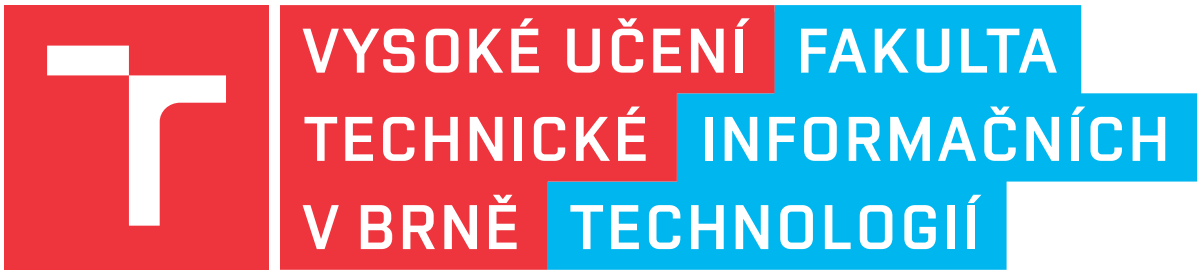


Dokumentace k projektu z předmětů IFJ a IAL Implementace překladače imperativního jazyka IFJ21.

Tým 119, varianta I

8. prosince 2021

Fesiun Bohdan (xfesiu00)	0%
Tikhonov Maksim (xtikho00)	60%
Sadovskyi Dmytro (xsadov06)	40%
Galliamov Eduard (xgalli01)	0%

1 Úvod

Cílem projektu bylo výtvořit program v jazyce C, který načítá zdrojový kód zapsáný v jazyce IFJ21 a přeloží ho do jazuka IFJ21code. Návratová hodnota programu je kód chyby (vrací 0 pokud překlad proběhnul bez chyb). IFJ21 je zjednodušenou podmnožinou jazyka Teal. Teal je staticky typovaný imperativní jazyk.

2 Návrh a implementace

2.1 Lexikální analýza

Lexikální analýzator je implementován ve zdrojovém souboru `scanner.c`, jehož hlavní funkcí je `get_next_token`, která rozpoznává jednotlivé lexémy, transformuje je na tokeny a předává tokeny syntaktickému analýzatoru. **Token** je struktura, která se skládá z **datového typu** (`type enum`), který poskytuje informace o přečteném lexému, a **atributu** (`type union`), který v závislosti na získaném řetězci může reprezentovat konstantní hodnotu (**integer**, **number**, **string**, **nil**), **klíčové slovo** nebo **identifikator**.

Lexikální analýza se provádí prostřednictvím přečtení zdrojového kódu znak po znaku pomocí nekonečného `while` cyklu a **deterministického konečného automatu** uvnitř něho. **Konečný automat** je implementován jako `switch-case`, kde každý case je jedním ze stavů automatu a na základě přijatého znaku se rozhoduje, do jakého stavu by měl automat v dalším kroku přejít.

