Implementácia prekladača imperatívneho jazyka IFJ23

DOKUMENTÁCIA

tým xbalog06, varianta TRP-izp

Michal Balogh, xbalog06, 38% - vedúci
Michal Cenek, xcenek04, 37%
Tadeáš Zobal, xzobal02, 25%
Dominik Doležal, xdolezal97, 0%

1. Popis implementácie a členenia projektu

Prekladač sme rozdelili do 4 hlavných celkov, na ktorých bolo možné individuálne a súbežne pracovať. V stručnosti si ich predstavíme, a zameriame sa na hlavné problémy, ktoré pri implementácii vznikli:

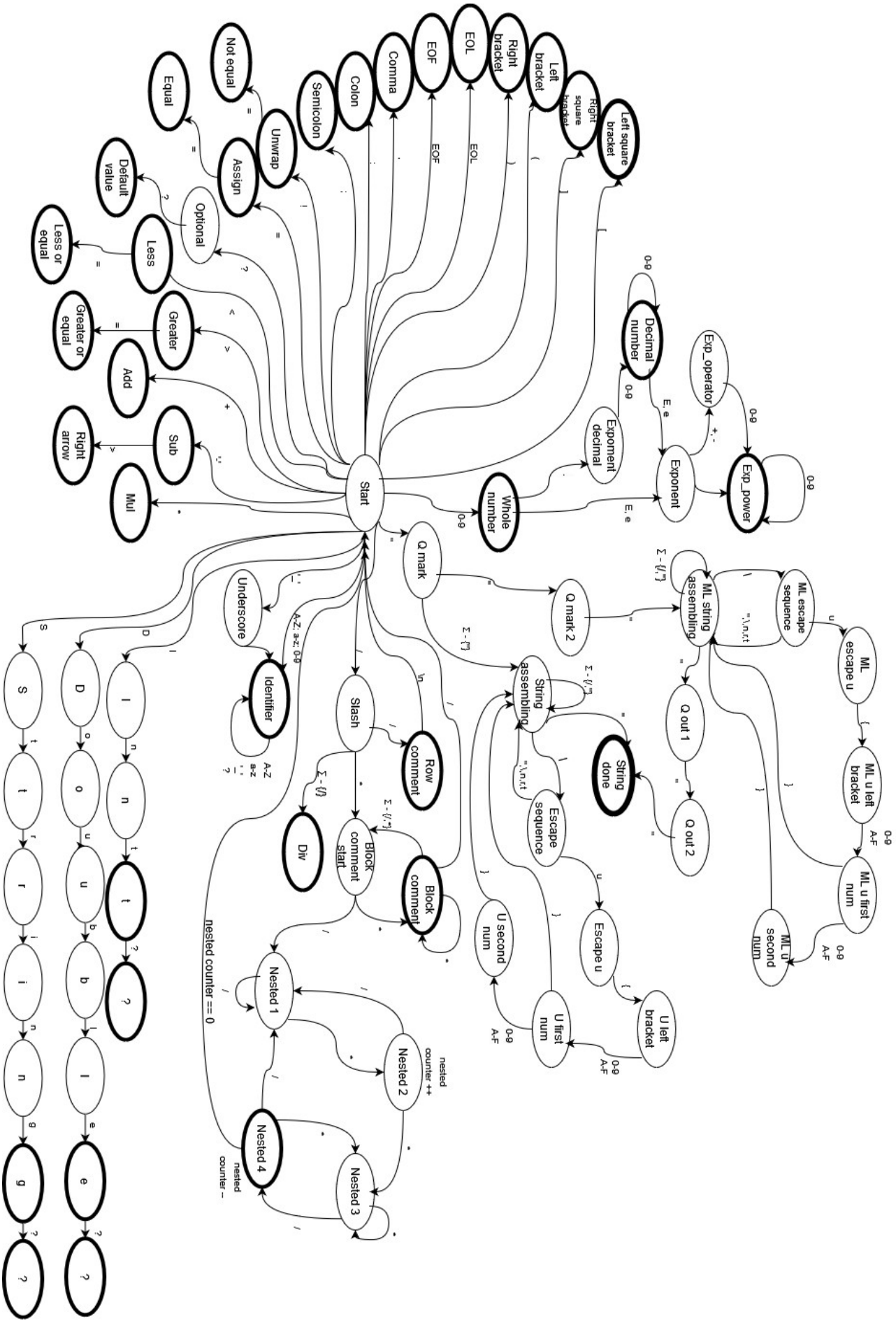
1.1 Lexikálny analyzátor

- súbor: scanner.c

Lexikálny analyzátor, alebo aj skener, je časť prekladača, ktorá interaguje priamo so zdrojovým kódom daného jazyka. Číta vstupný súbor znak po znaku a prostredníctvom najdôležitejšej funkcie celého súboru - get_next_token(), vytvára tzv. tokeny. Tieto sú definované štruktúrou, ktorá obsahuje typ a hodnotu tokenu. Medzi typy tokenov patria napríklad identifikátory, celé a desatinné čísla, kľúčové slová, reťazce, rôzne aritmetické operátory a iné znaky, ktoré existujú v jazyku IFJ23. Hodnota tokenu je definovaná pomocou dátovej štruktúry Union, ktorá obsahuje tri základné dátové typy: Int, Float a String. Tieto tokeny môžu obsahovať hodnotu, konkrétne identifikátory a reťazce s hodnotou dátového typu String, a celé a desatinné čísla s hodnotou dátového typu Int alebo Float. Ďalšou dôležitou funkciou súboru scanner.c je funkcia keyword_check(), ktorá kontroluje, či vstupný identifikátor nie je jedno z kľúčových slov jazyka IFJ23.

Lexikálny analyzátor bol implementovaný ako deterministický konečný automat. Pred vytvorením automatu bol zostrojený diagram celého automatu, aby jeho implementácia bola jasnejšia a nevznikali zbytočné problémy. Automat sa skladá z nekonečného cyklu while s jedným veľkým switchom, kde každý switch case zodpovedá danému stavu automatu. Ak príde znak, pre ktorý neexistuje žiadny switch case, automat prechádza do stavu lexikálnej chyby. Automat pracuje tzv. "žravým" spôsobom, čo znamená, že spracúva znaky dovtedy, kým môže. Vytvorením takéhoto tokenu funkcia get_next_token() končí.

