



Semestrální práce z KIV/FJP

Překladač C-- do virtuálního stroje PL/0

Václav Honzík, Jiří Noháč

https://github.com/honzikv/KIV_FJP

Zadání	2
Implementované konstrukce	3
Popis jazyka	3
Gramatika	4
Implementace	6
ANTLR Visitor	6
Struktury překladače	6
Překlad	7
PL0 Interpreter	7
Omezení jazyka	7
Ukázky a použití struktur jazyka	8
Práce s proměnnými	8
Definice funkce a její volání	8
Funkce s cykly a podmínkou	9
Is operátor (instanceof / typeof)	9
Typová kontrola	10
Chained přiřazení	10
Sestavení a spuštění projektu	11
Sestavení aplikace	11
Linux sestavení	11
Windows sestavení	11
Příklad spuštění	11
Argumenty programu	12
Help	12
Vstupní bod	12
Výstupní bod	12
Debug mód	13
Interpreter	13
Závěr	15

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 - nepište vlastní interpret, jednou z podúloh je vypořádat se s omezeními danými zvolenou platformou).

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. Je třeba vybrat alespoň jedno rozšíření z každé kategorie (tedy ne jen dělat ta triviální). Další rozšíření je možno doplnit po konzultaci, s ohodnocením podle odhadnuté náročnosti.

Implementované konstrukce

Kromě minimálních konstrukcí jsme vybrali a implementovali následující jazykové konstrukce:

- Datový typ boolean (1b)
- For cyklus, while cyklus, do while cyklus (2b - jeden cyklus byl povinný)
- Else větve (1b)
- Chained přiřazení ($x = y = z = 2$) (1b)
- Předávání parametrů do funkce hodnotou (2b)
- Návrátová hodnota funkce (2b)
- Typeof operátor (3b)
- Typová kontrola (3b)

Popis jazyka

C-- je inspirováno jazykem C. Jazyk využívá pure ("čisté") funkce s návratovými hodnotami a dvěma datovými typy - integer (int) a boolean (bool). Gramatika jazyka se nachází v souboru **CMM.g4** (ve složce src/main/antlr/grammar) a je napsána pro nástroj ANTLR.

Základními strukturami jazyka jsou definice funkcí (function definition) a příkazy (statement). Funkční program v C-- musí obsahovat alespoň jeden statement a nebo jednu definici funkce. Statementy se dále člení na specifické typy jako je např. blok kódu (obsahuje v sobě statementy), deklarace a inicializace proměnných, a další.

Programovací jazyk je designován na "top-level" zpracování příkazů - tzn. můžeme psát příkazy rovnou, bez toho abychom je museli definovat ve funkci. Definice funkcí nejsou na top-level statementech závislé (překladač je zpracuje jako první), takže můžeme dokonce i volat funkci před tím, než ji definujeme (podobně jako např. v Javě nebo C#).

Jazyk je striktně a silně typovaný a obsahuje základní typovou kontrolu, která uživateli např. nedovolí operace mezi odlišnými datovými typy, nebo přiřazovat proměnným nevalidní hodnoty. Nelze tím pádem např. sečíst celé číslo a booleovskou proměnnou.

Gramatika

Tato sekce obsahuje základní konstrukce jazyka. Zápis připomíná ANTLR, ale je pro přehlednost zjednodušený - některé tokeny jsou nahrazeny symboly. Symbol otazníku znamená nula až jeden výskyt.

entrypoint: (functionDefinition | statement)+

functionDefinition: functionDataTypes IDENTIFIER '(' functionParameters? ')'

functionBlockScope

functionBlockScope: '{' statement* returnStatement '}'

returnStatement: return (expression)? ';'

functionDataTypes: 'void' | 'int' | 'bool'

statement: blockScope | if parenthesesExpression blockscope (ELSE blockscope)?

