Dokumentace k projektu

Implementace překladače imperativního jazyka IFJ22

Tým **xlukas15 – varianta TRP**

Rozšíření: FUNEXP

Ondřej Lukášek (xlukas15) – Vedoucí	**25 %**	
Ondřej Koumar (xkouma02)	25 %	
Jonáš Morkus (xmorku03)	25 %	
Milan Menc (xmencm00)	25 %	7. prosince 2022

1. Úvod

Cílem našeho projektu bylo navrhnout a naimplementovat překladač v jazyce C pro jazyk IFJ22, který poté překladač přeloží do cílového jazyka IFJcode22. Jazyk IFJ22 je podmnožinou jazyka PHP. Zvolili jsme si variantu zadání, kde musíme pracovat s tabulkou symbolů pomocí tabulky s rozptýlenými položkami. K samotné implementaci jsme si přidali i rozšíření v podobě FUNEXP.

2. Lexikální analyzátor

Námi implementovaná lexikální analýza (soubory *lex.c* a *lex.h*) má za úkol postupný průchod vstupním souborem (podle volání syntaktického analyzátoru). Lexikální analyzátor přečte znaky a přidělí jim token, se kterým následně pracují další části našeho překladače.

Program rozděluje tokeny podle námi navrhnutého konečného automatu (obrázek níže) s výjimkou toho, že nejprve zkontroluje, že vstupní program obsahuje prolog, protože pokud se prolog nenačte, tak předá chybovou hlášku a ani nepřejde ke konečnému automatu. Konečný automat jsme museli během naší práce několikrát upravit, protože při testování lexikálního analyzátoru jsme narazili na chyby, které jsme v FSM neměli zahrnuté. Lexikální analyzátor obsahuje jednu základní funkci, která čte token (ReadToken). Po jednom znaku načítá každý znak podle FSM a podle něj se udělá token, případně se vyvolá chyba. Klíčová slova nenačítáme po znacích, ale načítáme rovnou celý řetězec. Taktéž načítáme jako token i epilog a podle něj zjišťujeme, jestli za ním něco následuje. Pokud by ovšem nastalo, že by se ihned po epilogu načetl EOF, tak lexikálnímu analyzátoru řekneme, že posledním tokenem je nadále epilog.

3. Syntaktický analyzátor

Návrh implementace syntaktického analyzátoru (soubory *parser.c* a *parser.h*) spočíval v tvorbě LL (1) gramatiky. Při pokusu o první převedení do LL tabulky jsme narazili na problém, a to ten, že naše gramatika nebyla LL (1), ale LL (7). Od prvního správného vytvoření LL (1) gramatiky proběhlo asi čtrnáct změn, než jsme se dostali do finální podoby, jelikož jsme nacházeli neustále další chyby anebo možnosti optimalizace. V gramatice zpracováváme i korektnost výrazu.

Samotná implementace syntaktického analyzátoru (soubory *parser.c* a *parser.h*) začala tím, že jsme si nadefinovali dvourozměrné pole, ve kterém byla celá naše gramatika, ale poté jsme přešli na klasičtější, rekurzivní sestup. V rekurzivním sestupu se volají funkce mezi neterminály a následně se postupuje podle gramatiky.

4. Sémantický analyzátor

V sémantickém analyzátoru (soubory *parser.c, parser.h, token.c, expression.c* a *expression.h*) využíváme precedenční tabulky, podle které se poté využívá dvou zásobníků, kde na jeden přijde expression od syntaktické analýzy a druhý zásobník se využívá pro vyhodnocení výrazu. Na druhý zásobník se postupně posouvají (instrukce push) části expressionu a pomocí precedenční tabulky se vyhodnocuje co se má posunout do zásobníku (instrukce push) a co nemá, aby byla zachována precedence operátorů.

Sémantická kontrola probíhá ve dvou průchodech. V prvním průchodu načteme nadeklarované/nadefinované funkce a abychom měli splněnou podmínku, že můžeme volat funkci před její deklarací/definicí. Zároveň sémantický analyzátor úzce spolupracuje s tabulkou symbolů, kde se kontrolují různé typy a jaké by měly často být např. při výstupu funkce že je typ takový jaký má být a jestli se nacházejí kompatibilní operátory mezi čísly. Nakonec se generuje kód expressionu, kde jsme využívali zásobníkových instrukcí.

