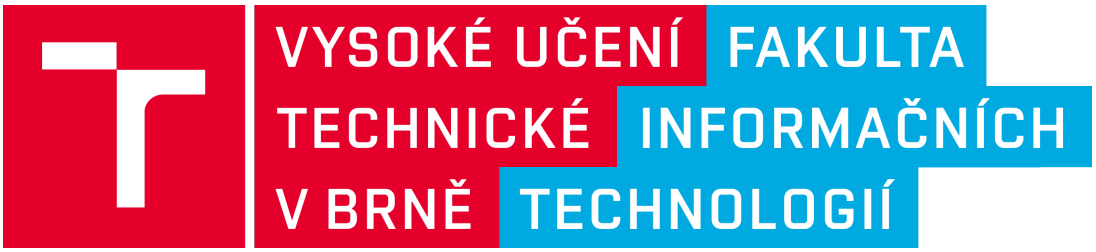


Projektová dokumentace

Implementace překladače imperativního jazyka IFJ22

Tým xmoskv01, varianta TRP

Radim Šafář (xsafar27) 25%
Lucie Jadrná (xjadrn03) 25%
Dominik Hofman (xhofma11) 25%
Jana Veronika Moskvová (xmoskv01) 25%

5. prosince 2022

Obsah

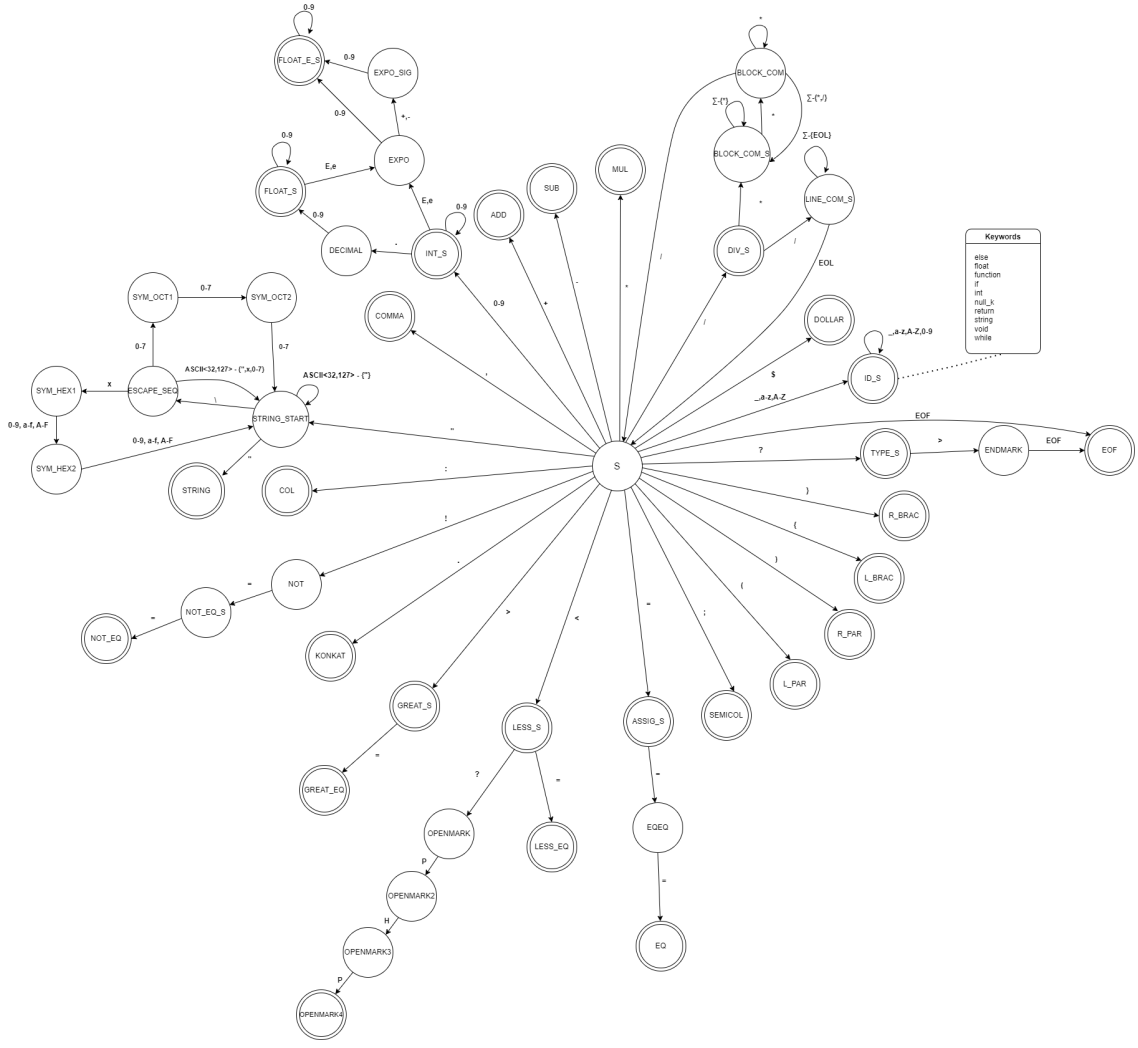
1 Úvod

Cílem projektu bylo vytvořit překladač jazyka IFJ22, založeném na programovacím jazyce PHP. Implementace projektu je v jazyce C.

2 Implementace

2.1 Lexikální analýza (Scanner)

Náš lexikální analyzátor je řízen konečným automatem. Jeho stavy jsou implementovány výčtovým datovým typem. Scanner funguje na principu načítání a analýzy lexémů, které následně jako tokeny předá parseru. Jeho implementace je v souborech `scanner.c` a `scanner.h`. Funkce `GetToken` načítá znaky ze standardního vstupu pomocí funkce `getchar()`. Tato funkce načítané znaky zpracuje v souladu s řídícím konečným automatem. Jestliže konečný automat na načítaném znaku nemůže udělat další přechod, funkce zkontroluje zda se nachází v konečném stavu, pokud ano tak vrátí token příslušné sekvence načítaných znaků, pokud se v koncovém stavu nenachází, program se ukončí s chybovou hodnotou 1 a vypíše chybovou hlášku.



Obrázek 1: Diagram konečného automatu

2.2 Syntaktická a sémantická analýza (Parser)

Pro syntaktickou analýzu byla využita metoda rekurzivního spádu. Syntaktický analyzátor si postupně volá funkci pro načtení dalšího tokenu, načtené tokeny následně porovnává s očekávanými tokeny podle LL a rozhoduje o chybě či pokračování v překladu. Jména identifikátorů, jenž byly cílovým operandem výrazu, se ukládala do dvou typů hashovacích tabulek **viz. 3.1 Tabulka symbolů**.
Vždy po načtení celého výrazu, ať už v podmínkách či nikoli, se zavolá Expression Parser, který provede následně sémantickou kontrolu výrazu a v případě úspěchu Syntaktickému analyzátoru vrací datový typ získaný z výsledku operace. Toto je velmi výhodné zejména při zpracovávání návratového typu funkce, kdy zkontrolujeme, že předpokládaný návratový typ odpovídá skutečnému návratovému typu. V průběhu celého překladu si průběžně ukládáme volání instrukcí do seznamu. Pokud celý překlad proběhne bez chyby, zavolá se funkce generátoru, která má následně za úkol projít tento seznam a vypsat instrukce na standartní výstup.

