
Semestrální práce z KIV/FJP

# Tvorba překladače zvoleného jazyka

Vladimíra Kimlová – A23N0102P
(jivl@students.zcu.cz)

Dominik Zappe – A23N0011P
(zapped99@students.zcu.cz)

`https://github.com/SpeekeR99/ZS23_FJP_Kimlova_Zappe`

Leden 2024

# Obsah

**1 Zadání** **1**

**2 Popis zvoleného jazyka** **4**
2.1 Zvolená bodovaná rozšíření . . . . . 4

**3 Popis implementace překladače** **7**
3.1 Popis adresářové struktury projektu . . . . . 7
3.2 Implementace . . . . . 7
3.2.1 Lexikální analýza . . . . . 7
3.2.2 Syntaktická analýza . . . . . 7
3.2.3 Sémantická analýza . . . . . 8
3.2.4 Optimalizace . . . . . 8
3.2.5 Generování instrukcí . . . . . 8
3.3 Datové typy . . . . . 8
3.3.1 Omezení datových typů . . . . . 9
3.4 Testování . . . . . 9

**4 Popis vylepšení interpreteru** **10**
4.1 Opravy a vylepšení existujícího . . . . . 10
4.2 Vlastní vylepšení . . . . . 10
4.2.1 Float . . . . . 10
4.2.2 Ostatní . . . . . 11

**5 Uživatelská příručka** **12**
5.1 Překlad . . . . . 12
5.2 Spuštění . . . . . 12

**6 Závěr** **13**

**A Gramatika jazyka** **14**

**B Grafická podoba parseru** **18**

**C Příklady** **19**

# 1 Zadání

Cílem práce bude vytvoření překladače zvoleného jazyka. Je možné inspirovat se jazykem PL/0, vybrat si podmnožinu nějakého existujícího jazyka nebo si navrhnout jazyk zcela vlastní. Dále je také potřeba zvolit si pro jakou architekturu bude jazyk překládán (doporučeny jsou instrukce PL/0, ale je možné zvolit jakoukoliv instrukční sadu pro kterou budete mít interpret).

Jazyk musí mít minimálně následující konstrukce:

- definice celočíselných proměnných
- definice celočíselných konstant
- přiřazení
- základní aritmetiku a logiku (+, -, *, /, AND, OR, negace a závorky, operátory pro porovnání čísel)
- cyklus (libovolný)
- jednoduchou podmínku (if bez else)
- definice podprogramu (procedura, funkce, metoda) a jeho volání

Překladač, který bude umět tyto základní věci bude hodnocen deseti body. Další body (alespoň do minimálních 20) je možné získat na základě rozšíření, jsou rozděleny do dvou skupin, jednodušší za jeden bod a složitější za dva až tři body. Další rozšíření je možno doplnit po konzultaci, s ohodnocením podle odhadnuté náročnosti.

Jednoduchá rozšíření (1 bod):

- každý další typ cyklu (for, do .. while, while .. do, repeat .. unitl, foreach pro pole)
- else větev
- datový typ boolean a logické operace s ním
- datový typ real (s celočíselnými instrukcemi)
- datový typ string (s operátory pro spojování řetězců)
- rozvětvená podmínka (switch, case)
- násobné přiřazení (a = b = c = d = 3;)
- podmíněné přiřazení / ternární operátor (min = (a ¡ b) ? a : b;)

- paralelní přiřazení (a, b, c, d = 1, 2, 3, 4;)
- příkazy pro vstup a výstup (read, write - potřebuje vhodné instrukce které bude možné využít)

Složitější rozšíření (2 body):

- příkaz GOTO (pozor na vzdálené skoky)
- datový typ ratio (s celočíselnými instrukcemi)
- složený datový typ (Record)
- pole a práce s jeho prvky
- operátor pro porovnání řetězců
- parametry předávané hodnotou
- návratová hodnota podprogramu
- objekty bez polymorfismu
- anonymní vnitřní funkce (lambda výrazy)

Rozšíření vyžadující složitější instrukční sadu než má PL/0 (3 body):

- dynamicky přiřazovaná paměť - práce s ukazateli
- parametry předávané odkazem
- objektové konstrukce s polymorfním chováním
- instanceof operátor
- anonymní vnitřní funkce (lambda výrazy) které lze předat jako parametr
- mechanismus zpracování výjimek

Vlastní interpret (řádkový, bez rozhraní, složitý alespoň jako rozšířená PL/0) je za 6 bodů.

