ARCIBISKUPSKÉ GYMNÁZIUM

Korunní 586/2 | Vinohrady | 120 00 Praha 2

Maturitní práce z předmětu

Informatika a Výpočetní Technika

Tvorba programovacího jazyka Rogalo

Jakub Svatuška 8.A

Vedoucí práce: Ing. Pavel Kryl

Oponent: Ing. Vladimír Nulíček, CSc.

Praha 2025

PROHLÁŠENÍ

Prohlašuji, že předložená práce je mým původním autorským dílem, které jsem vypracoval/a samostatně. Veškerou literaturu a další zdroje, z nichž jsem při zpracování čerpal/a, v práci řádně cituji a jsou uvedeny v seznamu použité literatury.

V Praze dne…….

…….

(podpis)

nebo:

Pokud jste práci už někde uplatnili (např. jako SOČ, seminární práci apod.), je toto potřeba zde v prohlášení uvést.

ABSTRAKT

Program slouží k překladu jednoduchého jazyka Rogalo do LLVM IR. K tomuto cíli používá nástrojů FLEX a BISON – usnadňující lexikální a syntaktickou analýzu. Výsledný LLVM IR kód je pak možno pomocí nástroje CLANG zkompilovat a vytvořit spustitelný kód.

Klíčové slova: lexikální analýza, syntaktická analýza, LLVM IR, překlad.

ABSTRACT

The progam was created to translate a simple programming language into LLVM IR. To acomplish this I used toolchains such as FLEX and BISON – facilitating generation of lexical and syntactical parsers. The resulting LLVM IR code is than ready to be compile into byte-code using tool-chain such as CLANG.

Keywords: lexical analysis, syntactical analysis, LLVM IR, translation.

OBSAH

1 ÚVOD

1.1 Přehled

Maturitní práce si dává za cíl navrhnout vlastní syntaxi programovacího jazyka se základními funkcemi. Jazyk by měl být procedurální, schopný vytvoření proměnných s dvěma číselnými typy, na nichž by mělo být možno provádět základní operace. Dále má jazyk obsahovat statické pole, základní implementaci textových řetězců, funkce a řídící struktury *if* a *loop*. Pro tento jazyk bylo cílem vytvořit gramatický lexer za pomocí nástroje FLEX. Pro vzniklé tokeny sestavit pravidla gramatiky pomocí parsovacího nástroje BISON a v rámci těchto pravidel generovat LLVM kód, který bude posléze možné zkompilovat pomocí vysoce optimalizovaného a uznávaného kompilátoru CLANG do spustitelného souboru.

2 TEORIE

2.1 Lexikální analyzátory

Lexikální analyzátor bere jako vstup textový soubor, který přepracovává na sadu tokenů – základní stavební prvky jazyka. Z klíčových slov jako *if, func, loop, ==, (,), ...* vytvoří tokeny jim odpovídající – *IF, DEF, LOOP, EQUALS, LPAREN, RPAREN, ...* Názvům proměnných se také přiřadí konkrétní token – *IDENT*. V něm však není zřejmý název proměnné, a tak lexer tvoří dvojice token-hodnota. Dvojice se tvoří i pro klíčová slova, přestože to vlastně není potřeba. Pro výraz `int x = 3+2` bude výstup následující: (*INT : „int“), (IDENT : „x“), (ASSIGN : „=“), (NUMBER : „3“), (PLUS : „+“), (NUMBER : „2“).* Kromě vytváření těchto dvojic muže lexikální analyzátor navíc zaznamenávat např. číslo řádku, na kterém se token nachází.

2.1.1 Konečný automat

Lexikální analyzátory fungují na principu konečného automatu, což je vlastně zjednodušený počítač, který má různé stavy. Mezi těmito stavy se přesouvá na základě toho, jaké symboly přečte na vstupu. Pamatuje si pouze aktuální stav a nic jiného. Konečný automat začne v počátečním stavu a přečte první token, podle svých pravidel se přesune do jiného stavu na základě přečteného tokenu, vykoná přiřazenou akci a vrátí se do stavu počátečního. Toto se opakuje, dokud automatu nedojdou tokeny.

