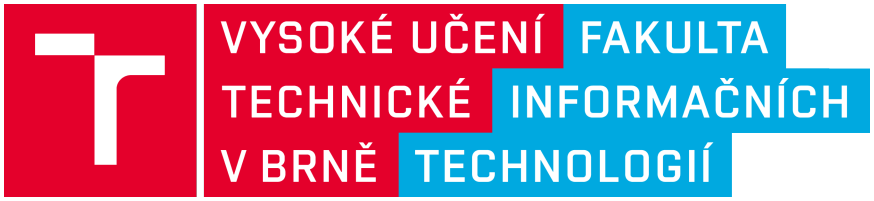


Projektová dokumentace

Implementace překladače imperativního jazyka IFJ22

Tým xnevar00, varianta BVS

Veronika Nevařilová	**(xnevar00)**	**25 %**
Patrik Michlian	(xmichl12)	25 %
Matěj Toul	(xtoulm00)	25 %
Lukáš Etzler	(xetzle00)	25 %

12. února 2023

Obsah

1 Úvod

V této dokumentaci je popsaný vývoj překladače pro jazyk IFJ22, který je zjednodušenou podmnožinou programovacího jazyka PHP. Zvolili jsme variantu BVS, která implementuje tabulku symbolů pomocí binárního vyhledávacího stromu.

2 Návrh

2.1 Lexikální analýza

První náš krok byla implementace lexikální analýzy v souboru `scanner.h` pomocí deterministického konečného automatu (DKA). Funkce `scan_lexeme` načítá znaky ze `stdin` a pomocí funkce `transition` se snaží dostat do stavu, ve kterém je možné vygenerovat token. Pokud se do tohoto stavu dostane, zavolá se funkce `generateLexeme`. Pokud se token nepodařilo vygenerovat, vrací se token s typem `SCANERROR`. V opačném případě se vrací struktura `Lexeme`, která obsahuje typ, uložený řetězec/hodnotu podle typu tokenu a číslo řádku, na kterém se nachází. Číslo řádku se ukládá pro případný přesnější chybový výstup i s číslem řádku.

Lexikální analýza se volá zvenku pomocí `get_token`. Jedná se o funkci, která má jeden parametr, inicializovaný binární vyhledávací strom, ve kterém jsou uložena potřebná klíčová slova. Protože pomocí DKA nelze vyhodnotit, zda se v řetězci vyskytuje klíčové slovo, tak pokud je typ tokenu `FUNCTION_ID`, zavolá se funkce `check_forKW`, která výskyt klíčového slova zkontroluje. Funkce `get_token` také přeskakuje tokeny typu `NULLLEX`, což jsou v našem případě komentáře.

2.2 Syntaktická analýza

Syntaktická analýza v souboru `parser.c` provádí kontrolu správnosti zápisu kódu metodou rekurzivního sestupu pomocí jednoho průchodu. Program může pracovat v jednu chvíli pouze s jedním tokenem, který mu vrátil lexikální analyzátor, proto se pro každý token podle jeho typu rozhoduje, v jakém kontextu se momentálně nachází. Jakmile dostane token, který nesedí ani do jedné povolené možnosti syntaxe jazyka IFJ22, znamená to syntaktickou chybu a parser zavolá vygenerování chyby s odpovídajícím chybovým hlášením, což je implemetováno v souboru `error.c`.

2.2.1 Precedenční syntaktická analýza

Pro korektní zpracování výrazů jsme zvolili precedenční syntaktickou analýzu (PSA). Implementace se nachází v souboru `expr_parser.c` a pro správné fungování využívá abstraktní datové struktury zásobník, a to hned dvakrát. Na prvním typu zásobníku se ukládají symboly a na druhém již zredukované výrazy. Implementace zásobníku se nachází v souboru `stack.c`.

Po tom, co proběhne předání řízení PSA, načítáme tokeny pomocí funkce `get_token`. Vyhodnocování operace (přesun, redukce, rovnost) probíhá ve funkci `check_operation` za pomoci funkce `precedence_lookup`, která vrací daný vztah symbolu na vstupu a na vrcholu zásobníku z dvourozměrného pole, jenž přímo koreluje s návrhem precedenční tabulky.

Vyhodnotí-li se, že má dojít k redukci, zavolá se funkce `check_rule`, která na základě symbolů na vstupu vyhodnotí pravidlo pro redukci. S těmito pravidly se formou výčtového typu `reduction_rule` pracuje dále při generování kódu.

Pro správné navrácení řízení jsme implementovali funkci `should_end`. Ta v závislosti na kontextu volání PSA vyhodnocuje podmínku pro ukončení, případně hlásí chybu, když výraz není ukončen korektně. Chybové hlášky generuje PSA také kdykoliv narazí na symbol, jež není syntakticky správně.

