Projektová dokumentace

Implementace překladače imperativního jazyka IFJ20

Tým 071, varianta 1

Petr Kabelka	**(xkabel09)**	25%
Daniel Moudrý	(xmoudr01)	25%
Michal Škrášek	(xskras01)	25%
Kryštof Glos	(xglosk01)	25%

9. prosince 2020

Implementovaná rozšíření: **BASE**

Obsah

1 Úvod

Cílem projektu bylo vytvořit v jazyce C program, který umí načíst kód zapsaný ve zdrojovém jazyce IFJ20, který je zjednodušenou podmnožinou jazyka Go 1.15 a následně ho přeloží do cílového jazyka IFJcode20 (mezikód).

Program funguje jako konzolová aplikace, která načítá zdrojový program ze standardního vstupu a výsledný mezikód vypisuje na standardní výstup, nebo v případě chyby končí s příslušným chybovým kódem.

2 Návrh a implementace

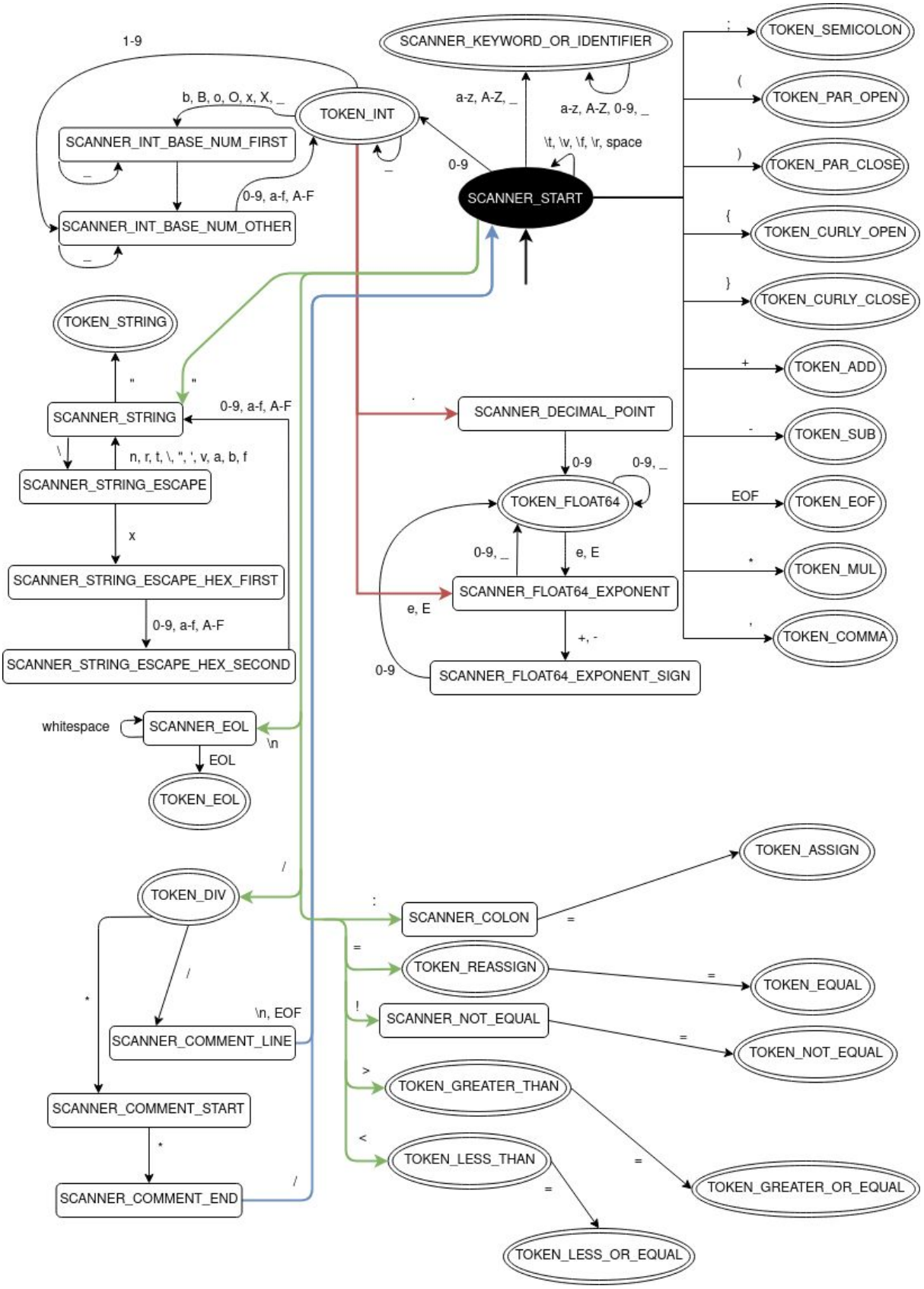
Projekt jsme si rozdělili na několik dílčích částí, které jsme postupně implementovali a napojovali na sebe. Všechny tyto části v této kapitole představíme a uvedeme, jakým způsobem byly vytvořeny a jak mezi sebou jednotlivé části spolupracují.

