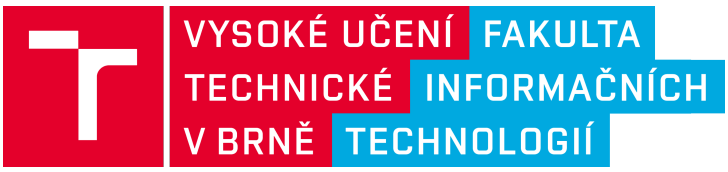


Projektová dokumentace

Implementace překladače imperativního jazyka IFJ22

Tým xnosal01, Varianta TRP, FUNEXP

Nikolas Nosál (xnosal01)	25 %	
Adam Mrkva (xmrkva04)	25 %	
Rostislav Navrátil (xnavra72)	25 %	
David Nevrlka (xnevrl00)	25 %	Brno, 7. prosince 2022

Obsah

1. Úvod

Cílem projektu bylo vytvořit program v jazyce C, který načte zdrojový kód v jazyce IFJ22, jenž je zjednodušenou podmnožinou jazyka PHP a následně jej přeloží do cílového jazyka IFJcode22. Jako tým jsme zvolili variantu TRP (Implementace tabulky symbolů pomocí tabulky s rozptýlenými položkami).

2. Implementace

V této kapitole je popsána implementace jednotlivých částí, které na sebe navazují a tvoří tak spolu výsledný překladač.

2.1. Lexikální analýza

Základním modulem lexikální analýzy je scanner. Tento scanner je implementován podle deterministického konečného stavového automatu. Diagram automatu je na obrázku 1. Většina lexikálního analyzátoru se nachází v souborech `lex.c` a `lex.h` a jeho pomocné funkce se nachází v souborech `string_lib.c.c` a `string_lib.h`. Hlavní je funkce `lex_tokenize`, která přečte zdrojový soubor a převádí jej na tokeny a ukládá je do pole tokenů. Při procházení zdrojového souboru je využit posuvný buffer, který umožňuje nahlížet dopředu. Tokeny obsahují typ, řádek, sloupec a obsah. Chyby jsou ošetřeny pomocí funkce `lex_error`, která vypisuje chybovou hlášku, řádek a obsah bufferu na místě chyby.

2.2. Syntaktická analýza

Syntaktická analýza se řídí LL-gramatikou (tabulka 3) a metodou rekurzivního sestupu. Je implementována v souborech `parser.c`, `parser.h`, `expr_parser.c`, `expr_parser.h`, `syntax_tree.c` a `syntax_tree.h`. Analýza je rozdělena na parsování výrazů a parsování všeho ostatního. Syntaktický analyzátor postupně prochází pole tokenů, pokud u tokenu nenalezne chybu, staví abstraktní syntaktický strom podle LL-gramatiky. Při vývoji jsme tento strom převáděli do textového formátu JSON a poté vizualizovali pomocí nástroje `https://vanya.jp.net/vtree/`(Ukázka na obrázku 2). Výrazy jsou parsovány pomocí funkce `expr_parse`, která vrací strom výrazů a ten je poté připojen do syntaktického stromu. Chyby jsou ošetřeny pomocí funkce `syntax_error`, která vypisuje chybovou hlášku a obsah daného tokenu.

2.3. Sémantická analýza

Sémantická analýza je implementována v souborech `semantic_analyzer.c` a `semantic_analyzer.h` a je založena na rekurzivním volání funkce `rec_check_types`. Nicméně hlavní funkce analýzy je `analyze_ast`. Funkce nejdříve prochází hlavičky funkcí, poté těla funkcí a nakonec blok kódu programu, všechno se zapisuje do tabulky symbolů. Pro každou funkci a blok kódu programu se vytváří nová tabulka symbolů, která se po analyzování dané funkce/bloku programu ruší. Tyto tabulky používáme 2. První uchovává proměnné a druhá funkce. Tabulky slouží primárně k referenci. Chyby jsou ošetřeny pomocí funkce `semantic_error`, která vypisuje, typ chyby, chybovou hlášku a obsah daného tokenu.

