je výstupem nějaké funkce řetězec, není nutné je opět zase přiřadit, program tak v případě potřeby učiní sám.

Pro konverze mezi typy jsou užity vestavěné funkce jazyka C# `(System.Type).TryParse()`.

4.3.8 Externí implementace instrukcí

Pokud je implementace instrukce v externím souboru, interpret očekává, že všechny metody volatelné z interpretu budou přístupné (public). Je možné implementovat třídu způsobem, že si bude udržovat hodnoty od svého vzniku, protože se v interpretu jednou vytvořená třída nezahazuje, ale používá znovu. Pokud je nutné definovat konstruktor, nesmí přijímat žádné argumenty. V případě, že by bylo potřeba inicializace pomocí vstupních hodnot, musí být implementována jako instrukce instrukční sady.

Jestliže instrukce pracuje se symboly, je nutné implementovat operaci pro každý platný vstupní typ. (Např.: Pro sčítání dvou hodnot bude existovat metoda přijímající dvě celá čísla a metoda stejného jména přijímající dvě reálná čísla, pokud jiné hodnoty nejsou platné.)

5 Možnosti rozšíření

Následující text se podívá na hotový projekt a zamyslí se nad možnostmi jeho rozšíření. Projekt byl vyvíjen s myšlenkou, že bude nutné ho upravovat a v mnoha ohledech je k tomu přímo uzpůsoben.

5.1 Rozšiřování instrukční sady

Instrukční sada je rozšiřitelná a pozměnitelná pomocí definičních XML souborů. Funkce každé instrukce lze implementovat ve zvláštním zdrojovém souboru. Pro jednoduché rozšiřování instrukční sady se vůbec nemusí zasahovat do zdrojové aplikace.

5.2 Rozšiřování interpretu

Tyto myšlenky byly již zmíněny v předchozích kapitolách jako možnosti rozšíření:

- možnost načítání instrukční pásky bez uložení do paměti (pokaždé číst ze souboru),
- kontrola instrukční pásky (lexikální, syntaktická, sémantická) a případné upozorňovaní na nedostatky vstupního programu,
- umožnění implementace instrukcí v jiných jazycích (např. JavaScript, F#),
- oddělení třídy `IRType` pro práci s typem vnitřních symbolů do zvláštního externího souboru pro usnadnění editace.

Většina zmíněných rozšíření nejsou pro správnou funkčnost a možnosti používání interpretu nijak zásadní. Dále by bylo zajímavé rozšířit interpret o:

- práci s polem a ukazatelem,
- kontrolu platnosti vstupních definic instrukcí,
- kompilaci do strojového kódu,
- umožnit tvorbu vestavěných funkcí pomocí instrukcí zavolaného zásobníku pro parametry funkcí a instrukcí volatelných funkcí (aby bylo možné definovat vestavěné funkce přijímající větší počet parametrů než je operandů instrukce)
- optimalizace vstupní instrukční pásky,
- krokování jednotlivých instrukcí s možností nahlédnout do informací běhu programu,
- napojení na ANTLRv4 nebo jiný nástroj, který by bylo možné využít pro načítání jiných zdrojových kódů a převádění do textového IR,
- a jistě mnoho dalších.

6 Testy

Tato kapitola se věnuje ověření vhodnosti finálního řešení bakalářské práce a jeho základní funkčnosti. Z následujících testů by mělo v ideálním případě vyplývat, že navržené instrukční sady (jejichž úplné definice jsou v příloze) a navržený interpret dokáží pokrýt všechny povinné parametry předešlých zadání projektu do IFJ shrnutých v kapitole 2.1.1.

V této kapitole se nebudeme zabývat přepisem vstupních zdrojových kódu projektu IFJ do programů ve vytvořených instrukčních sadách. Instrukce mají reprezentovat sekvenci interpretovaných dílčích částí a jejich vytvoření je součástí návrhu řešení projektu do IFJ. Navržená instrukční sada má být pouze pomůckou k vytvoření této přímo interpretovatelné sekvence. Testovací programy v této kapitole mají sloužit jako příklad možné interpretovatelné sekvence instrukcí po lexikální, syntaktické a případně sémantické analýze vstupního zdrojového kódu.

Z toho důvodu se tato kapitola na zadání projektů do IFJ dívá podle jejich povinných parametrů.

6.1 Typování

Kontrola typu je záležitost sémantické analýzy. Interpret se nezabývá tím, zda dojde k přepsání existujícího symbolu jisté hodnoty na symbol jiného typu. Všechna přiřazení jsou dynamická, proto při přepsání původní hodnoty na hodnotu jiného typu dojde i ke změně typu symbolu.

Testovací program 1 - Přiřazení hodnot při zapnutí přepínače pro statické typování symbolů

```
1    DEFS int, 2
2    DEFS bool, 1
3    DEFD string, "some string"
4    ASSIGN 4, int
5    ASSIGN 4, bool
6    ASSIGN true, bool
7    ASSIGN 4, string
8    ASSIGN "new string", string
9    PRINT int
10   PRINT bool
11   PRINT string
12   RET
```

V interpretu je ale možnost tyto změny typu zachytit. Použitím přepínače `--StaticTypeSymbols` lze získat chybová hlášení o změnách typu.

Zde je příklad takového výstupu (interpretován Testovací program 1):

```
ERROR: Static symbol error, changing value of type 1 to 2, Symbol name: bool
ERROR: Static symbol error, changing value of type 4 to 2, Symbol name: string
4Truenew string
```

Chybové hlášky zde vyvolal řádek 5 a řádek 7 (Testovací program 1). Ostatní přiřazovací instrukce proběhly v pořádku.

Testovací program 2 interaktivně testuje přesné chování.

Testovací program 2 - Program přijímající a ukládající různé vstupy

```
1     DEFS bool, 1
2     DEFS int, 2
3     DEFS double, 3
4     DEFS string, 4
5     PRINT "Write true or false: "
6     READ bool
7     PRINT "Write an integer: "
8     READ int
9     PRINT "Write a double: "
10    READ double
11    PRINT "Write some string: "
12    READ string
13    RET
```

Příklady vykonávání (Testovací program 2) s různými vstupy a se zapnutým přepínačem pro statické typování:

```
Write true or false: true
Write an integer: 4
Write a double: 3,7
Write some string: some string

Write true or false: 0
ERROR: Static symbol error, changing value of type 1 to 2, Symbol name: bool
Write an integer: 5,8
ERROR: Static symbol error, changing value of type 2 to 3, Symbol name: int
Write a double: aaa
ERROR: Static symbol error, changing value of type 3 to 4, Symbol name: double
Write some string: 56
ERROR: Static symbol error, changing value of type 4 to 2, Symbol name: string
```

6.2 Platnost proměnných

Platnost proměnných lze ovlivnit pomocí přepínače `--UseLocalVariables` (který ale platí pouze pro funkce) a pomocí k tomu učených instrukcí, které v jednom místě programu hodnotu definují a v jiném místě zase zahodí. Tímto způsobem lze mít jejich dobu platnosti pod kontrolou. Tuto funkčnost demonstruje Testovací program 3.