2.1 Lexikální analýza

Jako první část překladače jsme implementovali lexikální analyzátor. Primární funkcí lexikální analýzy je funkce *get_next_token*, která čte standardní vstup znak po znaku a vytváří strukturu *token*, která se skládá z typu a atributu. Typy tokenu jsou *EOF*, *EOL*, prázdný token, identifikátory, klíčová slova, celé či desetinné číslo, řetězec a také porovnávací a aritmetické operátory a ostatní symboly, které mohou být použity v jazyce IFJ20. Atribut tokenu je typu *union* a dělí se podle typu tokenu. Union se zde využívá hlavně kvůli šetření paměti při předávání hodnot tokenu. Atribut tokenu se využívá pouze pro tokeny typu *TOKEN_STRING*, *TOKEN_IDENTIFIER*, *TOKEN_FLOAT64*, *TOKEN_INT a TOKEN_KEYWORD*, v jiných případech atribut nelze využít. S vytvořeným tokenem posléze pracují další části překladače.

Celý lexikální analyzátor je implementován jako deterministický konečný automat podle vytvořeného diagramu (obr. 1). Konečný automat je v jazyce C implementovaný jako *switch* uvnitř nekonečného cyklu *while*, kde každý *case* je ekvivalentní jednomu stavu automatu. Pokud se načtený znak neshoduje s žádným znakem očekávaným v určitém stavu, program je ukončen a vrací chybu 1. V opačném případě se pokračuje do dalších stavů a načítají se další znaky, dokud se nedokončí celý token, ten se poté vrací a funkce končí.

Při zpracování identifikátorů a klíčových slov se řetězec načítá do struktury *string* a typ tokenu se nastaví na *TOKEN_KEYWORD*. Následně se porovnává s klíčovými slovy, pokud se najde shoda, nastaví se atribut *kw* na příslušné klíčové slovo, v opačném případě se nastaví typ tokenu na *TOKEN_IDENTIFIER* a nastaví se atribut *str* jehož hodnotou je načtený identifikátor.



obr. 1 - Konečný automat lexikálního analyzátoru (barvy pouze pro rozlišení cest)

2.2 Syntaktická analýza

Syntaktická analýza se celá, kromě výrazů, řídí LL(1) gramatikou a je implementovaná metodou rekurzivního sestupu který se řídí podle pravidel v LL tabulce (obr. 2). Jednotlivá pravidla jsou v kódu zapsána jako samostatné funkce, kterým se předává ukazatel na strukturu *data_t*, která obsahuje proměnné potřebné ke zpracování syntaktické, sémantické analýzy a také generování kódu. Syntaktická analýza získává tokeny pomocí funkce *get_next_token*. Tyto tokeny načítá lexikální analyzátor tak, jak je rozepsáno výše.