5. Tabulka symbolů

Tabulku symbolů lze najít v souborech *symtable.*c a *symtable.h*. Námi zvolené řešení bylo určeno volbou zadání, tudíž tabulka symbolů je provedena pomocí hashovací tabulky, jejíž velikost je řešena přes modulo, které si určíme. Naše hashovací funkce je udělána přes CRC32 a funguje na principu, že když se provede malá změna, tak se změna zapíše na místo vzdálené od původního.

6. Generování kódu

Kód generujeme přímo bez mezikroků v souborech *generator.c, generator.h, parser.c, parser.h, expression.c* a *expression.h*. Po týmové diskusi jsme se rozhodli, že hlavní běh programu bude procházet registrovými instrukcemi a pro evaluaci výrazů použijeme instrukce zásobníkové, resp. jejich kombinaci. Každý řádek kódu je reprezentován strukturou tCodeline, kdy je jedna instrukce, a její operandy, uložena v řetězci *code* a taktéž obsahuje ukazatel na další řádek cílového kódu IFJCode22. Instrukce hlavního kódu a funkcí jsou v oddělených seznamech, aby pak ve výsledném kódu funkce byly pohromadě před hlavním kódem.

7. Testování

Pro zjištění funkčnosti jednotlivých částí jsme si napsali vlastní IFJ22 programy, které jsme využívali u jednotlivých implementovaných částí a zjišťovali korektnost. Pro finální ladění jsme si připsali několik dalších testů pro části, u kterých jsme si byli vědomi pouze částečné funkčnosti.

V pozdější fázi našeho projektu jsme se uchýlili k vytvoření skriptu umožňujícího nám automaticky testovat manuálně vytvořené testové soubory v jazyce IFJ22.

8. Práce v týmu

Na projektu jsme začali pracovat na začátku října a nejdříve jsme se drželi každotýdenního setkání a případná krátká setkání prostřednictvím aplikace Discord. Od listopadu jsme se již nescházeli pravidelně, ale namísto toho jsme se častěji scházeli na online skrze Discord a občas jsme se sešli i prezenčně.

Jako komunikační kanál jsme zvolili již zmíněný Discord. Taktéž jsme hned založili GitHub repositář, který jsme využívali ke sdílení našich kódů a materiálů potřebných k práci.

Člen	**Práce, kterou udělal nebo na které se podílel**
Ondřej Lukášek	Vedení, dělení práce, lexikální analýza, testování, konzultace, syntaktická analýza, sémantická analýza
Ondřej Koumar	Lexikální analýza, testování, konzultace, syntaktická analýza, generování kódu, makefile, sémantická analýza
Jonáš Morkus	Lexikální analýza, testování, konzultace, syntaktická analýza, generování kódu, sémantická analýza
Milan Menc	Syntaktická analýza, testování, konzultace, dokumentace

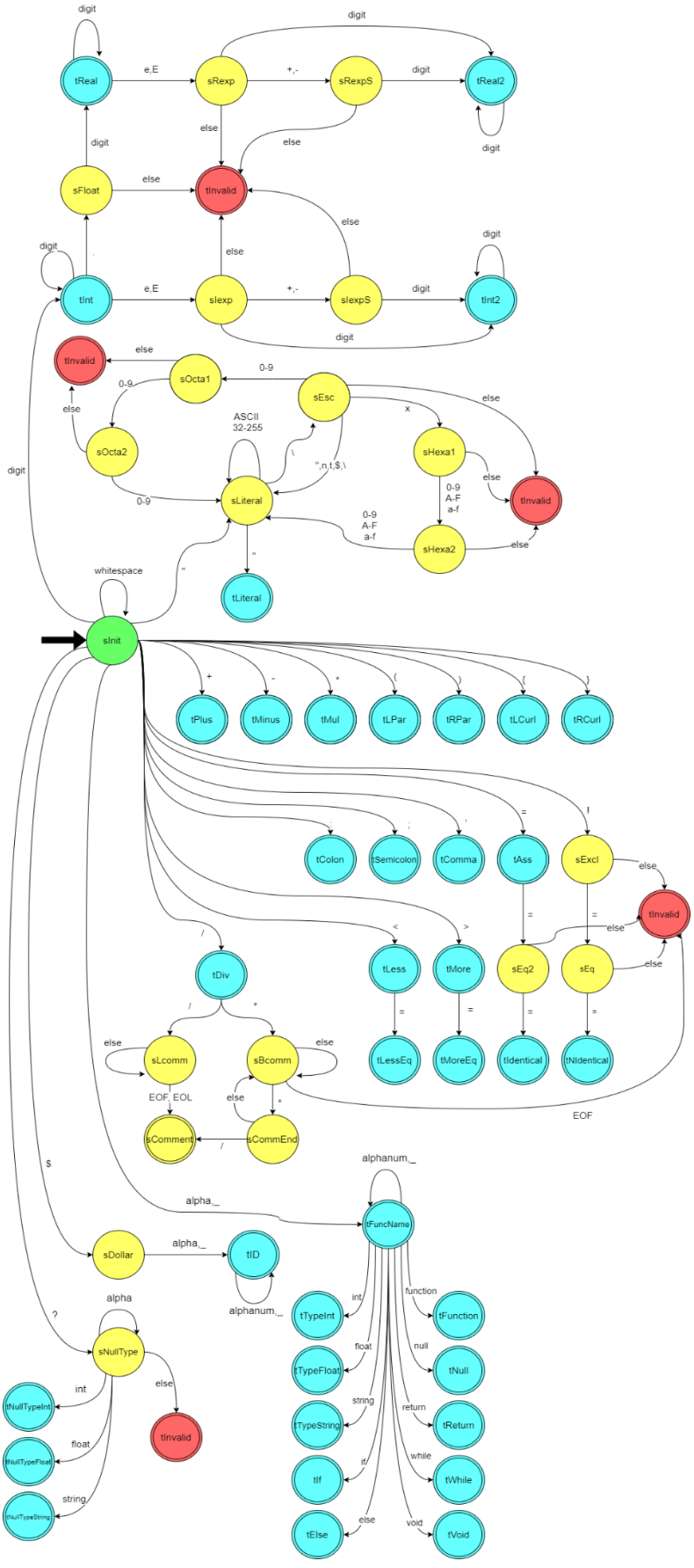
9. Speciální použité techniky a algoritmy