| FOR forExpression blockScope

| WHILE parenthesesExpression blockScope

| DO blockScope WHILE parenthesesExpression

| variableDeclaration | variableInitialization | constVariableInitialization | variableAssignment

| functionCall

chainAssignment: '=' IDENTIFIER

variableAssignment: IDENTIFIER chainAssignment* '=' (legalVariableLiterals | expression) ';'

variableDeclaration: legalDataTypes IDENTIFIER ';'

variableInitialization: legalDataTypes IDENTIFIER '=' (legalVariableLiterals | expression) ';'

constVariableInitialization: 'const' variableInitialization

parenthesesExpression: '(' expression ')'

expression: functionCall

| (IDENTIFIER | legalDataTypes) 'is' legalDataTypes

| valueExpr

| IDENTIFIER

| expression operation = ('/' | '%' | '*') expression

| expression operation = ('+' | '-') expression

| expression operation = ('>' | '>=' | '==' | '<' | '<=' | '!=') expression

| expression operation = ('&&' | '||') expression

| '(' expression ')'

| '!' expression

| operation = ('+' | '-') expression

legalDataTypes: 'int' | 'bool'

valueExpr: INTEGER_NUMBER | 'true' | 'false'

functionCall: IDENTIFIER '(' expressionChain? ')'

expressionChain: (IDENTIFIER | expression) (',' expressionChain)?

Gramatika “začíná” symbolem entypoint, který obsahuje vždy alespoň jednu definici funkce (functionDefinition) a nebo jeden příkaz (statement). Definice funkce má svůj návratový typ, identifikátor (viz dále), parametry (nemusí být definované) a tělo (functionBlockScope). Tělo funkce je stejné jako normální “scope” se složenými závorkami (blockScope), ale musí obsahovat jako poslední příkaz return - nelze se tedy z funkce vrátit předčasně, např. ve větvi if else.

Samotný neterminální symbol příkazu (statement) se může přepisovat na velké množství dalších symbolů, jako if-else, for cyklus, while cyklus nebo inicializaci / deklaraci proměnných (viz například symboly variableAssignment, variableDeclaration, variableInitialization, constVariableInitialization).

Dalším důležitým symbolem, který stojí za zmínění je výraz (expression), který se může přepisovat na velké množství operátorů, hodnotu (valueExpr), uzavřít se, identifikátor proměnné a nebo volání funkce.

Identifikátor proměnné / funkce může být cokoliv co začíná písmenem nebo podtržítkem (_) a případně obsahuje 0 až N písmen, číslic nebo podtržítkek. Posledním důležitým symbolem je volání funkce, které je opět ekvivalentní s jazykem C - tzn. identifikátor funkce, závorky a výrazy pro parametry.

Implementace

Překladač pro jazyk je implementovaný v jazyce Java 17 s několika přidávanými knihovnami:

- **ANTLR** - slouží ke generování lexeru a parseru
- **Apache Commons Lang3** - utilitní knihovna - abychom nemuseli implementovat základní věci jako parsování + využití Pair
- **Lombok** - generování konstruktorů, getterů, setterů, toString nebo equals s hashCode přes anotace, což sníží velké množství boiler plate kódu

Práce jako taková je strukturovaná ve formě Maven projektu. Základem aplikace je lexer a parser, který vygeneruje nástroj ANTLR z gramatiky CMM.g4.

ANTLR Visitor

Visitor je třídou pro průchod parsovacího stromu, který ANTLR vygeneruje při zpracování daného vstupu. Tento design pattern lze použít, abychom prošli celý strom a extrahovali jsme z něj potřebná data, která uložíme do vlastních struktur nutných pro překlad. Ve výchozím nastavení ANTLRu (přes antlr4-maven-plugin) si můžeme nechat vygenerovat prázdnou generickou implementaci **BaseVisitor**, jež lze extendovat a tím vložit potřebnou implementaci.

BaseVisitor obsahuje metody, které se volají, když se narazí na daný symbol gramatiky - např. inicializace proměnné, if statement, apod. Při oddělení této třídy můžeme jednotlivé metody přepsat a vytvořit tak visitory pro specifické jazykové konstrukce. Všechny vlastní visitory se nachází v balíku **parser.visitor**. Základním je EntrypointVisitor, který zpracovává kořen parsovacího stromu a rekurzivně volá visitory pro další konstrukce.

Struktury překladače

Po zavolání EntrypointVisitoru v programu získáme instanci třídy Entrypoint, která obsahuje seznam statementů a seznam definic funkcí, a které se využijí při samotném překladu. Pro překlad do jazyka se využívají tzv. "Processory" - tzn. objekt, který zpracuje daná data a vygeneruje k nim relevantní instrukce.