Syntaktický analyzátor si na zásobník *var_table* při vstupu do nového rámce platnosti ukládá novou tabulku symbolů pro aktuální rámec platnosti. Při vstupu do hlubšího rámce platnosti se také inkrementuje proměnná *scope_idx* ve struktuře *data_t*. Proměnné uložené v tabulce symbolů nesou informaci o jejich identifikátoru, datovém typu a indexu rámce platnosti. Při hledání proměnné se prochází tento zásobník a vyhledává se v dílčích tabulkách symbolů. Při odcházení z rámce platnosti se vyprázdní nejvrchnější tabulka symbolů a odstraní se ze zásobníku.

2.2.1 Pravidla LL gramatiky

```
(1)    parse → prolog EOL function EOF
(2)    function → func_header { EOL command EOL function
(3)    function → ε
(4)    func_header → func identifier ( func_args ) func_return_vals
(5)    func_args → identifier var_type , func_args
(6)    func_args → ε
(7)    func_return_vals → var_type
(8)    func_return_vals → ( var_type func_return_vals_par )
(9)    func_return_vals → ε
(10)    func_return_vals_par → , var_type func_return_vals_par
(11)    func_return_vals_par → ε
(12)    var_type → int
(13)    var_type → float64
(14)    var_type → string
(15)    command → identifier func_or_list_of_vars EOL command
(16)    command → is_inter_func ( call_func ) EOL command
(17)    command → if statement EOL command
(18)    command → for cycle EOL command
(19)    command → return returned_vals EOL command
(20)    command → }
(21)    statement → condition { EOL command else { EOL command
(22)    func_or_list_of_vars → , list_of_vars
(23)    func_or_list_of_vars → ( call_func )
(24)    func_or_list_of_vars → := assignment
(25)    func_or_list_of_vars → = reassignment
(26)    call_func → func_calling
(27)    func_calling → const_val_identifier func_calling_n
(28)    func_calling → ε
(29)    func_calling_n → , const_val_identifier func_calling_n
(30)    func_calling_n → ε
(31)    const_val_identifier → identifier
(32)    const_val_identifier → int_val
(33)    const_val_identifier → float64_val
(34)    const_val_identifier → string_val
(35)    list_of_vars → identifier list_of_vars_n list_of_vars_op
(36)    list_of_vars_n → , identifier list_of_vars_n
(37)    list_of_vars_n → ε
(38)    list_of_vars_op → := assignment
```

```
(39)    list_of_vars_op → = reassignment
(40)    is_inter_func → len
(41)    is_inter_func → ord
(42)    is_inter_func → chr
(43)    is_inter_func → substr
(44)    is_inter_func → float2int
(45)    is_inter_func → int2float
(46)    is_inter_func → print
(47)    is_inter_func → inputf
(48)    is_inter_func → intputi
(49)    is_inter_func → inputs
(50)    assignment → end_of_assignment
(51)    reassignment → end_of_assignment
(52)    end_of_assignment → expr
(53)    cycle → cycle_list_of_assign ; condition ; end_of_cycle { EOL command EOL
(54)    cycle_list_of_assign → identifier cycle_list_of_assign_n := assignment
(55)    cycle_list_of_assign → ε
(56)    cycle_list_of_assign_n → , identifier cycle_list_of_assign_n
(57)    cycle_list_of_assign_n → ε
(58)    cycle_list_of_reassign → identifier cycle_list_of_reassign_n = reassignment
(59)    cycle_list_of_reassign → ε
(60)    cycle_list_of_reassign_n → , identifier cycle_list_of_reassign_n
(61)    cycle_list_of_reassign_n → ε
(62)    end_of_cycle → cycle_list_of_reassign
(63)    condition → expr
(64)    returned_vals → next_returned_val
(65)    next_returned_val → , next_returned_val
(66)    next_returned_val → ε
```

