

# VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

## FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Dokumentace projektu z předmětů IFJ a IAL

### **Implementace překladače imperativního jazyka IFJ22**

Tým xbarto0g, varianta – TRP

2. prosince 2022

Petr Bartoš – xbarto0g (vedoucí)	40 %
Tomáš Rajsigl – xrajsi01	30 %
Lukáš Zedek – xzedek03	30 %
Dmytro Afanasiev – xafana01	0 %

# 1 Úvod

Cílem projektu je vytvořit program v jazyce C, který načte zdrojový kód zapsaný ve zdrojovém jazyce IFJ22 a přeloží jej do cílového jazyka IFJcode22. Jazyk IFJ22 je zjednodušenou podmnožinou jazyka PHP.

V následující dokumentaci je postup řešení tohoto projektu a popis implementace jeho jednotlivých částí.

## 2 Práce v týmu

Práci jsme si rozdělili jakožto čtyřčlenný tým, avšak z důvodu nespolupracování a nezájmu jednoho člena jsme byli nuceni si práci přerozdělit a projekt vypracovat pouze ve třech. Vzhledem k časové náročnosti a složitosti daných částí jsme se rozhodli pro nerovnoměrné rozdělení bodů.

### 2.1 Rozdělení práce

- Petr Bartoš
  - Návrh gramatik pro syntaktickou analýzu, syntaktická analýza, sémantická analýza, tabulka symbolů, chybové hlášky, kostra pro testování
- Tomáš Rajsigl
  - Návrh konečného automatu a gramatik pro syntaktickou analýzu, lexikální analýza, testování scanneru a parseru, dokumentace a prezentace
- Lukáš Zedek
  - Generování cílového kódu, testování jednotlivých modulů překladače, Makefile, dokumentace a prezentace

Kontrola kódu, jeho čitelnosti a opravě chyb jsme se věnovali všichni tři.

### 2.2 Vývojový cyklus

Při vypracovávání projektu jsme využívali verzovací systém Git. Zdrojové kódy jsme měli uloženy v repozitáři webové služby GitHub, díky čemuž jsme měli všichni vždy přístup k nejnovější verzi projektu.

Pro dané části projektu byly vytvořeny konkrétní branchy, kde jsme je testovali a upravovali. Před zahrnutím do master branchy byl vyžadován pull request, následný code review a schválení od jednoho či více členů.

Součástí vývojového cyklu byl i unit testing. Při každém commitu byly automaticky spuštěny testy a bylo tak hned možné vidět, jaký dopad bude commit mít na výsledný program.

## 3 Návrh a implementace překladače

Projekt byl rozdělen na několik konkrétních částí, které jsou postupně představeny v této kapitole.

### 3.1 Lexikální analýza

Prvním krokem překladače byl návrh a následná implementace lexikální analýzy. Celý scanner je implementován jako deterministický konečný automat, jehož diagram lze nalézt v obrázku 1. Řešení se nachází ve zdrojovém souboru `scanner.c` a hlavičkovém souboru `scanner.h`.

Hlavní funkce tohoto scanneru je `getToken`, pomocí které se čte znak po znaku ze zdrojového souboru a převádí se na strukturu `Token`, která se skládá z typu, hodnoty a z důvodu vypisování chybových hlášek<sup>1</sup> také pozice. Typy tokenu jsou EOF, prázdný token, identifikátory, klíčová slova, celé či desetinné číslo, řetězec a také porovnávací a aritmetické operátory a ostatní znaky, které mohou být použity v jazyce IFJ22. Hodnota tokenu je typu `union` a přiděluje se tokenům `TOKEN_KEYWORD`, `TOKEN_INT`, `TOKEN_FLOAT` a `TOKEN_STRING`.

V ojedinělých případech je nutno nahlédnout na další token<sup>2</sup>, v takové situaci je nastaven `bool peek` na hodnotu `true`. Token je poté vrácen do prvního volání s parametrem `peek` nastaveným na `false`.

