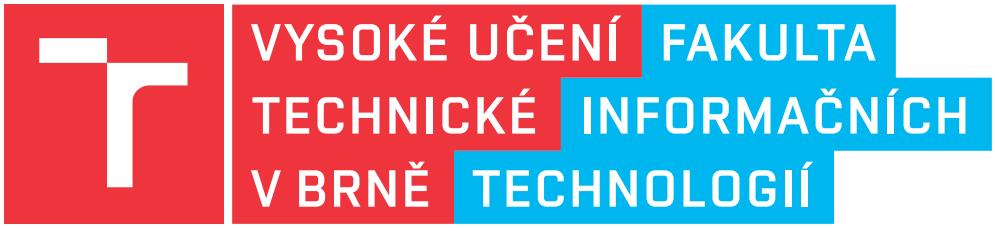


Projektová dokumentace

Implementace překladače imperativního jazyka IFJ24

Tým xstepa77, varianta TRP-izp

4. prosince 2024

Kovina Viktoriia	(xkovin00)	20 %
Litvinchuk Gleb	(xlitvi02)	25 %
Shmonin Gleb	(xshmon00)	30 %
Stepanov Pavel	**(xstepa77)**	25 %

Obsah

1 Úvod

V rámci tohoto projektu byl vytvořen program v jazyce C, který načte zdrojový kód zapsaný ve zdrojovém jazyce IFJ24 a přeloží jej do cílového jazyka IFJcode24

2 Návrh a implementace

2.1 Lexikální analýza

První části tvorby překladaču byla implementace lexikální analýzy. Lexikální analýza je realizována, včetně dalších podpůrných funkcí a struktur, ve zdrojovém souboru `scanner.c`.
Pro komunikaci mezi syntaktickým a sémantickým analyzátorem (parserem) byla implementována klíčová funkce `get_next_token`. Tato funkce slouží jako rozhraní mezi těmito dvěma částmi překladače a je odpovědná za postupné poskytování jednotlivých tokenů parseru. Datová struktura `Token`, kterou funkce vrací, obsahuje několik důležitých informací – typ tokenu, lexém a také jeho přesnou pozici ve zdrojovém souboru.
Funkcionalita lexikální analýzy je reprezentována konečným automatem na obrázku č.1.

2.2 Syntaktická a sémantická analýza

Pro syntaktickou a sémantickou analýzu byl implementovan modul `parser.c`, který je zodpovědný za zpracování vstupního zdrojového kódu a jeho převod do abstraktního syntaktického stromu (AST). Parser využívá techniku rekurzivního sestupu a zajišťuje jak syntaktickou, tak sémantickou kontrolu vstupního programu. Syntaktická a sémantická kontrola probíhá současně.
Klíčové funkce pro parser:

```
parse_program
    parse_import
    parse_function
        parse_parameter
        parse_block
            parse_statement
                parse_variable_declaration
                parse_variable_assigning
                parse_if_statement
                parse_while_statement
                parse_return_statement
                    parse_expression
                        parse_primary_expression
                    check_and_convert_expression
                        convert_to_float
```

Každá gramatická konstrukce má svou parsovací funkci, která může rekurzivně volat další funkce pro zpracování podřízených struktur. Tento přístup umožňuje přehlednou a modulární implementaci parseru.
Každá klíčová funkce, která implementuje logiku parseru, využívá strukturu ASTNode a vrací uzel této struktury, aby bylo možné při průchodu celým programem vytvořit abstraktní syntaktický strom. Tento strom bude následně využit generátorem kódu.

Při zpracování výrazů parser využívá precedenční analýzu pro správné zpracování aritmetických a logických operací. Výrazy jsou rozděleny do úrovní podle precedence operátorů:
Multiplikativní operátory $(*, /)$: zpracovávány funkcí `parse_multiplicative`.
Additivní operátory (+, -): zpracovávány funkcí `parse_additive`.

Relační operátory ($<, \leq, >, \geq$): zpracovávány funkcí `parse_relational`.
Rovnostní operátory (==, !=): zpracovávány funkcí `parse_equality`.

Každá z těchto funkcí zajišťuje správné pořadí operací a volá funkce pro nižší úroveň precedence. Precendenční tabulka je uvedena v tabulce č. 3

Parser využívá globální proměnnou `current_token` pro udržení aktuálního tokenu načteného ze scanneru. Funkce `get_next_token` získává další token ze vstupu. Funkce `expect_token` kontroluje, zda aktuální token odpovídá očekávanému typu, a v případě nesouladu vyvolá syntaktickou chybu.