Kromě toho že by program měl fungovat se zohledňují i další věci, které mohou pozitivně nebo negativně ovlivnit bodování:

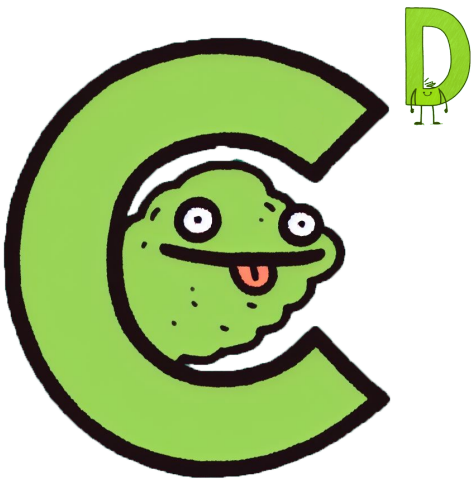
- Testování - tvorba rozumné automatické testovací sady +3 body (pro inspiraci hledejte test suit pro LLVM nebo se podívejte na Plum Hall testy, ale jde skutečně jen o inspiraci, stačí výrazně jednodušší řešení). Užitečné a stručné povídání na dané téma najdete také tady

- Kvalita dokumentace -x bodů až +2 body podle kvality a prohřešků (vynechaná gramatika, nesrozumitelné věty, příliš chyb a překlepů, bitmapové obrázky pro diagramy s kompresními artefakty, ...).
- Vedení projektu v GITu -x bodů až +2 body podle důslednosti a struktury příspěvků.
- Kvalita zdrojového textu -x bodů až +2 body podle obecně známých pravidel ze ZSWI, PPA a podobně (magická čísla, struktura programu a dekompozice problému, božské třídy a metody, ...)

Nepovinnou částí semestrální práce (za +1 až +5 bodů podle kvality) je článek na české Wikipedii který se bude týkat problematiky formálních jazyků, překladačů, automatů nebo příbuzných témat. Může jít i o překlad textu který je jen v angličtině nebo rozsáhlejší úpravu / opravu stávajícího textu.

# 2 Popis zvoleného jazyka

Námi zvolený jazyk je silně inspirován jazykem C. Z tohoto důvodu jsme ho pojmenovali **yadc** – „*yet another degenerated C*“ (je nám jasné, že nejsme první, co napodobují a degenerují jazyk C).



Obrázek 1: Logo yadc vygenerované od umělé inteligence

Jako cílová platforma byly zvoleny instrukce rozšířené PL/0. Instrukční sada rozšířené PL/0 nabízí instrukce jako základní instrukce PL/0 – tj. práce s literály (**LIT**), proměnnými (**STO**, **LOD**), aritmeticko-logické operace (**OPR**), skoky (**JMP**, **JMC**, **CAL**, **RET**) a I/O operace (**REA**, **WRI**). Rozšíření navíc nabízí instrukce pro práci s haldou (**NEW**, **DEL**, **LDA**, **STA**), dynamickou práci se zásobníkem (**PLD**, **PST**) a datový typ float (**ITR**, **RTI**) a aritmetiku s floaty (**OPF**).

Popis navrženého jazyka yadc v podobě gramatiky je v Příloze A.

## 2.1 Zvolená bodovaná rozšíření

Jazyk samozřejmě splňuje minimální základní konstrukce – definice celočíselných konstant a proměnných, přiřazení, základní aritmetiku a logiku, cykly, jednoduché podmínky a definice podprogramu a jeho volání. Tento základ by měl být hodnocen 10 body.

Ze seznamu jednoduchých rozšíření (hodnoceny 1 bodem) byla vybrána – další typy cyklů (konkrétně **while**, **do while**, **for**, **until do** a **repeat until**), else větev podmínky, datový

typ boolean + operace s ním, datový typ real + operace s ním, datový typ string + operace s ním, podmíněné přiřazení (ternární operátor) a příkazy pro vstup a výstup (I/O). Celkem by tato sekce měla být za 10 bodů.

Z listu složitějších vylepšení (hodnoceno 2 body) byla zvolena následující – příkaz goto, pole a práce s jeho prvky, operátor pro porovnání stringů, parametry předávané hodnotou, návratová hodnota z podprogramu a anonymní vnitřní funkce. Celkem by tato vylepšení měla být za 12 bodů.

Z poslední kategorie rozšíření vyžadujících složitější instrukční sadu než nabízí základní PL/0 byly vybrány – dynamicky přiřazovaná paměť, pointery a parametry předávané odkazem. Celkem by tato rozšíření měla být za 6 bodů.

Naše práce implementuje navíc více rozšíření, než jaká byla vyjmenována v zadání, nelze však jednoduše říci kolika body je hodnotit. Následuje výčet našich rozšíření:

- Chybové hlášky lexikálních, syntaktických a sémantických chyb
- Vlastní AST – modulární implementace spojená s návrhovým vzorem visitor
- Možnost deklarace globálních proměnných kdekoliv (klidně i mezi deklaracemi funkcí) (kdekoliv – myšleno v rámci globálního scopu)
- Možnost deklarace lokálních proměnných kdekoliv (nejen na začátku bloku)
- Možnost pouhé deklarace hlavičky funkce + dodeklarování těla později (inspirace v C)
- Příkazy break a continue uvnitř cyklů
- Návratová hodnota z hlavní funkce je zachycena v globálním bloku (bez globálních proměnných na adrese 3, jinak 3 + velikost datových typů globálních proměnných)
- Vyřešení built-in funkcí, které se při použití ve zdrojovém kódu vygenerují na začátku instrukcí
- Možnost explicitního přetypovávání
- Implicitní přetypovávání intu na floaty, když se jedná o binární operaci, kde jeden z operandů je float (je možné implicitní přetypování přebít explicitním a rozbít si zásobník)
- Jednoduchý type checking přiřazení, parametrů a return statementů
- Optimalizace nad AST (zjednodušení aritmetiky a logiky pro binární operace)
- Optimalizace nad vygenerovanými instrukcemi (zbavení se zbytečných skoků, které vedou na další nepodmíněné skoky)

