



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 skompilovat a vytvořit spustitelný kód.

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

ABSTRACT

The program was created to translate a simple programming language into LLVM IR. To accomplish this I used toolchains such as FLEX and BISON – facilitating generation of lexical and syntactical parsers. The resulting LLVM IR code is then ready to be compiled into bytecode using toolchain such as CLANG.

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

OBSAH

1 Úvod.....	5
1.1 Přehled.....	5
1.2 Lexikální analyzátory.....	6
1.3 Syntaktický analyzátor.....	6
1.3.1 Konečný automat.....	6
1.4 Typy syntaktických analyzátorů.....	7
1.4.1 Shora dolů.....	7
1.4.2 Zespoda nahoru.....	7
1.5 LL.....	7
1.6 LR.....	8
2 Jazyk Rogalo.....	9
2.1 Syntaxe.....	9
2.1.1 Komentáře.....	9
2.1.2 Proměnné.....	9
2.1.3 Operátory.....	9
2.1.4 Řídící struktury.....	9
2.1.5 Funkce.....	10
3 Využité nástroje.....	11
3.1 FLEX.....	11
3.2 BISON.....	11
3.3 LLVM.....	11
4 Gramatika.....	12
5 Generování kódu.....	13
5.1 Deklarace.....	13

5.1.1 Jednoduché proměnné.....	13
5.1.2 Funkce.....	13
5.1.3 Stringů.....	13
5.1.4 Pole.....	14
5.2 Řídící struktury.....	14
5.2.1 Loop.....	14
5.2.2 If.....	15
5.3 Změna proměnné.....	15
5.3.1 Jednoduché proměnné.....	15
5.3.2 Pole.....	15
6 Datové struktury.....	16
6.1 Tabulka symbolů.....	16
6.2 Tabulka funkcí.....	16
6.3 Spojový list.....	16
6.4 Zásobník.....	16
7 Spuštění.....	17
8 Závěr.....	18
9 Zdroje.....	19

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 nich by mělo být možno provádět základní operace. Dále má jazyk obsahovat statické pole, funkce a řídicí struktury if a loop. Pro tento jazyk bylo cílem vytvořit gramatický lexer za pomoci nástroje FLEX. Pro vytvořené 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 nějaký textový soubor. V textovém souboru je nějakým jazyk – lexovat se můžou i mluvené jazyky. Analyzátor text zpracuje a vrátí sadu po sobě jdoucích předem definovaných tokenů. Tokeny jsou elementárními nositeli významu v rámci daného jazyka. Jde tedy prakticky o reprezentaci (regex) pravidly daného kusu textu – může to tedy být například slovo, odstavec, mezera, závorka nebo v kontextu programovacích jazyků uplatnitelné dvojité rovná se, různá klíčová slova – if, loop, def, ..., název stringu atd. K tomuto kusu textu lexer přiřadí příslušný token, tedy pokud narazí na „if“ vrátí dvojici („if“ : IF), když dojde k „==“, vrátí („==“ : EQUALS). Díky tomu, že vytváří tyto dvojice, je možné zpětně tokeny využívat. Některé lexery při najití určitého tokenu ještě umožňují vykonat nějaké operace. Je tak možné vypisovat třeba na kterém řádku se token objeví či provádět různé operace s hodnotou tokenu.

2.2 Syntaktický analyzátor

Parser, tedy syntaktický analyzátor je dalším krokem v překladu jazyka. Za vstup bere tokeny vytvořené lexikálním analyzátozem 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 všechny 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.

2.2.1 Konečný automat

Syntaktické analyzátory fungují na principu konečného automatu. To je takový 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 a přečte další token. Jestliže je na konci ve stavu uznávaném jako konečný, byl vstup syntakticky správně strukturovaný, v opačném případě nikoliv.

2.3 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 základě 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 ty se mnohem efektivněji syntakticky analyzují. Před touto definicí občas ještě stojí další specifikace parseru – LALR, SLR (zjednodušené formy kanonického LR analyzátoru), GLR (analyzátor nedeterministických gramatik).

2.3.1 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 vztupem.

2.3.2 Zespoda nahoru

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

2.4 LL

LL parsování odpovídá systému shora dolů přičemž se využívá levé derivace. To znamená že analyzátor proměňuje symboly v pořadí zleva. Máme-li výraz $1 + 1 = 2$, provede následující kroky:

S

$\rightarrow S + S$

$\rightarrow S + S = S$

$\rightarrow 1 + S = S$

$\rightarrow 1 + 1 = S$

$\rightarrow 1 + 1 = 2$

2.5 LR

LR parser používá opačného principu a analyzuje odspoda nahoru s využitím pravé derivace. Tedy analogicky se jedná o expanzi symbolů zprava. Výše uvedený výraz by tedy s LR gramatikou vypadal takto:

S

$\rightarrow S = S$

$\rightarrow S + S = S$

$\rightarrow S + S = 2$

$\rightarrow S + 1 = 2$

$\rightarrow 1 + 1 = 2$

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 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 Operátory

Možné operátory jsou:

- a) číselné – plus `+`, minus `-`, krát `*`, děleno `/`
- b) komparativní – menší než `<`, větší než `>`, menší nebo rovno `<=`, větší nebo rovno `>=`, rovná se `==`, nerovná se `!=`

3.1.4 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]}`.