	prolog	EOF	EOL	;	:=	=	func	identifier	int	float64	string	int_val	float64_val	string_val	if	else	for	return	{	}	(	)	,	chr	ord	len	substr	float2int	int2float	print	inputf	intputi	inputs	$
parse	1																																
function		3					2																										
func_header							4																										
func_args								5														6											
func_return_vals									7	7	7										8												
func_return_vals_par																						11	10										
var_type									12	13	14																						
command								15							17		18	19		20				16	16	16	16	16	16	16	16	16	16	
statement																																	
func_or_list_of_vars					24	25															23		22										
call_func								26				26	26	26								26											
func_calling								27				27	27	27								28											
func_calling_n																						30	29										
const_val_identifier								31				32	33	34																			
const_val																																	
list_of_vars								35																									
list_of_vars_n					37	37																	36										
list_of_vars_op					38	39																											
is_inter_func																								42	41	40	43	44	45	46	47	48	49	
expression																																	
assignment																																	
reassignment																																	
end_of_assignment																																	
cycle				53				53																									
cycle_list_of_assign				55				54																									
cycle_list_of_assign_n					57																		56										
cycle_list_of_reassign								58											59														
cycle_list_of_reassign_n																							60										61
end_of_cycle																			62														
condition																																	
returned_vals			64																				64										
next_returned_val			66																				65										

obr. 2 - LL tabulka

2.2.2 Zpracování výrazů

Výrazy, jak již bylo řečeno v bodě 2.2, se tedy neřeší jako vše ostatní, ale pomocí precedenční syntaktické analýzy. Zpracování těchto výrazů se dělá v jiném souboru a to v *expression.c* kde se provede vše od syntaktické kontroly až po generaci samotného kódu.

Jakmile parser očekává výraz, tak je volána hlavní funkce *expression*, která načítá výraz a pomocí precedenční analýzy jej uloží do dvousměrně vázaného spojového seznamu *list*. Seznam obsahuje symboly, symbol je buď operátor, nebo konstanta (42, 3.14, “text”, ...), nebo proměnná (a, x, ...). Výsledkem precedenční syntaktické analýzy je výraz v postfixové notaci, ze které lze lehce generovat instrukce.

2.2.2.1 Redukce výrazů

Před generováním instrukcí pro výraz se však provádí zjednodušení částí výrazů obsahujících konstantní hodnoty. Redukce výrazů se odehrává v souboru *optimizer.c* ve funkci *optimize*. Tato funkce prochází celý seznam po jednom symbolu zleva doprava, pokud narazí na operátor (+, -, ...), tak se podívá na předchozí dva symboly, pokud tyto poslední znaky byly konstanty, tak se zredukují tyto tři symboly na jeden symbol př.: obsah *list*u [5 2 +] se zredukuje na [7], výraz [a b + 7 5 - +] v postfixovém zápisu se zredukuje na výraz [a b + 2 +]. Taktéž se při redukci provádí konkatenace řetězcových literálů.

Další situace při kterých je možné zredukování jsou např. násobení proměnné nulou, jedničkou, přičítání nuly, etc, kde je zbytečné provádět tyto operace protože výsledek je předem známý. Tato redukce se dá využít pro detekci dělení nulou ve složitějších výrazech složených z konstantních hodnot.

	+, -	***, /**	**(**	**)**	**<, >, ==, >=, <=, !=**	**id**	**$**
+, -	>	<	<	>	>	<	>
***, /**	>	>	<	>	>	<	>
(	<	<	<	=	<	<	
)	>	>		>	>		>
<, >, ==, >=, <=, !=	<	<	<	>		<	>
id	>	>		>	>		>
$	<	<	<		<	<	

tab. 1 - Precedenční tabulka

2.3 Sémantická analýza

Ve struktuře *data_t* jsou uloženy tabulky symbolů na dvou zásobnících - lokální pro lokální proměnné a globální pro funkce. Na vrcholech těchto zásobníků se vždy nachází tabulka s nejaktuálnějším rámcem platnosti. Tabulka symbolů je implementována jako binární vyhledávací strom, kam se ukládají informace o proměnných v aktuálním rámci platnosti. Informace z tabulky symbolů využíváme ke kontrole zda zadaný identifikátor vůbec existuje a posléze jestli souhlasí datový typ, který do něj ukládáme, či se kterým aktuálně pracujeme ve výrazu. U funkcí kontrolujeme počet a typ návratových hodnot a parametrů.