Pro kontrolu oblasti platnosti proměnné se do tabulky symbolů ukládá ukazatel na uzel AST, ve kterém jsou uloženy údaje o inicializaci proměnné. Následně funkce `scope_check_identifiers_in_tree` rekurzivně prochází celý strom a vyhledává uzly odpovídající identifikátorům proměnných. Pokud je nalezen některý z těchto uzlů, provede se dotaz do tabulky symbolů za účelem získání ukazatele na uzel deklarace proměnné. Od tohoto uzlu se rekurzivně dohledává původně nalezený uzel identifikátoru.

2.3 Tabulka symbolů

Tabulka symbolů je klíčovou součástí programu, která spravuje informace o symbolech, jako jsou proměnné, funkce a parametry. Modul je implementován v souboru `symtable.c` a používá hashovací tabulku pro efektivní vyhledávání a vkládání symbolů.

Funkce `symtable_init` inicializuje tabulku, zatímco `symtable_insert` vkládá nové symboly a řeší kolize. Pro vyhledávání symbolů slouží funkce `symtable_search`, která vrací symbol na základě jeho názvu.
Současně byly s pomocí tabulky symbolů částečně implementovány funkce pro sémantickou kontrolu, jako je `is_symtable_all_used`, `is_main_correct`.Tyto kontroly jsou nezbytné pro identifikaci potenciálních chyb v kódu před jeho generováním. Hashovací funkce `symtable_hash` určuje pozici symbolu v tabulce, což umožňuje rychlé vyhledávání. Tabulka symbolů je tedy nezbytná pro správnou analýzu a generaci kódu, zajišťující efektivní správu symbolů během celého procesu kompilace.

2.4 Generace kódu

Pro generování výsledného kódu byl implementovan modul `codegen.c`, který je zodpovědný za převod abstraktního syntaktického stromu (AST) do cílového jazyka IFJcode24. Generování kódu probíhá ve dvou hlavních fázích: sběr proměnných a samotná generace instrukcí.

V první fázi sbíráme všechny proměnné, včetně dočasných, které budou vygenerovány v kódu. To je realizováno funkcemi jako `collect_variables_in_expression` a `collect_variables_in_function_call`. Tyto funkce rekurzivně procházejí AST a shromažďují informace o proměnných, které jsou následně deklarovány na začátku každé funkce pomocí instrukce `DEFVAR`. Tím zajišťujeme, že všechny proměnné jsou deklarovány před svým použitím, což je požadavek jazyka IFJcode24.

Ve druhé fázi probíhá samotná generace kódu. Hlavní funkcí je `codegen_generate_program`, která postupně zpracovává všechny funkce v programu. Pro každou funkci je volána `codegen_generate_function`, která nejprve deklaruje všechny proměnné a poté generuje kód pro tělo funkce pomocí `codegen_generate_block`. Generování instrukcí je založeno na typu uzlu v AST. Například:

Výrazy: Funkce `codegen_generate_expression` generuje instrukce pro aritmetické a logické operace. Používá se při zpracování uzlů typu `NODE_BINARY_OPERATION, NODE_LITERAL` a `NODE_IDENTIFIER`.

Příkazy: Funkce `codegen_generate_statement` zpracovává příkazy jako přiřazení (`NODE_ASSIGNMENT`), deklarace proměnných (`NODE_VARIABLE_DECLARATION`), podmíněné příkazy (`NODE_IF`) a cykly (`NODE_WHILE`).

Volání funkcí: Pro volání funkcí je použita funkce `codegen_generate_function_call`, která se stará jak o uživatelsky definované funkce, tak o vestavěné funkce. U vestavěných funkcí, jako jsou `ifj.write`, `ifj.readi32`, `ifj.length` apod., jsou generovány specifické instrukce dle jejich požadavků.

Data z AST uzlů, jako jsou názvy proměnných a hodnoty literálů, jsou využívána při generování konkrétních instrukcí. Například při generování instrukce pro přiřazení se využívá název proměnné z `node->name` a výraz z `node->left`.
Instrukce jsou zapisovány do výstupního souboru (`output_file`) postupně během průchodu AST. Tento přístup umožňuje efektivní generování kódu a snadnou údržbu generátoru. Důležitou součástí je také práce s dočasnými proměnnými a unikátními návěštími pro skoky. K tomu slouží pomocné funkce jako `generate_unique_var_name` a `generate_unique_label`, které zajišťují, že vygenerovaná jména jsou jedinečná v rámci celého programu.