V naší práci jsme navíc rozšiřovali již existující semestrální práci z minulých let – jedná se o webový interpreter instrukční sady rozšířené PL/0. Naše práce na tomto interpreteru by se dala shrnout do následujících bodů:

- Opravení I/O jako celku – instrukce WRI např. mazala znak ze vstupu
- Opravení I/O při spouštění pomocí „run“ – I/O fungovalo pouze při krokování
- Oprava nápovědy – PLD a PST byly chybně popsány (adresa a level jsou prohozeny)
- Vylepšení o float – instrukce ITR, RTI a OPF (implementace instrukcí, vysvětlovač, nápověda)
- Vylepšení o nenutnost číslování instrukcí (instrukce lze nyní psát bez explicitního očíslování – lépe se píše ručně pseudo assembly PL/0)
- Vstupní pole nyní může přijímat speciální ASCII znaky jako CR, LF (použito jako ukončovací znak pro vstup)
- Kompletní překlad nápovědy do češtiny

Jako poslední bodované kritérium bylo v zadání zmíněno testování, dokumentace, vedení projektu v GITu a kvalita kódu. Testování bylo provedeno pomocí zmiňovaného webového interpreteru a nástroje Selenium WebDriver. Projekt je veden na GitHubu na adrese `https://github.com/SpeekeR99/ZS23_FJP_Kimlova_Zappe`.

# 3 Popis implementace překladače

Hlavní zvolenou technologií byl jazyk C, resp. C++. Z tohoto důvodu byly zvoleny nástroje lex a yacc, resp. flex a bison. Pro účely testování byl užit nástroj Selenium WebDriver a jazyk Python. Mimo standardní knihovny jazyka C++ nebyly užity žádné jiné externí.

## 3.1 Popis adresářové struktury projektu

Projekt obsahuje v kořenovém adresáři 5 důležitých adresářů – doc, examples, interpreter, src a test. V adresáři doc se nachází tato dokumentace, prezentace a logo jazyka. V adresáři examples se nachází soubory s příponou .yadc – název našeho jazyka. Jedná se o příklady použití různých konstrukcí v našem jazyce. Číslované příklady slouží jako tutoriály. Adresář interpreter bude detailněji popsán později, ale ve zkratce se jedná o fork semestrální práce z minulých let – webový intepreter rozšířené instrukční sady PL/0. Adresář src obsahuje zdrojové kódy překladače. Obsahuje 2 podadresáře a zdrojové kódy k abstraktnímu syntaktickému stromu, symbolické tabulce a hlavnímu skriptu. Zmiňované podadresáře se jmenují analysis a synthesis. Podadresář analysis obsahuje tokenizer napsaný ve flexu, parser napsaný v bisonu a třídu určenou pro syntaktickou analýzu. Podadresář synthesis obsahuje analyzér semántiky, generátor instrukcí a built-in funkcí a třídu starající se o optimalizace. Adresář test obsahuje skript napsaný v jazyce Python užívající nástroj Selenium WebDriver a testované případy (každý testovaný případ má svůj podadresář).

## 3.2 Implementace

### 3.2.1 Lexikální analýza

O veškerou lexikální analýzu se stará soubor /src/analysis/tokenizer.l – tedy nástroj flex. Z důvodu zpětného dohledání syntaktické chyby se stará o číslování řádků a sloupců. Precedence pravidel je zajištěná pořadím pravidel – předposlední pravidlo je pro identifikátor a úplně poslední pravidlo je pro zachycení neznámého tokenu.

### 3.2.2 Syntaktická analýza

Syntaktickou analýzu obaluje třída /src/analysis/SyntaxAnalyzer.h, která pouze předává vstupní soubor do skriptu nástroje bison /src/analysis/parser.y, a pak předává kořen vytvořeného abstraktního syntaktického stromu. Parser z bisonu se snaží odchytávat syntaktické chyby a používá k tomu předané informace z flexu o číslu řádku a sloupce. Předaná gramatika bisonu je atributovaná a už se tedy jedná i o první průchod

sémantickou analýzou, ve kterém se ale pouze vytváří abstraktní syntaktický strom, který je později užit pro možnost více průchodů sémantickou analýzou – modularita.

Vsuvka – bison umí generovat tzv. **DOT** formát (resp. GraphViz formát) souboru, který se dá použít pro vygenerování grafické podoby grafu parseru – viz Obrázek 3 v Příloze B.

### 3.2.3 Sémantická analýza

Jak již bylo řečeno, v prvním průchodu se pouze generuje abstraktní syntaktický strom a je již součást syntaktické analýzy.