Jedná se o dlouhý `switch`, kde každý případ `case` je ekvivalentní k jednomu stavu automatu. Pokud načtený znak nesouhlasí s žádným znakem, který jazyk povoluje, program je ukončen a vrací chybu 1. Jinak se přechází do dalších stavů a načítají se další znaky, dokud nemáme hotový jeden token, tj. dokud nedostaneme `tokenComplete = true` a token potom vrátíme a ukončíme tuto funkci.

Pro zpracovávání hexadecimálních a oktálních escape sekvencí v řetězci máme vytvořena dvě pole o velikosti 3 a 4, která jsou zpočátku vynulována a poté se postupně naplňují přečtenými čísly, vždy na pozici podle toho, kolikáté číslo zrovna uvažujeme a nakonec celé číslo převedeme do ASCII podoby.

#### 3.1.1 Prolog a epilog

Na zpracování prologu jsme se rozhodli vytvořit funkci `initialScan`, která ve scanneru kontroluje první část, a to otevírací značku `<?php`, druhou část – `declare(strict_types=1)`; posíláme jakožto jednotlivé tokeny dále syntaktické analýze, ve které provedeme dodatečnou kontrolu.

Pro zavírací značku volitelného epilogu je vytvořen `TOKEN_CLOSING_TAG`, za kterým je očekáván EOF a nebo `\n` a poté EOF, jinak vrací chybu 1.

#### 3.1.2 Řetězec s proměnnou délkou

K zpracovávání identifikátorů a klíčových slov používáme vlastní strukturu `vStr`, která je implementována ve zdrojovém souboru `vstr.c` a souboru hlavičkovém `vstr.h`.

Jedná se o pole znaků, které se dynamicky zvětšuje v závislosti na své délce – při dosáhnutí aktuální maximální délky řetězce je délka zdvojnásobena. Znaky načítáme a průběžně ukládáme do dynamického pole a jakmile odcházíme ze stavu identifikátoru, porovnáváme, jestli načtený řetězec není shodný s nějakým z klíčových slov a podle toho se rozhodneme, zda vrátíme typ a hodnotu tokenu jakožto klíčové slovo či identifikátor.

---

<sup>1</sup>K nalezení v souborech `error.{c,h}`

<sup>2</sup>tj. získat stejný token při více voláních

## 3.2 Syntaktická analýza

Syntaktická analýza, kterou lze nalézt v souborech `parser.c` a `parser.h`, vychází z LL–gramatiky, kterou je možno vidět v tabulce 1.

V souboru `parser.h` je vytvořena struktura `Parser`, ve které jsou data používaná jak pro syntaktickou, tak sémantickou analýzu. Struktura se skládá z aktuálního tokenu `Token currToken`, ukazatele na tabulku symbolů při vnoření do funkce `Symtable *localSymtable`, ukazatele na tabulku symbolů pro hlavní tělo programu `Symtable *symtable`, ukazatele na zásobník `Stack *stack`, značky pro potenciálně nedefinovanou proměnnou `bool condDec` a značky `bool outsideBody` pro situaci kdy je prováděna instrukce mimo hlavní tělo programu.

Parser postupně od scanneru získává jednotlivé tokeny, na základě kterých je vybráno derivační pravidlo. Neterminály tvořené těmito tokeny jsou poté rozkládány na primitivnější neterminály, dokud nejsou zcela rozloženy na terminály. Tento rozklad probíhá pomocí rekurzivního volání jednotlivých funkcí pro derivační pravidla. Tyto funkce naleznete ve zdrojovém souboru jsou například `parseWhile`, `parseIf` nebo `parseBody` a další.