Princip je opět stejný - formou rekurzivního sestupu, kdy processor buď zpracuje daný prvek rovnou - pokud může, a nebo zavolá jiný processor, který to udělá za něj. Specifické processory pro daný typ jazykové konstrukce jsou umístěné v balíku **compiler.compiletime.processor**.

Pro generování instrukcí se používá třída **GeneratorContext**, která také ukládá aktuální stav při generování - tabulku proměnných, odkaz na vrchol zásobníku nebo tabulku funkcí. Tento kontext se typicky vytváří nový při zpracování funkce nebo blokového scope, aby jsme mohli např. garantovat lokální proměnné a korektní operace s nimi. Kontext si ukládá referenci na rodiče a pokud hledáme proměnnou, která v něm není, zavolá metodu v rodiči.

Překlad

Než se začnou instrukce generovat, je potřeba provést preprocessing funkcí, protože potřebujeme nějakým způsobem vytvořit místo pro předávání argumentů a návratové hodnoty. To byl při vývoji relativně problém. Původně jsme zkoušeli si parametry umístit na vrchol zásobníku a ve funkci je vybrat, což ale bohužel není možné, protože nemáme jakým způsobem v programu konec zásobníku referencovat. Implementované řešení tedy místo toho alokuje paměť staticky a při každém volání funkce se na předem danou adresu uloží hodnoty.

Po preprocessingu se začne zpracovávat samotný kód funkcí (pomocí processorů), před který se ještě umístí instrukce pro nepodmíněný skok, aby se funkce v programu nezačaly vykonávat bez jejich volání. Tento způsob nicméně funguje pouze díky tomu, že jsou funkce pure - jinak bychom museli zpracovat první samotné příkazy a definovat nějakou hlavní funkci, která by sloužila jako vstupní bod. Po dokončení tohoto kroku se jako poslední zpracují samotné top-level statements.

PL0 Interpreter

Poslední částí naší práce bylo pak vytvořené instrukce do PL0 nějakým způsobem vyzkoušet, jestli doopravdy dělají, to co mají. Z tohoto důvodu jsme se rozhodli zpracovat a naimplementovat vlastní PL0 interpreter.

Interpreter jsme implementovali přímo uvnitř překladače jako spustitelný část po samotném vygenerování PL0 kódu. Třída zodpovědná za implementaci simulace PL0 instrukcí se nazývá **PL0Inter**. Náš simulátor nejprve načte všechny instrukce, které se mají provést do seznamu. Následně pak ve smyčce provádí načtení první instrukce. Provede konkrétní instrukci, na jejímž základě upraví konkrétní hodnoty na zásobníku nebo změny další volané instrukce. Tento proces funguje do té doby, dokud se nenarazí na poslední instrukci v sadě.

Pro správné fungování je samozřejmostí práce se zásobníkem, bází, vrcholem zásobníku, ale také se statickou a dynamickou bází při volání CAL operace. Součástí našeho simulátoru jsou pak základní operace INT, LIT, LOD, STO, CAL, RET, OPR, JMP, JMC.

Omezení jazyka

Mezi hlavní omezení jazyka patří:

- Pouze datové typy int a boolean
- Z funkce se lze vrátit pouze v posledním statementu
- Funkce jsou pure - void funkce tím pádem nic nedělá, parametry lze předávat pouze hodnotou
- Nelze provést dynamickou alokaci paměti
- Limitace na základní instrukční sadu PL0 bez vstupu a výstupu - stav lze zobrazit pouze v interpreteru

Ukázky a použití struktur jazyka

Tato sekce ukazuje základní použití jazyka. Další příklady jsou umístěné ve složce **testinputs**. V projektu je obsažen také python skript - **batch_run.py**, pomocí kterého lze celou složku přeložit (kromě souboru `type_control_compile_err.cmm`, který má cíleně selhat). Přeložený výstup se umístí do složky **batch_run_output**.