V druhém průchodu, který zajišťuje třída /src/analysis/SemanticAnalyzer.h se kontroluje smysluplnost syntakticky přijatých konstrukcí. Používá se k tomu tabulka symbolů (/src/SymbolTable.h), díky které je možné kontrolovat existenci referovaných proměnných, volaných funkcí atp. V tomto průchodu se také kontrolují typy při přiřazení, typy argumentů při volání funkcí, typy návratových hodnot atp. Sémantický analyzér je implementován pomocí návrhového vzoru visitor nad AST.

### 3.2.4 Optimalizace

Optimalizacemi se zabývá třída /src/analysis/Optimizer.h, nabízí funkce pro optimalizace nad AST a pro optimalizace nad vygenerovanými instrukcemi. Optimalizátor je opět implementací návrhového vzoru visitor nad AST – jediná optimalizace nad AST je optimalizace aritmetiky a logiky – tj. dělení 1, násobení 1 / 0, přičítání / odčítání 0, and true, or false atp. Optimalizované nad vygenerovanými instrukcemi jsou skoky – pokud skok vede na další nepodmíněný skok je zbytečné dělat „meziskoky“ a je možné se zbavit prostředních nepodmíněných skoků jednoduchou změnou destinace původního skoku. Důležité je však později správně opravit číslování všech zbylých instrukcí po tomto zásahu, který maže instrukce.

### 3.2.5 Generování instrukcí

Generátor instrukcí je opět implementací návrhového vzoru visitor nad AST (/src/analysis/InstructionGenerator.h). Vygenerované instrukce jsou vyspány do standardního výstupu a zároveň jsou uloženy do souboru instructions.txt.

## 3.3 Datové typy

Základní datové typy jsou – int, bool, float a string. Řetězce jsou realizovány jako pole znaků v haldě. Pole jsou obecně realizovány jako pointery do haldy. Pointery nemusejí

samozřejmě být pouze do haldy, lze si udělat pointer na „jakýkoliv“ datový typ a ukazovat na proměnnou daného typu.

### 3.3.1 Omezení datových typů

Hlavním omezením je nemožnost float pointeru, neboť instrukční sada rozšířené PL/0 na to není připravená. Nechtěli jsme předělávat instrukční sadu, ale v zásadě je hlavní problém v tom, že instrukce pracující dynamicky se zásobníkem (resp. instrukce pracující s haldou) očekávají na zásobníku adresu a hodnotu. V případě floatu (větší rozsah) ale není hodnota pouze na jednu datovou buňku, bylo by zapotřebí mít instrukci očekávající na zásobníku např. adresu a hodnotu a hodnotu. Tato skutečnost je ošetřená sémantickou chybou s hlášením, že float pointery nejsou podporované.

Dalším omezením u pointerů je level pointeru. Správně fungují pouze jednoduché pointery – ukazatel na proměnnou nebo ukazatel do haldy (např. 1D pole). Teoreticky je možné založit i pointery s vyšším levelem, ale není snadné v našem jazyce udělat 2D pole – muselo by se postupovat pozpátku, nejprve se vytvoří 1D pole (vnitřní pole), a poté se vytvoří „pole polí“ a postupně se reference dosadí do nadřazeného pole. Takto by šlo udělat libovolně velké pole, ale není to klasický postup, na který jsou programátoři zvyklý, tj. nejprve vytvořit nadřazené pole, a až později dodělávat podřazená pole s nižší dimenzí (viz /examples/matrix.yadc). Obecně je však problém s vícenásobnou dereferencí v případě řetězu pointerů, který končí pointerem na proměnnou.

## 3.4 Testování

Jak již bylo zmíněno, veškeré testované případy a testovací skripty se nacházejí v adresáři /src/test. Naše testy předpokládají, že je na adrese http://localhost:3000/ spuštěný webový interpreter. Jako defaultní prohlížeč se spouští Firefox. Po dokončení všech testů je vygenerován textový soubor results.txt obsahující výsledky jednotlivých testovaných případů. Obecně není možné testovat kontrolováním přesné podoby zásobníku (potažmo i haldy). Je tedy užíváno I/O operací a předpokládá se tedy jejich správné chování. Každý testovací případ má definovaný uživatelský vstup (nemusí také být žádný) a očekáváný výstup programu (ten musí existovat pro každý test, jinak by neměly smysl). Je tedy testován pouze správný výstup programu, nikoliv správný postup programu – to prakticky není možné.

# 4 Popis vylepšení interpreteru

V naší práci jsme chtěli zavést floaty, ale zjistili jsme, že webový interpreter rozšířené instrukční sady PL/0 neměl instrukce pro float implementované. Rozhodli jsme se proto si tyto instrukce dodělat sami. Vytvořili jsme proto fork původní práce `https://github.com/SpeekeR99/ZS23_FJP_PL0_INTERPRETER_Kimlova_Zappe`(původní práce: `https://github.com/lukasvlc3k/fjp`)

## 4.1 Opravy a vylepšení existujícího

Hlavním cílem našich oprav byl modul obsluhující I/O operace. Instrukce WRI, která má vypsat ASCII znak do výstupního pole navíc ubírala znak ze vstupního pole. Dále nebylo vůbec možné spustit programem módem „run“, když obsahoval instrukce REA a WRI (I/O operace). Bylo možné pouze program krokovat. Po našich úpravách již WRI neničí vstupní sekvenci a je navíc možné si program spustit bez nutnosti krokování, i když obsahuje I/O operace.