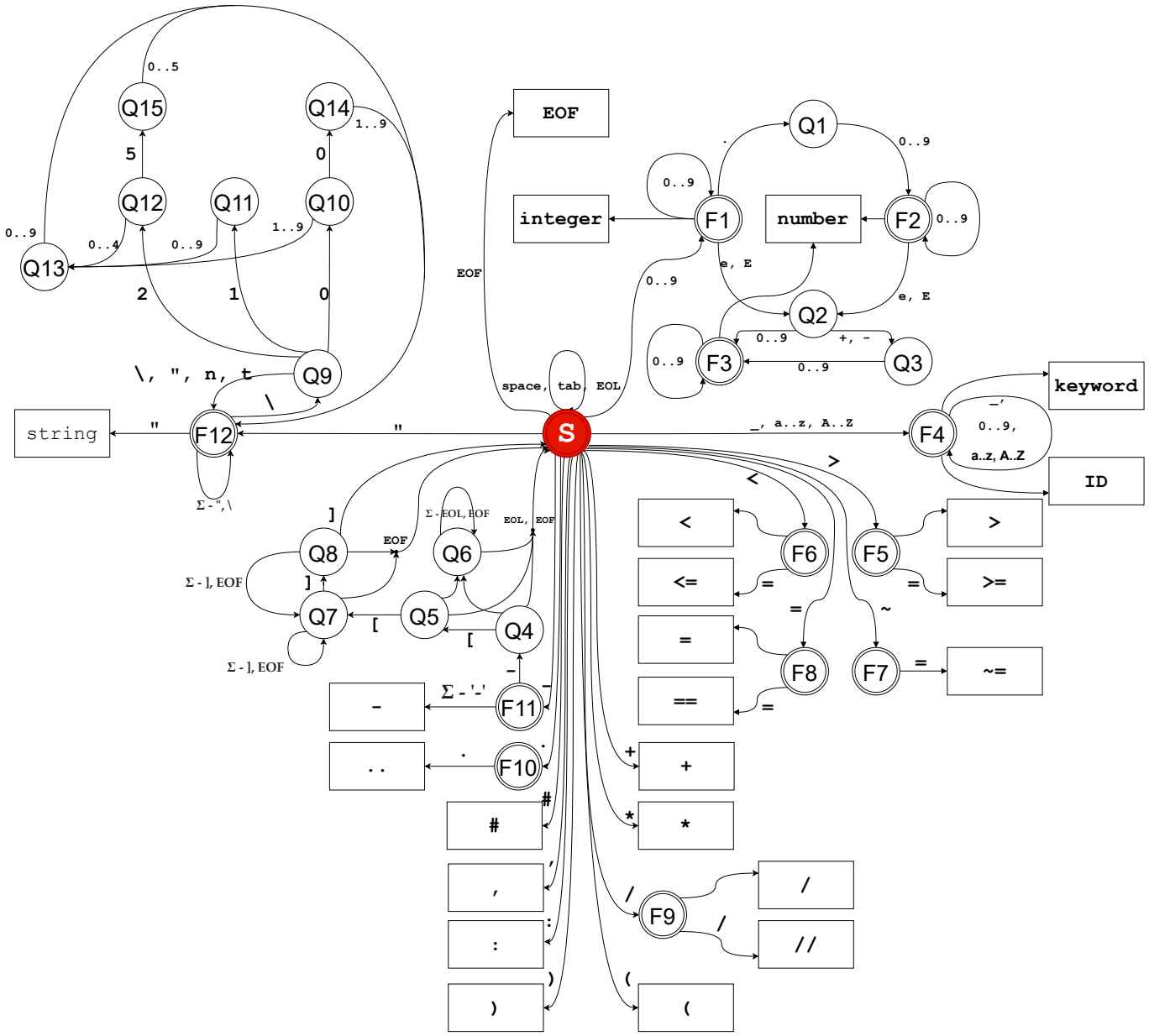
Pokud automat načte **token** předpokládaného typu T_IDE (**identifikátor**), proběhne kontrola, pokud se nejedná o **klíčové slovo** s následnou možnou změnou typu **tokenu** a přiřazením **klíčového slova** jako **atributu**.

Pokud automat načte číslo jakéhokoli druhu, bude provedena konverze z typu **string** do číselného typu (**float** nebo **integer**) pomocí funkce `strtod`

Při náčtení celé escape sekvence nebude řetězec tvaru `\XXX` vždy konvertovan do jediného znaku. Například pro všechny XXX $<$ 032 a pro XXX $=$ 034 platí že budou zapsaný do výsledného řetězce jako escape sekvence. Platí také to, že to funguje i obráceně (náčtené escape sekvence `'\n'`, `'\"'`, ' ' budou konvertovany do tvaru `\XXX`). Je navrženo tak především pro znaky `'\10'` (new line), `'\34'` (”), `'\32'` (Space) pro korektní zápis při generování kódu.¨

V případě náčtení nekorektní kombinace znaků vratí funkce `get_next_token` chybu 1 (chybná struktura aktuálního lexému)

2.1.1 Diagram konečného automatu, který specifikuje lexikální analyzátor



Obrázek 1: Diagram KA lexikalního analyzatoru

2.1.2 Legenda

S - počáteční stav, FX - koncové stavy, QX - nekoncové stavy.