2.4. Generování cílového kódu

Samotný generátor je implementován v souborech `code_generation.c` a `code_generation.h`. Kód je generován z abstraktního syntaktického stromu. Generování kódu je založeno na rekur-

zivním volání funkce `generate_code`, která volá všechny ostatní funkce podle typu uzlu ve stromu. Při generování je využíván zásobník, který se využívá na ukládání řetězců a char ukazatelů, je implementován v souboru `stack_lib.c` a `stack_lib.h`. Generátor využívá skoro všechny instrukce IFJ22 a podporuje zadáním danou funkcionalitu.

2.5. Testování

Při vývoji jsme se řídili námi vytvořenými jednotkovými testy. Později jsme přešli k testování automatickými testy.

3. Dynamické datové struktury

3.1. Posuvný buffer

Buffer do sebe postupně načítá znaky ze vstupu. Uchovává zdrojový soubor, informaci zda buffer obsahuje EOF, řádek a sloupec a kde se nachází první znak v bufferu. Dovoluje nám přistupovat k jednotlivým znakům a umožňuje nám tak dívat se dopředu, což je velmi výhodné při zjišťování klíčových slov. Je implementován v souborech `strings_lib.c` a `strings_lib.h`.

3.2. Pole tokenů

Jedná se o dynamicky alokované pole, do něhož jsou vkládány tokeny z lexikálního analyzátoru.Pole má definovanou počáteční velikost a zvětšuje se vždy na dvojnásobek své aktuální velikosti. Je implementován v souborech `tokens_lib.c` a `tokens_lib.h`.

3.3. Strom výrazů

Strom se skládá z uzlů, které jsou dynamicky alokované a provázané. Uchovávají v sobě tokeny výrazu. Strom reprezentuje prioritu operátoru a pořadí operandu. Je implementován v souborech `expr_tree.c` a `expr_tree.h`.

3.4. Syntaktický strom

Strom se skládá z uzlů, které jsou dynamicky alokované a provázané. Je implementován v souborech `syntax_tree.c` a `syntax_tree.h`.

3.5. Tabulka symbolů

Tabulka symbolů je implementována pomocí tabulky s rozptýlenými položkami (soubory `symtable.c`, `symtable.h`). Tuto implementaci jsme zvolili pro její jednoduchost a lepší možnosti vyhledávání a porovnávání pro naše řešení.

4. Práce v týmu

4.1. Komunikace

Na začátku semestru proběhla schůze týmu, kde jsme se dohodli na postupu při implementaci a na rozdělení práce v týmu. Pro komunikaci v týmu jsme používali hlavně discord server. Zde probíhali schůzky a řešení částí projektu, které vyžadovali spolupráci více členů týmu. Pro sdílení kódu jsme použili verzovací systém Git a servery GitHub.

5. Členění implementačního řešení

Lexikální analyzátor	`lex.c, lex.h`
Syntaktický analyzátor	`parser.c, parserh, expr_parser.c, expr_parser.h`
Sémantický analyzátor	`semantic_analyzer.c, semantic_analyzer.h`
Generování kódu	`code_generation.c, code_generation.h`
Posuvný buffer	`strings_lib.c, strings_lib.h`
Pole tokenů	`tokens_lib.c, tokens_lib.h`
Strom výrazů	`expr_tree.c, expr_tree.h`
Syntaktický strom	`syntax_tree.c, syntax_tree.h`
Tabulka symbolů	`symtable.c, symtable.h`
Funkce main	`ifj22_compiler.c`
Posuvný buffer a pomocné funkce	`string_lib.c, string_lib.h`
Ošetření chyb	`error_handler.c, error_handler.h`
Zásobník	`stack_lib.c, stack_lib.h`

