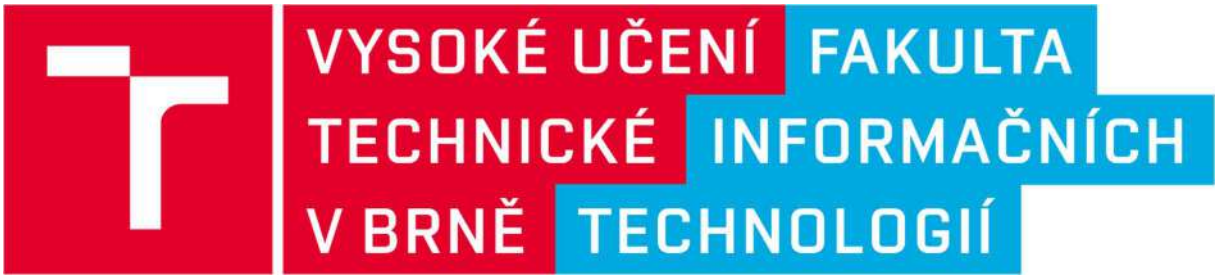


Projektová dokumentace

Tým číslo 099, varianta I

Matěj Žalmánek	**xzalma00**	**25%**
Igar Sauchanka	xsauch00	25%
Juraj Novosád	xnovos13	25%
Eva Mičánková	xmican10	25%

Implementovaná rozšíření: BOOLTHEN

OBSAH

1. Úvod

Cílem projektu bylo vytvořit překladač, který přeloží kód ze zdrojového jazyka IFJ21, jenž je zjednodušenou podmnožinou jazyka Teal, do cílového jazyka IFJcode21 (mezikód).

Překladač je implementován jako konzolová aplikace (bez grafického uživatelského rozhraní), tj. načítá zdrojový kód ze standartního vstupu a generuje mezikód na standartní výstup, přičemž všechna chybová hlášení generuje na standartní chybový výstup.

2. Návrh řešení a implementace

Celé řešení se sestává z námi implementovaných dílčích částí, které jsou popsány níže. Dále je popsáno, jak spolu tyto dílčí části komunikují a spolupracují. Celá implementace je provedena v jazyce C.

2.1 Lexikální analýza

První částí překladače je scanner, nebo také lexikální analyzátor, který je implementován za pomocí konečného automatu, který odpovídá námi předem definovaným regulárním výrazům, které popisují všechny konstrukce překládaného jazyka. Diagram konečného automatu, který reprezentuje námi implementovaný scanner je přiložený v bodě 6.

Pro komunikaci se scannerem slouží dvě funkce, *GetNextToken* a *TokenStore*. Jak již z nabízeného rozhraní napovídá, scanner ukládá tokeny do fronty tokenů, které si poté vnější části mohou dle libosti ukládat / vybírat. Jelikož pro ulehčení gramatiky jsme se rozhodli používat lexému ID_F, scanner se také snaží zjišťovat, zda se jedná o identifikátor funkce, či nikoliv a z toho vychází povinnost ukládat některé tokeny čtené scannerem navíc, ale s nižší prioritou, než kdyby je ukládala jiná část programu než scanner. Další důležitou částí námi implementovaného scanneru je pole řetězců, do kterého ukládá atributy tokenů za pomocí datové struktury *StringsArray* popsané níže.

2.2 Syntaktická analýza

Další důležitou částí překladače je syntaktická analýza, která je až na zpracování výrazů – precedenční syntaktickou analýzu zpracována metodou rekurzivního sestupu. Syntaktická analýza je implementována v souboru **parser.c** a řídí se pravidly LL-gramatiky, viz kapitola *7. LL-Gramatika*.

Pravidla jsou rozdělena do funkcí, které dostávají přes parametry ukazatel na struktury *Token* a *ScannerContext*. Struktura *Token* obsahuje informace o tokenu, zatímco struktura *ScannerContext* slouží k volání a obsluze lexikální analýzy.

Všechny funkce realizující gramatická pravidla mají návratový typ Boolean, díky kterému, pokud nastane chyba, dochází k okamžitému přerušení parsingu a vynoření ze syntaktické analýzy.

Komunikace mezi syntaktickou a lexikální analýzou se odehrává pomocí volání funkce *GetNextToken*, kdy lexikální analyzátor načte ze zdrojového kódu další token, přičemž provádí lexikální analýzu.