S — SCANNER_STATE_START	F4 — SCANNER_STATE_ID
F1 — SCANNER_STATE_INT Q1 — SCANNER_STATE_POINT F2 — SCANNER_STATE_DOUBLE Q2 — SCANNER_STATE_EXP F3 — SCANNER_STATE_EXP_NUM Q3 — SCANNER_STATE_EXP_SIGN	F5 — SCANNER_STATE_MT F6 — SCANNER_STATE_LT F7 — SCANNER_STATE_TILD F8 — SCANNER_STATE_EQUAL_SIGN
F9 — SCANNER_STATE_SLASH F10 — SCANNER_STATE_DOT F11 — SCANNER_STATE_FIRST_DASH	Q4 — SCANNER_STATE_COMMENT_LINE Q5 — SCANNER_STATE_COMMENT_LSB Q6 — SCANNER_STATE_COMMENT_READ Q7 — SCANNER_STATE_COMMENTBLOCK Q8 — SCANNER_STATE_COMMENTBLOCK_EXIT
F12 — SCANNER_STATE_STRING Q9 — SCANNER_STATE_ESC_SEC Q10 — SCANNER_STATE_ESC_ZERO Q11 — SCANNER_STATE_ESC_ONE Q12 — SCANNER_STATE_ESC_TWO Q13 — SCANNER_STATE_ESC_OTHER Q14 — SCANNER_STATE_ESC_ZERO_ZERO Q15 — SCANNER_STATE_ESC_TWO_FIVE	

2.2 Syntaktická a sémantická analýza

Syntatická a **sémantická analýza** je implementována v souborech `parser.c` (hlavní tělo) a `expression.c` (zpracování výrazů). Po inicializaci **dynamických řetězců** pro `scanner` a `generator kódu` a pomocné struktury `Parser_data`, která se skládá z inicializací **zásobníku tabulek symbolů**, globální tabulky symbolů, **pomocné fronty** a uložení **vestavěných funkcí** do globální tabulky se začíná analýza vstupního kódu. **Syntaktická analýza** se provádí metodou rekurzivního sestupu pomocí LL-gramatiky podle jednotlivých pravidel.

V průběhu analýzy `parser` mnohokrát používá funkce `get_next_token` a na základě získaného tokenů řeší jaké LL-pravidlo má použit. Všechna důležitá data, jako například aktuální index parametru funkce nebo ukazatel na volanou funkci, jsou během analýzy uložená do struktury `Parser_data`

Výsledkem všech funkcí reprezentujících LL-pravidla je kód chyby (0 pokud nedojde k žádné chybě)

Sémantická analýza v rámci souboru `parser.c` se provádí pomocí **tabulek symbolů**: koná se kontrola korespondence datových typů při příkazu přiřazení, definice nebo volání funkce; kontrola zda **identifikator** byl deklarovaný před jeho výskytem jako parametr funkce nebo jako součást výrazu. V souboru `expression.c` prochází kontrola správnosti konkretních výrazů a kompatibility typů operandů. Různé typy operandů nebo nevhodný typ operandů pro určité operace ve většině případů způsobí chybu 6 (sémantická chyba typové kompatibility ve výrazech). Výjimkou jsou typy **number** a **integer** pro které platí typova kompatibilita.

2.2.1 LL-tabulka

	REQUIRE	GLOBAL	FUNCTION	ID	var_ID	func_ID	LOCAL	IF	WHILE	RETURN	WRITE	EOF	INTEGER	NUMBER	STRING	=	,	value	EOF	$
<program>																				0
<function list>		1	2	3															4	
<function declaration>																				5
<declaration params>													6	6	6					7
<other declaration param>																	8			9
<function definition>																				10
<definition params>				11																12
<other definition param>																	13			14
<definition param>																				15
<output>													16	16	16					17
<other output type>																	18			19
<statement list>					20	21	22	23	24	25	26									27
<ID list>																				28
<other ID>																	29			30
<assignment values>																				31
<other assignment value>																	32			33
<assignment value>						34														35
<function call params>																				36
<other call param>																	37			38
<call param>																				39
<optional definition>																40				41
<value>						42														43
<return values list>																		44		45
<other return values>																	46			47
<return value>																				48
<write argument lits>																		49		50
<other write argument>																	51			52
<type>													53	54	55					