Práce s proměnnými

Proměnné lze deklarovat (proměnná `c`). Deklarace je “náhodně” (přes `Random` v Javě), což je jakýsi pokus o easter egg, protože v jazyce C dostaneme deklarací hodnotu v paměti, která je nepredikovatelná. Proměnné mohou být i konstanty (proměnná `b`), které nelze přepsat, a nebo je lze inicializovat přímo (proměnná `a`).

```
int a = 2;
const int b = 2;
int c;
c = 2;
```

Definice funkce a její volání

Funkce musí končit příkazem `return`. Funkce mohou být i návratového typu `void` (tzn. nic nevrací), nicméně to je efektivně k ničemu, protože nemohou modifikovat mimo svůj scope – tato (ne)funkcionalita nám zbyla navíc, protože jsme původně ještě přemýšleli o pointerech.

```
int foo(int bar) {
    return bar * bar + bar - bar;
}
```

```
int my_variable = foo(10);
```

Funkce s cykly a podmínkou

Top-level statementy můžou být v kódu před definicemi funkcí, nebo se mohou i mixovat.

```
int magic_variable1 = magic_function(5, true);
int magic_function(int magic_param1, bool magic_param2) {
    int sum = 0;
    if (magic_param2) {
        sum = 100 + magic_param1;
    }
    else {
        sum = 1000 + magic_param1;
    }

    for (i in 0..10) {
        sum = sum + 10;
    }

    while (false) {
        sum = 0;
    }

    do {
        sum = sum + 100;
    }
    while (false)
    return sum;
}
```

Is operátor (instanceof / typeof)

Pro operátor “is” můžeme použít název proměnné a nebo datový typ. Operátor vrací true, pokud se datové typy rovnají.

```
int a = 2;
if (a is int) {
    a = 3000;
}
if (!(int is int)) {
    a = -1;
}
```

Typová kontrola

Typová kontrola nám zabrání v alokaci nebo nevalidními operacemi mezi datovými typy.

Následující dva výrazy nejdou přeložit:

```
int a = true;
bool b = 200;
```

Zde dostaneme chybu, protože dělíme integer a boolean:

```
int a = 0;
bool b = true;

int c = b + a;
```

Nicméně ve výrazech, které se porovnávají se integer automaticky převede na boolean, aby šlo podmínku vyhodnotit:

```
int a;
if ((5 < 3 && 3 - 2 < 0) || true) {
    a = 10000;
}
else {
    a = -10000;
}
```

Chained přiřazení

Pomocí chained přiřazení můžeme nastavit hodnotu proměnných v jedné řádce:

```
int a;
int b;
int c;

const int constant = 200;
a = b = c = constant * constant - 2 * constant;
```

Sestavení a spuštění projektu

Sestavení aplikace

Pro sestavení aplikace jsme připravili dva soubory. Jeden soubor je určen pro *Linux* operační systém. Druhý je pak připraven pro Windows operační systém.

Linux sestavení

K sestavení programu je zapotřebí mít na lokálním stroji nainstalovaný Maven ve verzi 3.x a Javu ve verzi 17 a výše. Pro samotné sestavení je pak v projektu přiložen soubor **compile.sh**, který spustíme takto v příkazové řádce:

./compile.sh

Windows sestavení

K sestavení programu je zapotřebí mít na lokálním stroji nainstalovaný Maven ve verzi 3.x a Javu ve verzi 17 a výše. Pro samotné sestavení je pak v projektu přiložen soubor **compile.bat**, který spustíme takto v příkazové řádce:

./compile.bat

Po spuštění obou skriptů vznikne .jar soubor, který se pak nalezne v kořeni projektu s názvem:

fjp-sem-runnable.jar

Příklad spuštění

Nyní když již máme sestavený spustitelný program v .jar souboru, si můžeme předvést, jak vypadá základní spuštění aplikace. Naším cílem bude načíst náš kód C– do překladače a pak výstup zapsat do konzole. Zavoláme tedy z příkazové řádky tento příkaz:

java -jar fjp-sem-runnable.jar -i <vstupní_soubor>

```
d:\WorkSpaceFJPC\KIV_FJP>java -jar fjp-seme-runnable.jar -i testcode.txt

0 INT 0 6
1 JMP 0 33
2 INT 0 6
3 LIT 0 1444543815
4 LOD 1 3
5 STO 0 6
6 LIT 0 -160989287
7 LOD 1 4
8 STO 0 7
9 LOD 0 6
10 LOD 0 7
11 OPR 0 2
12 STO 1 5
13 INT 0 0
14 RET 0 0
15 INT 0 6
```

Obrázek č.1: Příklad spuštění překladače

Argumenty programu

Při samotném spuštění aplikace dokážeme přijmout několik vstupních argumentů a jejich parametrů.