Tabulka 1: Členění implementačního řešení

5.1. Rozdělení práce

Člen týmu	Přidělená práce
Nikolas Nosál	Tabulka symbolů, Generování kódu, Syntaktická analýza
Adam Mrkva	Ošetření chyb, Generování kódu
Rostislav Navrátil	Lexikální analýza, Syntaktická analýza, Sémantická analýza
David Nevrlka	Sémantická analýza, Dokumentace, Prezentace

Tabulka 2: Rozdělení práce v týmu

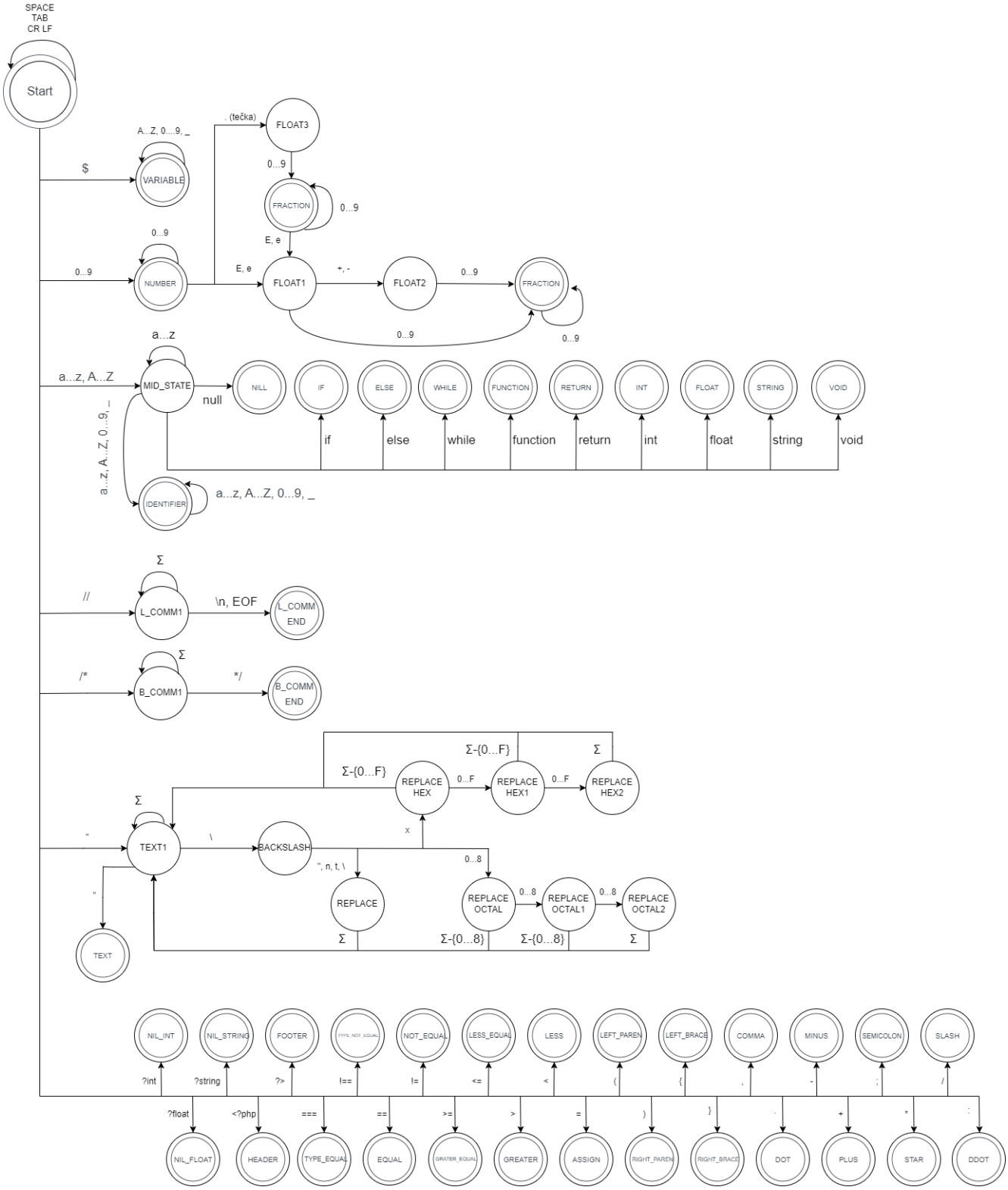
6. Dokumentace

Dokumentace patřila k finálním částem projektu. Byla vytvořena v Latexu v editoru Overleaf. Diagramy a tabulky byly navrženy pomocí online editorů.

7. Závěr

Projekt nás nejdříve zaskočil svou rozsáhlostí. Nejtěžší bylo přijít na to, jak takový překladač funguje a jak by se jeho částí měly chovat. Tým jsme sestavili prakticky hned na začátku semestru. Po nastudování všech náležitostí a rozdělení práce jsme mohli začít pracovat. Chtěli jsme zkusit pokusné odevzdání, ale bohužel jsme to nestihli, jelikož jsme se potýkali s řadou drobných nejasností. Nicméně hlavní termín zadání jsme stihli. Díky projektu jsme získali cenné zkušenosti s tvorbou překladačů, programování v C, PHP a prací v týmu.

A. Diagram konečného automatu specifikující lexikální analyzátor



Obrázek 1: Diagram konečného automatu lexikální analýzy

B. LL – Gramatika

1. <start> -> <?php declare(strict_types=1) <block> <end>
2. <block> -> <assign>
3. <block> -> <expression>;
4. <block> -> <func_def>
5. <block> -> <return>
6. <block> -> <ifelse>