### 3.2.1 Precedenční syntaktická analýza

Algoritmus pro zpracování výrazů byl implementován v souladu s materiály k předmětu IFJ, tedy metodou zdola nahoru. Řídí se dle precedenční tabulky, která je přiložena v tabulce 3. Podle ní může nastat 5 stavů a to: `reduce`, `shift`, `equal`, `error` a `ok`.

Funkce `parseExpression` postupně načítá tokeny do zásobníku, určuje jejich typ a pomocí tabulky určuje vztah mezi nejvyšším terminálem na zásobníku a tokenem na vstupu. Pokud žádné pravidlo neodpovídá situaci na vrcholu zásobníku, tak je program ukončen a vrací chybu 2.

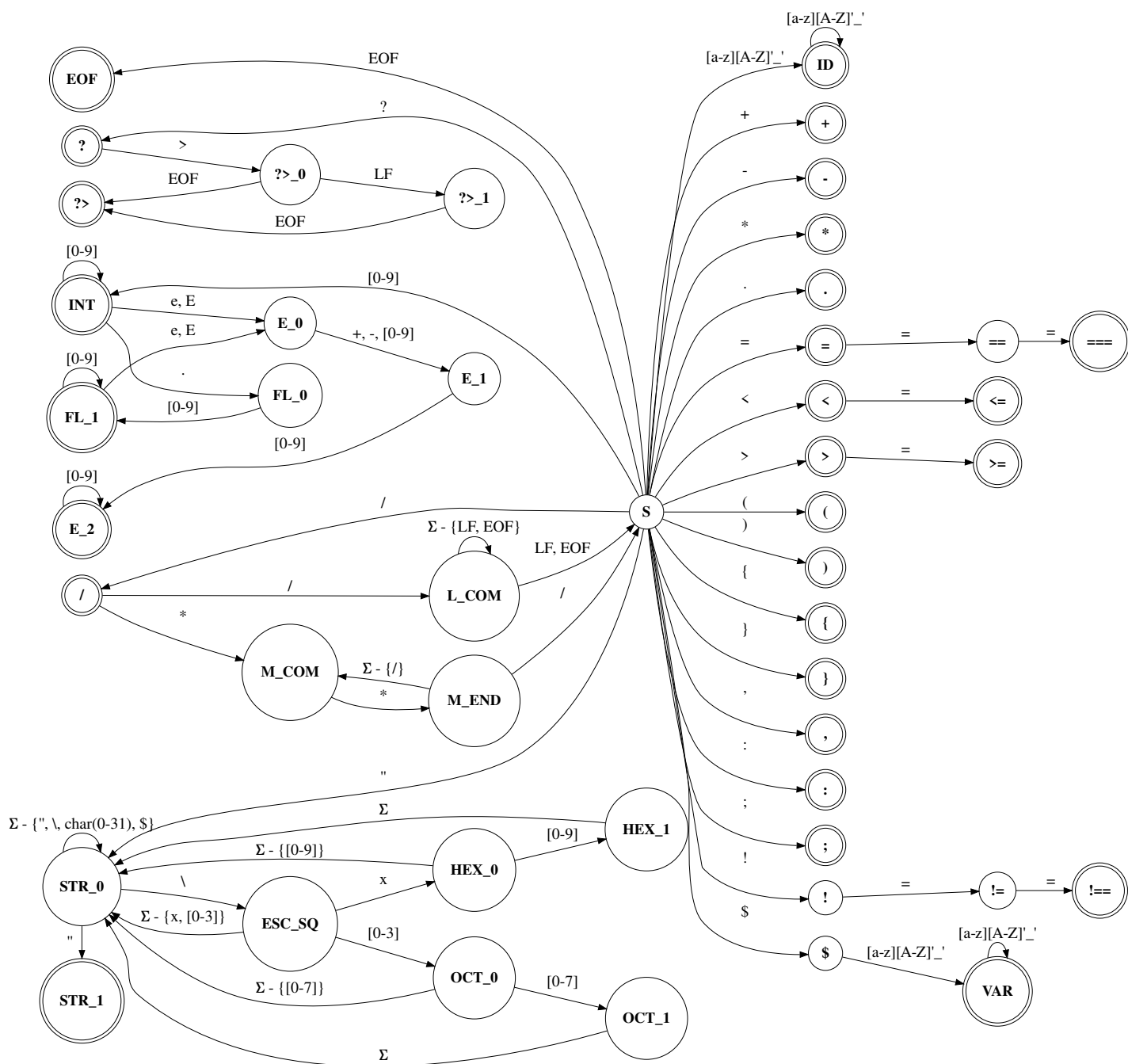
V úvahu by přišla záměna zásobníku za dvousměrně vázaný seznam kvůli jednoduchosti vkládání zářezek. V případě implementace se zásobníkem je třeba „překážející“ hodnoty dočasně uložit na jiný zásobník a poté vracet zpět, což je neefektivní.