2.2 Gramatika

Gramatika je soubor pravidel, které určují způsob, jak skládat tokeny do platných struktur. Určuje správnou posloupnost tokenů, aby byl text platný v kontextu daného jazyka. Příklad jednoduché LR gramatiky zpracovávající aritmetické výrazy:

E → E + T | T

T → T * F | F

F → (E) | *číslo*

2.3 Syntaktický analyzátor

Parser, tedy syntaktický analyzátor, je dalším krokem v překladu jazyka. Za vstup bere tokeny připravené lexikálním analyzátorem a vytvoří z nich hierarchický syntaktický strom, jenž reprezentuje strukturu textu v rámci předem definovaných pravidel. Během vytváření tohoto stromu kontroluje, zda je v kontextu stanovených pravidel jazyka text smysluplný. Toto je možné

za využití pravidel bezkontextové gramatiky, takové která si nepamatuje už zanalyzované části. Ta nemusí pojímat všechna pravidla jazyka, ale pokud by byla použita gramatika kontextová, nemohl by být text efektivně syntakticky analyzován, proto se používají jiné nástroje pro vymáhání kontextových pravidel – např. tabulka symbolů. Ta se vytváří díky přidaným pravidlům v gramatice a s její pomocí se tak může kontrolovat sémantická správnost textu. Společně se sémantickou kontrolou zároveň probíhá generování nového kódu, respektive druhého jazyka. Pokud by se jednalo o interpret, byl by syntaktický strom rovnou vyhodnocován.

2.4 Typy syntaktických analyzátorů

Syntaktické analyzátory jsou většinou označovány skupinou písmen – LR, LL(1). První písmeno značí, že parser text zpracovává zleva doprava, druhé jestli používá levou, či pravou derivaci. Za těmito písmeny stojí často číslo v závorce uvádějící počet tokenů, na který se může parser podívat směrem dopředu LR(k). Pro takzvaný lookahead k se zpravidla používá $k = 1$. Gramatika s nulovým lookaheadem by nebyla schopna jakékoliv rekurze a musela by se na základně jednoho symbolu rozhodnout. S $k = 1$ by bylo nemožné posoudit složitější struktury. Pro nejpoužívanější typ gramatiky LR jsou postupy, jak ty s $k > 1$ převést na LR(1), jelikož se poté mnohem efektivněji syntakticky analyzují. Před touto definicí občas ještě stojí další specifikace parseru – LALR, SLR (zjednodušené formy kanonického LR analyzátoru), GLR (analyzátor nedeterministických gramatik). Z každé gramatiky se dá poznat, jaký analyzátor se na ní dá použít – na některé pouze LL, na jiné jen LR a na nějaké oboje.

2.4.1 Zespoda nahoru

Častějším i praktičtějším způsobem je analýza začínající od terminálů. Těmito definitivními kameny začne a skládá z nich postupně více a více abstrahovaná pravidla, jež dohromady tvoří různé podstromy, až dojde k cílovému neterminálu, který podstromy spojí a vznikne tak finální syntaktický strom.

2.4.1.1 LR

LR parser používá opačného principu a analyzuje odspoda nahoru s využitím pravé derivace. Tedy analogicky se jedná o expanzi symbolů zprava. Analýza výrazu 1+1*3 by tedy se zmíněnou LR gramatikou vypadala takto:

1+1*3	(vstup)
1+1*F	(reducke *číslo* → F)
1+T*F	(redukce *číslo* → F → T)
1+T	(redukce T*F → T)
E+T	(redukce *číslo* → F → T → E)
E	(redukce E+T → E)

2.4.2 Shora dolů

Tento typ začíná analýzu od největšího stavebního kamene a snaží se ho postupně expandovat a výsledek napasovat na vstupní text. Postupné „rozbíjení“ větších kamenů vede až k posledním terminálním tokenům, které je možné porovnat se vstupem.