2.3 Sémantická analýza

Sémantická analýza se vykonává v průběhu syntaktické analýzy taktéž v souboru `parser.c`. Při čtení si ukládáme deklarované a definované funkce do binárního vyhledávacího stromu (BVS), jehož správa se nachází v souboru `symtable.c`. Každá funkce má svůj lokální BVS, kde se ukládají její parametry, počet parametrů, návratový typ, její lokální proměnné a další potřebná data. Kontrola deklarace probíhá pomocí vyhledávání v BVS. Všechny funkce jsou pak sdruženy do jednoho globálního stromu `globalFunctions`.

V případě chyby voláme naši knihovnu na zpracování chybového hlášení.

2.4 Generování cílového kódu v jazyce IFJcode22

Generování cílového kódu probíhá za běhu syntaktické a sémantické analýzy. Při generování kódu pro funkce se nejdříve vygeneruje kód pro tělo funkce a následně se s využitím obsahu BVS funkce vytvoří za speciálním návěštím kód pro deklaraci lokálních proměnných. Při zavolání funkce se nejdříve skočí na toto návěští a až následně se program vrátí na začátek funkce. Tímto způsobem se vyřešil problém opakované deklarace proměnných v cyklech. Každá podmínka a cyklus mají svoje unikátní číslo, které jim přiděluje parser, to se také používá při generování návěští.

Na začátku generování cílového kódu se vypíšou všechny vestavěné funkce jazyka. Ty se pak přeskočí skokem na návěští hlavního těla programu.

2.5 Generování chyb

Pro generování chyb jsme implementovali abstraktní datovou strukturu vázaný seznam. Pro maximální pohodlí probíhá hlášení chyb přes funkci `err`, která na vstupu přijímá číslo řádku, kde se chyba vyskytla (snadno lze získat ze struktury Lexeme), samotnou chybovou hlášku a v neposlední řadě také výčtový typ `error_code`. Jeho číselná reprezentace pak přímo koreluje s chybovými kódy ze zadání.

O výpis chyb se stará funkce `error_eval`, kterou voláme při návratu z funkce `main`. Postupně vypíše všechny chyby, jak šly za sebou a vrátí kód první nalezené chyby. Implementace systému chyb se nachází v `error.c`.

2.6 Použité datové struktury

Při práci s daty jsme využili datové struktury jednosměrného vázaného listu, zásobníku a binárního stromu.

Implementaci listu najdeme v souboru `error.c` a již podle názvu je určen pro generování chyb. Položky listu obsahují číslo řádku, kde se chyba vyskytla, chybovou hlášku a výčtový typ `error_code`. Pro přidání položky do listu voláme funkci `error`, pro vypsání chyb poté `error_eval`.

Datovou strukturu zásobníku jsme použili pro funkci precedenční analýzy, konkrétně dvakrát. Poprvé se jednalo o zásobník, který uchovával data pro kontrolu redukčních pravidel uvedených v precedenční tabulce. Podruhé šlo o `lex_stack`, přes který jsme poté spojili zredukované výrazy s odpovídajícími operacemi a operandy dle výstupu precedenční analýzy. Implementaci obou zásobníků najdeme v souboru `stack.c`. Implementace obsahuje např. pro zásobník typické funkce `init`, `push`, `pop` a dvě pomocné funkce pro chod precedenční syntaktické analýzy `non_terminal_check` a `push_after_terminal`.

Jako poslední byla využita datová struktura binárního vyhledávacího stromu. Jeho implementaci můžeme najít v souboru `symtable.c`. Obsahuje například funkce `node_init`, kterou inicializujeme list stromu, funkci `insert_node`, pomocí které vkládáme inicializovaný list pod námi určený list a funkci `tree_search`, pomocí které ve stromě hledáme dle řetězce `key`. Tato struktura je využita v drtivé většině programu.

2.7 Tabulka symbolů

Tabulka symbolů je implementována rekurzivně, formou binárního vyhledávacího stromu. Struktura podporuje všechny standardní operace, vyjma odebrání prvku. Tato operace pro náš projekt nebyla podstatná. Implementace se nachází v souboru `symtable.c`.

Vyhledávací strom zaobaluje strukturu data, která obsahuje podstromy pro parametry funkcí a proměnné, boolean hodnoty značící, zda jsou funkce a proměnné definované a další informace klíčové pro analýzu vstupu a generování kódu.

3 Implementace

3.1 Způsob práce v týmu

Pro komunikaci jsme výhradně používali vlastní server na platformě Discord, kde jsme měli několik místností pro zpřehlednění celé komunikace. Jako verzovací systém jsme zvolili Git, který jsme hostovali na GitHubu. Pro testování jsme využívali studenty vytvořený testovací nástroj [1], který jsme si nastavili na automatické spouštění při každém nahrání nového kódu na Git. Tímto jsme se snažili odhalit případný chybný kód hned v počátku. Na začátku projektu jsme se dohodli na pravidelných týdenních schůzkách, které se nám i podařilo dodržet.

