Dokumentace k projektu

Implementace překladače jazyka IFJ23

Tým **xkhoda01** – varianta BVS

Dmytro Khodarevskyi	**(xkhoda01)**	**25%**
Yaroslav Hryn	(xhryny00)	25%
Denys Chernenko	(xchern08)	25%
Dmytro Trifonov	(xtrifo00)	25%

06.12.2023

Rozdělení práce a komunikace v týmu

Rozdělení práce mezi členy týmu	
Řízení projektu, lexikální analýza,generování kódu, makefile, sémantická analýza	**xkhoda01**
dokumentace, generování kódu, sémantická analýza, zpracování chyb	xhryny00
sémantická analýza, lexikální analýza, dokumentace, syntaktická analýza, konzultace	xchern08
syntaktická analýza, sémantická analýza, generování kódu, testovani	xtrifo00

Jako tým jsme se zavázali k pravidelným týdenním setkáním, kde jsme diskutovali o pokroku, rozdělovali úkoly a sdíleli zpětnou vazbu. Pro usnadnění spolupráce a efektivity jsme využívali platformy GitHub a Discord. GitHub nám umožnil efektivně spravovat a sledovat změny v kódu, zatímco Discord byl klíčový pro denní komunikaci a rychlé řešení problémů. Tento přístup nám umožnil udržet vysokou úroveň organizace a adaptability, což bylo klíčové pro úspěch našeho projektu.

Obsah

1. Úvod

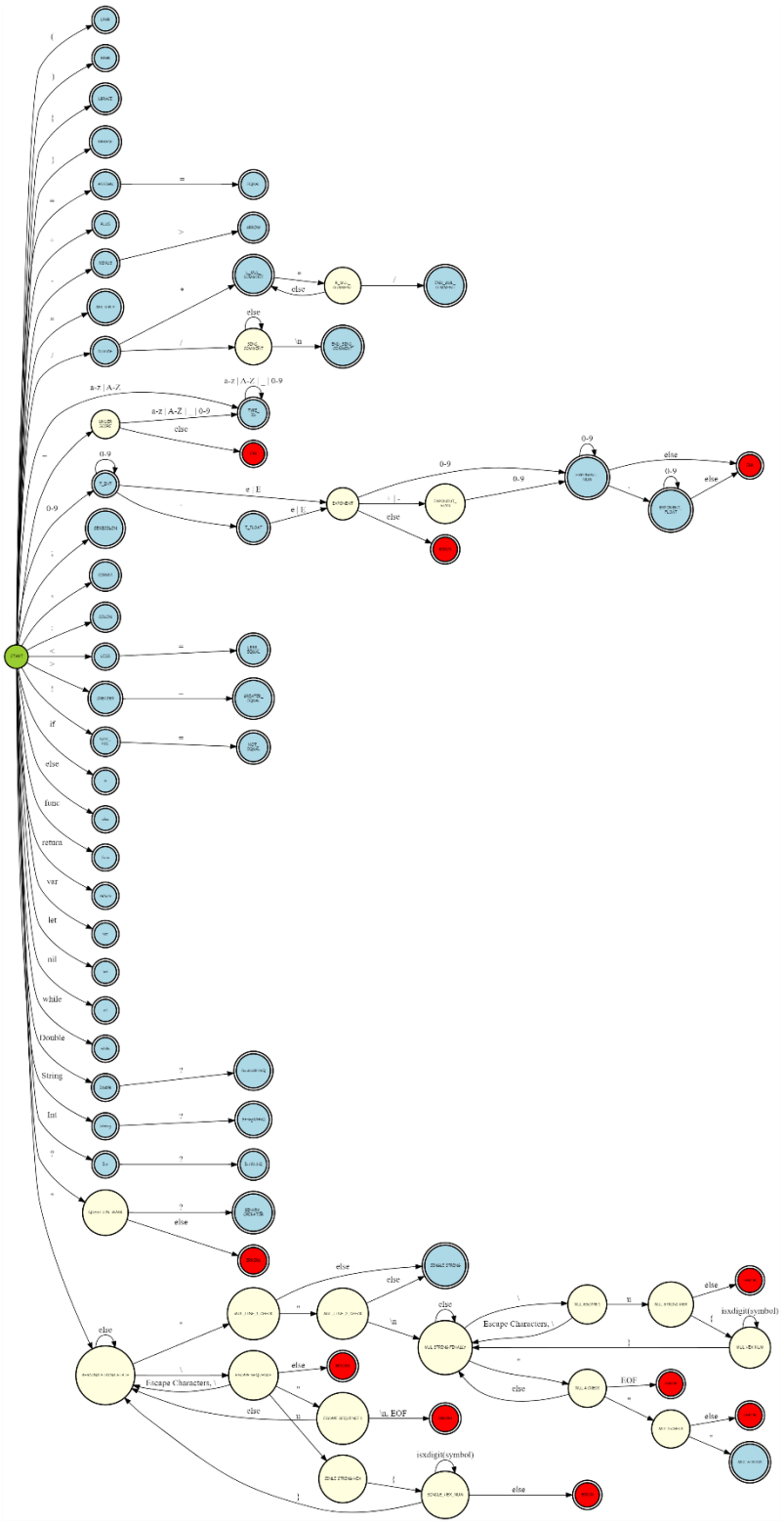
Našim hlavním úkolem bylo vytvořit překladač pro programovací jazyk IFJ23. Tento jazyk IFJ23 je zjednodušenou podmnožinou populárního jazyka Swift. Pro náš projekt jsme se rozhodli zaměřit na variantu BVS.