Testovací program 3 - Ovlivnění doby platnosti symbolu

```
1     DEF variable
2     DEF bool
3     EXISTS variable, bool
4     PRINT bool
5     UNDEF variable
6     EXISTS variable, bool
7     PRINT bool
8     RET
```

Platnosti hodnot v rámci funkce se věnuje kapitola 6.4.4.

6.3 Řízení toku

Následující testy by měli ověřit, že je možné pomocí navržených instrukčních sad pokrýt běžné konstrukce řízení toku.

6.3.1 Nepodmíněný skok

Testovací program 4: Interpret by měl při vykonávání nejprve vykonat instrukci `PRINT`, která vypíše na standardní výstup „Start “. Dále by měl vykonat skok na `someLabel` který je definovaný na řádku 4. Další instrukce v pořadí je opět `PRINT`, tentokrát ale vypíše „ End“.

Testovací program 4 – Příklad jednoduchého nepodmíněného skoku.

```
1    PRINT "Start "
2    JUMP someLabel
3    PRINT " WRONG "
4    LABEL someLabel
5    PRINT " End"
6    RET
```

Očekávaný výstup je tedy přeskočení výpisu textu „ WRONG “, který koresponduje s výstupem z interpretu v terminálu: `Start  End`

6.3.2 Podmíněný skok

Testovací program 5: Interpret by měl při vykonání nejprve vypsat textový řetězec, ve kterém uživatele žádá o napsání čísla. Následně by měl čekat na vstup potvrzený novým řádkem. Pokud je zadaná hodnota menší než 10, měl by skočit na řádek 9, jinak by měl pokračovat a skončit na řádku 8.

Testovací program 5 – Příklad jednoduchého podmíněného skoku se vstupem ze standardního vstupu

```
1    PRINT "Please write a number: "
2    DEF variable
3    READ variable
4    DEF boolean
5    LESS variable, 10, boolean
6    JTRUE lesser boolean
7    PRINT "inserted value is equal or greater than 10"
8    RET
9    LABEL lesser
10   PRINT "Inserted value is lesser than 10"
11   RET
```

Výstup z interpretu při zadání hodnoty 9:

```
Please write a number: 9
Inserted value is lesser than 10
```

Výstup při zadání hodnoty 10:

```
Please write a number: 10
Inserted value is equal or greater than 10
```

Při zadání neplatné hodnoty:

```
Please write a number: a
Assembly: Nonexistent method Lesser object Basic in file Basic.cs
Inserted value is greaterOrEqual than 10
```

Poslední chování je z toho důvodu, že interpret nenašel implementaci instrukce LESS pro porovnání textového řetězce a celého čísla, takže hodnota `boolean` zůstane implicitně 0. Jelikož není definované žádoucí chování pro tento případ, slouží příklad pouze jako ukázka vykonání neočekávaného (případně špatného) vstupu.

Stejně se chová program napsaný pomocí zásobníkových funkcí. (Testovací program 6)

Testovací program 6 - Příklad jednoduchého podmíněného skoku se vstupem ze standardního vstupu, zásobníkové instrukce

```
1    PRINT "Please write a number: "
2    PUSH 10
3    READ_S
4    LESS_S
5    JTRUE_S lesser
6    PRINT "Inserted value is equal or greater than 10"
7    RET
8    LABEL lesser
9    PRINT "Inserted value is lesser than 10"
10   RET
```

6.3.3 Přepis jednoduchého cyklu

Mějme definovaný cyklus v pseudokódu podobnému jazyku c:

```
for (i = 0; i < 10; i++)
{
      print(i);
}
```

Testovací program 7 ukazuje jeden ze způsobů, jak tento cyklus vyjádřit pomocí vytvořené instrukční sady. Vykonávání tohoto programu pomocí referenčního interpretu by mělo dojít k opakovanému tisknutí čísla uloženého v hodnotě `i`. Cyklus skončí, jakmile nebude platit podmínka: `i` je menší než 10.

Testovací program 7 - Jednoduchý cyklus

```
1     DEFD i, 0
2     DEF boolean
3     LABEL forCondition
4     LESS i, 10, boolean
5     JFALSE forEnd boolean
6     PRINT i
7     INC i
8     JUMP forCondition
9     LABEL forEnd
10    RET
```

Po vykonání (Testovací program 7 nebo jeho zásobníkové verze Testovací program 8) se na standardním výstupu dle očekávání objeví:

```
0123456789
```

Testovací program 8 - Jednoduchý cyklus, zásobníkové instrukce

```
1     DEF i
2     PUSH 0
3     LABEL forCondition
4     POP i
5     PUSH 10
6     PUSH i
7     LESS_S
8     JFALSE_S forEnd
9     PUSH i
10    PRINT_S
11    INC_S
12    JUMP forCondition
13    LABEL forEnd
14    RET
```

6.4 Funkce

V této části se budeme snažit otestovat, zda funkce fungují správně, zda je jejich opakované (i rekurzivní) volání možné, jestli je možné předávat parametry i návratovou hodnotu, zda si udržují lokální proměnné nebo ne.

6.4.1 Jednoduché volání

Testovací program 9 - Volání jednoduché funkce

```
1     PRINT "Function CALL..."
2     CALL function
3     PRINT "After function..."
4     RET
5
6     FUNC function
7     PRINT " This is inside the function. "
8     RET
9     END
```

Testovací program 9 zavolá funkci a po zavolání funkce program skončí. Výsledkem tohoto programu je výstup:

```
Function CALL... This is inside the function. After function...
```

To znamená, že k zavolání funkce došlo, a došlo k tomu pouze jednou a ve správném pořadí.

6.4.2 Návratová hodnota

Testovací program 10 (nebo jeho zásobníková obdoba Testovací program 11) slouží k otestování funkce instrukcí návratové hodnoty. Hodnota získaná při vykonávání instrukce `READ` (`READ_S`) se potom znovu vytiskne na standardní výstup již mimo funkci.

Testovací program 10 - Jednoduchá funkce s návratovou hodnotou

```
1     DEF retval
2     CALL function
3     GETRETV retval
4     PRINT retval
5     RET
6
7     FUNC function
8     DEF value
9     READ value
10    RETV value
11    RET
12    END
```

Testovací program 11 - Jednoduchá funkce s návratovou hodnotou pomocí zásobníkových instrukcí

```
1     CALL function
2     GETRETV_S
3     PRINT_S
4     RET
5
6     FUNC function
7     READ_S
8     RETV_S
9     RET
10    END
```

6.4.3 Předání parametru

Příklad předání parametru znázorňuje Testovací program 12 (a jeho zásobníková verze Testovací program 13), kde dochází k předání tří parametrů. Předání parametru interně probíhá pomocí zásobníku parametrů, proto je důležité dbát na pořadí, v jakém jsou parametry ve funkci vyzvednuty.