<PROG>	<?php <DECLARE> <ST_L> <OPT_EOF>
<DECLARE>	declare(strict_types=1);
<OPT_EOF>	?>
<OPT_EOF>	ε
<ST_L>	<STAT> ; <ST_L>
<ST_L>	if (<STAT_INSIDE>) { <ST_L> } <ELSE>
<ST_L>	while (\EXPRESSION_PARSER\ ) { <ST_L> }
<ST_L>	function ID (<PARAMS>) : <RET_TYPE> { <ST_L> }
<ELSE>	else { <ST_L> }
<STAT>	return
<STAT>	$ ID = <STAT_INSIDE>
<STAT_INSIDE>	\EXPRESSION_PARSER\
<PARAMS>	EPSILON
<PARAMS>	<P_LIST>
<P_LIST>	? <TYPE> <SEP>
<P_LIST>	<TYPE> <SEP>
<SEP>	ε
<SEP>	, <P_LIST>
<RET_TYPE>	? <TYPE>
<RET_TYPE>	<TYPE>
<TYPE>	int
<TYPE>	float
<TYPE>	string
<TYPE>	void

Obrázek 2: LL Gramatika

2.3 Zpracování výrazů (Expression parser)

Pro zpracování výrazů byla použita, podle zadaní, precedenční syntaktická analýza. Tento expression parser dostal od hlavního parseru seznam načtených tokenů, které již měly některá pravidla zkontrolovány. Tím byla eliminována potřeba kontrolovat některé chyby, například opakování operátorů. Po úspěšné analýze byl navrácen datový typ výrazu.

Zároveň zde probíhala generace instrukcí pro následné zpracování výrazů v generovaném kódu. Byly zvoleny zásobníkové instrukce z důvodu již prováděnému převodu výrazu do postfixu. Pro snažší generaci jsou veškeré hodnoty typu `int` zpracovány jako typ `float`. Předejte se tak případnému převádění všech operandů na jiný typ. Když je finální typ celého výrazu `int`, tak je vložena na konec zpracování instrukce na převod typu.

:)	+	-	*	/	.	<	>	<=	>=	===	!==	(	)	ID	INT	FLT	STR	NULL	$
+	>	>	<	<	>	>	>	>	>	>	>	<	>	<	<	<	X	<	>
-	>	>	<	<	>	>	>	>	>	>	>	<	>	<	<	<	X	<	>
*	>	>	>	>	>	>	>	>	>	>	>	<	>	<	<	<	X	<	>
/	>	>	>	>	>	>	>	>	>	>	>	<	>	<	<	<	X	<	>
.	>	>	<	<	>	>	>	>	>	>	>	<	>	<	<	<	<	<	>
<	<	<	<	<	<	X	X	X	X	X	X	<	>	<	<	<	<	<	>
>	<	<	<	<	<	X	X	X	X	X	X	<	>	<	<	<	<	<	>
<=	<	<	<	<	<	X	X	X	X	X	X	<	>	<	<	<	<	<	>
>=	<	<	<	<	<	X	X	X	X	X	X	<	>	<	<	<	<	<	>
!==	<	<	<	<	<	X	X	X	X	X	X	<	>	<	<	<	<	<	>
(	<	<	<	<	<	<	<	<	<	<	<	<	=	<	<	<	<	<	X
)	>	>	>	>	>	>	>	>	>	>	>	X	>	X	X	X	X	X	>
ID	>	>	>	>	>	>	>	>	>	>	>	X	>	X	X	X	X	X	>
INT	>	>	>	>	X	>	>	>	>	>	>	X	>	X	X	X	X	X	>
FLT	>	>	>	>	X	>	>	>	>	>	>	X	>	X	X	X	X	X	>
STR	X	X	X	X	>	>	>	>	>	>	>	X	>	X	X	X	X	X	>
NULL	>	>	>	>	>	>	>	>	>	>	>	X	>	X	X	X	X	X	>
$	<	<	<	<	<	<	<	<	<	<	<	<	X	<	<	<	<	<	#

Obrázek 3: Precedenční tabulka

2.4 Generace cílového kódu

Po úspěšném dokončení všech analýz přichází na řadu generace. Jednotlivé instrukce pro generaci byly vkládány do seznamu v průběhu kontrolování kódu. Generátor nejprve vygeneruje hlavičku a vestavěné funkce. Následně prochází seznam instrukcí a postupně je generuje.

Převážně se pracovalo se zásobníkem. Ať při vyhodnocování výrazů tak při předávání argumentů do funkcí a následného navrácení hodnoty.