### **3.3 Sémantická analýza**

### **3.4 Generování cílového kódu**

## 4 Přílohy

### 4.1 Diagram konečného automatu



Obrázek 1: Konečný automat<sup>2</sup> specifikující lexikální analyzátor

<sup>2</sup>Z důvodu úspory místa a čitelnosti byly jednotlivé stavy přejmenovány, legenda stavů následuje na další stráně

### 4.1.1 Legenda

S	STATE_INITIAL	/	STATE_SLASH
ID	STATE_IDENTIFIER	L_COM	STATE_LINE_COM
+	STATE_PLUS	M_COM	STATE_MULTILINE_COM
-	STATE_MINUS	M_END	STATE_POT_MULTILINE_END
*	STATE_MULTIPLY	STR_0	STATE_STRING_0
.	STATE_CONCATENATE	STR_1	STATE_STRING_1
=	STATE_EQUAL_OR_ASSIGN	ESC_SQ	STATE_STRING_ESCAPE
==	STATE_EQUAL_0	HEX_0	STATE_STRING_HEXA_0
===	STATE_EQUAL_1	HEX_1	STATE_STRING_HEXA_1
<	STATE_LESSTHAN	OCT_0	STATE_STRING_OCTA_0
>	STATE_GREATERTHAN	OCT_1	STATE_STRING_OCTA_1
(	STATE_LEFT_BRACKET		
)	STATE_RIGHT_BRACKET		
{	STATE_LEFT_BRACE		
}	STATE_RIGHT_BRACE		
EOF	STATE_EOF		
?	STATE_OPTIONAL		
?>_0	STATE_CLOSING_TAG_0		
?>_1	STATE_CLOSING_TAG_1		
?>	STATE_CLOSING_TAG_2		
,	STATE_COMMA		
:	STATE_COLON		
;	STATE_SEMICOLON		
!	STATE_NOT_EQUAL_0		
!=	STATE_NOT_EQUAL_1		
!==	STATE_NOT_EQUAL_2		
\$	STATE_VARID_PREFIX		
INT	STATE_NUMBER		
E_0	STATE_FLOAT_E_0		
E_1	STATE_FLOAT_E_1		
E_2	STATE_FLOAT_E_2		
FL_0	STATE_FLOAT_0		
FL_1	STATE_FLOAT_1		