2.4.2.1 LL

LL parsování odpovídá systému shora dolů, přičemž se využívá levé derivace. To znamená že analyzátor proměňuje symboly v pořadí zleva. Vzhledem k tomu, že výše zmíněná gramatika je pouze LR (má levou rekurzi), potřebujeme ji převést na LL:

E → T E'

E' → + T E' | ε (prázdný symbol)

T → F T'

T' → * F T' | ε

F → (E) | *číslo*

Vezmeme-li v potaz následující výraz: 1+1*3, provede LL parser následující kroky:

E	(vstup)
TE’	(expanze E → TE’)
FT’E’	(expanze T → FT’)
1T’E’	(expanze F → *číslo*)
1E’	(expanze T’ → ε)
1+TE’	(expanze E’ → +TE’)
1+FT’E’	(expanze T → FT’)
1+1T’E’	(expanze F → *číslo*)
1+1*FT’E’	(expanze T’ → *FT’)
1+1*3T’E’	(expanze F → *číslo*)
1+1*3E’	(expanze T’ → ε)
1+1*3	(expanze E’ → ε)

3 JAZYK ROGALO

Jazyk Rogalo je jednoduchým procedurálním jazykem s funkcemi. Syntaxe je proto intuitivní a podobná jiným, již existujícím jazykům. Nicméně jsem ji upravil tak, aby se mi v něm kód psal dobře a myslím, že by se s ním daly napsat i komplikovanější programy, jak je možné vidět na příkladě využití algoritmizace řazení seznamu bubble-sort.

3.1 Syntaxe

3.1.1 Komentáře

Komentáře pro lepší orientaci v kódu a popis se označují dvojitým ##, případně ##[[*[komentář]*]]##.

3.1.2 Proměnné

Proměnné se zakládají takto: *[typ] [název proměnné] = [výraz]*. Pod pojmem výraz se rozumí číselné operace, jiné proměnné a také volání funkcí. Případně ještě prvek seznamu. Pole se pouze inicializují, nebo se celé kopírují z jiného pole: arr *[typ prvků] [název pole] [specifikace pole]* nebo arr *[typ prvků] [název pole] [specifikace pole] = [jiné pole]*. Specifikace pole znamená kolik prvků bude pole mít, případně jestli to bude pole polí – arr int pole[2][3][4] vytvoří pole o dvou polích, každé se třemi poli o čtyřech prvcích.

3.1.3 Typy

Podporované typy proměnných jsou:

- *int*
- *byte*
- *arr* (speciální případ)
- *str*

Funkce nemůže vracet typ *str*, ale zato je možné uvést typ *void*, čímž se naznačí, že funkce nic vracet nebude.

3.1.4 Operátory

Možné operátory jsou:

a) číselné – plus +, binární minus -, krát *, děleno /

Číselné operátory je možné použít ve výrazech – na číslech a proměnných kromě polí a stringů. Unární mínus pro proměnné není implementované

b) komparativní – menší než <, větší než >, menší nebo rovno <=, větší nebo rovno >=, rovná se ==, nerovná se !=

Komparativní operátory je možné použít pouze v podmínce *if* na číslech a proměnných kromě polí a stringů (bohužel nerekurzivně – na jednu podmínku může být použit vždy maximálně jeden operátor).

3.1.5 Řídící struktury

Řídící struktury jazyk nabízí dvě:

a) If struktury vypadají následovně: if (*[podmínka]*){*[kód]*}. Podmínka znamená porovnání jednoho výrazu s druhým.

b) Loop: loop (*[proměnná přes kterou se iteruje]*; *[začáteční hodnota iterování]*→*[konečná hodnota do které se bude iterovat (včetně)]*; *[vynechatelný krok o který se proměnná bude zvyšovat, výchozí hodnota je 1]*){*[kód]*}.

3.1.6 Funkce

Funkce jsou definovány takto: func *[datový typ návratové hodnoty] [jméno funkce]*(*[argumenty]*) {*[kód]*}. Argumenty jsou definovány typem a názvem proměnné.