Dalším cílem oprav byla nápověda, kde popis instrukcí PLD a PST je zavádějící, protože je zaměněno pořadí adresy a levelu.

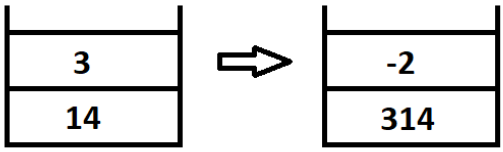
## 4.2 Vlastní vylepšení

### 4.2.1 Float

Největším vylepšením na webovém interpreteru je zavedení datového typu float. Chtěli jsme, aby float neměl jednotkovou velikost, aby si programátor musel uvědomit, že ne každý datový typ nutně zabírá jednu datovou buňku (= má jednotkovou velikost). Byl tedy po domluvě se cvičícím navržen tak, že zabírá dvě datové buňky, kde jedna datová buňka reprezentuje mantisu a druhá reprezentuje exponent (v desítkové soustavě).

Instrukce ITR by měla přetvářet celočíselný typ na float. V praxi to funguje tak, že se vezmou dvě hodnoty ze zásobníku a reprezentují se jako celá část desetinného čísla a desetinná část desetinného čísla. Na pozadí se tyto dvě čísla převedou na mantisu odpovídající exponent a jsou vloženy zpět na zásobník. Příklad převedení na float je vidět na Obrázku 2

Instrukce RTI je inverzní operací k instrukci ITR. Ze zásobníku se vezmou dvě čísla, reprezentují se jako mantisa a exponent a převedou se na celou a desetinnou část čísla.




Obrázek 2: Převod celočíselné části (3) a desetinné části (14) na float ve tvaru $314 \cdot 10^{-2}$

Instrukce OPF je obdobnou instrukce OPR, jedná se o aritmetiku a logiku nad floaty. Veškerá implementovaná logika je nad mantisou a exponentem, nikoliv nad aproximací opravdové float hodnoty v TypeScriptu – tím by se ztrácelo na přesnosti. Jedná se tedy prakticky o implementaci plovoucí desetinné čárky v desítkové soustavě. Implementace této instrukce byla nejvíce náročná. Samotná logika float aritmetiky byla poměrně netriviální a dále byla obtížná implementace vysvětlovače v prostředí cizího projektu – navzdory tomu, že vysvětlovač byl od původních autorů dobře připravený a rozšiřitelný.

V neposlední řadě byla doplněna nápověda (anglická i česká) o popis zmíněných instrukcí.

### 4.2.2 Ostatní

Vstupní pole pro I/O operace bylo změněno tak, aby bylo možné v něm odenterovat a nechat tak na vstup protéct speciální znaky jako CR, LF atp. (např. ASCII 10). Konkrétně ASCII 10 je v naší práci užita jako ukončovací znak vstupní sekvence.

Celá nápověda byla dostupná pouze v anglickém jazyce, tak jsme ji kompletně přeložili do češtiny.

V kódu jsme doplnili přepínač při kontrole vstupních instrukcí – v naší verzi není nutné číslovat jednotlivé instrukce. Je tedy mnohem jednodušší psát ručně pseudo assembly PL/0, protože se instrukce nemusí uvozovat číslem řádky.

# 5 Uživatelská příručka

Aplikace je přeložitelná a spustitelná pouze na platformě linux, neboť je závislá na linuxových nástrojích lex/flex a yacc/bison. Za předpokladu existence portu těchto aplikací pro platformu Windows by aplikace měla být schopná fungovat i v prostředí Windows.

Nejprve je vhodné si stáhnout, resp. naklonovat, projekt z GitHubu pomocí příkazu:

git clone https://github.com/SpeekeR99/ZS23_FJP_Kimlova_Zappe.git

(repozitář sice obsahuje submodule, ale není nutné ho inicializovat, jedná se pouze o upravovaný interpreter – postačuje tedy obyčejný git clone příkaz)

## 5.1 Překlad

Aplikace je přeložitelná za pomoci nástroje **cmake**.

Na platformě Linux se projekt přeloží následovně:

```
user@pc:/yadc$ mkdir build
user@pc:/yadc$ cd build
user@pc:/yadc/build$ cmake ../
user@pc:/yadc/build$ make
```

(předpokladem je, že adresář pojmenován „yadc“ označuje kořenový adresář projektu z GitHubu)

## 5.2 Spuštění

Aplikace přebírá jeden až dva parametry z příkazové řádky. Nutný a postačující parametr pro správné spuštění programu je vstupní soubor obsahující kód napsaný v jazyce yadc. Druhý, tentokrát volitelný, parametr je přepínač optimalizací. Defaultně jsou optimalizace zapnuté a je možné je vypnout parametrem ve tvaru -o=0.