Testovací program 12 - Příklad užití 3 vstupních parametrů pro funkci

```
1. DEF value
2. PRINT "Please insert a valid number: "
3. READ value
4. PUSHP value
5. PUSHP 10
6. PUSHP "This will be printed before result of multiplication: "
7. CALL function
8. RET

9. FUNC function
10. DEF p1
11. POPP p1
12. DEF p2
13. POPP p2
14. DEF p3
15. POPP p3
16. PRINT p1
17. DEF result
18. MULT p2, p3, result
19. PRINT result
20. END
```

Testovací program 13 - Příklad užití 3 vstupních parametrů pro funkci pomocí zásobníkových instrukcí

```
1    PRINT "Please insert a valid number: "
2    READ_S
3    PUSHP_S
4    PUSH 10
5    PUSHP_S
6    PUSH "This will be printed before result of multiplication: "
7    PUSHP_S
8    CALL function
9    RET
10
11   FUNC function
12   POPP_S
13   PRINT_S
14   POPP_S
15   POPP_S
16   MULT_S
17   PRINT_S
18   END
```

Výsledek při zadání čísla 3 je:

```
Please insert a valid number: 3
This will be printed before result of multiplication: 30
```

6.4.4 Platnost hodnoty ve funkci

Testovací program 14 by měl otestovat, že nedojde k přepsání ani smazání hodnoty ve volané funkci při zapnutí přepínače `--UseLocalVariables`.

Testovací program 14 - Ukázka použití lokálních hodnot ve funkci

```
1    DEFD variable, 4
2    PRINT "Main: "
3    PRINT variable
4    CALL function
5    PRINT " Main again: "
6    PRINT variable
7    RET
8
9    FUNC function
10   DEFD variable, 8
11   PRINT " In function: "
12   PRINT variable
13   END
```

Výstupem je dle očekávání: `Main: 4 In function: 8 Main again: 4`

6.4.5 Rekurze

Testovací program 15 je trochu složitější, než testovací programy uvedené doposud, proto se ho pokusíme trochu vysvětlit.

Testovací program 15 - Složitější program s rekurzí

```
1    PUSHP 4
2    CALL function
3    PRINT " last retval:"
4    DEFS retval, 2
5    GETRETV retval
6    PRINT retval
7    RET
8
9    FUNC function
10   DEFS local, 2
11   POPP local
12   PRINT " function:"
13   PRINT local
14   DEFS new, 2
15   SUB local, 1, new
16   DEFS bool, 1
17   EQUAL local, 0, bool
18   JTRUE function_end, bool
19   PUSHP new
20   CALL function
21   PRINT " return val:"
22   DEFS retval, 2
23   GETRETV retval
24   PRINT retval
25   LABEL function_end
26   RETV local
27   RET
28   END
```

Funkce `function` přijme jeden parametr. Tento parametr nejprve uloží do lokální proměnné a potom ho vytiskne. Odečte od něj 1 a uloží novou hodnotu. Pokud je tato nová hodnota rovna 0, skočí na konec funkce, jinak pokračuje a volá opět samu sebe s touto novou hodnotou.

Jelikož na počátku dostane první volaná funkce jako parametr 4, mělo by volání proběhnout celkově pětkrát. Toto tvrzení koresponduje s výstupem interpretu:

```
 function:4 function:3 function:2 function:1 function:0 return val:0 return
val:1 return val:2 return val:3 last retval:4
```

7 Závěr

V kapitole věnující se výzkumu a analýze jsme shrnuli zadání projektu do IFJ z minulých let a vyjmenovali pár důležitých vlastností, které mají všechna zadání společná. Zamysleli jsme se nad existujícími řešeními daného problému, možnosti využití ANLTRv4 nebo jiného nástroje, vyjmenovali několik existujících intermediálních reprezentací a zamysleli se nad jejich vhodností pro naše účely. Během zpracovávání těchto údajů jsme vyvodili, co vlastně je předmětem této bakalářské práce, jaké jsou možnosti jejího řešení a požadavky na výslednou práci. Rozhodli jsme se pro tvorbu vlastní intermediální reprezentace, knihovny pro načítání a zápis textové reprezentace a interpretu této reprezentace.

V návrhové části jsme se podívali už na možné praktické řešení. Intermediální reprezentace v textové podobě co nejblíže kopíruje tří adresný kód. Vyjmenovali jsme z funkčního hlediska instrukce, na které nesmíme při implementaci zapomenout a které ovlivní stavbu interpretu. Také jsme na základě požadavků určili, jak bude vypadat knihovna pro načítání a zápis intermediální reprezentace a struktura, která bude reprezentovat načtenou instrukci. U návrhu referenčního interpretu jsme došli k závěru, že nebude stačit instrukční pásku projít pouze jednou. Také jsme si uvědomili, jaké informace budou zapotřebí pro správné vykonání každé instrukce, a jakým způsobem ji bude možné vykonat. Dále jsme si načrtli vnitřní architekturu interpretu, která sestávala z definice instrukční sady, knihovního obalu, dekodéru instrukčních definic, interpretu instrukce a modulu uchovávajícího informace doby běhu programu.

Implementační část navázala na návrh. Pevně definovala textový formát instrukce i souboru s instrukční páskou a strukturu definičních souborů v XML formátu i s příklady (jeden pro knihovnu na načítání a zápis instrukcí a druhý pro interpret). Popsali jsme knihovnu pro načítání, která je napsaná v jazyce C++, aby byla snadno zahrnutelná do projektu do IFJ, který je psán v jazyce C. Definovali jsme její vstupy a výstupy, detailně popsali rozhraní, které sestává ze čtyř funkcí, popsali jsme algoritmus knihovny. U interpretu, který je psaný v jazyce C# jsme popsali, vstupy i se vstupními parametry a výstup, kterým je samotná interpretace vstupní instrukční pásky. Specifikovali jsme architekturu interpretu a jeho základní třídy, které kopírovaly funkční celky popsané v návrhu. Samotný algoritmus interpretu není složitý, ale jeho nejdůležitější části – vykonávání jedné instrukce z instrukční pásky – byla věnována celá samostatná kapitola. Zde jsme podrobně rozepsali výpočet vstupních parametrů pro každou instrukci, ukládání návratové hodnoty a vše jsme dovysvětlili na příkladu.

Zvláštní pozornost byla věnována i objektu Symbol a práci s interním typem. Variabilita těchto prvků je klíčová kvůli různorodosti zadání projektu do IFJ. Stručně byla také popsaná externí implementace instrukce načítaná ze zdrojových souborů v jazyce C# nebo Visual Basic.

Výsledkem práce je vytvořená intermediální reprezentace ve formě po sobě jdoucích instrukcí a interpret této reprezentace rozšiřitelný pomocí konfiguračních souborů v XML formátu a vnějších zdrojových souborů. Díky těmto definicím v externích konfiguračních souborech, které jsou snadno editovatelné i člověkem, je tento nástroj opravu snadno rozšiřitelný nebo pozměnitelný.

Kapitola „Možnosti rozšíření“ se zamyslela jak nad možnými budoucími potřebami rozšiřování, také nad vlastnostmi, které by z výsledného nástroje vytvořily ještě lepší pomůcku pro účely tvorby a testování projektu do IFJ.

Část zabývající se testováním se snaží ukázat vhodnost výsledné práce. Obsahuje také mnoho příkladů přeložitelných programů v definovaných instrukčních sadách a naznačuje, jakým způsobem tvořit program v definované instrukční sadě pro jednotlivá zadání projektu do IFJ

Když se nyní podíváme na začátek celé této práce a zamyslíme se nad otázkou, zda se nám podařilo dosáhnout vytyčeného cíle, odpověď by byla: ano, podařilo. Jeho skutečnou využitelnost ukáže čas, nejspíš jej bude třeba v jistých ohledech upravit (pár užitečných úprav zmiňuje kapitola

Možnosti rozšíření), ale na většinu těchto změn upozorní až samotní uživatelé tohoto nástroje. Tento nástroj byl v tomto duchu navržen a programován, a tudíž jeho rozšiřování či pozměňování by nemělo být překážkou v jeho užívání.

