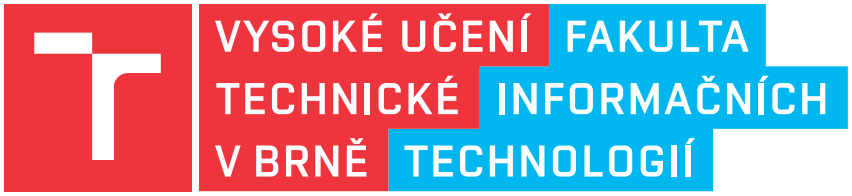


Projektová dokumentácia
Implementácia prekladača imperatívneho jazyka IFJ23

6. decembra 2023

Tím xkolia00, Varianta - vv-BVS
FUNEXP

Autori:

Nikita Koliada - vedúci tímu - 29%	xkolia00@stud.fit.vutbr.cz
Pavlo Butenko - 29%	xbuten00@stud.fit.vutbr.cz
Maksym Podhornyi - 29%	xpodho08@stud.fit.vutbr.cz
Juraj Remeň - 13%	xremen02@stud.fit.vutbr.cz

Obsah

1 Úvod

Cieľom projektu je vytvorenie prekladača imperatívneho jazyka IFJ23 v jazyku C, ktorý načíta zdrojový kód napísaný v danom jazyku IFJ23. Jazyk IFJ23 je podmnožinou jazyka SWIFT a prekladač ho prekladá do cieľového jazyka IFJcode23.

Jedná sa o konzolovú aplikáciu, ktorá načíta zdrojový program zo štandardného vstupu a generuje výsledná medzikód na štandardný výstup. V prípade akejkoľvek chyby vracia odpovedajúci chybový kód.

2 Implemantácia

2.1 Časti prekladača

- lexikálna analýza
- syntaktická analýza
- semantická analýza
- generátor kódu

2.2 Lexikálna analýza

Lexikálna analýza sa opiera o štruktúru `token`, v ktorej sa ukladajú všetky dôležité informácie o tokene, ako napr. aké dáta obsahuje, aký je typ tokeu a na akom riadku sa nachádza token v zdrojovom kóde.

Hlavná funkcia lexikálneho analyzátora je `get_next_token()` - obsahuje `switch`, ktorého vetvenie slúži ako konečný automat, na ktorom je lexikálna analýza postavená. Funkcia vracia 0 alebo adekvátnu chybu, ak ju zdetekuje. Ukladá všetky zistené informácie do štruktúry `token`, ktorý používa ako svoj argument.

V súbore `scanner.c` sú aj ďalšie pomocné funkcie, správne identifikujúce prichádzajúce tokeny.

2.3 Syntaktická analýza

Syntaktická analýza vychádzala z LL gramatiky a jej tabuľky, podľa ktorých sa vytvorili všetky pravidlá a funkcie pre kontrolu syntaktickej správnosti kódu. Pre správny chod tejto súčasti sme potrebovali aj tabuľku symbolov, ktorá je implementovaná ako výškovo vyvážený binárny vyhľadávací strom.

Súbor`analysis.h` obsahuje deklarované funkcie a štruktúry, ktoré boli nutné k ýápisu hlavnej časti syntaktickej analýzy v súbore `analysis.c`, v ktorom som vytvoril makrá zabezpečujúce jednoduchosť a čitateľnosť kódu.

Funkcie v tomto súbore sú zodpovedné za pravidla, ktoré vychádzajú z LL gramatiky a jej tabuľky. Napríklad Funkcia `function(analyse_data_t* data)` - kontroluje či je syntax funkcie zo zdrojového kódu zapísaná správne. Kontroluje takisto jej argumenty, návratový typ a pridáva všetky hodnoty, teda typ, názov a identifikátory, do jednoduchej tabuľky vychádzajúcej z tabuľky symbolov.

Ďalšia funkcia `def_var(analyse_data_t* data)` - definuje premenné, ktoré prečíta zo zdrojového kódu. Má za úlohu skontrolovať syntakticky správny tvar premennej a tiež pridáva modifikátor `let` alebo `var`, typ, ktorý je nepovinný a volá funkcie zodpovedné za sémantickú analýzu a kontrolu v súbore `expression.c`.