Spuštění na platformě Linux může vypadat následovně:

```
user@pc:/yadc/build$ ./yadc input.yadc -o=0
```

(přípona .yadc není nutná)

Po dokončení překladu je v konzoli buď vypsána chybová hláška, nebo vygenerované instrukce, které jsou rovněž uloženy do souboru instructions.txt.

# 6 Závěr

V rámci semestrální práce byl vytvořen překladač pro námi vymyšlený jazyk yadc. Překladač byl vytvořen pomocí nástrojů flex, pomocí něhož byla udělána lexikální analýza, a bison, který byl využit pro syntaktickou analýzu, navíc díky atributované gramatice byl nástroj také užit pro první průchod sémantické analýzy. Další průchody byly prováděny nad AST s pomocí tabulky symbolů, jejich implementace byla provedena v C++. Cílovou platformou pro nás byly instrukce rozšířené PL/0, ale vzhledem k modularitě řešení by se do budoucna dalo překládat i pro jiné platformy. Kromě samotné tvorby překladače byl navíc opraven a rozšířen webový interpreter pro instrukční sadu rozšířené PL/0.

# A Gramatika jazyka

Neterminální symboly psané velkými písmeny označují tokeny z předchozí lexikální analýzy.

⟨*program*⟩ ::= ⟨*program*⟩ ⟨*decl_var_stmt*⟩
| ⟨*program*⟩ ⟨*decl_func_stmt*⟩
| /* empty */

⟨*decl_var_stmt*⟩ ::= TYPE ID SEMICOLON
| TYPE ⟨*ptr_modifier*⟩ ID SEMICOLON
| CONSTANT TYPE ID SEMICOLON
| CONSTANT TYPE ⟨*ptr_modifier*⟩ ID SEMICOLON
| TYPE ID ASSIGN_OP ⟨*expr*⟩ SEMICOLON
| TYPE ⟨*ptr_modifier*⟩ ID ASSIGN_OP ⟨*expr*⟩ SEMICOLON
| CONSTANT TYPE ID ASSIGN_OP ⟨*expr*⟩ SEMICOLON
| CONSTANT TYPE ⟨*ptr_modifier*⟩ ID ASSIGN_OP ⟨*expr*⟩ SEMICOLON
| TYPE ID ASSIGN_OP ⟨*expr*⟩ QUESTION ⟨*expr*⟩ COLON ⟨*expr*⟩ SEMICOLON
| TYPE ⟨*ptr_modifier*⟩ ID ASSIGN_OP ⟨*expr*⟩ QUESTION ⟨*expr*⟩ COLON ⟨*expr*⟩ SEMICOLON
| CONSTANT TYPE ID ASSIGN_OP ⟨*expr*⟩ QUESTION ⟨*expr*⟩ COLON ⟨*expr*⟩ SEMICOLON
| CONSTANT TYPE ⟨*ptr_modifier*⟩ ID ASSIGN_OP ⟨*expr*⟩ QUESTION ⟨*expr*⟩ COLON ⟨*expr*⟩ SEMICOLON

⟨*ptr_modifier*⟩ ::= ⟨*ptr_modifier*⟩ DEREF
| DEREF

⟨*decl_func_stmt*⟩ ::= TYPE ID L_BRACKET ⟨*params*⟩ R_BRACKET ⟨*block*⟩
| TYPE ID L_BRACKET ⟨*params*⟩ R_BRACKET SEMICOLON

⟨*params*⟩ ::= ⟨*params_list*⟩
| /* empty */

⟨*params_list*⟩ ::= ⟨*params_list*⟩ COMMA TYPE ID
| ⟨*params_list*⟩ COMMA TYPE ⟨*ptr_modifier*⟩ ID

| TYPE ID
| TYPE ⟨*ptr_modifier*⟩ ID

⟨*block*⟩ ::= BEGIN_BLOCK ⟨*stmts*⟩ END_BLOCK

⟨*stmts*⟩ ::= ⟨*stmts*⟩ ⟨*stmt*⟩
| ⟨*stmts*⟩ ID COLON ⟨*stmt*⟩
| /* empty */

⟨*stmt*⟩ ::= SEMICOLON
| ⟨*decl_func_stmt*⟩
| ⟨*decl_var_stmt*⟩
| ⟨*assign_stmt*⟩
| ⟨*if_stmt*⟩
| ⟨*loop_stmt*⟩
| ⟨*jump_stmt*⟩
| ⟨*expr*⟩ SEMICOLON

⟨*if_stmt*⟩ ::= IF L_BRACKET ⟨*expr*⟩ R_BRACKET ⟨*block*⟩ ⟨*else_stmt*⟩

⟨*else_stmt*⟩ ::= ELSE ⟨*block*⟩
| /* empty */

⟨*loop_stmt*⟩ ::= ⟨*while_stmt*⟩
| ⟨*do_while_stmt*⟩
| ⟨*until_do_stmt*⟩
| ⟨*repeat_until_stmt*⟩
| ⟨*for_stmt*⟩