Literatura

[1] Meduna, A., Lukáš, R.: *Doplňující informace předmětu IFJ, přednášky* [online]. Brno, poslední aktualizace 2015-09-22 [cit. 2016-02-02]. Dostupné pro na URL: < http://www.fit.vutbr.cz/study/courses/IFJ/public/materials/index.php>

[2] Meduna, A.: *Elements of Compiler Design.* Taylor & Francis, New York, 2008. ISBN 9781420063233

[3] Parr, T.: *The Definitive ANTLR 4 Reference*. The Pragmatic Bookshelf, Dallas, 2012. ISBN 978-1-93435-699-9

[4] Toal, R.: *Intermediate Representation* [online]. Loyola Marymount University, Los Angeles, [cit. 2016-02-04]. Dostupné na URL: <http://cs.lmu.edu/~ray/notes/ir/>

[5] Johnson, M., Zelenski, J.: *Compilers: Principles, Techniques, and Tools, Intermediate Representation* [online]. Aktualizováno 2008-09-28 [cit. 2016-02-08]. Dostupné na URL <http://dragonbook.stanford.edu/lecture-notes/Stanford-CS143/16-Intermediate-Rep.pdf>

[6] Rouse, M.: *What is Common Language Infrastracture (CLI)* [online]. 2007, [cit. 2016-02-08]. Dostupné na URL: <http://searchsoa.techtarget.com/definition/Common-Language-Infrastructure>

[7] Haggar, P.: *Java bytecode: Understanding bytecode makes you a better programmer* [online]. Aktualizováno 2001-07-01 [cit. 2016-02-08]. Dostupné na URL: < http://www.ibm.com/developerworks/library/it-haggar_bytecode/>

[8] *Home | Mono* [online]. c2016. Dostupné na URL: <http://www.mono-project.com/>

[9] *GitHub - The .NET Compiler Platform ("Roslyn") provides open-source C# and Visual Basic compilers with rich code analysis APIs* [online]. Aktualizováno 2016-05-15 [cit 2016-05-16] Dostupné na URL: <https://github.com/dotnet/roslyn>

[10] Dawes, B., Abrahams, D., Rivera R.: *Boost C++ Libraries* [online]. c1998-2007, aktualizováno 2016-05-13 [cit. 2016-05-15]. Dostupné na URL: <http://www.boost.org/>

[11] *Microsoft Developer Network: System.CodeDom.Compiler Namespace* [online]. c2016, [cit. 2016-02-12]. Dostupné na URL: <https://msdn.microsoft.com/en-us/library/system.codedom.compiler(v=vs.110).aspx>

Seznam příloh

PŘÍLOHA 1

PODPORA VÝVOJE A TESTOVÁNÍ INTERPRETŮ JEDNODUCHÝCH JAZYKŮ

MANUÁL

1 Než začnete

Projekt sestává z částí:

- Konfigurační soubory se vzorovou instrukční sadou, jež se skládají z:
  - XML definice pro knihovnu,
  - XML definice pro interpret,
  - zdrojové soubory pro instrukce,
- Knihovna pro načítání a zápis instrukční sady,
- Interpret instrukční sady.

Knihovna ani interpret bez svých platných konfiguračních souborů nedokáže fungovat.
Knihovna dokáže fungovat samostatně se svým definičním souborem.
Interpret nedokáže fungovat bez knihovny.

2 Instalace

2.1 Instalace knihovny

Knihovna je dostupná ve formě zdrojových kódů v C++, proto není potřeba žádné instalace. Stačí zdrojové soubory přiložit ke svému C/C++ projektu a používat hlavičkový soubor IR_Reader.h a elementy popsané v něm.

Důležité! Knihovna ve své zdrojové podobě potřebuje zdrojové soubory knihovny boost < http://www.boost.org/>. Cesty jsou relativní – buď tedy o lokaci knihovny boost bude vědět kompilátor, nebo složku se zdrojovými soubory vložte do adresáře s knihovnou.

2.1.1 Kompilace knihovny do nativního formátu

Tato část je potřeba pro správné fungování interpretu. Interpret musí být schopen přistupovat ke čtyřem (reálně třem) metodám, které jsou popsány v rozhraní v bakalářské práci. Kompilace DLL pomocí MS Visual Studia proběhla pomocí souboru (MSVS\IR_Reader\IR_Reader_DLL\Source.cpp):

```
#include "IR_Reader.h"
#include "IR_Reader.cpp"
#include "IRInstructionList.h"
#include "IRInstructionList.cpp"

extern "C" __declspec(dllexport) IR_Reader_Obj* E_IR_Reader_Construct(char* defSource,
char* irSource) {
	return IR_Reader_Construct(defSource, irSource);
}

extern "C" __declspec(dllexport) void E_IR_Reader_GetInstruction(IR_Reader_Obj* ptr,
TInstruction* i) {
	return IR_Reader_GetInstruction(ptr, i);
}

extern "C" __declspec(dllexport) void E_IR_Reader_SetInstruction(IR_Reader_Obj*
reader_ptr, int id, char* codename, char* op1, char* op2, char* op3) {
	return IR_Reader_SetInstruction(reader_ptr, id, codename, op1, op2, op3);
}

extern "C" __declspec(dllexport) void E_IR_Reader_Destroy(IR_Reader_Obj* ptr) {
	return IR_Reader_Destroy(ptr);
}
```

Tento soubor nedělá nic jiného, než že zpřístupní některé funkce. Obdobným způsobem je nutné vytvořit i ostatní nativní knihovny pro správnou funkci interpretu.

2.2 Instalace interpretu

Interpret přímo využívá knihovnu pro načítání a zápis. Pro potřeby interpretu je nutné mít knihovnu ve formě dynamicky linkované knihovny, která je často pro každou platformu jiná.

2.2.1 Před instalací

Pro správné instalování a spuštění potřebujete:

- spouštět na operačním systému MS Windows, nebo
- mít nainstalované nejnovější Mono <http://www.mono-project.com/>,
- mít zkompilovanou knihovnu pro načítání a zápis v nativním formátu (.dll pro Windows, .so pro Unix)

2.2.2 Instalace na MS Windows

Žádná zvláštní instalace na tuto platformu není potřeba, knihovna v nativním formátu je přiložená v rámci adresáře interpretu.

2.2.3 Instalace na jinou platformu

1. Zkompilujte zdrojové knihovny pro načítání do nativní podoby. Návod pro tento postup je uvedený v kapitole 2.1.12.1.1.
2. Vytvořenou knihovnu vložte do adresáře s interpretem.

3 Spuštění interpretu

Přehled platných vstupních argumentů je uveden v bakalářské práci v kapitole 4.3.2.1.