7. <block> -> <while>
8. <end> -> EOF
9. <end> -> ?>
10. <assign> -> VARIABLE = <expression>;
11. <func_def> -> function IDENTIFIER <params>:<return_type> <body>
12. <return> -> return <expression>;
13. <ifelse> -> if (<expression>) <body>
14. <ifelse> -> if (<expression>) <body> else <body>
15. <while> -> while (<expression>) <body>
16. <params> -> ()
17. <params> -> (<param>)
18. <params> -> (<param> ... <param>)
19. <param> -> <param_type> VARIABLE
20. <param> -> , <param_type> VARIABLE
21. <return_type> -> void
22. <return_type> -> ?int
23. <return_type> -> ?float
24. <return_type> -> ?string
25. <return_type> -> int
26. <return_type> -> float
27. <return_type> -> string
28. <body> -> {<block>}
29. <param_type> -> int
30. <param_type> -> float
31. <param_type> -> string

Tabulka 3: LL–Gramatika

C. LL–Tabulka

	<start>	<block>	<end>	<assign>	<func_def	<return>	<ifelse>	<while>	<params>	<param>	<return_type>	<body>	<param_type>
HEADER	1												
FOOTER			9										
IDENTIFIER					11								
=				10									
;		3				12							
VARIABLE				10						19, 20			
FUNC					11								
:				10	11								
RETURN		5				12							
IF		6					13, 14						
ELSE		6					14						
WHILE		7						15					
()							13, 14	15	16, 17, 18				
,										20			
VOID											21		
NIL_INT											22		
NIL_FLOAT											23		
NIL_STRING											24		29
INT											25		30
FLOAT											26		31
STRING											27		
{}												28	