⟨*while_stmt*⟩ ::= WHILE L_BRACKET ⟨*expr*⟩ R_BRACKET ⟨*block*⟩

⟨*do_while_stmt*⟩ ::= DO ⟨*block*⟩ WHILE L_BRACKET ⟨*expr*⟩ R_BRACKET SEMICOLON

⟨*until_do_stmt*⟩ ::= UNTIL L_BRACKET ⟨*expr*⟩ R_BRACKET ⟨*block*⟩

⟨*repeat_until_stmt*⟩ ::= DO ⟨*block*⟩ UNTIL L_BRACKET ⟨*expr*⟩ R_BRACKET SEMICOLON

⟨*for_stmt*⟩ ::= FOR L_BRACKET ⟨*expr*⟩ SEMICOLON ⟨*expr*⟩ SEMICOLON ⟨*expr*⟩ R_BRACKET ⟨*block*⟩
| FOR L_BRACKET ⟨*decl_var_stmt*⟩ ⟨*expr*⟩ SEMICOLON ⟨*expr*⟩ R_BRACKET ⟨*block*⟩

⟨*jump_stmt*⟩ ::= ⟨*break_stmt*⟩
| ⟨*continue_stmt*⟩
| ⟨*return_stmt*⟩
| ⟨*goto_stmt*⟩

⟨*break_stmt*⟩ ::= BREAK SEMICOLON

⟨*continue_stmt*⟩ ::= CONTINUE SEMICOLON

⟨*return_stmt*⟩ ::= RETURN ⟨*expr*⟩ SEMICOLON
| RETURN SEMICOLON

⟨*goto_stmt*⟩ ::= GOTO ID SEMICOLON

⟨*expr*⟩ ::= ID
| INT_LITERAL
| BOOL_LITERAL
| STRING_LITERAL
| FLOAT_LITERAL
| L_BRACKET ⟨*expr*⟩ R_BRACKET
| ⟨*assign_expr*⟩
| ⟨*arithm_expr*⟩
| ⟨*logic_expr*⟩
| ⟨*compare_expr*⟩
| ⟨*cast_expr*⟩
| ⟨*call_func_expr*⟩
| ⟨*memory_expr*⟩

⟨*assign_expr*⟩ ::= ID ASSIGN_OP ⟨*expr*⟩
| ID ASSIGN_OP ⟨*expr*⟩ QUESTION ⟨*expr*⟩ COLON ⟨*expr*⟩
| ⟨*expr*⟩ ASSIGN_OP ⟨*expr*⟩
| ⟨*expr*⟩ ASSIGN_OP ⟨*expr*⟩ QUESTION ⟨*expr*⟩ COLON ⟨*expr*⟩

⟨*arithm_expr*⟩ ::= ⟨*expr*⟩ SUM ⟨*expr*⟩
| ⟨*expr*⟩ SUB ⟨*expr*⟩
| ⟨*expr*⟩ MUL ⟨*expr*⟩
| ⟨*expr*⟩ DIV ⟨*expr*⟩
| ⟨*expr*⟩ MOD ⟨*expr*⟩
| U_MINUS ⟨*expr*⟩ /* unary minus */

⟨*logic_expr*⟩ ::= ⟨*expr*⟩ AND ⟨*expr*⟩
| ⟨*expr*⟩ OR ⟨*expr*⟩
| NOT ⟨*expr*⟩

⟨*compare_expr*⟩ ::= ⟨*expr*⟩ EQ ⟨*expr*⟩
| ⟨*expr*⟩ NEQ ⟨*expr*⟩
| ⟨*expr*⟩ LESS ⟨*expr*⟩
| ⟨*expr*⟩ LESSEQ ⟨*expr*⟩
| ⟨*expr*⟩ GRT ⟨*expr*⟩
| ⟨*expr*⟩ GRTEQ ⟨*expr*⟩