4 VYUŽITÉ NÁSTROJE

4.1 FLEX

FLEX[1] je open-source nástroj, který generuje lexikální analyzátor. Umožňuje zefektivnit a výrazně zkrátit vytváření analyzátoru. Struktura FLEX programu je následující: začíná definicemi, následuje uživatelský kód v c, nebo c++ (v mém případě c) uzavřený v závorkách s procenty `% { } %`. Pak jsou v souboru deklarace stavů jiných než INITIAL, dále dvojitá procenta `%%`, po kterých přijdou na řadu samotná pravidla pro lexování. Pokud bychom chtěli používat pouze lexikální analýzu bez syntaktické, dal by se ještě na konec souboru připojit driver kód. Nástroj se volá z příkazové řádky následovně: flex *[možnosti] [název souboru]*.l

4.2 BISON

BISON [2] je obdobně praktickým a výrazně práci ulehčujícím nástrojem. Generuje syntaktický generátor na základě gramatických pravidel. Struktura je velice podobná jako u FLEX souboru. Tedy začíná uživatelským kódem statických deklarací v závorkách s procenty `{% %}`, následují definice datových typů tokenů, pak definice tokenů samotných, neterminálů, předností jednotlivých symbolů i startovacího symbolu. Dále se v souboru vyskytují gramatická pravidla uzavřená v dvojitých procentech `%% [kód] %%`. Na konci souboru jsou definice na začátku deklarovaných funkcí společně s hlavní main funkcí, která celý analyzátor spouští. BISON se volá také z příkazové řádky: bison *[možnosti] [název souboru]*.y

4.3 LLVM

LLVM[3] se zaměřuje na meziplatformovou reprezentaci kódu – LLVM IR. Tento „srozumitelnější assembly kód“ postupně vylepšuje v několika po sobě jdoucích průchodech. Clang jakožto součást LLVM projektu obsahuje tento optimalizátor, ale i kompilátor ze samotného LLVM IR do strojového kódu. Clang za mě tedy spustí LLVM optimalizátor na LLVM IR vygenerované mým kompilátorem a zároveň z optimalizované verze IR vytvoří spustitelný soubor.

---

1 https://github.com/westes/flex/tree/master

2 https://www.gnu.org/software/bison/

3 https://llvm.org/docs/index.html

5 GRAMATIKA JAZYKA ROGALO

Budu ji vysvětlovat způsobem shora dolů, i když jsem si vědom, že BISON využívá LR princip, a tedy analýzu zespoda nahoru. Celý soubor je uzavřen v neterminálu *st*, jenž se skládá z neterminálů začátku souboru a programu. Program se rekurzivně skládá z neterminálů *statement*. Tedy může jich tam být 1 až *n* za sebou. *Statement* je expandován buďto jako deklarace, řídící struktura, změna proměnné, nebo komentář. Pod deklarace spadá vytvoření nové proměnné, funkce, textového řetězce i pole. Mezi řídící struktury patří *loop* a *if*. Dále ukážu, jak jsem konkrétní struktury přeložil z jazyka Rogalo do LLVM IR.

6 GENEROVÁNÍ KÓDU

6.1 Deklarace

6.1.1 Jednoduché proměnné

Proměnné v mém kódu jsou reprezentovány jako pointer. To je proto, aby se mohla jejich hodnota měnit a nemusela se pokaždé vytvářet nová proměnná, jelikož LLVM IR je SSA (Static Single Asignment). To znamená, že každé proměnné může být přiřazena hodnota právě jednou. Při deklaraci proměnné tedy vytvořím pointer s příslušným typem a k jeho názvu připojím nulu, aby bylo jasné, že to je pointer, a poté na místo, na které ukazuje, uložím hodnotu, jež je proměnné přiřazována.

6.1.2 Funkce

Definice funkcí by měly být na nejvyšší úrovní LLVM IR programu, tedy mimo funkci *main*. Proto mám otevřené dva soubory – *out.ll* a *temp_out.ll*. To mi umožňuje psát definice funkcí do *out.ll* a zbytek do *temp_out.ll* a posléze připojit *temp_out.ll* k *out.ll*, a tak dosáhnout definice funkcí na začátku souboru. Reprezentace funkce v Rogalu je takováto: func *[datový typ návratové hodnoty] [jméno funkce]*(*[argumenty]*){*[kód]*}. Datový typ je stejný jako u proměnné, s tím rozdílem, že funkce může vracet i pole, které je popsané níže a také je schopná nevracet nic – datový typ *void*. Argumenty funkce jsou předávány normální SSA proměnnou, a tak je potřeba je na začátku funkce převést na pointer, abych s nimi mohl dál pracovat. Tělo funkce vypadá stejně jako normální kód, ale kromě tokenu *statement* akceptuje ještě *return*.