2.2.2 Seznam pravidel

0. <program> -> REQUIRE "ifj21" <function list>
1. <function list> -> <function declaration> <function list>
2. <function list> -> <function definition> <function list>
3. <function list> -> <entry point> <function list>
4. <function list> -> EOF
5. <function declaration> -> GLOBAL ID : FUNCTION (<declaration params>) <output>
6. <declaration params> -> <type> <other declaration param>
7. <declaration params> -> ε
8. <other declaration param> -> , <type> <other declaration param>
9. <other declaration param> -> ε
10. <function definition> -> FUNCTION ID (<definition params>) <output> <statement list> END
11. <definition params> -> <definition param> <other definition param>
12. <definition params> -> ε
13. <other definition param> -> , <param> <other definition param>
14. <other definition param> -> ε
15. <definition param> -> ID : <type>
16. <output> -> <type> <other output type>
17. <output> -> ε
18. <other output type> -> , <type> <other output type>
19. <other output type> -> ε
20. <statement list> -> <ID list> <assignment values> <statement list>
21. <statement list> -> ID (<function call param list>) <statement list>
22. <statement list> -> LOCAL ID : <type> <optional definition> <statement list>
23. <statement list> -> IF <expression> THEN <statement list> ELSE <statement list> END <statement list>
24. <statement list> -> WHILE <expression> DO <statement list> END <statement list>
25. <statement list> -> RETURN <return list> <statement list>
26. <statement list> -> WRITE (<write argument list>) <statement list>
27. <statement list> -> ε
28. <ID list> -> ID <other ID>

29. <other ID> -> , ID <other ID>
30. <other ID> -> ε
31. <assignment values> -> = <assignment value> <other assignment value>
32. <other assignment value> -> , <assignment value> <other assignment value>
33. <other assignment value> -> ε
34. <assignment value> -> ID(<function call params>)
35. <assignment value> -> <expression>
36. <function call params> -> <call param> <other call param>
37. <other call param> -> , <call param> <other call param>
38. <other call param> -> ε
39. <call param> -> <expression>
40. <optional definition> -> = <value>
41. <optional definition> -> = ε
42. <value> -> ID(<function call params>)
43. <value> -> <expression>
44. <return values> -> <return value> <other return value>
45. <return value list> -> ε
46. <other return value> -> , <return value> <other return value>
47. <other return value> -> ε
48. <return value> -> <expression>
49. <write argument list> -> <expression> <other write argument>
50. <write argument list> -> ε
51. <other write argument> -> , <expression> <other write argument>
52. <other write argument> -> ε
53. <type> -> INTEGER
54. <type> -> NUMBER
55. <type> -> STRING

2.3 Zpracování výrazů

Zpracování výrazů je implementováno v souboru `expression.c` a provádí se funkcí `expression`, kterou volá **syntakticky analyzátor**. **Výrazy** se zpracovávájí pomocí **precedenční tabulky** a **zásobníku symbolů**. Zpracování jednotlivých výrazů se začíná s inicializací zásobníku symbolů a ukládání symbolu dolaru na jeho vrchol, pak token po tokenu začíná funkce analýzovat vstupní výraz. Z každého tokenu funkce bere potřebnou informace (symbol, datový typ (pokud je typ tokenu T_IDE (**identifikátor**), pak ho najde v (**tabulkách symbolů** a předá typ jeho hodnoty), a informace, pokud je tento token konstantní nula. Pak na základě nejbližšího k vrcholu zásobníku terminálu a vstupního symbolu, který je získan z následujícího tokenu, funkce provádí jednu z 4 operací - **shift**, **reduce**, **equal**, **blank**; všechny možné kombinace vstupních symbolů a prvních terminálů na vrcholů jsou popsány precedenční tabulkou.

Pokud znak z tabulky je <, vloží menšítko před prvním terminálem vloží náčtený symbol na vrchol zásobníku (funkce `shift`) a zavolá funkce `get_next_token`.