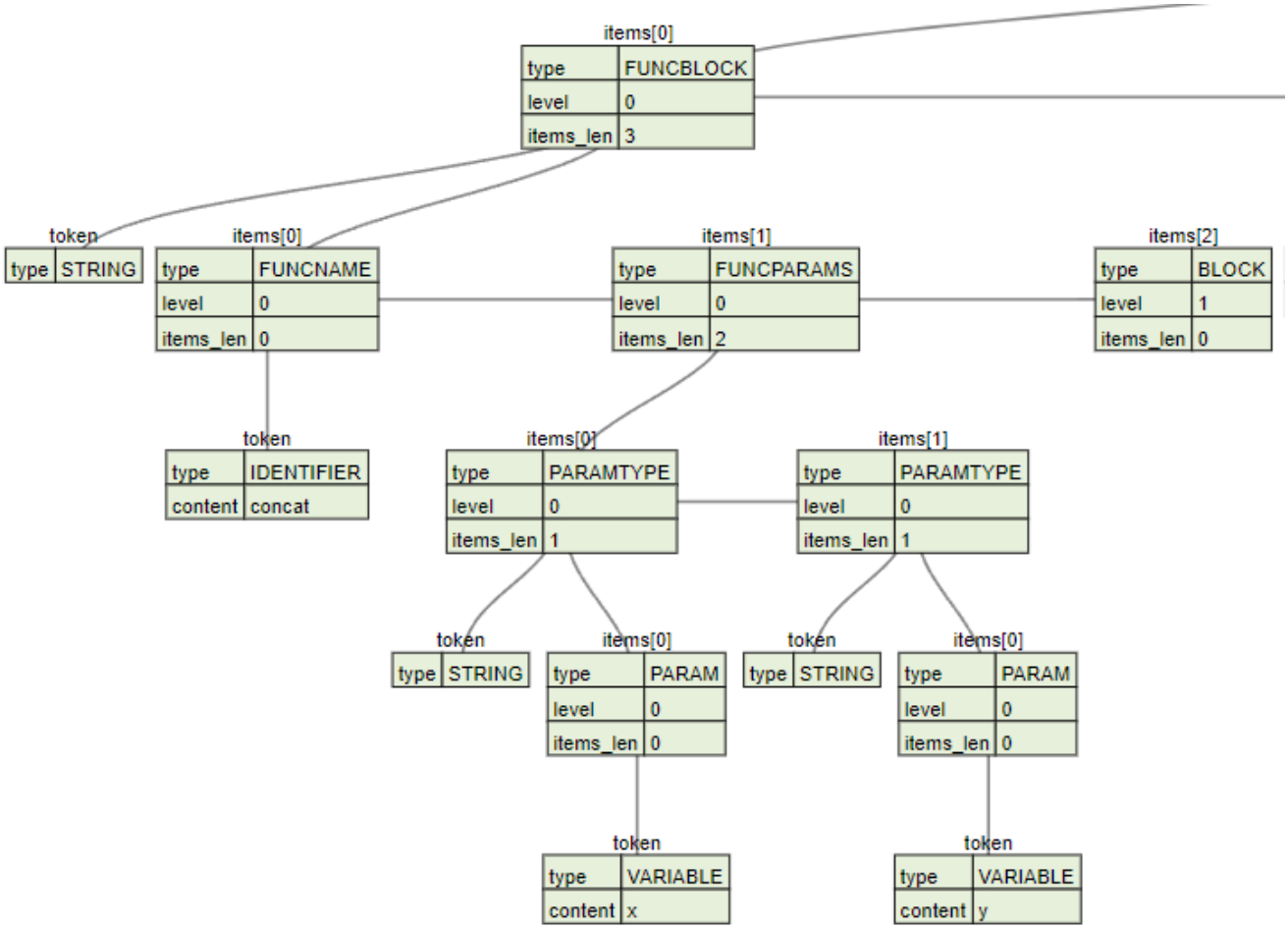
Tabulka 4: LL-tabulka

D. Precedenční tabulka

	+ - .	* /	<,>,<=,>= ==,!=	===,!==	(	)
+ - .	>	<	>	>	<	>
* /	>	>	>	>	<	>
<, >,<=,>= ==, !=	<	<	X	X	<	>
===, !==	<	<	X	X	<	>
(	<	<	<	<	<	=
)	>	>	>	>	X	>

Tabulka 5: Precedenční tabulka

E. Ukázka stromu



Obrázek 2: Ukázka funkce ve stromu