3.2 Rozdělení práce v týmu

Každý člen týmu se podílel na programové implementaci, tvorbě dokumentace, na testování a na generování výsledného kódu.

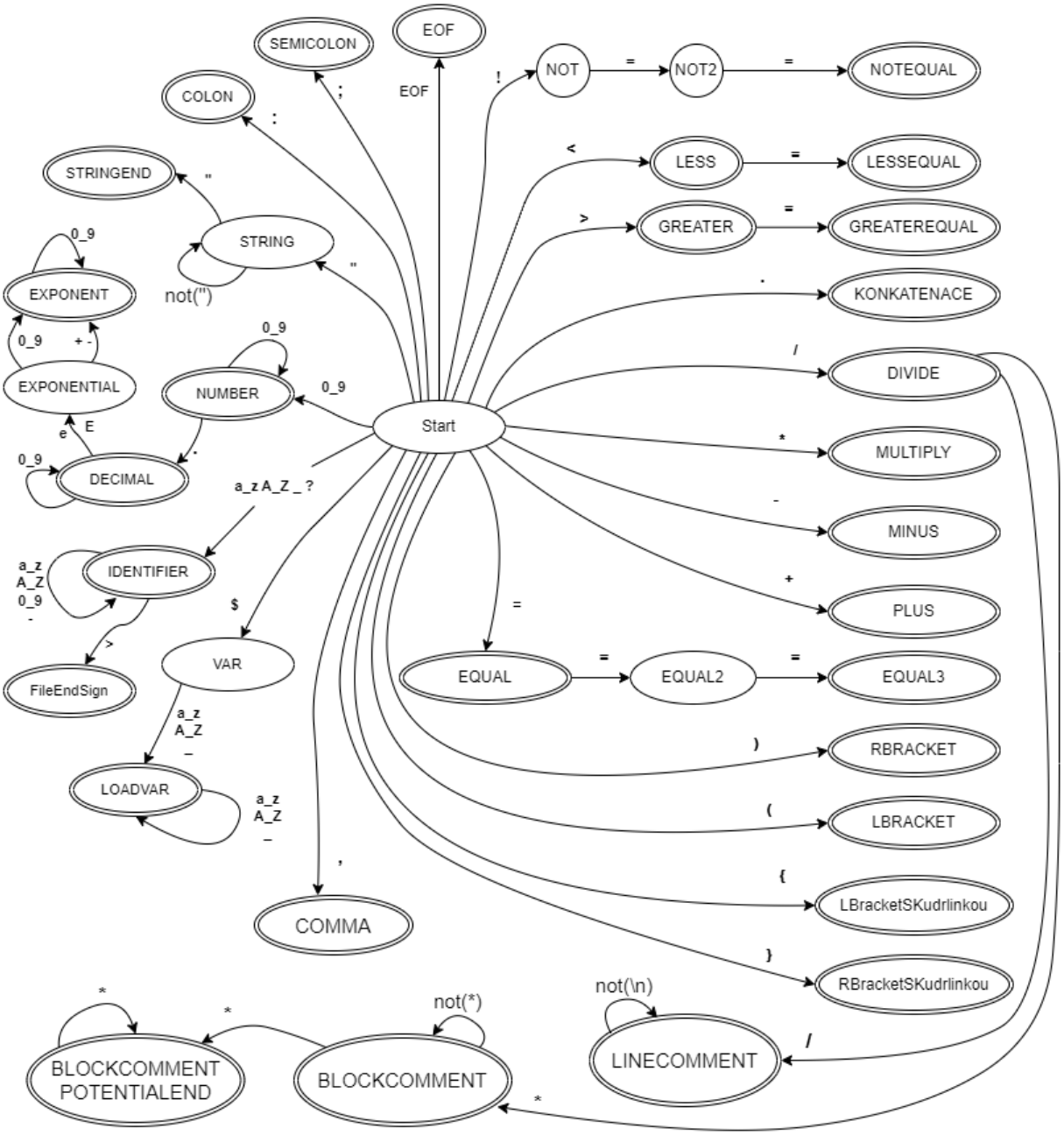
Veronika Nevařilová	Vedení týmu, návrh DKA, lexikální analýza, syntaktická analýza a sémantická analýza
Patrik Michlian	Lexikální analýza, syntaktická a sémantická analýza
Matěj Toul	Binární vyhledávací strom, návrh LL gramatiky, precedenční syntaktická analýza a generování chybových hlášení
Lukáš Etzler	Binární vyhledávací strom, návrh LL gramatiky, precedenční syntaktická analýza a generování chybových hlášení

4 Shrnutí

Projekt se zdál být na první pohled velmi náročný. Osvědčilo se nám rozdělit si ho do menších částí, vždy si dané téma nastudovat a následně aplikovat. Z projektu si odnášíme znalost toho, jak funguje překladač programovacího jazyka, zlepšení práce v jazyce C a v neposlední řadě také praktické pochopení práce s binárním vyhledávacím stromem. Pokud bychom měli příležitost dělat projekt znovu, tak bychom na něm určitě začali pracovat dříve a lépe bychom využili pokusného odevzdání.