6.1.3 Stringy

Textové řetězce mám definované jako globální konstanty na začátku souboru. Je tedy obdobně jako u definicí funkcí potřeba ošetřit, aby byly před funkcí *main*. Zde ovšem nastává konflikt, pokud jsem v definici funkce a vytvářím string, mohlo by se stát, že bych konstantu vytvořil uprostřed funkce, a tím pádem ji vytvářel při každém volání funkce. Proto jsem vytvořil dočasný string, do kterého se deklarace stringů ukládají v případě, že je kód v deklaraci funkce. Když pak program dojde na konec funkce, daný string vytiskne za funkci, čímž se zajistí, že se stringy nebudou tvořit vícekrát. Samotný datový typ je statické pole znaků. V kódu tedy deklarace stringů vypadá takto: @*[název stringu]* = private constant [*[délka stringu]* x i8] c“*[string samotný]*“.

6.1.4 Pole

U deklarace polí jsem se je rozhodl pouze inicializovat. Tedy nepřiřazuji každému prvku nějakou hodnotu, to je povoleno jenom jednotlivě, pro každý prvek pole zvlášť. Jako u proměnné se vytvoří pointer s daným typem a názvem s připojenou nulou na konci. Proto je deklarace, která kopíruje celé jiné pole povolená, stačí pouze načíst hodnotu pointeru a uložit jí do jiného. Ještě musí proběhnout kontrola, zda se jeho typ shoduje s typem zadaným. A právě typ pole je v tomto případě trochu komplikovanější. Specifikace pole vypadá takto: *[výraz][výraz][výraz]...* Výraz nejvíce napravo je nejvnitřnější, tedy deklarace `arr int pole[2][4][3]` vytvoří pole s dvěma prvky – poli o čtyřech prvcích, které jsou také pole a mají v sobě tři prvky typu int. Tuto specifikaci ukládám v zásobníku, z kterého pak tvořím typ v LLVM IR vypadající takto: `[2 x [4 x [3 x i32]]]`. Přistupuji k tomu takovým způsobem, že vytvořím první závorku se základním typem seznamu `[3 x i32]` a potom k ní zvenku přidávám vrchní položky ze zásobníku. Používám zásobník, jelikož k hodnotám přistupuji „odzadu“.

6.2 Řídící struktury

6.2.1 Loop

Řídící struktura loop je jedna z nejkomplikovanějších částí projektu, znát je to na množství kódu, který při jejím tvoření generuji. Začínám přípravným blokem kódu. Ten vytvoří proměnou, přes kterou bude *loop* iterovat, a uloží do ní startovní hodnotu. Zároveň vytvoří proměnnou s maximální hodnotou, k níž se bude iterovat. Také se vytvoří proměnná, která ukazuje, zda je startovní hodnota vyšší, nebo nižší než cílová. Touto proměnnou se *loop* dělí na dvě části. V první vždy kontroluje, zda je proměnná menší než cílová hodnota a krok přičítá, zatímco v druhé větvi kontroluje, zda je proměnná větší a krok odečítá. Kontrola probíhá před spuštěním kódu uvnitř smyčky, přidávání či odčítání po něm. Zároveň s tím se skočí zpět na kontrolu, a pokud už byly podmínky splněny ze, smyčky se vystoupí. Každý *loop* má své proměnné a bloky kódu, kam skáče, pokud tedy nastane situace, že se volá *loop* ve struktuře *loop,* musí si původní *loop* nějakým způsobem zapamatovat, jak pojmenovávat své proměnné, aby nedošlo ke kolizi. Toho lze docílit pomocí zásobníku – když *loop* začíná, přidá se do zásobníku číslo aktuální struktury *loop*. Pokud parser narazí na další strukturu *loop*, opět přidá její číslo do zásobníku a vygeneruje potřebný kód. Na konci struktury číslo ze zásobníku odstraní, načež může generovat zbytek kódu z původní struktury *loop*, jelikož její číslo je stále uložené v zásobníku.