Další komponentou, se kterou syntaktická analýza komunikuje, je precedenční syntaktická analýza. Ke komunkaci dochází prostředictvím ukazatele na strukturu *ScannerContext*.

Precedenční syntaktická analýza je volána tehdy, pokud syntaktická analýza narazí na výraz, přiřazení do proměnné, nebo na příkaz `return`.

Poslední komponentou, se kterou je zajištěna komunikace, je sémantická analýza. Komunikace je uskutečněna voláním funkcí, které provádí příslušné sémantické akce.

2.2.1 PRECEDENČNÁ SYNTAKTICKÁ ANALÝZA

Precedenčná syntaktická analýza vyhodnocuje správnosť zápisu výrazov a podmienok. Je založená na princípe zásobníkového automatu, nasledujúci stav je určený precedenčnou tabuľkou. V tabuľke sa indexuje pomocou prichádzajúceho tokenu a terminálu na vrchole zásobníka. Automat pozná stavy, ktoré sú kódované nasledovne: ‘<’, ‘>’, ‘#’, ‘&‘, ‘=‘ .

Popis stavov:

- ‘<‘ - Operátor na vrchole zásobníka má menšiu prioritu ako prichádzajúci. Ulož ho na zásobník, tak aby mal handle s najbližším neterminálom, a získaj ďalší token.
- ‘>’ – Operátor na vrchole zásobníka má väčšiu prioritu. Spracuj ho až po handle na zásobníku a zavolaj precedečnú sémantickú akciu s predchádzajúcim tokenom.
- ‘#’ – Operátory na vrchole zásobníka a prichádzajúci nemôžu nijakým spôsobom interagovať. Nastala syntaktická chyba, opusti spracovanie s chybovým kódom.
- ‘=‘ – Operátory majú rovnakú prioritu, Ulož na zásobník aktuálny operátor a načítaj ďalší token.
- ‘&‘ – Očakávaný koniec výrazu. Ukončí sémantické spracovanie výrazu. Ak sa do tohto stavu dostane s neprázdnym zásobníkom, ukonči spracovanie s chybou

Tabuľka je implementovaná v **precedence_analyzer.h**, automat je v **precedence_analyzer.c.**

2.2.2 PRECEDENČNÁ SÉMANTICKÁ ANALÝZA

Precedenčná sémantická akcia je implementovaná ako automat. Pričom stavy rozlišujú, či bola zavolaná s identifikátorom, konštantou, aritmetickým operátorom alebo relačným operátorom.

Výrazy sa v precedenčnej analýze prevedú na postfixovú notáciu a následne sa pri sémantických akciách tvorí abstraktný syntaktický strom. ASS je využívaný na zisťovanie návratových hodnôt jednotlivých operácii aj na tvorenie výsledného kódu. Knižnica na podporu práce s výrazovým stromom je implementovaná v **expression_tree.***. Knižnica na tvorbu kódu zo stromu je v **cg_expression_tree.c**, je to nadstavba nad generátorom kódu. Pričom expression_tree je nadstavba nad symtable stromom.

Popis stavov precedenčnej sémantickej analýzy:

- Identifikátor – skontroluje či je daná premenná definovaná, vytvorí z nej nový strom a uloží ho na zásobník stromov
- Konštanta – vytvorí nový strom a uloží na zásobník stromov.
- Aritmetický operátor – Načíta zo zásobníka stromov 2 najvrchnejšie. Z tabuľky návratových hodnôt podľa operácie a dátových typov koreňov vyhodnotí, či je možná daná operácia alebo je nutná implicitná konverzia niektorého z operandov, prípadne ju vykoná. Následne vytvorí nový koreň pod ktorý zlúči pravý a ľavý podstrom. Ten sa uloží ako nový vrchol zásobníka a vykoná sa generácia výsledného kódu.
- Relačný operátor – Spracovanie je podobne ako pri aritmetickom, ale postup

Typová kontrola je na základe tabuľky návratových hodnôt. Tabuľka návratových hodnôt je implementovaná ako trojrozmerné pole, pričom prvý index je typ operácie, druhý typ ľavého operandu, tretí typ pravého operandu. Určuje aký dátový typ bude mať výsledok operácie,

prípadne či daná operácia môže prebehnúť nad danými typmi a či nedochádza k neočakávanému narábaniu s hodnotou nil.