Help

Pokud si uživatel není jist co všechno překladač nabízí, připravili jsme jednoduchý výpis všech argumentů a jejich stručný popis do konzole. Help si lze zobrazit parametrem **-h** či **-H** jako argument za název .jar souboru překladače. Help se také zobrazí, pokud zadá uživatel neznáme argumenty! Příklad zápisu:

java -jar fjp-sem-runnable.jar -h -i <vstupní_soubor>

Vstupní bod

Překladač pro správné spuštění aplikace vyžaduje nějaký vstupní soubor, ze kterého načte konkrétní kód ke zpracování. K tomu slouží argument **-i** či **-I**. Společně s tímto argument je nutno dodat název souboru jako další argument. Pokud nebyl argument s předáním názvu souboru nalezen nebo soubor neexistuje, je vyhozena výjimka s konkrétní chybou a program ukončen. Příklad spuštění:

java -jar fjp-sem-runnable.jar -i <vstupní_soubor>

Výstupní bod

Výstupním bodem aplikace může být buď výpis do konzole nebo zápis instrukcí do souboru. Za tímto účelem pak slouží parametr **-o** či **-O**(jedná se o, jako Oliver, né o nulu). Společně s tímto

argument je nutno dodat název souboru jako další argument. Pokud nebyl argument s předáním názvu souboru nalezen, je vyhozena výjimka s konkrétní chybou a program ukončen. Příklad spuštění:

```
java -jar fjp-sem-runnable.jar -i <vstupní_soubor> -o <výstupní_soubor>
```

Debug mód

V rámci vývoje překladače jsme používali různé druhy výpisů do konzole pro naši kontrolu. Časem jsme pak nějaké z nich odebrali, ale část z nich jsme schovali za volitelně nastavitelný debug mód. K jeho aktivaci stačí doplnit **-d** či **-D** jako argument při spuštění aplikace. Příklad spuštění:

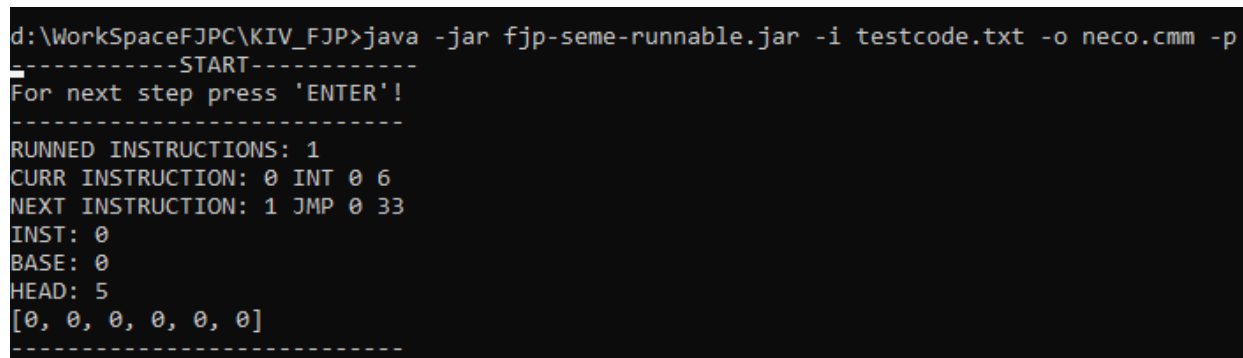
```
java -jar fjp-sem-runnable.jar -i <vstupní_soubor> -d
```

Debug mód je užitečný pro výpis adres proměnných, protože jejich pozice v paměti rozhoduje hashování v hashsetu

