VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Dokumentace projektu z předmětů IFJ a IAL

Implementace překladače imperativního jazyka IFJ22

Tým xbarto0g, varianta – TRP

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

2. prosince 2022

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 a sémantická analýza, tabulka a zásobník symbolů, chybové hlášky, kostra pro testování
- Tomáš Rajsigl
  - Návrh konečného automatu a gramatik pro syntaktickou analýzu, lexikální analýza, testování jednotlivých částí překladače, dokumentace
- Lukáš Zedek
  - Generování cílového kódu, testování jednotlivých částí překladače, Makefile, dokumentace a prezentace

Kontrole 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 větve, kde jsme je testovali a upravovali. Před zahrnutím do hlavní branche 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 lexikální analyzátor 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 lexikálního analyzátoru 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[1] 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[2], v takové situaci je nastaven `bool peek` na hodnotu `true`. Token je poté vracen do prvního volání s parametrem `peek` nastaveným na `false`.

Ve zdrojovém souboru se jedná 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 vracíme 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 v lexikálním analyzátoru 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` s příslušným hlavičkovým souborem `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 vracíme typ a hodnotu tokenu jakožto klíčové slovo či identifikátor.

[1] K nalezení v souborech `error.{c,h}`.

[2] 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.

Syntaktický analyzátor postupně od lexikálního analyzátoru 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 naleznutelné ve zdrojovém souboru jsou například `parseWhile`, `parseIf` nebo `parseBody` a další.

Jak již bylo dříve zmíněno v sekci 3.1.1 funkcí `checkPrologue` probíhá v syntaktickém analyzátoru kontrola druhé části prologu, kde ze lexikálního analyzátoru převzaté tokeny jednotlivě kontrolujeme oproti správnému formátu.

3.2.1 Precedenční syntaktická analýza

Výrazy se zpracovávají odděleně od zpracování syntaxe a to v souborech `expression.{c,h}`. Algoritmus pro zpracování výrazů byl implementován v souladu s materiály k předmětu IFJ, tedy metodou zdola nahoru řídící se dle precedenční tabulky. V přílohách se jedná o tabulku 3, podle které 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.

3.2.2 Tabulka symbolů

Tabulku symbolů[3] jsme implementovali jako tabulku s rozptýlenými položkami (hashovací tabulku). Každý index pole ukazuje na lineárně vázaný seznam prvků, které jsou přidávány do tabulky zepředu tak, že nejstarší prvky na daném indexu jsou poslední v seznamu. Klíčem do hashovací tabulky jsou samotné názvy identifikátorů.

Každý prvek tabulky kromě klíče a ukazatele na následující prvek obsahuje typ, data pro funkci a data pro proměnnou. Data pro funkci obsahují počet parametrů a ukazatel na lineárně vázaný seznam parametrů.

Hashovací funkce byla inspirována z 2. IJC projektu [1]. Použití této hashovací funkce jsme vyhodnotili jako nejvhodnější.

3.2.3 Zásobník symbolů

Pro syntaktickou a sémantickou analýzu jsme implementovali zásobník v souboru `structures.c`, jehož rozhraní je v souboru `structures.h`. Má implementovány základní zásobníkové operace jako `stackPush`, `stackPop` a `stackFree`.

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

[3] V souborech `symtable.{c,h}`.

3.3 Sémantická analýza

Sémantická analýza je prováděna podle sémantických pravidel jazyka IFJ22. Z části se vykonává zároveň se syntaktickou analýzou a z části se vykonává až po skončení syntaktické analýzy – tudíž za běhu programu. Taktéž používá strukturu `Parser`, ve které jsou potřebná data pro sémantické kontroly.

Na příslušných místech v souboru `parser.c` jsou tedy prováděny kontroly zda jsou používané funkce a proměnné definované – pokud ne, tak se pro funkce jedná o chybu 3 a pro proměnné o chybu 5. Dále jestli sedí počet/typ parametrů u volání funkce či typ návratové hodnoty z funkce, což by v případě nesplnění byla chyba 4.

Další sémantická kontrola se provádí za běhu programu. V souboru `codegen.c` kontrolujeme typovou kompatibilitu při operacích s výrazy. Při takovýchto[4] operacích musíme případně operandy přetypovat.