Tabuľka je implementovaná v **semantic_bottom_up.h**, automat je v **semantic_bottom_up.c**.

2.3 SÉMANTICKÁ ANALÝZA

Sémantická analýza (dál SA) je implementovaná prostřednictvím sémantických akcí v souboru **semantic_action.h** a **semantic_action.c**. V souborech **semantic_global.h** a **semantic_global.c** je definována struktura *SemanticGlobals,* do které SA ukládá informaci o současném stavu překladu.
Gramatickým pravidlům jsou přiřazena odpovídající sémantické akce, které pak jsou volané syntaktickou analýzou. Sémantické akce jsou přímo spojené s generátorem kódu a volají jeho funkce ve svém průběhu. Pomocí pomocné struktury *SemanticGlobals* SA komunikuje s generátorem kódu, jestli se příkazy, které jsou teď zpracovávané nacházejí uvnitř konstrukce while.
Zodpovědnost SA je všechny příkazy a konstrukce mimo aritmetických, řetězcových a relačních výrazů (za výrazy je zodpovědná precedenční sémantická analýza (dál PSA)). SA komunikuje s PSA u zpracování příkazu přiřazení a příkazu return, prostřednictvím fronty přiřazení. SA ukládá do fronty proměnné, se kterými pak pracuje PSA. Po provedení přiřazení SA zkontroluje stav fronty a buď doplňuje chybějící hodnoty (u příkazu return), nebo nahlásí chybu (u přiřazení).

2.4 GENERÁTOR KÓDU

Generátor kódu má tri základné druhy funkcií.
Prvý typ funkcií je na formátovanie premenných, konštánt a labelov. V našej implementácii sa pri podmienke netvorí nový FRAME v IFJcode, a preto je nutné zaistiť nepremenovanie premennej alternáciou jej názvu, pridaním predpon a prípon oproti jej menu v IFJ21. Konštanty sa formátujú ako je zadané v zadaní.
Zformátované premenné a labely sa zadajú do funkcií, ktoré už priamo vygenerujú reťazec príkazu v IFJcode21. Pre každý príkaz, v IFJcode, je separátna funkcia.
Vygenerovaný reťazec príkazu dostane ako parameter funkcia cg_envelope. Podľa toho či je práve spracovávaná konštrukcia while, príkaz vypíše na konzolu alebo ho odloží na frontu. Príkazy deklarácie premenných sa vždy vypíšu, takto je všetko čo je deklarované vo while deklarované pred ním. Po ukončení najvrchnejšieho whilu sa všetky pozdržané príkazy naraz vypíšu.

2.5 IMPLEMENTÁCIA TABUĽKY SYMBOLOV

Tabuľka symbolov ja implementovaná ako binárny vyhľadávací strom. Základné funkcie na prácu so stromom a dátové štruktúry sú implementované v **symtable** knižnici. Funkcie na pracovanie s premennými sú implementované v **ts_handler** knižnici. Funkcie na prácovanie s funkciami sú v **fun_data** a **fun_table** knižniciach.
Pridávanie a vyhľadávanie v tabuľke symbolov je implementované pomocou hashovania. Z mena každej premennej alebo funkcie sa vypočíta hash, pomocou ktorého sa potom orientuje v príslušnom strome ako v klasickom binárnom. V prípade ak by malo viac mien rovnaký hash, sa daný nový uzol uloží ako najľavejší od uzlu s rovnakým hashom.
Premenné sú uložené v stromoch uložených v zreťazenom zozname. Nový strom sa vytvára vždy pri riadiacej štruktúre(if alebo while), takto je zabezpečená možnosť znova deklarovať rovnomenné premenné v jednej funkcii. Pri deklarovaní sa stačí pozerať na najvrchnejší

strom, či už nie je deklarovaná. Pri hľadaní sa prehľadávajú všetky stromy. Aby sa zabezpečila unikátnosť mena premenných, každý prvok zreťazeného zoznamu stromov má svoj identifikátor. Identifikátor je pridelený pri deklarácii prvku zoznamu. Identifikátor je v podstate počítadlo riadiacich konštrukcií vo funkcii. Identifikátor je dôležitý pri generovaní kódu. Vynuluje sa pri ukončení funkcie. Pri ukončení riadiacich konštrukcií sa dealokuje najvrchnejší prvok zo zoznamu tabuľky symbolov aj s jeho stromom premenných. Tabuľka symbolov pre funkcie pozostáva z troch stromov, pre deklarované definované a vstavané funkcie. Každý uzol stromu obsahuje názov funkcie, jej parametre a návratové hodnoty.