2. Lexikální analýza

Implementujeme lexikální analyzátor v souborech **tokenizer.c** a **tokenizer.h** . Klíčovou funkcí je **'get_token'**, ve které používáme dynamický řetězec pro čtení znaků ze souboru. Znak po znaku a pomocí 'switch case' určujeme:

- Zda máme vrátit token.
- Zda máme vrátit chybu (1 - lexikální chyba).
- Přidat znak do dynamického řetězce pro další určení tokenu

2.1 Diagram konečného automatu



3. Syntaktická analýza

Pro implementaci syntaktické analýzy jsme vytvořili soubory parse.c a parse.h. Tato část našeho projektu je postavena na principu LL(1) gramatiky, která je známá svou přehledností a efektivitou při analýze syntaxe. Při implementaci využíváme metodu rekurzivního sestupu, což je technika založená na pravidlech LL tabulky, umožňující systematické zpracování programového kódu. Díky integraci s naším tokenizerem (používáme funkce : **get_token()** a **peekNextToken()**), získáváme tokeny. Tyto tokeny pak efektivně používáme k určení, zda je syntaxe analyzovaného programu správná. Tento přístup nám umožňuje detekovat a správně interpretovat různé syntaktické struktury při zpracování zdrojového kódu.

get_token() - *analyzuje a vrátí následující token ze zadaného souboru*

peekNextToken() *- slouží k nahlédnutí následujícího tokenu souboru bez posunutí čtecí pozice ve streamu*

3.1 LL Gramatika

S ::= [PROGRAM] $

[PROGRAM] ::= [DECLARE_GLOBAL_FUNC]

[DECLARE_GLOBAL_FUNC] ::= [STMT_LIST]

[STMT_LIST] ::= [STMT] [STMT_LIST]

[STMT_LIST] ::= ''

[STMT] ::= if [IF_EXP] { [STMT_LIST] } [ELSE]

[STMT] ::= while [WHILE_EXP] { [STMT_LIST] }

[STMT] ::= [KWRD_STMT]

[STMT] ::= [ASSIGN_STMT]

[STMT] ::= [FUNC_DECL]

[STMT] ::= return [EXP]

[FUNC_DECL] ::= func id ([PARAM_LIST]) [RETURN_TYPE] { [STMT_LIST] }

[RETURN_TYPE] ::= -> [TYPE]

[RETURN_TYPE] ::= ''