Ostatné funkcie súboru scanner.c, ktoré ešte neboli spomenuté, slúžia na spracovanie dynamického poľa znakov. Keďže identifikátory a ani reťaze nemajú v jazyku IFJ23 obmedzenú veľkosť, pamäť na ukladanie týchto reťazcov musí byť alokovaná dynamicky. Toto pole sa využíva pri načítavaní hodnoty identifikátorov a čísel. Poľu je na začiatku priradená pevná základná veľkosť a súčasťou funkcie, slúžiacej na pridávanie znakov, je aj automatická kontrola stavu poľa a jeho prípadná realokácia v pamäti v prípade potreby zväčšenia.



Konečný automat, pomocou ktorého bola implementovaná lexikálna analýza. Koncové stavy sú označené hrubou čiarou okolo oválu. Z pravých spodných vetiev automatu Int?, Double? a String? vedú z každého stavu else vetvy do stavu Identifier, ktoré kvoli prehľadnosti nie sú uvdené.

1.2 Syntaktická analýza

Syntaktická analýza je rozdelená na dve časti:

- rekurzívny parser: synt_recursive_parser.c, synt_recur_rules.c
  - slúži na spracovanie všetkého okrem výrazov
- precedenčný parser: synt_precedence_parser.c, synt_prec_rules.c
  - slúži na spracovanie výrazov

Spustenie celého programu začína v súbore main.c volaním rekurzívneho parsera, ktorý získava ďalšie tokeny zo scanneru prostredníctvom funkcie get_next_token(). Rekurzívny parser kontroluje správnosť programu pomocou vytvorenej LL gramatiky, kde každé pravidlo reprezentuje vlastnú funkciu, ktorá vracia pravdivostnú hodnotu. Keď narazí na potenciálny začiatok výrazu, nasleduje nasledujúci postup:

- ak prvý token potenciálneho výrazu je konštanta, rekurzívny parser prechádza do precedenčného módu, ktorý slúži na kontrolu správnosti výrazov (v LL-gramatike sa toto prepnutie zobrazuje zmenou pravidla na => expression)
- ak však prvý token potenciálneho výrazu je identifikátor, vzniká nejednoznačnosť, pretože identifikátor môže byť súčasťou výrazu alebo môže predstavovať volanie funkcie - id alebo id()

Tento problém sme vyriešili tým, že sme prvý token uložili do globálnej premennej "stash" a pozreli sme sa na nasledujúci token. Na základe toho sme sa rozhodli, či je daný identifikátor súčasťou výrazu alebo predstavuje volanie funkcie.