Spuštění na MS Windows přes CMD:

```
IR_Interpreter.exe --InputFile=test_programs\Testovaci_program_4.txt
--IRDefDLL=IRDefLib.xml --IRDefInterpret=IRDefInterpret.xml
```

Spuštění přes mono:

```
mono IR_Interpreter.exe --InputFile=test_programs\Testovaci_program_4.txt
--IRDefDLL=IRDefLib.xml --IRDefInterpret=IRDefInterpret.xml
```

Pokud se na terminálu objeví „`Start End`“, gratuluji. Nyní můžete přejít do kapitoly 4.

3.1 Spuštění se nezdařilo

Testování aplikace probíhalo na platformě MS Windows s tím, že byla spouštěna i pomocí mono.

Pokus o spuštění probíhal na systému CentOS 6.7 s instalací mono, bohužel kvůli špatně vytvořené knihovně interpret nenačetl knihovnu a nebyl schopen pokračovat. Pokud se tento krok nepodaří nějak rozumně uskutečnit, bude nutné knihovnu převést do jiného formátu nebo přímo do zdrojového kódu snadno přeložitelného pro Mono.

4 Psaní instrukční pásky

Při psaní programů v definované instrukční sadě je nutné se držet formátu popsaného v bakalářské práci. Na začátek doporučuji prostudovat vzorové programy a přiloženou instrukční sadu.

5 Změny v instrukční sadě

Pro popis práce s definičními soubory doporučuju pročíst bakalářskou práci. Každá instrukce musí mít jedinečné klíčové jméno. Pro knihovnu je potřeba mít její definici v jednodušší formě. Pro interpret je nutné instrukci doplnit i v druhém definičním souboru a určit, kde má hledat operaci pro vykonání instrukce, kterou je nutné implementovat.

K propojení instrukcí dochází pomocí klíčového jména, proto je nutné použít stejné v obou definičních souborech!

5.1 Přidání instrukce

K přidání instrukce do existující instrukční sady je nutné vytvořit nový element v seznamu instrukcí a naplnit ho správnými hodnotami. Při implementaci úplně nové instrukční operace je nutné implementaci této instrukce uvést z definičním souboru pro interpret.

5.2 Smazání instrukce

Pokud si předeme nějakou instrukci smazat, stačí smazat její definiční element v XML souborech. Pokud si pouze nepřejeme danou instrukci akceptovat, stačí smazat její definici v definičním souboru pro knihovnu. Při čtení této instrukce bude knihovna vracet instrukci UNKNOWN.

PŘÍLOHA 2

PODPORA VÝVOJE A TESTOVÁNÍ INTERPRETŮ JEDNODUCHÝCH JAZYKŮ

NAVRŽENÁ INSTRUKČNÍ SADA

ID	instrukce	zásobníková obdoba	stručný popis	operandy
		Hodnoty		
4	DEF/GDEF*		Definice hodnoty jen podle jména	%ID
5	DEFS/GDEFS*		Definice hodnoty staticky (s typem)	%ID, %TYPE
6	DEFD/GDEFD*		Definice hodnoty dynamicky (hodnotou)	%ID, value
7	ASSIGN		Přiřazení	value, %ID
8	UNDEF		Smazání hodnoty *pro globální symboly	%ID
		Řízení toku		
11	LABEL		definice místa pro skok	%LABEL
12	JUMP		nepodmíněný skok na %LABEL	%LABEL
13	JTRUE	JTRUE_S	podmíněný skok	%LABEL[, %ID/value]
14	JFALSE	JFALSE_S	podmíněný skok	%LABEL[, %ID/value]
15	EXIT		ukončení programu	
		Funkce		
19	FUNC		definice funkce	%FUNCTION
20	CALL		volání funkce	%FUNCTION
21	END		konec funkce	
22	RET		návrat z funkce	
23	RETV	RETV_S	nastavení návratové hodnoty	[%ID/value]
24	GETRETV	GETRETV_S	vyzvednutí návratové hodnoty	[%ID]
25	PUSHP	PUSHP_S	nastavení parametru funkce	[%ID/value]
26	POPP	POPP_S	vyzvednutí parametru funkce	[%ID]
		Zásobník		
30		PUSH	vloží prvek na vrchol zásobníku	%ID/value
31		POP	uloží prvek ma vrcholu zásobníku do proměnné	%ID
32		TOP	vrátí vrchol zásobníku aniž by ho smazal	%ID
		Aritmerické operace		
36	ADD	ADD_S	a + b = c	[%ID/value, %ID/value, %ID]
37	SUB	SUB_S	a - b = c	[%ID/value, %ID/value, %ID]
38	MULT	MULT_S	a * b = c	[%ID/value, %ID/value, %ID]

39	DIV	DIV_S	a / b = c	[%ID/value, %ID/value, %ID]
40	MODULO	MODULO_S	a % b = c	[%ID/value, %ID/value, %ID]
41	DEC	DEC_S		[%ID]
42	INC	INC_S		[%ID]
		Logické operace		
46	NEG	NEG_S	!a = b	[%ID/value, %ID]
47	AND	AND_S	a & b = c	[%ID/value, %ID/value, %ID]
48	OR	OR_S	a | b = c	[%ID/value, %ID/value, %ID]
49	XOR	XOR_S	a ^ b = c	[%ID/value, %ID/value, %ID]
		Porovnávací		
53	EQUAL	EQUAL_S	a == b = c	[%ID/value, %ID/value, %ID]
54	NEQUAL	NEQUAL_S	a != b = c	[%ID/value, %ID/value, %ID]
55	LESS	LESS_S	a < b = c	[%ID/value, %ID/value, %ID]
56	GREAT	GREAT_S	a > b = c	[%ID/value, %ID/value, %ID]
		Vestavěné		
62	LENGHT	LENGTH_S	vratí délku stringu	[%ID/value, %ID]
63	PRINT	PRINT_S	vytiskne string na standardní výstup	[%ID/value]
64	READ	READ_S	přečte string ze standardního vstupu	[%ID]
65	GETTYPE		vrátí číselnou hodnotu typu	%ID/value, %ID
66	SORT	SORT_S	seřadí textový řetězec	[%ID/value, %ID]
67	FIND		najde pozřetězec a v řetezci b a vrátí jeho pozici v c	%ID/value, %ID/value, %ID
69	CONVERT	CONVERT_S	změní typ hodnoty na %TYPE	%TYPE[, %ID/value, %ID]
71	EXISTS	EXISTS_S	zjistí, zda symbol a existuje, hodnotu uloží do b	%ID[, %ID]

Vysvetlivky	
[]	zasobnikove funkce berou udaje z [] ze zasobniku
a, b, c	operandy instrukce v tomto pořadí