2.4 Sémantická analýza

Sémantická pozostavá zo súborov `expression.c`, `expression.h`, `stack.c`, `stack.h`. Jedná sa o časť analyzátoru výrazov, ktorá je založená na precedenčnej sémantickej analýze, vychádzajúcej z precedenčnej tabuľky. Táto časť analyzuje typy hodnôt, zároveň porovnáva správnu syntax pri volaniach funkcie `call()`, definície premenných, priradeniu hodnoty k premennej alebo výrazu a podmienkach pri konštrukciách `if` a `while`.

Kontroluje sa pomocou zásobníku, na základe precedenčnej tabuľky, založenej na vstupnom symbole a terminále, nachádzajúcom sa na vrchole zásobníku. Môžné operácie v analýze sú `shift()`, `reduce()`, `failure()` alebo `equal()`.

Pre implementáciu rozšírenia FUNEXP jsme pridali funkciu či operáciu `function()` - spustí LL analýzu, `shift()` - vezme ďalší symbol zo vstupu, `reduce()` - príjma pravidlá a spracováva sémantiku, `equal()` - číta ďalší znak a `failure()` - skúma či je terminál na vrchole zásobníka $ (temrinál dolár), ak áo tak je analýza úspešná, ak nie nastáva chyba a program končí s adekvátnym chybovým kódom.

2.5 Generátor kódu

Generátor kódu je v našom prípade časť prekladača, ktorá je volaná ostatnými časťami, v ktorých je kontrolovaná syntax aj sémantika zdorjového kódu.

Sú v ňom implementované všetky vstavané funkcie zo zadania a niektoré z nich sú implementované pomocou trojadresného kódu, ako napríklad funckie `generate_readString()` a aj ostatné `generate_read()` funkcie - ktoré načítavajú hodnoty zo vstupu a vypíšu ju na výstup.Alebo funkcia `generate_substring()` - ktorá zisťuje či je vstupný reťazec podreťazdom druhého vstupného reťazca a ďalšie. Takisto aj pár funkcií, implementovaných ako zásobníkové funkcie ako napr. `gen_int2double()` - má za úlohu prevod reálneho čísla na celé číslo, odstránením desatinnej časti vstupného čísla a `gen_operation()` - generuje všetky operácie pre matematiku a aritmetiku. Všetky vstavané funkcie sú potom definované v súbore `generator.c` vo funkcii `define_built_in_functions()` a volané v syntaktickom aj sémantickom analyzátore.

Aj ostatné konštrukcie zdrojového programu sú generované v súboroch `generator.*`. Jedná sa o konštrukcie `if`, `else`, `while`, `call` a `variable`. Všetký konštrukcie sú opäť volané v syntaktickom a sémantickom analyzátore s potrebnými overeniami.

2.6 Tabuľka symbolov

Tabuľka symbolov je implementovaná ako výškovo vyvážený binárny vyhľadávací strom. Dané funkcie využívajú základy z predmetu IAL, z jednej z domácich úloh. Na začiatku sú implemntované funkcie pre BVS, ktoré sú rekurzívne volané vo funkciách na prácu s tabuľkou symbolov.

Na zabezpečenie vyškovej vyváženosti BVS používame funkcie:

- `int height(bst_node_ptr N)` - vracia výšku stromu
- `int get_balance(bst_node_ptr N)` - vypočíta výšku stromu rozdielom oboch ukazovateľov na strom
- `int right_rotate(bst_node_ptr N)` - otočí strom vpravo čím ho vyváži, vypočíta výšku a vráti údaje o novom koreni stromu
- `int left_rotate(bst_node_ptr N)` - otočí strom vľavo čím ho vyváži, vypočíta výšku a vráti údaje o novom koreni stromu

3 Práca v tíme

3.1 Verzovací systém a iné projektové nástroje

Na správu verzií sme použili verzovací systém Git, náš vzdialený repozitár bol umiestnený na GitHube.

Kvôli prehľadnosti sme používali takisto portál Trello, kde sme pravidelne prispievali na našu tabuľu a mali tak prehľad o tom kto čo robí a čo mu ešte treba spraviť.