Ďalší problém, ktorý vznikol počas implementácie syntaktickej analýzy, sa týkal situácie, keď medzi jednotlivými tokenmi mohol byť ľubovoľný počet bielych znakov, vrátane koncov riadkov, ktoré sú povinné pre ukončenie jednotlivých štruktúr jazyka IFJ23. To znamená, že sme museli niekde ignorovať koncové riadky a inde ich naopak zaznamenať. Túto situáciu sme riešili pomocou pomocnej funkcie consume_optional_EOL, ktorá odstraňuje všetky koncové riadky, kým nenarazí na iný token ako koniec riadku. Táto funkcia nastavuje globálny príznak EOL_flag na hodnotu True, ak odstránila aspoň jeden koniec riadku; v opačnom prípade nastaví príznak na False. Tento príznak slúži ako kontrola tam, kde je potrebný koniec riadku - napríklad na oddelenie dvoch príkazov alebo výrazov.

Precedenčná syntaktická analýza je implementovaná pomocou precedenčnej tabuľky, ktorá je zjednodušená tým, že niektoré operátory majú rovnaké precedencie; napríklad + a - sú v rovnakom políčku, rovnako ako všetky relačné operátory. Z tohto dôvodu sme implementovali aj pomocnú funkciu na určenie indexu v tabuľke, napríklad + a - musia mať rovnaký index v tabuľke.

Oproti implementácii preberanej v predmete IFJ je rozdiel aj v ukončovacej podmienke precdenčnej analýzy. Namiesto kontroly konca výrazu – čo je v jazyku

IFJ23 koniec riadku, kontroluje či daný token môže byť súčasťou výrazu. Ide zase o rovnaký problém, ktorý vznikol aj pri rekurzívnom parseri – koniec riadku sa môže vyskytovať pred aj za ľubovoľnými tokenmi. Zásobník pre precedenčnú analýzu má tiež pár špeciálnych vlastností, aby sa s ním lepšie pracovalo pri spracovaní výrazov. Oproti normálnemu zásobníku môže vracať aj druhý a tretí prvok zhora – čo sa využíva pri redukovaní zásobníku napríklad pomocou pravidla „*operand operátor operand*“.

LL-gramatika

1. <program> -> <stat_list> EOF

2. <stat_list> -> <statement> EOL <stat_list>
3. <stat_list> -> EPSILON

4. <statement> -> <let_or_var> <var_assignment>
5. <statement> -> id <after_id>
6. <statement> -> func id (<param_list>) <return_type> { <func_stat_list> }
7. <statement> -> if <condition> { <brack_stat_list> } else { <brack_stat_list> }
8. <statement> -> while <expression> { <brack_stat_list> }
9. <statement> -> EPSILON

10. <brack_stat_list> -> <statement> EOL <brack_stat_list>
11. <brack_stat_list> -> EPSILON

12. <brack_statement> -> <let_or_var> <var_assignment>
13. <brack_statement> -> id <after_id>
14. <brack_statement> -> if <condition> { <brack_stat_list> } else { <brack_stat_list> }
15. <brack_statement> -> while <expression> { <brack_stat_list> }
16. <brack_statement> -> EPSILON

17. <let_or_var> -> let id
18. <let_or_var> -> var id

19. <var_assignment> -> : type <val_assigment>
20. <var_assignment> -> = id <fn_or_exp>
21. <var_assignment> -> = const <expression>

22. <val_assignment> -> = id <fn_or_exp>
23. <var_assignment> -> = const <expression>
24. <val_assignment> -> EPSILON

25. <fn_or_exp> -> <expression>
26. <fn_or_exp> -> (<input_param_list>)

27. <after_id> -> = id <fn_or_exp>
28. <after_id> -> = const <expression>
29. <after_id> -> (<input_param_list>)

30. <input_param_list> -> <input_param> <input_param_next>
31. <input_param_list> -> EPSILON

32. <input_param_next> -> , <input_param> <input_param_next>
33. <input_param_next> -> EPSILON