3. SPECIÁLNÍ POUŽITÉ TECHNIKY A ALGORITMY

V řešení projektu jsme použili následující speciální techniky a algoritmy.

3.1 POLE ŘETĚZCŮ

Tato datová struktura slouží pro ukládání atributů tokenů a byla navržena výhradně pro tyto potřeby. Funguje za pomocí statického pole, které se vkládá jako prvek do fronty těchto polí. Výsledkem je tedy fronta polí, kde jedno pole má předem definovanou velikost, v našem případě 1024B. Jelikož se snaží tato datová struktura vyhovět požadavkům pro vkládání řetězců ze scanneru a šetřit paměť, nabízí invalidovat vložené znaky po poslední vložený separátor a tím začít vkládat znovu na pozice, na které již bylo jednou vkládáno. Pro ulehčení práce s touto datovou strukturou nám tato datová struktura nabízí získat ukazatel na začátek posledního vkládaného řetězce.

4. ROZDĚLENÍ PRÁCE

Matěj Žalmánek	Lexikální analýza, testování, dokumentace
Igar Sauchanka	Sémantická analýza, testování, dokumentace
Juraj Novosád	Precedenční syntaktická analýza, sémantická analýza, testování, dokumentace
Eva Mičánková	Syntaktická analýza, testování, dokumentace

5. PRÁCE V TÝMU

Na projektu jsme začali pracovat v půlce října, kdy jsme si prostřednictvím videohovoru rozdělili práci. Poté jsme se téměř každý víkend scházeli v seminárních místnostech na fakultě, kde jsme si ujasňovali látku probranou na přednáškách a pečlivě rozmýšleli aplikaci nově nabytých znalostí na řešení projektu.

5.1 KOMUNIKACE

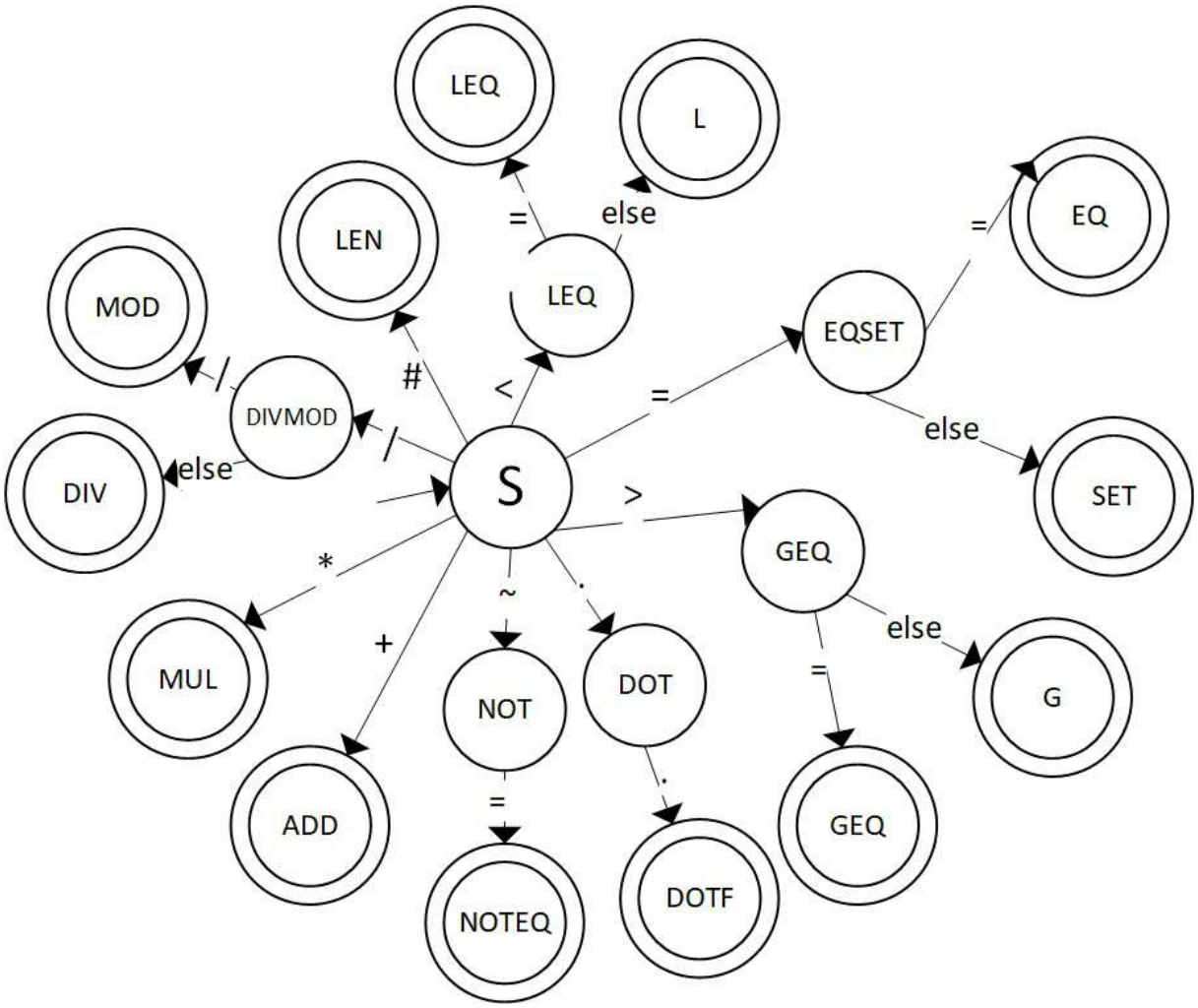
Komunikace v týmu probíhala primárně prostřednictvím aplikace Messenger, kde jsme si domlouvali schůzky a rezervaci seminárních místností, později také různé požadavky na řešení, nalezené chyby atd. Příležitostně jsme komunikovali taky přes aplikaci Discord, kde jsme využili hlavně hlasový kanál pro rychlé hovory a diskusi.