Vďaka obom nástrojom sme si vyskúšali prácu ako na reálnom tímovom projekte v práci a určite nám to pomôže v budúcom živote. V podstate sme si odsimulovali taký takmer dvoj mesačný Sprint v Agile projektovom manažmente.

3.2 Komunikácia v tíme

Na komunikáciu sme využívali primárne aplikáciu Discord, kde sme na serveri viedli diskusie a takisto mali pravidelné hovory. Pár krát sme sa stretli aj osobne, v prípade, že sme chceli vyriešiť nejaký väčší problém.

3.3 Deľba práce v tíme

- **Nikita Koliada** – organizácia, syntaktická analýza, sémantická analýza, tabuľka symbolov, testovanie
- Pavlo Butenko – lexikálna analýza, syntaktická analýza, sémantická analýza
- Maksym Podhornyi – lexikálna analýza, syntaktická analýza, generátor kódu, testovanie
- Juraj Remeň – organizácia, lexikálna analýza, tabuľka symbolov, generátor kódu, dokumentácia, prezentácia

3.4 Bodové rozdiely v rozdelení

Niektorí z nás urobili na projekte o mnoho viac práce ako druhí a tak dostali viac %. Všetci s daným rozdelením súhlasili.

4 Záver

Projekt je jednoznačne jeden z náročnejších, na ktorých sme doteraz pracovali. Začiatok sme trochu podcenili, zabudli sme na registráciu tabuľky symbolov a tak sme museli implementovať BVS namiesto TRP, ktorú mal jeden z členov tímu pripravenú z iného predmetu a žiadala len upravenie pre projekt. Takisto implementácia nešla podľa predstáv, dávali sme si načas a ku koncu nás to dobiehalo.

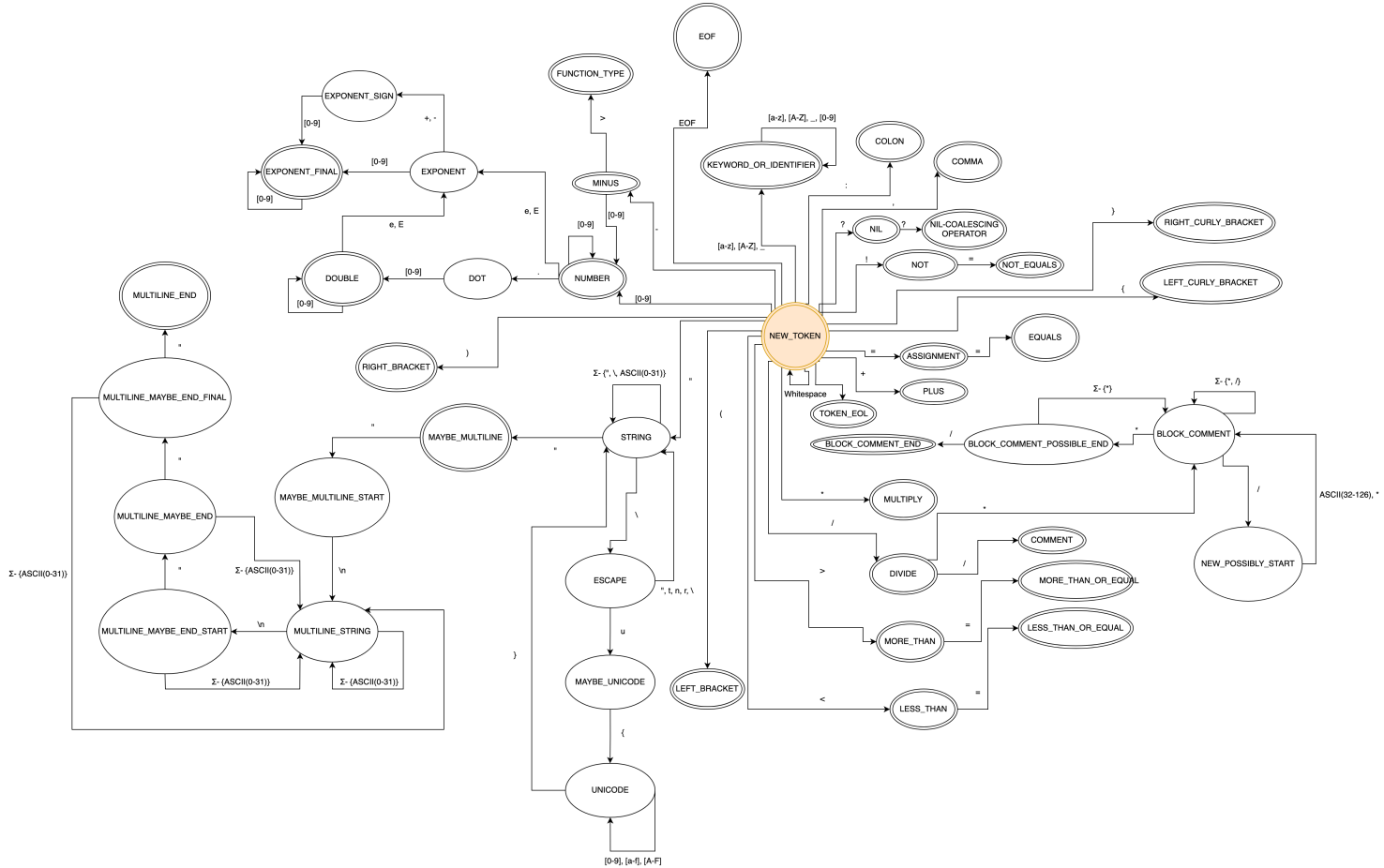
Ale hlavne vďaka prednáškam a pomocným videám na internete z minulých rokov, sme nabrali potrebné tempo a projekt dokončili v stanovenom čase. Pokusné odovzdani nám ukázali, koľko nám ešte chýba a dokončili to ako sa patrí.