6.2.2 If

U řídící struktury *if* se jednoduše vytvoří proměnná, která rozhoduje, zda se má přesunout do bloku s kódem *if* struktury, nebo se má pokračovat dál. Pro tento účel se v LLVM IR využívá příkaz *icmp:* %condition = icmp *[operace] [typ] [první proměnná]*, *[druhá proměnná]*.Ve výsledku to pak může vypadat zhruba takto: %condition_if1 = icmp sgt i32 %t10, 12. Ptáme se, jestli je proměnná *t10* (může být i negativní, proto *sgt*) větší než 12, pokud ano bude hodnota *%condition_if1* nastavena na 1, jinak na 0. Proměnná se pak využije pro určení, zda pokračovat dál, nebo se vnořit do *if*: br i1 *[proměnná]*, label *[název štítku, nakterý se má skočit, když je proměnná 1]*, label *[když je proměnná 0]*.

6.3 Změna proměnné

6.3.1 Jednoduché proměnné

Změna jednoduché proměnné spočívá v tom, že si vyhledám její typ v tabulce, do které si je ukládám a potom uložím přiřazovanou hodnotu do pointru proměnné: store *[typ] [přiřazovaná hodnota]*, ptr *[název proměnné]*0.

6.3.2 Pole

Statické pole je reprezentováno pouze místem v paměti na určitý počet prvků daného typu. Pokud chci změnit prvek *[10][2]* nějakého pole, na které mám uložený pointer na jeho začátek, nejde o nic jiného, než vypočítat, kde se tento prvek nachází. K tomu slouží instrukce *getelementptr*: *[proměnná, do které chci výsledek uložit]* = getelementptr *[typ prvků]*, ptr *[název pointru na pole]*, i32 0, i32 *[první index]*, i32 *[druhý index]*, … Na začátku výčtu indexů je 0, protože pole je v LLVM IR implementované jako pointer, který musí být také indexován. Poté stačí uložit přiřazovanou hodnotu této vypočítané adrese: store *[typ] [ukládaná proměná]*, ptr %idx.

7 DATOVÉ STRUKTURY

7.1 Tabulka symbolů

Tabulka slouží k orientaci v programu – ukládá mimo jiné klíčová slova (*if, loop, call*), podle kterých se dá jednoduše poznat způsob, jakým parser čte soubor. Další využití tkví v ukládání typů proměnných i polí a návratových typů funkcí. Také je u každého záznamu poznamenáno, na jakém řádku je definován, což také usnadňuje orientaci. Tabulka je staticky alokována, předpokládám, že nebude mít více než 1024 záznamů.

7.2 Tabulka funkcí

Tabulku funkcí používám výhradně k uložení typů argumentů, ale v rámci toho, že ji na konci tisknu do standardního výstupu, rozhodl jsem se zdvojit informaci o návratovém typu a řádku definice. Takhle jsou všechny funkce přehledně na jednom místě. Stejně jako tabulka symbolů je i tabulka funkcí alokována staticky, avšak pouze pro 64 záznamů, jelikož funkcí je výrazně míň než všech symbolů.

7.3 Spojový list

Pro uložení argumentů funkce používám spojový list. Jeden prvek listu je reprezentován stringem s názvem proměnné argumentu, stringem s jeho typem a ukazatelem na další prvek.

7.4 Zásobník

Pro uložení specifikace pole a čísla řídících struktur *loop* a *if* používám zásobník. Vlastní implementace je stejná jako spojový list, s tím rozdílem, že vždy operuji pouze s posledním prvkem.

8 ZPRACOVÁNÍ CHYB

Syntaktický analyzátor sám kontroluje, zda je kód validní v rámci gramatiky, to ovšem ještě neznamená, že je kód bezchybný – parser kontroluje text bezkontextově, je ho ještě potřeba zkontrolovat sémanticky. Proto například probíhá kontrola s tabulkou symbolů, zda volaná funkce nebo proměnná zmiňovaná ve výrazu již existuje. Co ale kontrolou projde a bylo by potřeba opravit jsou aritmetické operace na proměnné s typem *str*.