Generování uživatelských funkcí probíhalo v hlavním těle programu, přičemž před funkcí byl vygenerován skok na návěští, které bylo za jejím koncem. Tímto způsobem se funkce přeskočila a nebyla tak spouštěna, dokud nebyla zavolána.

Funkce `write` má upravenou generaci. Funkce tiskne hodnotu na vrcholu zásobníku, dokud nenarazí na hodnotu `nil@nil`, která je na zásobník vkládána pokaždé jako nultý argument. Argumenty se na zásobník vkládaly od prvního, to způsobilo že funkce tiskla text v opačném pořadí než v jakém jí byl zadán. Proto v momentě kdy je vkládána instrukce na generaci volání `write`, tak jsou veškeré instrukce až do implicitně vložené `nil@nil` vyjmuty a vloženy v opačném pořadí.

3 Abstraktní datové struktury

3.1 Tabulka symbolů

Varianta TRP nám určila použití tabulky s rozptýlenými položkami. Byla využita poupravená implementace z předmětu IJC. Tabulek bylo využíváno více. Jedna tabulka byla určena na ukládání funkcí. Následně každá funkce, včetně hlavního těla, měla vlastní tabulku symbolů pro lokální proměnné.

3.2 Obousměrně vázaný seznam

V řešení jsou použity dva obousměrně vázané seznamy. Jeden z nich na ukládání instrukcí pro následné generování kódu `IFJcode22` a druhý pro ukládání tokenů, který je následně předáván expression parseru.

Oba seznamy jsou implementovány zvlášť, mnohem efektivnější by byla implementace pomocí maker. Stačila by tak jedna implementace pro několik datových typů.

3.3 Jednosměrně vázaný seznam hashovacích tabulek

Pro snažší generování proměnných ve funkcích jsou lokální tabulky symbolů ukládány do jednosměrně vázaného seznamu. Následně při nalezení instrukce na generaci proměnných se vezme následující tabulka ze seznamu a podle ní se vygenerují definice proměnných. Tímto způsobem se vyhneme potenciálním kolizím co se týče redefinice proměnných.

3.4 Zásobník

Zásobník je hlavní součást implementace expression parseru. Je implementovaný nad seznamem. Kromě standardních metod zásobníku dovoluje podívat se na jinou položku než je vrchol. Tato metoda je využívána při kontrole typů u redukce zásobníku.

4 Testování

K testování bylo využito několik různých přístupů. Některé části, jako zásobníky a tabulky, se daly otestovat unit testy. Postupem času se již musel projekt testovat jako celek, a tak byly sepsány různé testovací soubory shell script na jejich automatické spouštění.

V průběhu projektu byly zveřejněny vcelku propracované automatické testy na letošní projekt. Ke konci vývoje byly primárně používány právě tyto automatické testy.

5 Práce v týmu

5.1 Bodové rozdělení a rozdělení práce

Radim Šafář	xsafar27	25%	Symtable, Expression parser, Generace
Lucie Jadrná	xjadrn03	25%	Generace, Scanner
Dominik Hofman	xhofma11	25%	Parser, Generace, Error
Jana Veronika Moskvová	xmoskv01	25%	Scanner, Dokumentace

5.2 Způsob práce

Tým se pravidelně osobně scházel a zároveň probíhala komunikace přes různé internetové platformy. Osobní schůze byly primárně věnovány probrání složitějších problematik. Projekt jsme si rozdělili na větší logické celky, na kterých se primárně podílel jeden člen. Pro verzování a sdílení kódu byl zvolen systém git na platformě github.

6 Závěr

Projekt nám dal zkušenosti s rozsáhlejším programem a prací v týmu. Zároveň nám dopomohl k pochopení problematiky formálních jazyků a překladačů. Pro některé členy to byl i důvod se naučit s verzovacím systémem git, který jistě využijí v dalších projektech. Díky výběru jazyka PHP nám projekt dal i základy do dalších předmětů, jako je například IPP.