Jednotlivé časti sme riešili skôr indivudálne s konzultáciou. Na začiatku projektu sme si rozdelili prácu, pracovali samostatne a v prípade akéhokoľvek problému, obrátili na kolegov.

Implementácia projektu nám priniesla mnoho nových znalostí, ktoré využijeme nie len v programovaní ale takisto v práci v tíme alebo dokonca aj vedenia tímu.

5 Prílohy

5.1 Diagram konečného automatu



Obr. 1: Diagram konečného automatu

5.2 Precedenčná tabuľka

	+-	*/	(	)	r	??	!	id	$
+-	>	<	<	>	>	>	<	<	>
*/	<	>	<	>	>	>	<	<	>
(	<	<	<	=	<	<	<	<	Failure
)	>	>	Failure	>	>	>	<	>	>
r	<	<	<	>	Failure	>	<	<	>
??	<	<	<	>	<	<	<	<	>
!	>	>	<	>	>	>	Failure	<	>
id	>	>	Function	>	>	>	>	Failure	>
$	<	<	<	Failure	<	<	<	<	Failure

Obr. 2: Precedenčná tabuľka

5.3 LL - gramatika

1. ⟨ program ⟩ ⟶ ⟨ statement ⟩
2. ⟨ statement ⟩ ⟶ ⟨ function ⟩EOL ⟨ statement ⟩
3. ⟨ statement ⟩ ⟶ ⟨ if_else ⟩EOL ⟨ statement ⟩
4. ⟨ statement ⟩ ⟶ ⟨ while ⟩EOL ⟨ statement ⟩
5. ⟨ statement ⟩ ⟶ ⟨ assignment ⟩EOL ⟨ statement ⟩
6. ⟨ statement ⟩ ⟶ ⟨def_var ⟩ EOL ⟨ statement ⟩
7. ⟨ statement ⟩ ⟶ ⟨ f_call ⟩EOL ⟨ statement ⟩
8. ⟨ statement ⟩ ⟶ ⟨ return_kw ⟩⟨ expression ⟩EOL ⟨ statement ⟩
9. ⟨ statement ⟩ ⟶ EOL ⟨ statement ⟩
10. ⟨ statement ⟩ ⟶ ⟨ end ⟩
11. ⟨ statement ⟩ ⟶ ε
12. ⟨ function ⟩ ⟶ func ⟨ ID ⟩(⟨ args ⟩) ⟨ func_ret ⟩{ EOL ⟨ statement ⟩ }
13. ⟨ func_ret ⟩ ⟶ ⟨ FUNCTION_TYPE ⟩⟨ type ⟩
14. ⟨ func_ret ⟩ ⟶ ε
15. ⟨ args ⟩ ⟶ id id : ⟨type ⟩⟨args_n ⟩
16. ⟨ args ⟩ ⟶ ε
17. ⟨ args_n ⟩ ⟶ , id id : ⟨type ⟩⟨args_n ⟩
18. ⟨ args_n ⟩ ⟶ ε
19. ⟨ if_else ⟩ ⟶ if ⟨ expression ⟩{ ⟨ statement ⟩} ⟨ possible_EOL ⟩else { ⟨ statement ⟩}
20. ⟨ while ⟩ ⟶ while ⟨ expression ⟩{ ⟨ statement ⟩}
21. ⟨ assignment ⟩ ⟶ ⟨ id ⟩ = ⟨ expression ⟩
22. ⟨ def_var ⟩ ⟶ ⟨ modifier ⟩⟨ id ⟩⟨ def_var_body ⟩