34. <input_param> -> id <with_name>
35. <input_param> -> const

36. <with_name> -> : <id_or_const>
37. <with_name> -> EPSILON

38. <id_or_const> -> id
39. <id_or_const> -> const

```
40. <param_list> -> <param> <param_next>
41. <param_list> -> EPSILON

42. <param> -> <id_or_underscore> id : <type>

43. <id_or_underscore> -> _
44. <id_or_underscore> -> id

45. <param_next> -> , <param> <param_next>
46. <param_next> -> EPSILON

47. <return_type> => -> <type>
48. <return_type> => EPSILON

49. <func_stat_list> -> <func_stat> EOL <func_stat_list>
50. <func_stat_list> -> EPSILON

51. <func_stat> -> <let_or_var> <var_assignment>
52. <func_stat> -> id <after_id>
53. <func_stat> -> return <ret_val> EOL <func_stat_list>
54. <func_stat> -> if <condition> { <func_stat_list> } else { <func_stat_list> }
55. <func_stat> -> while <expression> { <func_stat_list> }
56. <func_stat> -> EPSILON

57. <ret_val> -> <expression>
58. <ret_val> -> EPSILON

59. <condition> -> <expression>
60. <condtion> -> let id

62. <type> -> Int
63. <type> -> Int?
64. <type> -> Double
65. <type> -> Double?
66. <type> -> String
67. <type> -> String?

68. <expression> -> id
69. <expression> -> const
70. <expression> -> ( <expression> )
71. <expression> -> <expression> arithm_op <expression>
72. <expression> -> <expression> ?? <expression>
73. <expression> -> <expression>!  (force unwrap)
74. <expression> -> <expression> rel_op <expression>
```

- <expression> značí prepnutie do precedenčného parseru.
- <statement>, <brack_statement> a <func_stat> sú skoro rovnaké, no potrebujú vlasnté pravidlá, ináč by bola gramatika nejednoznačná – <brack_statement> môže byť oproti obyčajnému <statement> ukončený aj znakom „}“ a vo funkcii môžeme mať klúčové slovo return, kdežto inde nie.
- V kóde by sa dala táto problematika vyriešiť pridaním nastavovania príznakov, či sme medzi zátvorkami, v hlavnom tele programu alebo vo funkcii.

LL-tabuľka

	EOL	let	var	if	while	id	const	func	return	(	)	{	}	,	->	=	_	:	EOF
program	1	1	1	1	1	1		1											1
stat_list	3	2	2	2	2	2		2											3
statement	9	4	4	7	8	5		6											9
brack_stat_list	11	10	10	10	10	10		10					11						
brack_statement	16	12	12	14	15	13							16						
let_or_var		17	18																
var_assignment						20	21											19	
val_assignment	24					22	23						24						24
fn_or_exp	25									26			25						25
after_id						27.1	28.1			29						27,28			
input_param_list						30					31								
input_param_next											33			32					
input_param						34	35												
with_name											37			37				36	
id_or_const						38	39												
param_list						40					41						40		
param						42											42		
id_or_underscore						43											44		
param_next											46			45					
return_type												48			47				
func_stat_list	50	49	49	49	49	49			49				50						
func_stat	56	51	51	54	55	52			53				56						
ret_val	58					57	57			57			58						
condition		60				59	59			59									

Poznámka: pravidlá 27 a 28 sú v podstate jedno pravidlo, najprv sa skontroluje znak „=“ a potom sa ďalšie pravidlo volí podľa ďalšieho tokenu – identifikátor alebo konštanta

Precedenčná tabuľka:

- **REL** značí všetky relačné operátory
- **i** značí identifikátory a konštanty

	+ -	* /	??	!	REL	(	)	i	$
+ -	>	<	>	<	>	<	>	<	>
* /	>	>	>	<	>	<	>	<	>
??	<	<	<	<	<	<	>	<	>
!	>	>	>		>		>		>
REL	<	<	>	<		<	>	<	>
(	<	<	<	<	<	<	=	<	
)	>	>	>	>	>		>		>
i	>	>	>	>	>		>		>
$	<	<	<	<	<	<		<	

1.3 Sémantická analýza

- súbor: expr.c, symtable.c, istack.c semantic_analysis.c symtablestack.c

Funkcie sémantickej analýzy sú volané pri simulácii generovania derivačného stromu. Uchováva v pomocných premenných informácie o relevantných uzloch abstraktného syntaktického stromu a akonáhle má dostatok informácií, vykoná sémantickú akciu. Po kontrole sémantickej správnosti aktuálneho uzla abstraktného syntaktického stromu prenáša riadenie generátoru cieľového kódu.

Informácie o type a inicializácii premenných a o návratovej hodnote a parametroch funkcií sú ukladané do tabuľky symbolov, ktorá je implementovaná ako tabuľka s rozptýlenými položkami. Z dôvodu podpory zatienenia premenných sa používa zásobník tabuliek symbolov, pričom pri vstupe do nového bloku sa na zásobník pridá nová tabuľka a pri výstupe z bloku sa z vrcholu zásobníka tabuľka odoberá.