[1] `https://github.com/galloj/IFJ22_Tester`

A Diagram konečného automatu specifikující lexikální analyzátor



Obrázek 1: Diagram konečného automatu specifikující lexikální analyzátor

B LL – gramatika

1. <prog> -> <prolog> <body>
2. <prolog> -> PROLOG declare (strict_types = 1) ;
3. <body> -> <stat> ; <body>
4. <body> -> FILE_END_SIGN LEXEOF
5. <body> -> LEXEOF
6. <body> -> KW_FUNCTION FUNCTION_ID (<decl-param>) : <type> { <st-list> } <body>
7. <body> -> <control> <body>
8. <control> -> KW_IF (<expr>) { <st-list> } KW_ELSE { <st-list> }
9. <control> -> KW_WHILE (<expr>) { <st-list> }
10. <st-list> -> <stat> ; <st-list>
11. <st-list> -> <control> <st-list>
12. <st-list> -> eps
13. <stat> -> VARIABLE_ID = <expr>
14. <stat> -> <expr>
15. <stat> -> VARIABLE_ID = FUNCTION_ID (<param>)
16. <stat> -> FUNCTION_ID (<param>)
17. <stat> -> KW_RETURN <ret-expr>
18. <type> -> KW_INT
19. <type> -> KW_FLOAT
20. <type> -> KW_STRING
21. <type> -> KW_VOID
22. <type> -> KW_OPTIONALINT
23. <type> -> KW_OPTIONALFLOAT
24. <type> -> KW_OPTIONALSTRING
25. <decl-param> -> eps
26. <decl-param> -> <type> VARIABLE_ID <decl-param2>
27. <decl-param2> -> eps
28. <decl-param2> -> , <type> VARIABLE_ID <decl-param2>
29. <param> -> eps
30. <param> -> VARIABLE_ID <param2>
31. <param2> -> eps
32. <param2> -> , VARIABLE_ID <param2>
33. <param> -> <expr>
34. <ret-expr> -> eps
35. <ret-expr> -> <expr>

Tabulka 1: LL – gramatika řídící syntaktickou analýzu

C LL – tabulka

	PROLOG	FILE_END_SIGN	LEXEOF	KW_FUNCTION	KW_IF	KW_WHILE	VARIABLE_ID	FUNCTION_ID	KW_RETURN	KW_INT	KW_FLOAT	KW_STRING	KW_VOID	KW_OPTIONALINT	KW_OPTIONALFLOAT	KW_OPTIONALSTRING	.	eps
<prog>	1																	
<prolog>	2																	
<body>		4	5	6	7	7	3	3	3									
<st-list>					11	11	10	10	10									12
<stat>							13/15	16	17									
<expr>																		
<decl-param>										26	26	26	26	26	26	26		25
<decl-param2>																	28	27
<param>							30											29
<param2>																	32	31
<type>										18	19	20	21	22	23	24		
<control>					8	9												
<ret-expr>																		34

Tabulka 2: LL – tabulka použitá při syntaktické analýze

D Precedenční tabulka

Terminál na vrcholu zásobníku	Vstupní token																
		*	/	+	-	.	<	>	<=	>=	===	!==	(	)	i	$	NULL
	*	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>	<
	/	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>	<
	+	<	<	>	>	>	>	>	>	>	>	>	<	>	<	>	<
	-	<	<	>	>	>	>	>	>	>	>	>	<	>	<	>	<
	.	<	<	>	>	>	>	>	>	>	>	>	<	>	<	>	<
	<	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>	<
	>	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>	<
	<=	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>	<
	>=	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>	<
	===	<	<	<	<	<	<	<	<	<	>	>	<	>	<	>	<
	!==	<	<	<	<	<	<	<	<	<	>	>	<	>	<	>	<
	(	<	<	<	<	<	<	<	<	<	<	<	<	=	<		<
	)	>	>	>	>	>	>	>	>	>	>	>		>		>	
	i	>	>	>	>	>	>	>	>	>	>	>		>		>	
	$	<	<	<	<	<	<	<	<	<	<	<	<		<		<
	NULL	>	>	>	>	>	>	>	>	>	>	>		>		>	

Tabulka 3: Precedenční tabulka použitá při precedenční syntaktické analýze výrazů