23. ⟨ def_var_body ⟩⟶ ⟨ def_type ⟩= ⟨expression ⟩
24. ⟨ def_var_body ⟩⟶ id :⟨type ⟩
25. ⟨ write ⟩ ⟶ write (⟨ expression ⟩, . . .)
26. ⟨f_call ⟩⟶ id (⟨fc_args ⟩)
27. ⟨ fc_args ⟩⟶ id: expression ⟨fc_ n_args ⟩
28. ⟨fc_args ⟩⟶ ε
29. ⟨fc_n_args ⟩⟶ , id: expression ⟨fc_ n_args ⟩
30. ⟨fc_n_args ⟩⟶ ε

31. ⟨ modifier ⟩⟶ let
32. ⟨ modifier ⟩⟶ var
33. ⟨ def_type ⟩⟶ :⟨type ⟩
34. ⟨def_type ⟩⟶ ε
35. ⟨ end ⟩ ⟶ EOF
36. ⟨ possible_EOL ⟩ ⟶ EOL
37. ⟨ possible_EOL ⟩ ⟶ ε
38. ⟨ id ⟩ ⟶ identifier
39. ⟨ type ⟩ ⟶ ⟨ p_type ⟩
40. ⟨ p_type ⟩ ⟶ Double
41. ⟨ p_type ⟩ ⟶ String
42. ⟨ p_type ⟩ ⟶ Int
43. ⟨ p_type ⟩ ⟶ StringNullable
44. ⟨ p_type ⟩ ⟶ DoubleNullable
45. ⟨ p_type ⟩ ⟶ IntNullable

5.4 Tabuľka LL - gramatiky

	EOL	⟨return_kw⟩	⟨expression⟩	ε	func	⟨FUNCTION_TYPE⟩	id	:	,	if	while	write	let	var	EOF	identifier	Double	String	Int	StringNullable	DoubleNullable	IntNullable
⟨program⟩	1	1		1	1		1			1	1		1	1	1	1						
⟨statement⟩	9	8		11	2		7			3	4		6	6	10	5						
⟨function⟩					12																	
⟨func_ret⟩				14		13																
⟨args⟩				16			15															
⟨args_n⟩				18					17													
⟨if_else⟩										19												
⟨while⟩											20											
⟨assignment⟩																21						
⟨def_var⟩													22	22								
⟨def_var_body⟩				23			24	23														
⟨write⟩												25										
⟨f_call⟩							26															
⟨fc_args⟩				28			27															
⟨fc_n_args⟩				30					29													
⟨modifier⟩													31	32								
⟨def_type⟩				34				33														
⟨end⟩															35							
⟨possible_EOL⟩	36			37																		
⟨id⟩																38						
⟨type⟩																	39	39	39	39	39	39
⟨p_type⟩																	40	41	42	43	44	45

Obr. 3: Tabuľka LL - gramatiky