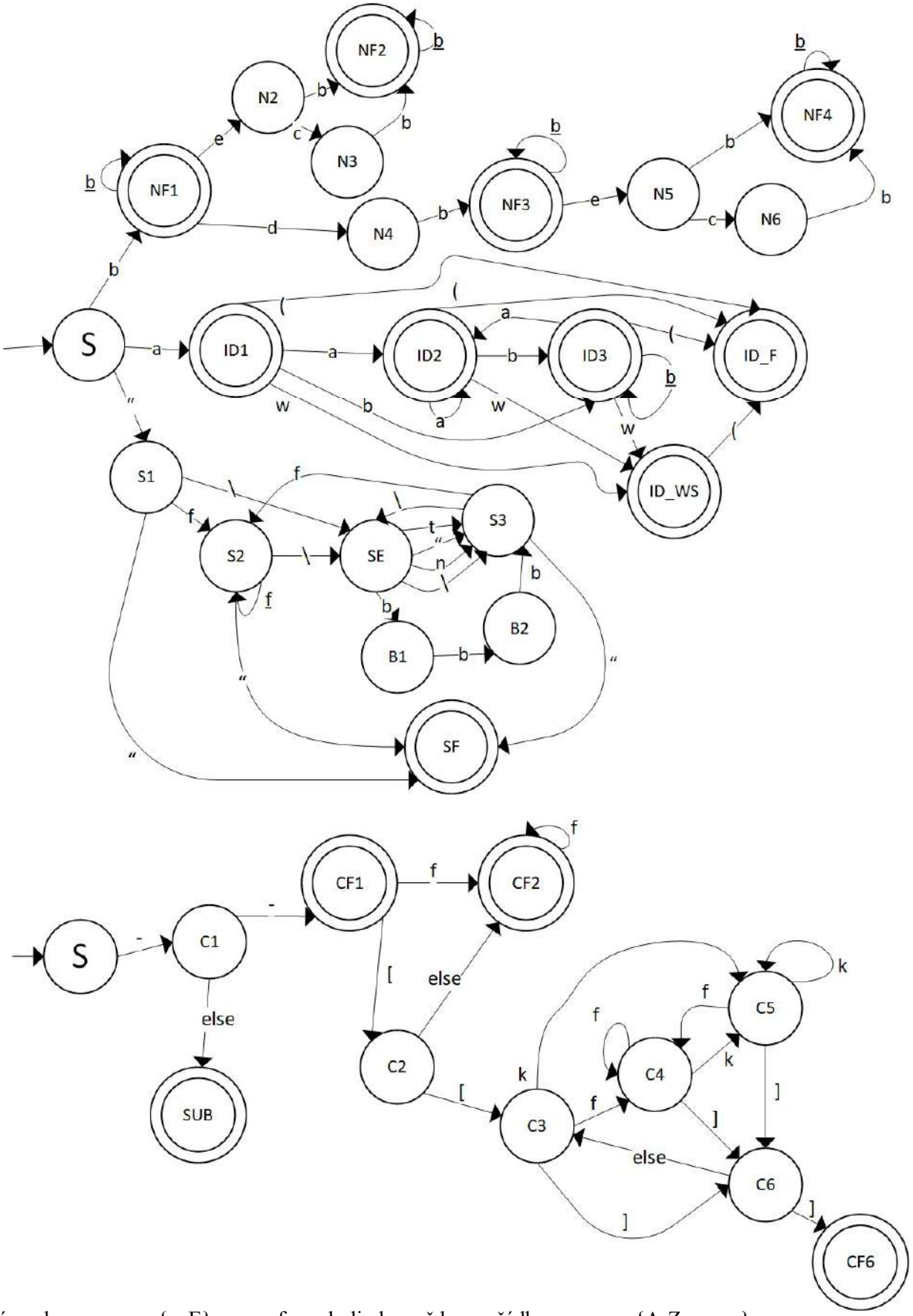
5.2 VERZOVACÍ SYSTÉM

Pro verzování souborů jsme využili systém Git a repozitář na GitHubu, který nám umožnil pracovat ve větvích, na více částech projektu zároveň. Po důkladném otestování jsme své úpravy začlenili do hlavní vývojové větve.

6. DIAGRAM KONEČNÉHO AUTOMATU

Poznámka: Pro přehlednost je stavový automat rozdělen do tří diagramů. Stav Start (S) je vždy jeden a ten samý stav ve všech třech diagramech.





w = bílý znak e = {e, E} f = cokoliv kromě konce řádku a = {A-Z, a-z, _}

d = tečka b = cifra k = konec řádku c = {+,-}

7. LL – GRAMATIKA

1	<prog>	→	require
2	<prog>	→	function id_f (<params_list> <return_fc> end <prog>
3	<prog>	→	global id : function (<types_list> <fc_decl_ret>
4	<prog>	→	EOF
5	<prog>	→	<function_call>
6	<params_list>	→	)
7	<params_list>	→	<first_param> <next_params>
8	<first_param>	→	<param>
9	<next_params>	→	)
10	<next_params>	→	<param> <next_params>
12	<types_list>	→	<first_type> <next_types>
13	<types_list>	→	)