3.1.5 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

FLEX¹ 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 dvojité 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

BISON² 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 procetech `%% [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

LLVM³ je optimalizuje kód, soustředí se na meziplatformovou reprezentaci kódu – LLVM IR. Tento „srozumitelnější assembly kód“ postupně zlepšuje v několika po sobě jdoucích průchodech. Normálně dostane optimalizátor původní IR od nějakého kompilátoru, v mém případě mu ho generuji sám. 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 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

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 – n za sebou. *Statement* může být buďto deklarace, řídicí struktura, změna proměnné nebo komentář. Pod deklarace spadá vytvoření nové proměnné, funkce, textového řetězce i pole. Mezi řídicí struktury patří *loop* a *if*. Dále ukáži, 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 Assignment). 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, která je proměnné přiřazována.

6.1.2 Funkce

Definice funkcí by měly být na nejvyšší úrovni 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 vrátit i pole, které je popsáno níže a také je schopná nevrátit 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 Stringů

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 nepřisuzuji 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 je 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 na pravo 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, který vypadá 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 nejkomplicovanějších částí projektu, znát je to na množství kódu, který při jejím tvoření generuji. Začínám přípravným blokem kódu. Ten vytvoří proměnnou, přes kterou bude *loop* iterovat a uloží do ní startovní hodnotu. Zároveň vytvoří proměnnou s maximální hodnotou, ke které 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. Kontrolování probíhá před spuštěním kódu uvnitř *loopu*, Přidávání či odčítání po něm. Zároveň s tím se skočí na zpět na kontrolu a pokud už byly podmínky splněny z *loopu* se vyskočí. 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 docílím pomocí zásobníku – když *loop* začíná přidám do zásobníku číslo momentální struktury *loop*, když narazím na další *loop*, přidám opět jeho číslo do zásobníku, vygeneruji ho, a pak číslo ze zásobníku odstraním, pak může být vygenerován zbytek kódu z původní struktury *loop*, jelikož její číslo je stále uloženo v zásobníku.

6.2.2 If

U řídicí struktury *if* se jednoduše vytvoří proměnná, která rozhoduje, zda se má skočit do bloku s kódem v *if* struktuře, 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, na který 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 pointeru proměnné: `store [typ] [přiřazovaná hodnota], ptr [název proměnné]`.

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 pointeru na pole], i32 0, i32 [první index], i32 [druhý index], ...`. Na začátku výčtu indexů je 0, to je proto, že pole samotné, tak jak je v LLVM IR implementované, vzhledem k tomu, že to je pointer, musí být také indexováno. Poté stačí uložit přiřazovanou hodnotu této vypočítané adresy: `store [typ] [ukládána proměnná], ptr %idx`.

7 DATOVÉ STRUKTURY

7.1 Tabulka symbolů

Tabulka slouží jak k orientaci v programu – ukládá mimo jiné klíčová slova, ze kterých se dá jednoduše poznat, jak soubor parser čte, tak i k ukládání typů proměnných i polí a návratových typů funkcí. Také je u každého záznamu poznamenáno, na kterém 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 printuji, tak jsem se rozhodl zdvojit informaci o návratovém typu a řádku definice. Takhle jsou všechny funkce přehledně na jednom místě. Stejně jako tabulka symbolů je tabulka funkcí alokována staticky a to na 50 záznamů.

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, akorát vždy operuji pouze s posledním prvkem.

8 SPUŠTĚNÍ

Pro spuštění kódu psaného v jazyku Rogalo musíte mít nainstalované toolchainy *bison*, *flex* a *clang*. První krok je vytvoření syntaktického analyzátoru. 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 skompilujeme do strojového kódu pomocí *clang* následovně: `clang out.ll` To ovšem nestačí, jelikož Rogalo používá funkci *mingw_printf*, která je součástí standartní c knihovny *stdio.h*. Proto je potřeba soubor ještě propojit s touto knihovnou. Výsledný příkaz může vypadat takhle: `clang out.ll -o final -LC:\dev\clang\llvm-mingw-20250114-msvcrt-x86_64\include\stdio.h`, obecněji pak takto: `clang out.ll -o [název souboru pro uložení] -L[cesta ke knihovně stdio.h]`. Tím vznikne finální soubor, který už jednoduše spustíme.

9 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. Na konec jsem uvedl, jak si může čtenář svůj vlastní úryvek kódu Rogalo pomocí připojených souborů *grammar.l* a *parser.y* sám přeložit a následně i skompilovat. Projekt splnil všechny interní cíle, dokonce se mi povedlo implementovat funkci *print*, tedy interace s *stdout*.

10 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].