Celkově náš generátor kódu transformuje vstupní program reprezentovaný AST do spustitelného kódu v jazyce IFJcode24, přičemž dbá na správnou deklaraci proměnných, správné pořadí instrukcí a optimalizuje práci s dočasnými hodnotami.

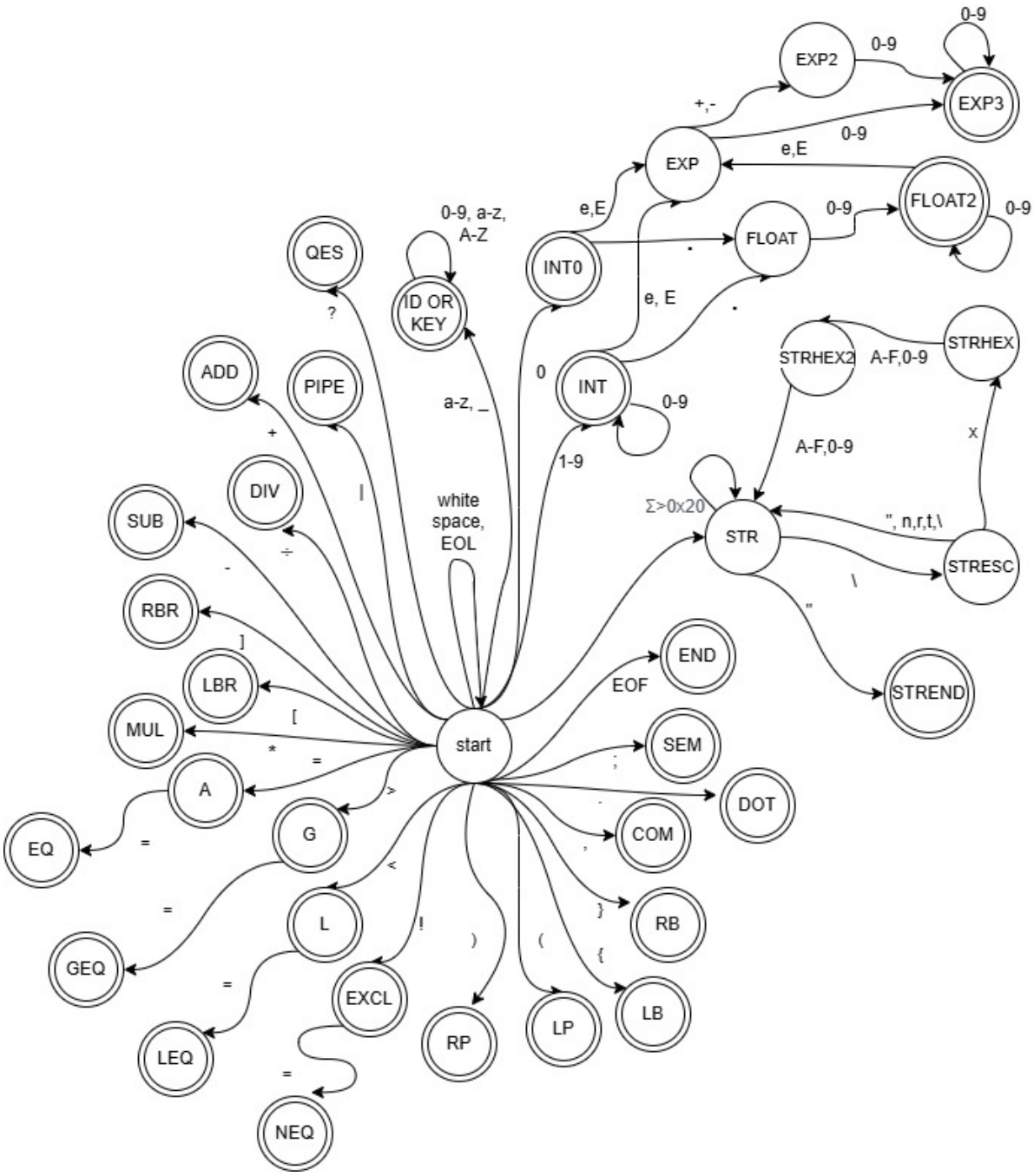
3 Práce v týmu

3.1 Rozdělení práce mezi členy týmu

Viktoriia Kovina
Práce na syntaktickém a sémantickém analyzátoru, precedenční tabulky, dokumentace
Gleb Litvinchuk
Generovnáí cílového kódu, práce na lexikálním analyzátoru
Gleb Shmonin
Implementace syntaktického a sémantického analyzátoru, tvorba gramatických pravidel, vypracování tabulky LL gramatiky,
Pavel Stepanov
Vedení týmu, tvorba a návrh lexikálního analyzátoru, generovnáí cílového kódu, implementace tabulky symbolů