[WHILE_EXP] ::= ([EXP] [MB_WHILE_EXP])
[MB_WHILE_EXP] ::= [COMP_OP] [EXP]

[MB_WHILE_EXP] ::= ''

[IF_EXP] ::= ([EXP])

[IF_EXP] ::= let id

[ELSE] ::= else { [STMT_LIST] }

[KWRD_STMT] ::= let id [MB_STMT_LET]

[KWRD_STMT] ::= var id [MB_STMT_LET]

[ASSIGN_STMT] ::= id [MB_ASSIGN_STMT]

[MB_ASSIGN_STMT] ::= = [EXP]

[MB_ASSIGN_STMT] ::= [FUNC_CALL]

[MB_ASSIGN_EXPR] ::= = [EXP]

[MB_ASSIGN_EXPR] ::= [FUNC_CALL]

[MB_ASSIGN_EXPR] ::= ''

[MB_STMT_LET] ::= : [TYPE] [MB_ASSIGN_EXPR]

[MB_STMT_LET] ::= = [EXP]

[TYPE] ::= String

[TYPE] ::= Int

[TYPE] ::= Double

[TYPE] ::= String?

[TYPE] ::= Int?

[TYPE] ::= Double?

[COMP_OP] ::= ==

[COMP_OP] ::= !=

[COMP_OP] ::= >

[COMP_OP] ::= >=

[COMP_OP] ::= <

[COMP_OP] ::= <=

[EXP] ::= [T] [EXP_LIST]

[EXP_LIST] ::= + [T] [EXP_LIST]

[EXP_LIST] ::= - [T] [EXP_LIST]

[EXP_LIST] ::= ''

[T] ::= [FAC] [T_LIST]

*[T_LIST] ::= * [FAC] [T_LIST]*

[T_LIST] ::= / [FAC] [T_LIST]

[T_LIST] ::= ''

[FAC] ::= ([EXP])

[FAC] ::= id [FAC_TAIL]

[FAC_TAIL] ::= [FUNC_CALL]

[FAC_TAIL] ::= ''

[FUNC_CALL] ::= ([ARG_LIST])

[ARG_LIST] ::= [ARG] [ARG_LIST_REST]

[ARG_LIST] ::= ''

[ARG_LIST_REST] ::= , [ARG] [ARG_LIST_REST]

[ARG_LIST_REST] ::= ''

[ARG] ::= [PREFIX] : [ARG_NAME]

[PREFIX] ::= ''

[PREFIX] ::= id

[ARG_NAME] ::= id

[PARAM_PREFIX] ::= _

[PARAM_PREFIX] ::= id

[PARAM_LIST] ::= [PARAM] [PARAM_LIST_REST]

[PARAM_LIST] ::= ''

[PARAM_LIST_REST] ::= , [PARAM] [PARAM_LIST_REST]

[PARAM_LIST_REST] ::= ''

[PARAM] ::= [PARAM_PREFIX] [PARAM_NAME] : [TYPE]