PŘÍLOHA 3

PODPORA VÝVOJE A TESTOVÁNÍ INTERPRETŮ JEDNODUCHÝCH JAZYKŮ

ZDROJOVÝ TEXT, DEFINIČNÍ SOUBOR PRO KNIHOVNU, NEZÁSOBNÍKOVÁ INSTRUKČNÍ SADA

```
<?xml version="1.0" encoding="utf-8"?>
<IRdefinition>
  <instructions>
    <instruction>
      <codename>DEF</codename>
      <ID>4</ID>
      <Operands>1</Operands>
    </instruction>
    <instruction>
      <codename>DEFS</codename>
      <ID>5</ID>
      <Operands>2</Operands>
    </instruction>
    <instruction>
      <codename>DEFD</codename>
      <ID>6</ID>
      <Operands>2</Operands>
    </instruction>
    <instruction>
      <codename>GDEF</codename>
      <ID>4</ID>
      <Operands>1</Operands>
    </instruction>
    <instruction>
      <codename>GDEFS</codename>
      <ID>5</ID>
      <Operands>2</Operands>
    </instruction>
    <instruction>
      <codename>GDEFD</codename>
      <ID>6</ID>
      <Operands>2</Operands>
    </instruction>
    <instruction>
      <codename>ASSIGN</codename>
      <ID>7</ID>
      <Operands>2</Operands>
    </instruction>
    <instruction>
      <codename>UNDEF</codename>
      <ID>8</ID>
      <Operands>1</Operands>
    </instruction>
    <instruction>
      <codename>LABEL</codename>
      <ID>11</ID>
      <Operands>1</Operands>
    </instruction>
    <instruction>
      <codename>JUMP</codename>
      <ID>12</ID>
      <Operands>1</Operands>
    </instruction>
    <instruction>
```

```
  <codename>JTRUE</codename>
  <ID>13</ID>
  <Operands>2</Operands>
</instruction>
<instruction>
  <codename>JFALSE</codename>
  <ID>14</ID>
  <Operands>2</Operands>
</instruction>
<instruction>
  <codename>EXIT</codename>
  <ID>15</ID>
  <Operands>0</Operands>
</instruction>
<instruction>
  <codename>FUNC</codename>
  <ID>19</ID>
  <Operands>1</Operands>
</instruction>
<instruction>
  <codename>CALL</codename>
  <ID>20</ID>
  <Operands>1</Operands>
</instruction>
<instruction>
  <codename>END</codename>
  <ID>21</ID>
  <Operands>0</Operands>
</instruction>
<instruction>
  <codename>RET</codename>
  <ID>22</ID>
  <Operands>0</Operands>
</instruction>
<instruction>
  <codename>RETV</codename>
  <ID>23</ID>
  <Operands>1</Operands>
</instruction>
<instruction>
  <codename>GETRETV</codename>
  <ID>24</ID>
  <Operands>1</Operands>
</instruction>
<instruction>
  <codename>PUSHP</codename>
  <ID>25</ID>
  <Operands>1</Operands>
</instruction>
<instruction>
  <codename>POPP</codename>
  <ID>26</ID>
  <Operands>1</Operands>
</instruction>
<instruction>
  <codename>ADD</codename>
  <ID>36</ID>
  <Operands>3</Operands>
</instruction>
```

```
<instruction>
  <codename>SUB</codename>
  <ID>37</ID>
  <Operands>3</Operands>
</instruction>
<instruction>
  <codename>MULT</codename>
  <ID>38</ID>
  <Operands>3</Operands>
</instruction>
<instruction>
  <codename>DIV</codename>
  <ID>39</ID>
  <Operands>3</Operands>
</instruction>
<instruction>
  <codename>MODULO</codename>
  <ID>40</ID>
  <Operands>3</Operands>
</instruction>
<instruction>
  <codename>DEC</codename>
  <ID>41</ID>
  <Operands>1</Operands>
</instruction>
<instruction>
  <codename>INC</codename>
  <ID>42</ID>
  <Operands>1</Operands>
</instruction>
<instruction>
  <codename>NEG</codename>
  <ID>46</ID>
  <Operands>2</Operands>
</instruction>
<instruction>
  <codename>AND</codename>
  <ID>47</ID>
  <Operands>3</Operands>
</instruction>
<instruction>
  <codename>OR</codename>
  <ID>48</ID>
  <Operands>3</Operands>
</instruction>
<instruction>
  <codename>XOR</codename>
  <ID>49</ID>
  <Operands>3</Operands>
</instruction>
<instruction>
  <codename>EQUAL</codename>
  <ID>53</ID>
  <Operands>3</Operands>
</instruction>
<instruction>
  <codename>NEQUAL</codename>
  <ID>54</ID>
  <Operands>3</Operands>
```

```
    </instruction>
    <instruction>
      <codename>LESS</codename>
      <ID>55</ID>
      <Operands>3</Operands>
    </instruction>
    <instruction>
      <codename>GREAT</codename>
      <ID>56</ID>
      <Operands>3</Operands>
    </instruction>
    <instruction>
      <codename>PRINT</codename>
      <ID>63</ID>
      <Operands>1</Operands>
    </instruction>
    <instruction>
      <codename>READ</codename>
      <ID>64</ID>
      <Operands>1</Operands>
    </instruction>
    <instruction>
      <codename>LENGHT</codename>
      <ID>62</ID>
      <Operands>2</Operands>
    </instruction>
    <instruction>
      <codename>GETTYPE</codename>
      <ID>65</ID>
      <Operands>2</Operands>
    </instruction>
    <instruction>
      <codename>SORT</codename>
      <ID>66</ID>
      <Operands>1</Operands>
    </instruction>
    <instruction>
      <codename>FIND</codename>
      <ID>67</ID>
      <Operands>3</Operands>
    </instruction>
    <instruction>
      <codename>CONVERT</codename>
      <ID>69</ID>
      <Operands>3</Operands>
    </instruction>
    <instruction>
      <codename>EXISTS</codename>
      <ID>71</ID>
      <Operands>2</Operands>
    </instruction>
  </instructions>
</IRdefinition>
```