3.2 Způsob práce v týmu

Práce byla postupně rozdělena mezi účastníky týmu. Během osobních setkání byly sledovány výsledky práce na jednotlivých částech a diskutovány další kroky. Pro komunikaci a šíření důležitých informací byl také používán Discord. K verzování zdrojových kódů byl použit GitHub, ten umožnil pracovat na více úkolech na projektu současně a sledovat změny. Z důvodu nerovnoměrného rozdělení práce mezi členy týmu máme odchylku od rovnoměrného rozdělení bodů.



Obrázek 1: Diagram konečného automatu specifikující lexikální analyzátor

Legenda:
$\sum > 0x20$ - všechny tisknutelné znaky

1. <program> -> eof
2. <program> -> <import> <program>
3. <program> -> <func_def> <program>
4. <import> -> const id = @ import (<expr>)
5. <func_def> -> pub fn id (<params>) <type> { <body_stats> }
6. <func_def> -> fn id (<params>) <type> { <body_stats> }
7. <params> -> eps
8. <params> -> <param>
9. <params> -> <param> , <params>
10. <param> -> id <type_id>
11. <type> -> i32
12. <type> -> f64
13. <type> -> []u8
14. <type> -> void
15. <type> -> ?i32
16. <type> -> ?f64
17. <type> -> ?[]u8
18. <type_id> -> : <type>
19. <type_id> -> eps
20. <body_stats> -> eps
21. <body_stats> -> <body> ; <body_stats>
22. <body> -> id = <expr>
23. <body> -> var <type_id> = <expr>
24. <body> -> const <type_id> = <expr>
25. <body> -> if (<expr>) <id_null> { <body_stats> } <else>
26. <body> -> while (<expr>) <id_null> { <body_stats> }
27. <body> -> return <expr>
28. <else> -> else { <body_stats> }
29. <id_null> -> | id |
30. <id_null> -> eps
31. <func_call> -> (<args>)
32. <func_call> -> eps
33. <func_call> -> . id (<args>)
34. <args> -> eps
35. <args> -> <arg> , <args>
36. <arg> -> <expr>
37. <expr> -> id <func_call>
38. <expr> -> literal
39. <expr> -> eps

Tabulka 1: LL – gramatika řídící syntaktickou analýzu

	pub	fn	var	const	id	literal	while	if	else	return	void	i32	f64	[]u8	?i32	?f64	?[]u8	=	(	)	{	}	:	;	,	.	|	EOF
<program>	3	3		2																								1
<import>				4																								
<func_def>	5	6																										
<params>					10																							
<param>					8															7								
<type>											14	11	12	13	15	16	17											
<type_id>																		19		19			18					
<body_stats>			21	21	21		21	21		21												20						
<body>			23	24	22		26	25		27																		
<else>									28																			
<id_null>																					30			21			29	
<func_call>																			31	32				32	32	33		
<args>					35	35														34					35			
<arg>					36	36																			36			

Tabulka 2: LL – tabulka použitá při syntaktické analýze

Operátor	*	/	+	-	==	!=	<	>	<=	>=	(	)
*	=	=	>	>	>	>	>	>	>	>	<	>
/	=	=	>	>	>	>	>	>	>	>	<	>
+	<	<	=	=	>	>	>	>	>	>	<	>
-	<	<	=	=	>	>	>	>	>	>	<	>
==	<	<	<	<	=	=	>	>	>	>	<	>
!=	<	<	<	<	=	=	>	>	>	>	<	>
<	<	<	<	<	<	<	=	=	=	=	<	>
>	<	<	<	<	<	<	=	=	=	=	<	>
<=	<	<	<	<	<	<	=	=	=	=	<	>
>=	<	<	<	<	<	<	=	=	=	=	<	>
(	<	<	<	<	<	<	<	<	<	<	<	=
)	>	>	>	>	>	>	>	>	>	>	=	>

Tabulka 3: Precedenční tabulka použitá při precedenční syntaktické analýze výrazů