2.4 Generování instrukcí

Jednotlivé části kódu jsou generovány v průběhu analýz a jsou průběžně ukládány do 4 proměnných, *for_assigns* (přiřazení v hlavičkách *for* cyklů), *func_declaration* (deklarace všech proměnných ve funkci), *func_body* (tělo funkce) a *ifjcode20_output* do které se na konci zpracování funkce připojí obsahy *func_declaration* a *func_body*, ty se následně vyprázdní pro použití v další zpracovávané funkci. Po úspěšném dokončení všech analýz se vytiskne obsah *ifjcode20_output* na standardní výstup.

2.4.1 Rozhraní generátoru instrukcí

Obsahuje funkce, které fungují jako šablony pro generované instrukce. Funkce přijímají například název funkce, či index rámce platnosti.

2.4.2 Začátek generování instrukcí

Na začátku se inicializují řetězce pro generování instrukcí (*for_assigns*, *func_declaration*, *func_body*, *ifjcode20_output*), které budou potřeba v průběhu analýzy kódu, tyto struktury se na konci generování opět uvolní. Dále se musí vygenerovat hlavička mezikódu, ta se skládá z deklarace pomocných globálních proměnných a skokem na funkci *main*.

2.4.3 Deklarace proměnných

Všechny proměnné použité uvnitř funkce jsou deklarovány na začátku funkce a pro vypsání deklarací na začátek funkce slouží řetězec *func_declaration*. Důvodem pro deklarace proměnných na začátku funkce je umožnění deklarace proměnných uvnitř cyklu *for*. Pro uložení informací o všech proměnných uvnitř funkce slouží zásobník *defvar_table*, který obsahuje tabulky symbolů ze všech rámců platnosti uvnitř funkce. Díky těmto tabulkám v *defvar_table* se pro proměnnou se vygeneruje deklarační instrukce pouze při prvním průchodu cyklem. Zásobník *defvar_table* se narozdíl od *var_table* vyprazdňuje až na konci funkce. Každá proměnná v tabulce obsahuje index rámce platnosti ve kterém se nachází.

Pokud se při analýze objeví proměnná, která nebyla v žádném rámci platnosti deklarována, přidá se do nejvrchnější tabulky v *defvar_table* a vygeneruje se pro ni instrukce `DEFVAR` do řetězce *func_declaration*.

Proměnné je nutné indexovat pro umožnění překrývání proměnných v hlubších rámcích platnosti. Aby bylo toto umožněno, každý rámec platnosti má svůj index (hloubku) a všechny proměnné uvnitř něj mají tento index specifikovaný za % v jejich názvu. Například `LF@var%1` identifikuje proměnnou `var` uvnitř funkce bez dalšího zanoření.

3 Použité algoritmy a struktury

Při implementaci byly využity struktury: dvousměrný spojový seznam, string, binární vyhledávací strom a generický zásobník, které v této kapitole budou popsány.

3.1 Dvousměrný spojový seznam

Tato struktura je zde použita kvůli zpracování výrazů a přistupujeme k němu jako k zásobníku. Další využití této struktury bylo při ukládání argumentů u funkce *print* kdy funkce vypisovala argumenty v opačném pořadí a použili jsme ji jako zásobník pro obrácení pořadí argumentů.

3.2 Struktura string

Tato struktura implementující řetězce proměnlivé délky, je v celém projektu využívána ze všech nejčastěji. Funkce které nad touto strukturou fungují jsou určeny pro zjednodušení práce s řetězci, jako přidávání znaků a literálů, či kopírování řetězců. Tato struktura je použita pro string literály a identifikátory proměnných.

Při implementaci[1] jsme se na chvíli zasekli u alokování paměti kdy v některých případech způsobovala funkce segmentation fault, tato chyba byla detekována při testech a ihned odstraněna.