9 SPUŠTĚNÍ

Pro spuštění kódu psaného v jazyku Rogalo je nutné mít nainstalované tool-chainy *bison*, *flex* a *clang*. V prvním kroku vytvoříme syntaktický analyzátor. Ten se vytváří první, jelikož *flex* využívá soubor *y.tab.h* ve kterém jsou definované tokeny a generuje ho právě volání *bisonu* se svolenou možností *-d*. Příkaz může vypadat následovně: `bison -d parser.y` V souboru *parser.y* musí být ve funkci main uveden váš soubor jako input. Poté vytvoříme lexikální analyzátor pomocí nástroje *flex*: `flex grammar.l` Následně zkompilujeme oba dva vygenerované analyzátory do jednoho souboru pomocí nástroje *clang*: `clang lex.yy.c parser.tab.c` a spustíme vytvořený kód.

Nyní se vygeneruje soubor v LLVM IR, nazvaný v mé verzi *out.ll*. Tento soubor tedy zoptimalizujeme a zkompilujeme do strojového kódu pomocí *clang* následovně: `clang out.ll.`Tím vznikne finální soubor, který už jednoduše spustíme.

Celý tento proces je uložený ve skriptu *run.bat.* Ten nejdřív zavolá jiný skript *compile.bat*, který vytvoří LLVM IR soubor *out.ll* Původní skript *run.bat* posléze tento soubor zkompiluje a spustí.

10 ZÁVĚR

V práci jsem popsal základní fungování principů spojených s překladem programovacího jazyka. Konkrétněji jsem rozvedl, jak se generuje kód při zpracování různých akcí v jazyce Rogalo a popsal datové struktury, které jsem k tomu použil. Nakonec jsem uvedl, jak si může čtenář svůj vlastní úryvek kódu Rogalo pomocí připojených souborů *grammar.l* a *parser.y* sám přeložit a následně i zkompilovat. Projekt splnil všechny interní cíle, dokonce se mi povedlo implementovat funkci *print,* tedy interakci s *stdout.*

11 ZDROJE

(1) *Get the number of digits in an int*. Online. In: Stackoverflow. 2012. Dostupné z: https://stackoverflow.com/a/11151570. [cit. 2025-03-16].

(2) *Inserting char string into another char string*. Online. In: Stackoverflow. 2010. Dostupné z: https://stackoverflow.com/a/2016015. [cit. 2025-03-16].

(3) *A Gentle Introduction to LLVM IR*. Online. In: Mcyoung.xyz. 2023. Dostupné z: https://mcyoung.xyz/2023/08/01/llvm-ir/. [cit. 2025-03-16].

(4) *A Quick Introduction to Handling Conflicts in Yacc Parsers*. Online, Obecně-vzdělávací. Tucson: The University of Arizona, 2008. Dostupné z: https://www2.cs.arizona.edu/~debray/Teaching/CSc453/DOCS/conflicts.pdf. [cit. 2025-03-16].

(5) *Example program for the lex and yacc programs*. Online. In: IBM. 2023. Dostupné z: https://www.ibm.com/docs/en/aix/7.1?topic=information-example-program-lex-yacc-programs. [cit. 2025-03-16].

(6) *Bison 3.8.1*. Online. In: Gnu. 2021. Dostupné z: https://www.gnu.org/software/bison/manual/bison.html#Examples. [cit. 2025-03-16].

(7) *Online Documentation for Flex and Bison*. Online. In: Kiv.zcu. 2002. Dostupné z: https://www.kiv.zcu.cz/~rohlik/vyuka/fjp/Cv03_2002/manualy/Using%20Lex%20and%20Yacc.htm. [cit. 2025-03-16].

(8) *How to Build a C Compiler Using Lex and Yacc*. Online. In: Medium. 2021. Dostupné z: https://medium.com/codex/building-a-c-compiler-using-lex-and-yacc-446262056aaa. [cit. 2025-03-16].