## 4.2 LL – gramatika

1. <program>	-> <body_main> <program_e>
2. <program_e>	-> $\epsilon$
3. <program_e>	-> ?>
4. <type_p>	-> float
5. <type_p>	-> int
6. <type_p>	-> string
7. <type_n>	-> ? <type_p>
8. <type>	-> void
9. <type>	-> <type_p>
10. <type>	-> <type_n>
11. <return>	-> return <return_p>
12. <return_p>	-> expr
13. <return_p>	-> $\epsilon$
14. <body_main>	-> <body> <body_main>
15. <body_main>	-> <func_def> <body_main>
16. <body_main>	-> $\epsilon$
17. <body>	-> <if>
18. <body>	-> <while>
19. <body>	-> expr ;
20. <body>	-> identifier_var = <assign_v> ;
21. <body>	-> <func> ;
22. <body>	-> <return> ;
23. <assign_v>	-> expr
24. <assign_v>	-> <func>
25. <pe_body>	-> <body> <pe_body>
26. <pe_body>	-> $\epsilon$
27. <if>	-> if ( expr ) { <pe_body> } else { <pe_body> }
28. <while>	-> while ( expr ) { <pe_body> }
29. <params_dn>	-> $\epsilon$
30. <params_dn>	-> , <type_n> identifier_var <params_dn>
31. <params_dp>	-> <type_n> identifier_var <params_dn>
32. <params_dp>	-> $\epsilon$
33. <params_cnp>	-> lit
34. <params_cnp>	-> identifier_var
35. <params_cn>	-> $\epsilon$
36. <params_cn>	-> , <params_cnp> <params_cn>
37. <params_cp>	-> lit <params_cn>
38. <params_cp>	-> identifier_var <params_cn>
39. <params_cp>	-> $\epsilon$
40. <func_def>	-> function identifier_func ( <params_dp> ) : <type> { <pe_body> }
41. <func>	-> identifier_func ( <params_cp> )

Tabulka 1: LL – gramatika řídící syntaktickou analýzu



4.3 LL – tabulka

	?>	float	int	string	?	void	return	expr	;	identifier_var	=	if	(	)	{	}	else	while	,	lit	function	identifier_func	:	\$
<program>	1						1	1		1		1						1			1	1		1
<program_e>	3																							2
<type_p>		4	5	6																				
<type_n>					7																			
<type>		9	9	9	10	8																		
<return>							11																	
<return_p>								12	13															
<body_main>	16						14	14		14		14						14			15	14		16
<body>							22	19		20		17						18				21		
<assign_v>								23														24		
<pe_body>							25	25		25		25			26			25				25		
<if>												27												
<while>																		28						
<params_dn>														29					30					
<params_dp>					31									32										
<params_cnp>										34										33				
<params_cn>														35					36					
<params_cp>										38				39						37				
<func_def>																					40			
<func>																						41		

Tabulka 2: LL – tabulka použitá při syntaktické analýze

## 4.4 Precedenční tabulka

	<b>+ -</b>	<b>* /</b>	<b>i</b>	<b>\$</b>	<b>Rel</b>	<b>Cmp</b>	<b>(</b>	<b>)</b>
<b>+ -</b>	>	<	<	>	>	>	<	>
<b>* /</b>	>	>	<	>	>	>	<	>
<b>i</b>	>	>		>	>	>		>
<b>\$</b>	<	<	<		<	<	<	
<b>Rel</b>	<	<	<	>			<	>
<b>Cmp</b>	<	<	<	>			<	>
<b>(</b>	<	<	<		<	<	<	=
<b>)</b>	>	>		>	>	>		>

Tabulka 3: Precedenční tabulka<sup>3</sup> použitá při precedenční syntaktické analýze výrazů

---

<sup>3</sup>Rel značí relační operátory a Cmp je zkratka pro komparační operátory

## 5 Reference

1. AHO, Alfred V.; LAM, Monica S.; SETHI, Ravi; ULLMAN, Jeffrey D. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN 0321486811.
2. NYSTROM, R. *Crafting Interpreters*. USA: Genever Benning, 2021. ISBN 9780990582939.
3. PERINGER, Petr. *IJC: DU2* [online]. FIT VUT, 2022. Dostupné také z: <https://www.fit.vutbr.cz/study/courses/IJC/public/DU2.html.cs>.
4. MEDUNA, Alexander; KŘIVKA, Zbyněk. *Podklady k přednáškám* [online]. FIT VUT, 2022. Dostupné také z: <http://www.fit.vutbr.cz/study/courses/IFJ/public/materials/>.