3.3 Generický zásobník

Struktura implementovaná pomocí jednosměrně vázaného lineárního seznamu. Vrchol zásobníku ukazuje na první element seznamu. Prvek zásobníku je struktura která obsahuje ukazatel na prvek pod ním a ukazatel typu void na data. Ukazatel na void umožňuje ukládání

[1] Implementace této knihovny je inspirována knihovnou *str.c* nalezenou v souboru *jednoduchy_interpret.zip*: https://www.fit.vutbr.cz/study/courses/IFJ/public/project/

ukazatele na libovolný datový typ. Díky tomuto zásobníku není třeba implementovat zásobníky pro specifické účely, avšak programátor si musí dát pozor na datový typ dat která ukládá na zásobník, aby nepřetypoval ukazatel na data na špatný typ při přístupu k nim.

S touto strukturou a její implementací nebyl žádný zásadní problém až na zmíněnou problematiku přetypování při přístupu k datům uložených ve složitějších strukturách.

3.4 Binární vyhledávací strom

Strukturu jsme využili pro implementování tabulky symbolů jak nám bylo zadáno. Celá tato struktura byla implementována podle postupů vyučovaných v předmětu IAL. Při implementaci jsme využili nerekurzivní přístup proto, aby kód byl kód co nejvíce efektivní až na funkci *symtable_delete_node*, která by byla zbytečně složitá nerekurzivně.

4 Práce v týmu

4.1 Kooperace v týmu

Na samotném projektu jsme začali pracovat v polovině listopadu kvůli zhoršeným komunikačním podmínkám z důvodu uzavření škol a také kvůli prvotnímu nepochopení látky z přednášek. Neměli jsme dopředu rozhodnuto kdo bude co dělat a úkoly jsme si rozdělovali podle potřeby v průběhu práce.

Komunikace mezi členy byla vzhledem ke zhoršeným podmínkám horší než obvykle, protože probíhala převážně přes Facebook Messenger a TeamSpeak.

4.2 Verzovací systém

Na verzování projektu a pro správu souborů a jednotlivých částí projektu jsme použili verzovací systém Git.

V projektu jsme pracovali na více částech současně a hned je nahrávali do master větve kde jsme je poté testovali a dále upravovali. Až za polovinou projektu jsme si uvědomili že to byla chyba a měli jsme verzování dělat následovně: Všechny vyvíjené části překladače jsme měli ukládat nejdříve do vývojových větví kde bychom je všechny postupně testovali a následně po všech úspěšných testech připojovali do master větve kde už by se nijak zásadně neupravovali jak pro přehlednost, tak kvůli tomu aby každý commit v historii repozitáře byl funkční.

4.3 Rozdělení práce

Práce na projektu byla rozdělena rovnoměrně mezi všechny členy týmu vzhledem k podílu práce všech na projektu.

Petr Kabelka - vedení týmu, organizace práce, LL gramatika, LL tabulka, lexikální analýza, testování, generování cílového kódu, dokumentace
Daniel Moudrý - syntaktická analýza, redukce výrazů, prezentace
Michael Škrášek - LL gramatika, syntaktická analýza, sémantická analýza, precedenční syntaktická analýza
Kryštof Glos - tabulka symbolů, generování cílového kódu, testování, dokumentace

5 Závěr

Projekt byl zatím nejdelší a nejsložitější z dosavadních projektů. Vzhledem k tomu kdy byl zadán, jsme neměli hned dostatek znalostí na to abychom na něm mohli začít pracovat a tudíž jsme začali docela pozdě, ale přesto jej stihli s malou časovou rezervou.

Při řešení se objevila spousta nejasností vyplívajících ze zadání, která byla řešena na fóru předmětu nebo v kanále předmětu na Discordu.

Celkově byl tento projekt přínosný zejména z pohledu spolupráce ve složitějších podmínkách a na větších projektech, více jsme se naučili pracovat v týmu a pochopili různé aspekty Gitu a hlavně se poučili ze svých chyb, při řešení různých problémů.