PŘÍLOHA 4

PODPORA VÝVOJE A TESTOVÁNÍ INTERPRETŮ JEDNODUCHÝCH JAZYKŮ

ZDROJOVÝ TEXT, DEFINIČNÍ SOUBOR PRO INTERPRET, NEZÁSOBNÍKOVÁ INSTRUKČNÍ SADA

```
<?xml version="1.0" encoding="utf-8"?>
<ArrayOfInstructionDefinition xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <InstructionDefinition>
    <Codename>DEF</Codename>
    <Operation>@RuntimeInfo.DefineSymbol</Operation>
    <NumOfInputOperands>1</NumOfInputOperands>
    <OperationDestination>null</OperationDestination>
    <SourceOfOperands>raw</SourceOfOperands>
  </InstructionDefinition>
  <InstructionDefinition>
    <Codename>DEFD</Codename>
    <Operation>@RuntimeInfo.DefineDynamicSymbol</Operation>
    <NumOfInputOperands>2</NumOfInputOperands>
    <OperationDestination>null</OperationDestination>
    <SourceOfOperands>raw</SourceOfOperands>
  </InstructionDefinition>
  <InstructionDefinition>
    <Codename>DEFS</Codename>
    <Operation>@RuntimeInfo.DefineStaticSymbol</Operation>
    <NumOfInputOperands>2</NumOfInputOperands>
    <OperationDestination>null</OperationDestination>
    <SourceOfOperands>raw</SourceOfOperands>
  </InstructionDefinition>
  <InstructionDefinition>
    <Codename>GDEF</Codename>
    <Operation>@RuntimeInfo.DefineGlobalSymbol</Operation>
    <NumOfInputOperands>1</NumOfInputOperands>
    <OperationDestination>null</OperationDestination>
    <SourceOfOperands>raw</SourceOfOperands>
  </InstructionDefinition>
  <InstructionDefinition>
    <Codename>GDEFD</Codename>
    <Operation>@RuntimeInfo.DefineGlobalDynamicSymbol</Operation>
    <NumOfInputOperands>2</NumOfInputOperands>
    <OperationDestination>null</OperationDestination>
    <SourceOfOperands>raw</SourceOfOperands>
  </InstructionDefinition>
  <InstructionDefinition>
    <Codename>GDEFS</Codename>
    <Operation>@RuntimeInfo.DefineGlobalStaticSymbol</Operation>
    <NumOfInputOperands>2</NumOfInputOperands>
    <OperationDestination>null</OperationDestination>
    <SourceOfOperands>raw</SourceOfOperands>
  </InstructionDefinition>
  <InstructionDefinition>
    <Codename>ASSIGN</Codename>
    <Operation></Operation>
    <NumOfInputOperands>1</NumOfInputOperands>
    <OperationDestination>code</OperationDestination>
    <SourceOfOperands>code</SourceOfOperands>
  </InstructionDefinition>
  <InstructionDefinition>
```

```
  <Codename>UNDEF</Codename>
  <Operation>@RuntimeInfo.DeleteSymbol</Operation>
  <NumOfInputOperands>1</NumOfInputOperands>
  <OperationDestination>null</OperationDestination>
  <SourceOfOperands>raw</SourceOfOperands>
</InstructionDefinition>
<InstructionDefinition>
  <Codename>LABEL</Codename>
  <Precompile>@RuntimeInfo.RegisterLabel</Precompile>
  <NumOfInputOperands>1</NumOfInputOperands>
  <OperationDestination>null</OperationDestination>
  <SourceOfOperands>raw</SourceOfOperands>
</InstructionDefinition>
<InstructionDefinition>
  <Codename>JUMP</Codename>
  <Operation>@RuntimeInfo.JumpToLabel</Operation>
  <NumOfInputOperands>1</NumOfInputOperands>
  <OperationDestination>null</OperationDestination>
  <SourceOfOperands>raw</SourceOfOperands>
</InstructionDefinition>
<InstructionDefinition>
  <Codename>JTRUE</Codename>
  <Operation>@RuntimeInfo.JumpIfTrue</Operation>
  <NumOfInputOperands>2</NumOfInputOperands>
  <OperationDestination>null</OperationDestination>
  <SourceOfOperands>raw</SourceOfOperands>
</InstructionDefinition>
<InstructionDefinition>
  <Codename>JFALSE</Codename>
  <Operation>@RuntimeInfo.JumpIfFalse</Operation>
  <NumOfInputOperands>2</NumOfInputOperands>
  <OperationDestination>null</OperationDestination>
  <SourceOfOperands>raw</SourceOfOperands>
</InstructionDefinition>
<InstructionDefinition>
  <Codename>EXIT</Codename>
  <Operation>@RuntimeInfo.ExitProgram</Operation>
  <NumOfInputOperands>0</NumOfInputOperands>
  <OperationDestination>null</OperationDestination>
  <SourceOfOperands>null</SourceOfOperands>
</InstructionDefinition>
 <InstructionDefinition>
  <Codename>FUNC</Codename>
  <Precompile>@RuntimeInfo.RegisterFunction</Precompile>
  <Operation>@RuntimeInfo.JumpToEnd</Operation>
  <NumOfInputOperands>1</NumOfInputOperands>
  <OperationDestination>null</OperationDestination>
  <SourceOfOperands>raw</SourceOfOperands>
</InstructionDefinition>
<InstructionDefinition>
  <Codename>CALL</Codename>
  <Operation>@RuntimeInfo.CallFunction</Operation>
  <NumOfInputOperands>1</NumOfInputOperands>
  <OperationDestination>null</OperationDestination>
  <SourceOfOperands>raw</SourceOfOperands>
</InstructionDefinition>
<InstructionDefinition>
  <Codename>END</Codename>
  <Precompile>@RuntimeInfo.RegisterEnd</Precompile>
```

```
  <Operation>@RuntimeInfo.Return</Operation>
  <NumOfInputOperands>0</NumOfInputOperands>
  <OperationDestination>null</OperationDestination>
  <SourceOfOperands>null</SourceOfOperands>
</InstructionDefinition>
<InstructionDefinition>
  <Codename>RET</Codename>
  <Operation>@RuntimeInfo.Return</Operation>
  <NumOfInputOperands>0</NumOfInputOperands>
  <OperationDestination>null</OperationDestination>
  <SourceOfOperands>null</SourceOfOperands>
</InstructionDefinition>
<InstructionDefinition>
  <Codename>RETV</Codename>
  <Operation>@RuntimeInfo.SetReturnValue</Operation>
  <NumOfInputOperands>1</NumOfInputOperands>
  <OperationDestination>null</OperationDestination>
  <SourceOfOperands>raw</SourceOfOperands>
</InstructionDefinition>
<InstructionDefinition>
  <Codename>GETRETV</Codename>
  <Operation>@RuntimeInfo.GetReturnValue</Operation>
  <NumOfInputOperands>0</NumOfInputOperands>
  <OperationDestination>code</OperationDestination>
  <SourceOfOperands>null</SourceOfOperands>
</InstructionDefinition>
<InstructionDefinition>
  <Codename>PUSHP</Codename>
  <Operation>@RuntimeInfo.PushToCallStack</Operation>
  <NumOfInputOperands>1</NumOfInputOperands>
  <OperationDestination>null</OperationDestination>
  <SourceOfOperands>raw</SourceOfOperands>
</InstructionDefinition>
<InstructionDefinition>
  <Codename>POPP</Codename>
  <Operation>@RuntimeInfo.PopFromCallStack</Operation>
  <NumOfInputOperands>0</NumOfInputOperands>
  <OperationDestination>code</OperationDestination>
  <SourceOfOperands>null</SourceOfOperands>
</InstructionDefinition>
<InstructionDefinition>
  <Codename>ADD</Codename>
  <Operation>Basic.cs@Basic.Add</Operation>
  <NumOfInputOperands>2</NumOfInputOperands>
  <OperationDestination>code</OperationDestination>
  <SourceOfOperands>code</SourceOfOperands>
</InstructionDefinition>
<InstructionDefinition>
  <Codename>SUB</Codename>
  <Operation>Basic.cs@Basic.Sub</Operation>
  <NumOfInputOperands>2</NumOfInputOperands>
  <OperationDestination>code</OperationDestination>