1.4 Generovanie kódu

- súbor: generator.c

Akonáhle má sémantická analýza všetky informácie o sémantickej akcii a potvrdila ich správnosť, zavolá príslušnú funkciu modulu generátor, ktorá pre sémantickú akciu vygeneruje kód.

Návratová hodnota funkcie je prenášaná cez zásobník.

Každá tabuľka symbolov v zásobníku tabuliek symbolov má unikátne číslo, ktoré je pridané k názvu premennej pri generovaní jej identifikátora v kóde kvôli rozlíšeniu zatienených premenných.

Generovanie výrazu silne spolupracuje so sémantickou analýzou, ktorá kontroluje výraz v postfixovej notácii. Operandy priradí na vrchol zásobníka a akonáhle narazí na operátor, vyjme zo zásobníka príslušný počet operandov a ak je operácia sémanticky správna, zavolá funkciu generátora, ktorá pre ňu vygeneruje kód. Medzivýsledky sú ukladané na zásobník. Výsledok výrazu je po jeho vyhodnotení tiež na vrchole zásobníka.

Varianta TRP-IZP

Na výber boli dve varianty implementácie tabuľky symbolov. My sme si zvolili implementáciu pomocou tabuľky s rozptýlenými položkami – konkrétne s implicitným zreťazením položiek. Pomocou hashovacej funkcie sa získa hash vkladaného (alebo hľadaného) prvku, a ten slúži ako index do tabuľky. Pri kolízii sa index posunie na najbližšie miesto vpravo od pôvodného hashu. Tabuľka je implementovaná pomocou poľa, a keďže dopredu nepoznáme počet prvkov, ktoré budeme chcieť do tabuľky vkladať, tabuľka je implementovaná pomocou nafukovacieho pola.

Práca v týme:

S projektom sme začali hneď, ako bolo zverejnené zadanie. Na začiatku dostal každý za úlohu prečítať si zadanie a rozhodnúť sa, ktorú časť prekladača by chcel robiť. Keď sme si podelili projekt na časti, každý si začal študovať tú svoju, predovšetkým prostredníctvom samoštúdia a pozerania prednášok z minulých rokov.

Spoločný kód sme mali uložený na GitHube, kde sme vytvorili aj projekt, ktorý slúžil na priraďovanie úloh, sledovanie míľnikov a poskytoval nám celkový prehľad o stave projektu. Každý si vytvoril svoju vetvu na GitHube, ktorá vždy musela najprv prejsť kontrolou od nejakého člena tímu predtým, ako bola spojená do hlavnej vývojovej vetvy. V tejto fáze projektu sme funkčnosť kódu kontrolovali pomocou jednotkových testov.

V neskorších fázach projektu, keď sme už jednotlivé moduly spájali dohromady, sme mali pravidelné schôdze, kde sme prediskutovali problémy a potrebné zmeny. V tejto fáze sme vytvorili aj integračné testy, aby sme overili správne fungovanie vzájomnej spolupráce jednotlivých modulov.

Na záver boli vytvorené ešte testy na overenie správneho generovania kódu.

Rozdelenie práce:

Michal Balogh:

- štruktúra projektu, riadenie projektu, návrh a implementácia syntaktickej analýzy, LL-gramatika, LL-tabuľka, precedenčná tabuľka, implementácia tabuľky symbolov, testovanie, dokumentácia, prezentácia

Tadeáš Zobal:

- lexikálna analýza, dokumentácia, dynamický reťazec, konečný automat

Michal Cenek:

- implementácia sémantickej analýzy, generovanie kódu, časti tabuľky symbolov a dátových štruktúr, ktoré sú použité v sémantickej analýze a v generovaní kódu, lineárny zoznam, zásobník, testovanie

Odchylky od rovnomerného rozdelenia bodov sú dané množstvom vykonanej práce a času stráveného pracovaním na projekte. Keďže sa Dominik Doležal nezúčastnil na riešení projektu, jeho prácu sme ohodnotili na 0%.

Zdroje:

- Prednášky a democvičenia: https://www.fit.vutbr.cz/study/courses/IFJ/public/materials/ https://www.fit.vutbr.cz/study/courses/IFJ/public/project/
- FIT VUT logo: https://www.overleaf.com/edu/but#templates
- Hashovacia funkcia: https://craftinginterpreters.com/