14	<next_types>	→	)
15	<next_types>	→	, <type> <next_types>
17	<type>	→	nil
18	<param>	→	id : <type>
19	<type>	→	integer
20	<type>	→	string
21	<type>	→	number
22	<return_fc>	→	: <first_ret> <next_rets>
23	<return_fc>	→	<function_body>
24	<first_ret>	→	<type>
25	<next_rets>	→	, <type> <next_rets>
26	<next_rets>	→	<function_body>
27	<first_type>	→	<type>
28	<fc_decl_ret>	→	: <fc_ret_first_type> <fc_ret_next_types>
29	<fc_ret_first_type>	→	<type>
30	<fc_ret_next_types>	→	, <type> <fc_ret_next_types>
31	<fc_ret_next_types>	→	<prog>
32	<fc_decl_ret>	→	<prog>
33	<function_call>	→	id_f (<args_list>
34	<args_list>	→	)
35	<args_list>	→	<first_arg> <next_args>
36	<first_arg>	→	<value>
37	<next_args>	→	, <value> <next_args>
37.1	<value>	→	id
37.2	<value>	→	string_value
37.3	<value>	→	number_value
37.4	<value>	→	number_int_value
37.5	<value>	→	nil
38	<next_args>	→	)
39	<function_body>	→	Ɛ
40	<function_body>	→	<return>
41	<function_body>	→	local id : <type> <assignment> <function_body>