  <SourceOfOperands>code</SourceOfOperands>
</InstructionDefinition>
<InstructionDefinition>
  <Codename>MULT</Codename>
  <Operation>Basic.cs@Basic.Mult</Operation>
  <NumOfInputOperands>2</NumOfInputOperands>
  <OperationDestination>code</OperationDestination>
```

```
  <SourceOfOperands>code</SourceOfOperands>
</InstructionDefinition>
<InstructionDefinition>
  <Codename>DIV</Codename>
  <Operation>Basic.cs@Basic.Div</Operation>
  <NumOfInputOperands>2</NumOfInputOperands>
  <OperationDestination>code</OperationDestination>
  <SourceOfOperands>code</SourceOfOperands>
</InstructionDefinition>
<InstructionDefinition>
  <Codename>MODULO</Codename>
  <Operation>Basic.cs@Basic.Modulo</Operation>
  <NumOfInputOperands>2</NumOfInputOperands>
  <OperationDestination>code</OperationDestination>
  <SourceOfOperands>code</SourceOfOperands>
</InstructionDefinition>
<InstructionDefinition>
  <Codename>DEC</Codename>
  <Operation>@RuntimeInfo.Decrement</Operation>
  <NumOfInputOperands>1</NumOfInputOperands>
  <OperationDestination>null</OperationDestination>
  <SourceOfOperands>raw</SourceOfOperands>
</InstructionDefinition>
<InstructionDefinition>
  <Codename>INC</Codename>
  <Operation>@RuntimeInfo.Increment</Operation>
  <NumOfInputOperands>1</NumOfInputOperands>
  <OperationDestination>null</OperationDestination>
  <SourceOfOperands>raw</SourceOfOperands>
</InstructionDefinition>
<InstructionDefinition>
  <Codename>NEG</Codename>
  <Operation>Basic.cs@Basic.Neg</Operation>
  <NumOfInputOperands>1</NumOfInputOperands>
  <OperationDestination>code</OperationDestination>
  <SourceOfOperands>code</SourceOfOperands>
</InstructionDefinition>
<InstructionDefinition>
  <Codename>AND</Codename>
  <Operation>Basic.cs@Basic.And</Operation>
  <NumOfInputOperands>2</NumOfInputOperands>
  <OperationDestination>code</OperationDestination>
  <SourceOfOperands>code</SourceOfOperands>
</InstructionDefinition>
<InstructionDefinition>
  <Codename>OR</Codename>
  <Operation>Basic.cs@Basic.Or</Operation>
  <NumOfInputOperands>2</NumOfInputOperands>
  <OperationDestination>code</OperationDestination>
  <SourceOfOperands>code</SourceOfOperands>
</InstructionDefinition>
<InstructionDefinition>
  <Codename>XOR</Codename>
  <Operation>Basic.cs@Basic.Xor</Operation>
  <NumOfInputOperands>2</NumOfInputOperands>
  <OperationDestination>code</OperationDestination>
  <SourceOfOperands>code</SourceOfOperands>
</InstructionDefinition>
<InstructionDefinition>
```

```
  <Codename>EQUAL</Codename>
  <Operation>Basic.cs@Basic.Equal</Operation>
  <NumOfInputOperands>2</NumOfInputOperands>
  <OperationDestination>code</OperationDestination>
  <SourceOfOperands>code</SourceOfOperands>
</InstructionDefinition>
<InstructionDefinition>
  <Codename>NEQUAL</Codename>
  <Operation>Basic.cs@Basic.NotEqual</Operation>
  <NumOfInputOperands>2</NumOfInputOperands>
  <OperationDestination>code</OperationDestination>
  <SourceOfOperands>code</SourceOfOperands>
</InstructionDefinition>
<InstructionDefinition>
  <Codename>LESS</Codename>
  <Operation>Basic.cs@Basic.Lesser</Operation>
  <NumOfInputOperands>2</NumOfInputOperands>
  <OperationDestination>code</OperationDestination>
  <SourceOfOperands>code</SourceOfOperands>
</InstructionDefinition>
<InstructionDefinition>
  <Codename>GREAT</Codename>
  <Operation>Basic.cs@Basic.Greater</Operation>
  <NumOfInputOperands>2</NumOfInputOperands>
  <OperationDestination>code</OperationDestination>
  <SourceOfOperands>code</SourceOfOperands>
</InstructionDefinition>
<InstructionDefinition>
  <Codename>PRINT</Codename>
  <Operation>Basic.cs@Basic.Print</Operation>
  <NumOfInputOperands>1</NumOfInputOperands>
  <OperationDestination>null</OperationDestination>
  <SourceOfOperands>code</SourceOfOperands>
</InstructionDefinition>
<InstructionDefinition>
  <Codename>READ</Codename>
  <Operation>Basic.cs@Basic.Read</Operation>
  <NumOfInputOperands>0</NumOfInputOperands>
  <OperationDestination>code</OperationDestination>
  <SourceOfOperands>null</SourceOfOperands>
</InstructionDefinition>
<InstructionDefinition>
  <Codename>LENGHT</Codename>
  <Operation>Basic.cs@Basic.GetLenght</Operation>
  <NumOfInputOperands>1</NumOfInputOperands>
  <OperationDestination>code</OperationDestination>
  <SourceOfOperands>code</SourceOfOperands>
</InstructionDefinition>
<InstructionDefinition>
  <Codename>GETTYPE</Codename>
  <Operation>@RuntimeInfo.GetSymbolType</Operation>
  <NumOfInputOperands>1</NumOfInputOperands>
  <OperationDestination>code</OperationDestination>
  <SourceOfOperands>raw</SourceOfOperands>
</InstructionDefinition>
<InstructionDefinition>
  <Codename>SORT</Codename>
  <Operation>Basic.cs@Basic.Sort</Operation>
  <NumOfInputOperands>1</NumOfInputOperands>
```

```
    <OperationDestination>code</OperationDestination>
    <SourceOfOperands>code</SourceOfOperands>
  </InstructionDefinition>
  <InstructionDefinition>
    <Codename>FIND</Codename>
    <Operation>Basic.cs@Basic.Find</Operation>
    <NumOfInputOperands>2</NumOfInputOperands>
    <OperationDestination>code</OperationDestination>
    <SourceOfOperands>code</SourceOfOperands>
  </InstructionDefinition>
  <InstructionDefinition>
    <Codename>CONVERT</Codename>
    <Operation>@RuntimeInfo.ConvertSymbol</Operation>
    <NumOfInputOperands>2</NumOfInputOperands>
    <OperationDestination>code</OperationDestination>
    <SourceOfOperands>raw</SourceOfOperands>
  </InstructionDefinition>
  <InstructionDefinition>
    <Codename>EXISTS</Codename>
    <Operation>@RuntimeInfo.Exists</Operation>
    <NumOfInputOperands>1</NumOfInputOperands>
    <OperationDestination>code</OperationDestination>
    <SourceOfOperands>raw</SourceOfOperands>
  </InstructionDefinition>
</ArrayOfInstructionDefinition>
```

PŘÍLOHA 5

PODPORA VÝVOJE A TESTOVÁNÍ INTERPRETŮ JEDNODUCHÝCH JAZYKŮ

OBSAH PŘILOŽENÉHO CD/DVD

- Složka Docs – obsahuje všechny vytvořené dokumenty
    - Složka Original – originální soubory *.docx
    - Soubory *.pdf
- Složka ExeWindows – obsahuje všechny potřebné soubory pro spuštění na platformě MS Windows a definiční soubory instrukčních sad
    - Složka test_programs – testovací soubory, programy napsané v instrukční sadě
    - Soubor IR_Interpreter.exe
    - Soubor IR_Reader_DLL.dll
    - Jiné soubory
- Složka IR_Reader – obsahuje zdrojové soubory knihovny pro načítání a zápis intermediální reprezentace
    - Složka Other – pro kompilaci prostředím GCC
    - Složka Win – pro kompilaci na MS Windows
- Složka MSVSProjekt – obsahuje projekt vytvořený v MS Visual Studio 2015 a zdrojové soubory interpretu
    - Soubor projektu je v další složce IR_Reader s názvem IR_Reader.sln
- Soubor README.txt