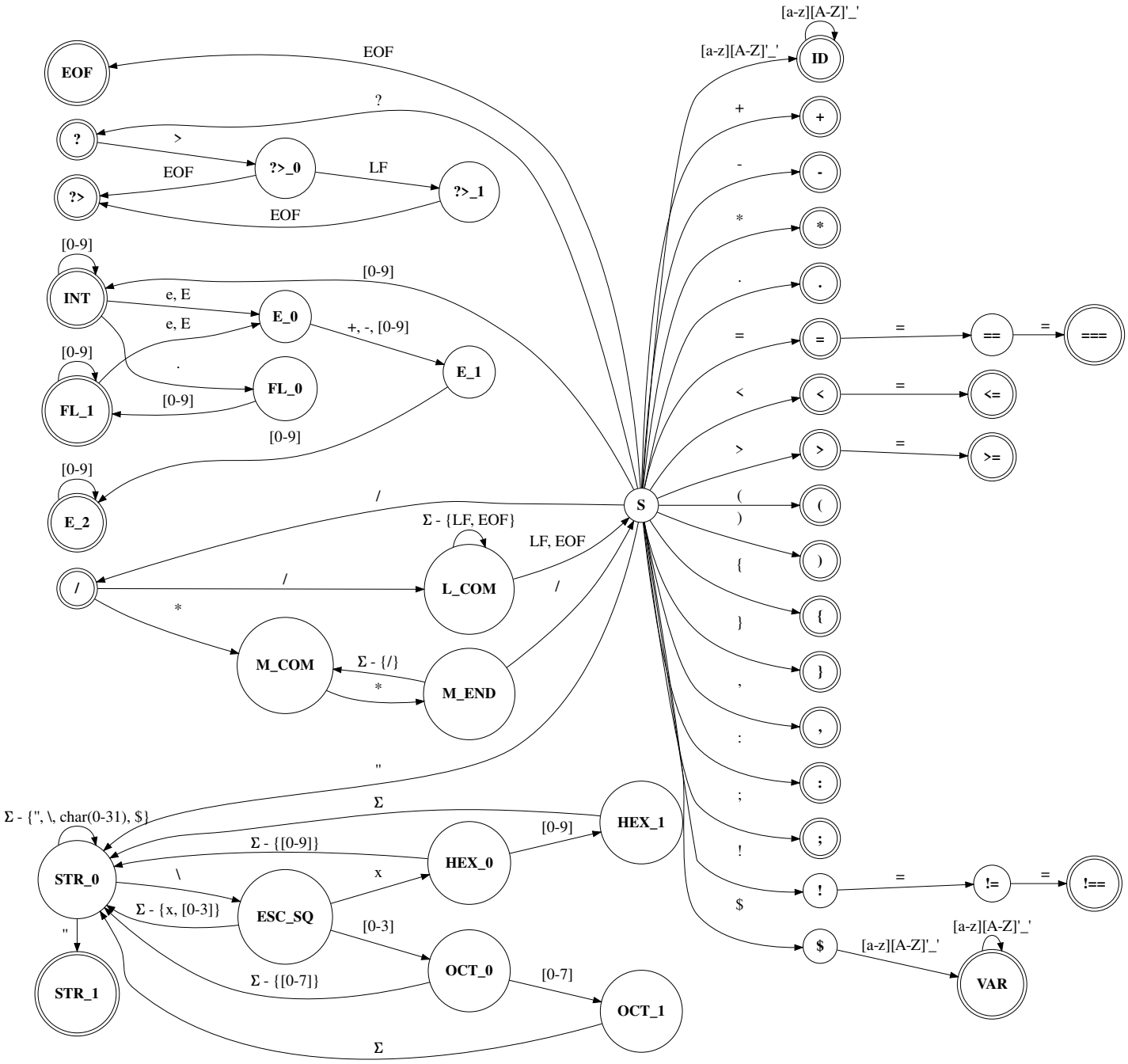
3.4 Generování cílového kódu

Generování výstupního kódu IFJcode22 je poslední částí našeho překladače. Generátor `codegen.c` se skládá z několika funkcí, které se volají z odpovídajících míst v parseru.

[4] Např. jeden operand typu float a druhý je typu integer.

4 Přílohy

4.1 Diagram konečného automatu



Obrázek 1: Konečný automat[5] specifikující lexikální analyzátor

[5]Z důvodu úspory místa a čitelnosti byly jednotlivé stavy přejmenovány, legenda stavů následuje na další stráně.

4.1.1 Legenda

S	STATE_INITIAL
ID	STATE_IDENTIFIER
+	STATE_PLUS
-	STATE_MINUS
*	STATE_MULTIPLY
.	STATE_CONCATENATE
=	STATE_EQUAL_OR_ASSIGN
==	STATE_EQUAL_0
===	STATE_EQUAL_1
<	STATE_LESSTHAN
>	STATE_GREATERTHAN
(	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
/	STATE_SLASH
L_COM	STATE_LINE_COM
M_COM	STATE_MULTILINE_COM
M_END	STATE_POT_MULTILINE_END
STR_0	STATE_STRING_0
STR_1	STATE_STRING_1
ESC_SQ	STATE_STRING_ESCAPE
HEX_0	STATE_STRING_HEXA_0
HEX_1	STATE_STRING_HEXA_1
OCT_0	STATE_STRING_OCTA_0
OCT_1	STATE_STRING_OCTA_1

4.2 LL – gramatika

1. <program> -> <body_main> <program_e>
2. <program_e> -> ε
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> -> ε
14. <body_main> -> <body> <body_main>
15. <body_main> -> <func_def> <body_main>
16. <body_main> -> ε
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> -> ε
27. <if> -> if (expr) { <pe_body> } else { <pe_body> }
28. <while> -> while (expr) { <pe_body> }
29. <params_dn> -> ε
30. <params_dn> -> , <type_n> identifier_var <params_dn>
31. <params_dp> -> <type_n> identifier_var <params_dn>
32. <params_dp> -> ε
33. <params_cnp> -> lit
34. <params_cnp> -> identifier_var
35. <params_cn> -> ε
36. <params_cn> -> , <params_cnp> <params_cn>
37. <params_cp> -> lit <params_cn>
38. <params_cp> -> identifier_var <params_cn>
39. <params_cp> -> ε
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

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

Tabulka 3: Precedenční tabulka[6] použitá při precedenční syntaktické analýze výrazů

[6]Rel značí relační operátory a Cmp je zkratka pro komparační operátory.

5 Reference

1. PERINGER, Petr. *IJC: DU2* [online]. FIT VUT, 2022. Dostupné také z: `https://www.fit.vutbr.cz/study/courses/IJC/public/DU2.html.cs`.
2. 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.
3. NYSTROM, R. *Crafting Interpreters*. USA: Genever Benning, 2021. ISBN 9780990582939.
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/`.