Pokud znak z tabulky je >, ověří se že mezi < a > je validní výraz (funkce `test_rule` pro nalezení vhodného pravidla a funkce `test_semantics` pro ověření typové kompatibility, vhodnosti typů operandů pro konkretní operace a možná i, pokud je to nutné, přetypování některých operandů) a zredukuje získaný výraz do nonteminálu E s výsledným datovým typem (funkce `reduce`).

Pokud znak z tabulky je =, provede `push` a zavolá funkce `get_next_token`

Pokud znak z tabulky je blank, začne redukovat všechno, co se zůstalo v **zásobníku** až do okamžiku, kdy prvním terminalem v **zásobníku** bude dolar pak zkontroluje symbol následujícího **tokenů** a v případě korektnosti **výrazu** nastaví booleovskou proměnnou `success` na true, což způsobí východ z loopu.

Po zjištění, že výraz je korektní, funkce uloží výsledek do globální proměnné GF@%expResult, a pokud ukazatel na aktualní proměnnou `lhsId` není NULL, uloží do ní.

2.3.1 Precedenční tabulka

Stack top\Input	+, -	*, /, //	rel.	(	)	i	#	..	$
+, -	>	<	>	<	>	<	<	>	>
*, /, //	>	>	>	<	>	<	<	>	>
rel.	<	<		<	>	<	<	<	>
(	<	<	<	<	=	<	<	<	
)	>	>	>		>		>	>	>
i	>	>	>		>		>	>	>
#	>	>	>	<	>	<	<	>	>
..	<	<	>	<	>	<	<	<	>
$	<	<	<	<		<	<	<	

2.4 Generování kódu

Generování kódu je implementováno v souboru `code_generator.c` provádí se pomocí použití maker `ADD_CODE` a `ADD_LINE` (`ADD_CODE + '\n'`), které utilizují funkce `ds_add_chars` **dynamického řetězce** která doplňuje výsledný kód (v průběhu programy je reprezentován jako **dynamický řetězec**) novými příkazy v mezikódu IFJ21code.

Generování kódu je nikoliv samostatná součást překladače ale se vždy příkazy z touto komponenty používají během **syntaktické analýzy** (včetně **zpracování výrazů**) a **sémantické analýzy**. Pro spravné pojmenování proměnných a pro indexování návěští while a if ve struktuře `Parser_data` jsou

Po nalezení prologu budou vygerovány prolog mezikódu, pomocné globální proměnné a vestavěné funkce.

Při generování definicí funkcí bude vygenerován JUMP `"nad nimi"` kvůli tomu, že interpret hledá první volání funkce a nemusí spouštět části kódu, které nebyly volány. Pak vygeneruje navěští funkce ve tvaru `$functionId` a PUSHFRAME aby dočasný rámec přikryl aktuální. Pro předání parametrů jsou používány LF@%N, kde N je libovolné celé číslo. Pokud funkce má návratové hodnoty LF@%retvalN budou deklarovány na začátku těla funkce a budou mít hodnotu nil@nil. Těsně před koncem těla funkce se používá příkaz POPFRAME pro přesun lokálního rámce do dočasného.

Před voláním funkce vždy bude vytvořen nový rámec příkazem CREATEFRAME a budou do proměnných TF@%N předány hodnoty, které jsou uvedeny jako parametry, pak bude zavolána funkce. Pro předání výsledků do proměnných (pokud volání je v příkazu přiřazení nebo definice) budou použity proměnné TF@%N z dočasného rámce.

Při generování kódu pro výrazy hodně se používá zásobník a zásobníkové operace. Jsou implementovány příkazy pro generace kódu s hodnotami na zásobníku a to jsou LENS (#), CATS (..), LETS, (<=), METS (>=), NEQS (=). Výsledek vypracovaného výrazu se ukládá do globální proměnné GF@%expResult

Při generování kódu pro návěští if a while se používají data ze struktury `Parser_data` pro správné indexování.
Návěští pro if mají tvar : `$functionId$ifIndex$if$součást_if`,
pro while : `$functionId$whileIndex$loop$součást_while`

Výsledný kód se vypisuje až na konci běhu programu.