42	<function_body>	→	<function_call> <function_body>
43	<function_body>	→	<ids> <expressions> <function_body>
44	<function_body>	→	<while>
45	<function_body>	→	<if>
46	<ids>	→	id <next_id>
47	<next_id>	→	, id <next_id>
48	<next_id>	→	=
49	<expressions>	→	<exp_first> <next_expr>
51	<exp_first>	→	<expression>
52	<next_expr>	→	, <expression> <next_expr>
53	<next_expr>	→	ε
54	<expression>	→	<exp>
55	<exp>	→	call precedenční analýza
56	<expression>	→	<function_call>
57	<assignment>	→	= <expression>
58	<assignment>	→	ε
59	<return>	→	return <list>
60	<list>	→	<expressions>
62	<while>	→	while <exp_cond> do <function_body> <end>
63	<if>	→	if <exp_cond> then <function_body> <elsif>
64	<elseif>	→	elseif <exp_cond> then <function_body> <elseif>
65	<elseif>	→	else <function_body> <end>
66	<elseif>	→	<end>
67	<end>	→	end <function_body>
68	<exp_cond>	→	call precedenční analýza

8. LL – TABULKA

	)	string_value	number_value	number_int_value	nil	id	,	nil	integer	string	number	:	end	local	return	id_f	while	if	elsif	else	global	exp	function	=	EOF	require
<prog>																5					3		2		4	1
<first_arg>		36	36	36	36	36																				
<next_args>	38						37																			
<first_param>						8																				
<value>		37.2	37.3	37.4	37.5	37.1																				
<next_params>	9					10																				
<param>						18																				
<type>								17	19	20	21															
<return_fc>						23						22	23	23	23	23	23	23								
<first_ret>								24	24	24	24															
<next_rets>						26	25						26	26	26	26	26	26								
<first_type>								27	27	27	27															
<next_types>	14						15																			
<fc_decl_ret>												28				32					32		32		32	
<fc_ret_first_type>								29	29	29	29															
<fc_ret_next_types>							30									31					31		31		31	
<function_call>																33										
<function_body>						43							39	41	40	42	44	45								
<ids>						46																				
<next_id>							47																	48		
<expressions>																49						49				
<exp_first>																						51				
<next_expr>						53	52						53	53	53	53	53	53								
<assignment>						58							58	58	58	58	58	58				58		57		
<return>															59											
<list>						60										60						60				
<while>																	62									
<if>																		63								
<elsif>													66						64	65						
<end>													67													
<params_list>	6					7																				
<types_list>	13							12	12	12	12															
<args_list>	34	35	35	35	35	35	35																			
<exp>																						56				

9. PRECEDENČNÁ TABULKA

Typ na vrchole zásobníku \ Typ príchodzieho tokenu	#	/ // *	+ -	..	>< ~ =	i	f_id	(	)	$
#	#	>	>	>	>	<	<	#	>	>
/ // *	<	>	>	>	>	<	<	<	>	>
+ -	<	<	>	>	>	<	<	<	>	>
..	<	<	<	<	>	<	<	<	>	>
>< ~ =	<	<	<	<	>	<	<	<	>	>
i	>	>	>	>	>	>	#	#	>	>
f_id	>	>	>	>	>	>	#	#	>	>
(	<	<	<	<	<	<	<	<	=	#
)	>	>	>	>	>	>	#	#	>	>
$	<	<	<	<	<	<	<	<	#	&

10. TABUĽKA NÁVRATOVÝCH TYPOV

Jedna dimenzia tabuľky návratových typov. Konkrétne pre sčítanie a odčítanie. Inak má tabuľka osem dimenzií pre ostatné aritmetické, relačné, priraďovacie operácie. Tabuľky pre ostatné operátory vyzerajú podobne, len s inými návratovými hodnotami.

Ľavá strana \ Pravá strana	Number	Integer	String	Bool	Nil
Number	Number	Number	NO_TYPE	NO_TYPE	Nil
Integer	Number	Integer	NO_TYPE	NO_TYPE	Nil
String	NO_TYPE	NO_TYPE	NO_TYPE	NO_TYPE	Nil
Bool	NO_TYPE	NO_TYPE	NO_TYPE	Bool	Nil
Nil	Nil	Nil	Nil	Nil	Nil

- NO_TYPE - znamená, že nad danými dvoma dátovými typmi nie je možné vykonať zadanú operáciu
- Number, Bool, String, Integer – aký dátový typ bude mať návratova´hodnota operácie
- Nil – značí neočakávané narábanie s hodnotou nil