⟨*cast_expr*⟩ ::= L_BRACKET TYPE R_BRACKET ⟨*expr*⟩

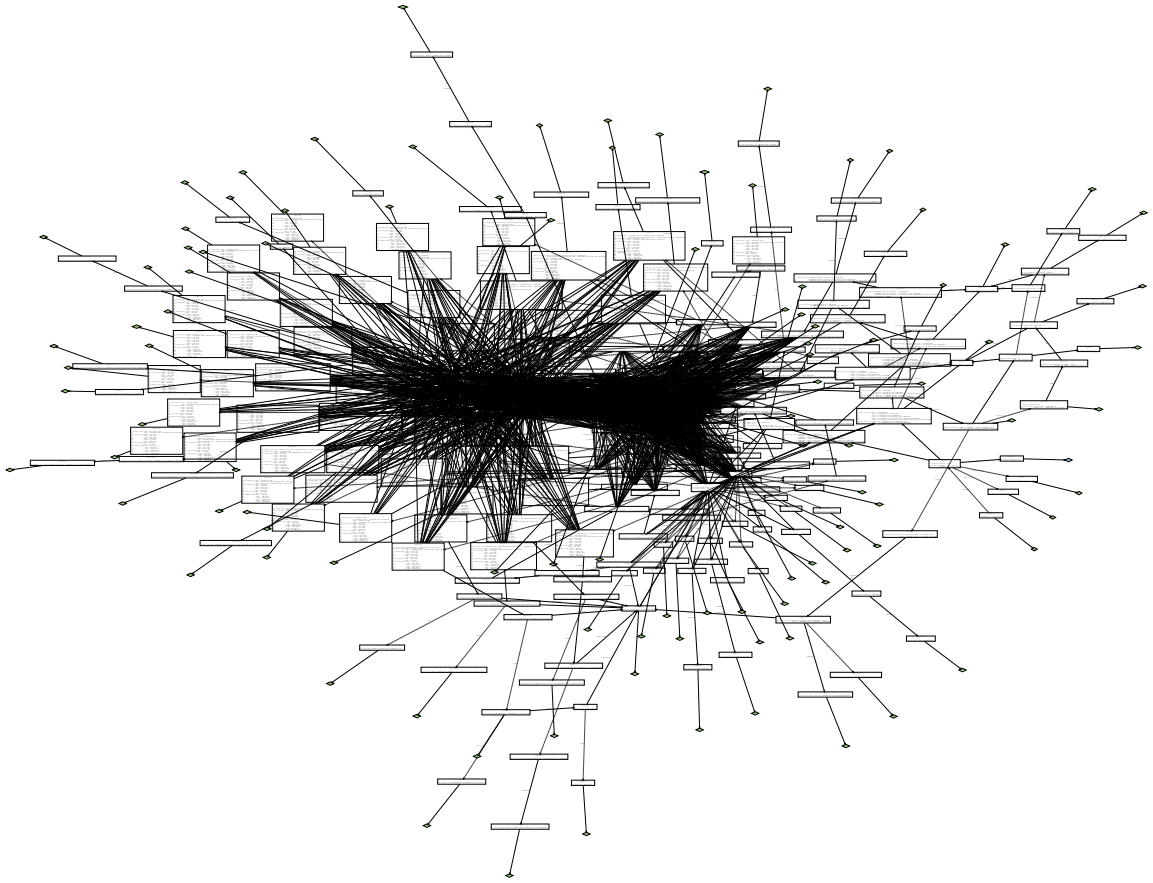
⟨*call_func_expr*⟩ ::= ID L_BRACKET ⟨*args*⟩ R_BRACKET

⟨*args*⟩ ::= ⟨*args_list*⟩
| /* empty */

⟨*args_list*⟩ ::= ⟨*args_list*⟩ COMMA ⟨*expr*⟩
| ⟨*expr*⟩

⟨*memory_expr*⟩ ::= NEW L_BRACKET TYPE COMMA ⟨*expr*⟩ R_BRACKET
| DELETE ⟨*expr*⟩
| DEREF ⟨*expr*⟩
| REF ID
| SIZEOF L_BRACKET TYPE R_BRACKET

# B Grafická podoba parseru



Obrázek 3: Grafická podoba parseru po použití linux utility sfdp s dodatečným přepínačem -Goverlap=prism, který slouží k ubrání překryvů v grafu

# C Příklady

Příklad 1: součet globální a lokální proměnné

```
1 int a = 5;
2
3 int main() {
4     int b = 2;
5     int c = a + b;
6
7     return 0;
8 }
```

```
0  JMP  0  1
1  INT  0  4
2  LIT  0  5
3  STO  0  3
4  JMP  0  17
5  INT  0  3
6  INT  0  2
7  LIT  0  2
8  STO  0  3
9  LOD  1  3
10 LOD  0  3
11 OPR  0  2
12 STO  0  4
13 LIT  0  0
14 STO  0  −1
15 RET  0  0
16 INT  0  −2
17 INT  0  1
18 CAL  0  5
19 RET  0  0
```

Příklad 2: dodeklarování těla funkce a volání funkce

```
int sum(int a, int b);

int sub(int a, int b) {
    return a - b;
}

int main() {
    int c = sum(1, 2);
    return 0;
}

int sum(int a, int b) {
    return a + b;
}
```

```
0  JMP  0  1
1  INT  0  3
2  JMP  0  4
3  JMP  0  31
4  JMP  0  17
5  INT  0  3
6  INT  0  2
7  LOD  0  -2
8  STO  0  3
9  LOD  0  -1
10 STO  0  4
11 LOD  0  3
12 LOD  0  4
13 OPR  0  3
14 STO  0  -3
15 RET  0  0
16 INT  0  -2
17 JMP  0  30
18 INT  0  3
19 INT  0  1
20 INT  0  1
21 LIT  0  1
22 LIT  0  2
23 CAL  1  3
24 INT  0  -2
25 STO  0  3
26 LIT  0  0
27 STO  0  -1
28 RET  0  0
29 INT  0  -1
30 JMP  0  43
31 INT  0  3
32 INT  0  2
33 LOD  0  -2
34 STO  0  3
35 LOD  0  -1
36 STO  0  4
37 LOD  0  3
38 LOD  0  4
39 OPR  0  2
40 STO  0  -3
41 RET  0  0
42 INT  0  -2
43 INT  0  1
44 CAL  0  18
45 RET  0  0
```

Jelikož je množství vygenerovaných instrukcí poněkud velké, rádi bychom se spíše odkázali na složku s příklady přímo v projektu – /src/examples.