Interpreter

Posledním argumentem je pak samotné vyvolání simulace PL0 vygenerovaných instrukcí. Pro tento stav je pak možné volitelně doplnit argumenty **-p** či **-P**. Následně pak po dokončení generace instrukcí se hned spustí samotná simulace přímo do konzole příkazové řádky. PL0 interpreter pak provede první instrukci a čeká na uživatele, dokud nestiskne **ENTER**. Tím se provede další instrukce. Pokud by uživatel nechtěl vidět každé spuštěné instrukce zvlášť, je možné za pomoci sekundárního argumentu **-pa** či **-PA** docílit kompletního výpisu všech stavů simulace do konzole najednou. Příklad pro interpret řízený uživatelem:

```
java -jar fjp-sem-runnable.jar -i <vstupní_soubor> -p
```



```
d:\WorkspaceFJPC\KIV_FJP>java -jar fjp-seme-runnable.jar -i testcode.txt -o neco.cmm -p
-----START-----
For next step press 'ENTER'!
-----
RUNNED INSTRUCTIONS: 1
CURR INSTRUCTION: 0 INT 0 6
NEXT INSTRUCTION: 1 JMP 0 33
INST: 0
BASE: 0
HEAD: 5
[0, 0, 0, 0, 0, 0]
-----
```

Obrázek č.2: Spuštění interpreteru s uživatelským řízením

Simulace kompletně vypsaná do konzole bez interakce s uživatelem:

```
java -jar fjp-sem-runnable.jar -i <vstupní_soubor> -pa
```

```

d:\WorkSpaceFJPC\KIV_FJP>java -jar fjp-seme-runnable.jar -i testcode.txt -o neco.cmm -pa
-----START-----
For next step press 'ENTER'!
-----
RUNNED INSTRUCTIONS: 1
CURR INSTRUCTION: 0 INT 0 6
NEXT INSTRUCTION: 1 JMP 0 33
INST: 0
BASE: 0
HEAD: 5
[0, 0, 0, 0, 0, 0]
-----
RUNNED INSTRUCTIONS: 2
CURR INSTRUCTION: 1 JMP 0 33
NEXT INSTRUCTION: 33 LIT 0 -153924694
INST: 1
BASE: 0
HEAD: 5
[0, 0, 0, 0, 0, 0]
-----
RUNNED INSTRUCTIONS: 3
CURR INSTRUCTION: 33 LIT 0 -153924694
NEXT INSTRUCTION: 34 LIT 0 192803496
INST: 33
BASE: 0
HEAD: 6
[0, 0, 0, 0, 0, 0, -153924694]
-----
•
•
•
-----
RUNNED INSTRUCTIONS: 63
CURR INSTRUCTION: 49 INT 0 -2
NEXT INSTRUCTION: 50 INT 0 -3
INST: 49
BASE: 0
HEAD: 5
[0, 0, 0, 512, 512, 2048]
-----
RUNNED INSTRUCTIONS: 64
CURR INSTRUCTION: 50 INT 0 -3
NEXT INSTRUCTION: 51 RET 0 0
INST: 50
BASE: 0
HEAD: 2
[0, 0, 0]
-----
-----DONE-----

```

Obrázek č.3: Spuštění interpretru a kompletní výstup do konzole

Závěr

Semestrální práce z toho předmětu byla vysoce zajímavá a řešení úlohy bylo vskutku zajímavé, ale vybralo si to svou daň v podobě časové náročnosti a obtížnosti řešit některé problémy, které nebyly již tak triviální. Práce ve skupině je vždy přínosem, jelikož každý člen může přinést zajímavé názory a řešení problémů jiným způsobem a tím pádem poukázat na nedostatky různých návrhů. Spolupráce probíhala kolaborativně bez zádrhelů, snad jen pro příště si lépe rozvrhnout práci, kdo co bude dělat. Co se týče samotného držení se tématu a zadání, museli jsme nakonec zavrhnout pár funkcí, které jsme vzhledem ke stavu projektu, náročnosti a času nutného ke zpracování nestihli, či po úvaze omezili. Mezi konkrétní případy můžeme zařadit pure funkce nebo omezení datových typů. Pokud bychom měli zmínit nějaké funkce, které by se asi měli v budoucnu doplnit je určitě změna deklarace proměnných a jejich zjišťování stavu inicializace. Další rozšíření by mohly být implementace polí a možná alespoň float reprezentace datového typu.