2.5 Pomocné struktury

2.5.1 Zásobník tabulek symbolů

Zásobník tabulek symbolů je implementován jako seznam prvků a každý jeho prvek poskytuje informace o všech identifikátorech, deklarovaných v konkretních blocích. Struktura **prvku zásobníku** zahrnuje "hloubku"konkretního bloku a **tabulku symbolu** pro proměnné z tohoto bloku. Tabulka, která se nachází v prvku s hloubkou 0, je globální a obsahuje pouze **identifikátory funkcí**; všechny ostatní obsahují **identifikátory proměnných** konkretních bloku v tělech funkcí.

Vstup do nového bloku (začátek if nebo while) znamená inkrementace hloubky, inicializace nové tabulky symbolu a push nového prvku zásobníku. Východ z bloku znamená likvidace aktuální tabulky symbolů a odstranění vrcholu zásobníku.

2.5.2 Tabulka symbolů

Tabulka symbolů je implementována jako binární vyhledávací strom.Tabulka symbolů obsahuje všechny potřebné informace o **identifikátorech** ve vybraném bloku. Každý prvek tabulky symbolů obsahuje informace o **identifikátoru** (struktura `Item_data`), vlastní klíč pro vyhledávání a ukazately na dva další prvky. Funkce pro práci s zásobníkem tabulek symbolů a s samotnými tabulkami jsou implementovány v souboru `symtable.c`, struktury zásobníku a tabulky symbolů jsou v `symtable.h`.

2.5.3 Fronta

Fronta je implementována jako seznam prvků. Slouží jako pomocná struktura při práci s příkazem přiřazení. Jedíným elementem struktury prvku fronty (kromě ukazatelu na příští prvek) jsou data o konkretním **identifikátoru**. **Fronta** plní se prvky, které se nachází na levé straně od rovnítka. Vyprázdní se během zpracování všeho, co je na pravé straně od rovnítka (každý element fronty koresponduje jednomu **výrazu** nebo jedné z návratových hodnot funkce). Pokud není fronta prázdná po zpracování příkazu přiřazení program hlasí chybu 7 (ostatní sémantické chyby). **Fronta** je implementována v souboru `parser.h`.

2.5.4 Dynamický řetězec

Dynamický řetězec je pomocná strukturá, která se skládá z třech elementů - rozměru (velikost přidelené dynamické paměti), délky (počet symbolů v řetězci) a samotného řetězcu. Struktura se používá ve všech případech, kdy předem neznáme počet znaků v řetězci: při lexikální analýzě, generování kódu, definice parametrů/návratových hodnot funkce. Struktura a funkce pro práci s ní jsou implementovány v souborech `string_processor.h` a `string_processor.c`.

2.5.5 Zásobník symbolů

Zásobník symbolů je implementován jako seznam prvků. Každý prvek zásobníku obsahuje symbol pro rozhodnutí o následující operace v rámci vypracování výrazu a datový typ pro sémantickou analýzu, také obsahuje booleovskou proměnnou `isZero` pro nalezení výrazů, ve kterých dochází k dělení 0. Navíc k základním operacím se zásobníkem jsou implementovány operace pro uložení prvku před prvním terminálem a operace pro získání typu/symbolu elementu na vrcholu **zásobníku**. Struktura a funkce pro práci s ní jsou implementovány v souborech `stack.h` a `stack.c`.

3 Práce v týmu

3.1 Rozdělení práce

Bohdan Fesiun		**0%**
Maksim Tikhonov	Lexikální analýza Syntaktická analýza Generování kódu Zpracování výrazů Organizace práce Dokumentace	**60%**
Dmytro Sadovskyi	Tabulka symbolů Generování kódu Zpracování výrazů Dokumentace Testování	**40%**
Eduard Galliamov		**0%**

3.2 Zdůvodnění odchylek od rovnoměrnosti bodů

Dva členy týmu ji opustili.
Špatná komunikace mezi členy týmu na začátku práce.
Nerovnoměrné rozdělení práce

Reference

[1] Alexander Meduna: *Elements of compiler design*. Boca Raton: Auerbach Publications; 2008, ISBN: 1-4200-6323-5