3.2 LL-tabulka

	if	var	while	func	let	id	return	$	}	->	{	(	!=	>	<	==	<=	>=	)	else	=	:	String	Int	Double	String?	Int?	Double?	+	-	*	/	,	_
S	1	1	1	1	1	1	1																											
[PROGRAM]	2	2	2	2	2	2	2																											
[DECLARE_GLOBAL_FUNC]	3	3	3	3	3	3	3	3																										
[STMT_LIST]	4	4	4	4	4	4	4	5	5																									
[STMT]	6	8	7	10	8	9	11																											
[FUNC_DECL]				12																														
[RETURN_TYPE]										13	14																							
[WHILE_EXP]												15																						
[MB_WHILE_EXP]													16	16	16	16	16	16	17															
[IF_EXP]					19							18																						
[ELSE]																				20														
[KWRD_STMT]		22			21																													
[ASSIGN_STMT]						23																												
[MB_ASSIGN_STMT]												25									24													
[MB_ASSIGN_EXPR]	##	28	28	28	28	28	28	28	28			27									26														
[MB_STMT_LET]																					30	29												
[TYPE]																							31	32	33	34	35	36						
[COMP_OP]													38	39	41	37	42	40																
[EXP]						43						43																							
[EXP_LIST]	##	46	46	46	46	46	46	46	46				46	46	46	46	46	46	46										44	45				
[T]						47						47																							
[T_LIST]	##	50	50	50	50	50	50	50	50				50	50	50	50	50	50	50										50	50	48	49		
[FAC]						52						51																							
[FAC_TAIL]	##	54	54	54	54	54	54	54	54			53	54	54	54	54	54	54	54										54	54	54	54		
[FUNC_CALL]												55																						
[ARG_LIST]						56													57			56												
[ARG_LIST_REST]																			59														58	
[ARG]						60																60												
[PREFIX]						62																61												
[ARG_NAME]						63																												
[PARAM_PREFIX]						65																												64
[PARAM_LIST]						66													67															66
[PARAM_LIST_REST]																			69														68	
[PARAM]						70																												70
[PARAM_NAME]						71																												

3.3 Precedenční tabulka

	"+-"	"*/"	rel	!	??	(	)	i
"+-"	R	S	R	S	R	S	R	S
"*/"	R	R	R	S	R	S	R	S
rel	S	S	E	S	S	S	R	S
!	R	R	R	E	R	E	R	E
??	S	S	S	S	S	S	R	S
(	S	S	S	S	S	S	EQ	S
)	R	R	R	R	R	E	R	E
i	R	R	R	R	R	E	R	E
$	S	S	S	S	S	S	E	S

4. Sémantická analýza

Hlavní část semantické analýzy našeho překladače je realizována prostřednictvím souborů **parse.h, parse.c, symtable_stack.c, symtable_stack.h, symtable.c, symtable.h, expression_parse.h a expression_parse.c**. Kritickou roli v semantické analýze hraje symbolická tabulka (symtable), která obsahuje strukturu symdata. Tato struktura umožňuje určit základní atributy jako typ, jméno a rozlišení mezi globálními a lokálními proměnnými, a také identifikovat, zda je daný prvek funkcí nebo identifikátorem atd. Pro implementaci různých scope (dosahů proměnných) využíváme soubor symtable_stack, ve které se nacházejí symbolické tabulky. Nejnižší prvek v tomto zásobníku je vždy globální symbolická tabulka, zatímco s vyšším umístěním prvku ve zásobníku se dostáváme do více lokálních scope. Tento přístup nám umožňuje efektivně spravovat proměnné a funkce v závislosti na jejich dosahu v rámci programu.

5. Generování kódu

Generování kódu je implementováno v souborech codegenerator.h a codegenerator.c. Hlavní funkcí generátoru kódu je **generate_code**, která vytváří řetězce pro zápis instrukcí pomocí deklarovaných příkazů (například DEFVAR, MOVE atd.). Instrukce jsou poté přidávány do jednosměrně vázaného seznamu, který je následně zapisován na stdout. Při implementaci funkce se pro ni vytvoří samostatný jednosměrný seznam. Pomocí symbolického stolu zásobníku(**symtable_stack**) určujeme hloubky oblastí, aby bylo možné správně identifikovat identifikátory a labelu.

6. Speciální použité techniky a algoritmy

exitWithError

Funkce exitWithError představuje klíčovou součást našeho překladače a je zodpovědná za zpracování různých typů chyb, které mohou nastat během procesu kompilace. Přijímá dva parametry - zprávu o chybě (message) a kód chyby (ErrCode). Kód chyby kategorizuje druh chyby, což umožňuje jasnou identifikaci problémů ve zdrojovém kódu.

Formát Chybových Zpráv

Chybové zprávy generované funkcí exitWithError jsou formátovány s použitím barevného kódování pro lepší viditelnost. Číslo řádku a popisná zpráva jsou prezentovány tak, aby asistovaly vývojářům při rychlém identifikování a řešení problémů ve zdrojovém kódu.