Pro snazší přístup k pomocným funkcím jsme si vytvořili soubory *support.h* a *support.c*.

Vytvořili jsme si náš vlastní způsob alokace, který funguje na principu mallocu, ale změna je v tom, že se všechno ukládá do jednosměrného vázaného seznamu a při konci programu se to všechno odalokuje. Jedná se o funkce *safe_malloc, safe_free* a *safe_free_all*.

Pro výpis na standardní output jsme si vytvořili funkci *dbgMsg*, která zastává funkci *printf*. Abychom nemuseli před odevzdáním mazat nepotřebné řádky, tak *dbgMsg* využívá námi definovanou globální proměnnou, která pokud se rovná 1, tak nám funkce bude vypisovat hlášky a pokud se bude rovnat nule, tak nic nevykoná.

V našem programu využíváme námi implementované funkce *rearrangeStack*. Když program narazí na unární minus (např. -5), tak to přeskládá tak, abychom s tím mohli počítat (např. „(0-5)“).



Konečný automat lexikálního analyzátoru (FSM)

LL Gramatika

programs -> program programs
programs -> ε

program -> tFunction tFuncName tLPar arguments tRPar tColon type tLCurl statements tRCurl
program -> statement

statements -> statement statements
statements -> ε

statement -> tIf tLPar expression tRPar tLCurl statements tRCurl tElse tLCurl statements tRCurl
statement -> tWhile tLPar expression tRPar tLCurl statements tRCurl
statement -> tSemicolon
statement -> tIdentifier nextTerminal
statement -> tReturn returnValue tSemicolon
statement -> preExpression

functionCall -> tFuncName tLPar parameters tRPar

returnValue -> expression
returnValue -> ε

nextTerminal -> tAssign expression tSemicolon
nextTerminal -> expression2 tSemicolon

preExpression -> tMinus minusTerm expression2 tSemicolon
preExpression -> const expression2 tSemicolon
preExpression -> functionCall expression2 tSemicolon
preExpression -> tLPar const expression2 tRPar tSemicolon

expression -> term expression2
expression -> tLPar expression tRPar expression2

expression2 -> tPlus expression
expression2 -> tMinus expression
expression2 -> tMul expression
expression2 -> tDiv expression
expression2 -> tConcat expression
expression2 -> tLess expression
expression2 -> tLessEq expression
expression2 -> tMore expression
expression2 -> tMoreEq expression
expression2 -> tIdentical expression
expression2 -> tNotIdentical expression
expression2 -> ε

arguments -> type tIdentifier argumentVars
arguments -> ε

argumentVars -> tComma type tIdentifier argumentVars
argumentVars -> ε

parameters -> expression parameters2
parameters -> ε

parameters2 -> tComma expression parameters2
parameters2 -> ε

term -> tMinus minusTerm
term -> const
term -> tIdentifier
term -> functionCall

minusTerm -> const
minusTerm -> tIdentifier
minusTerm -> functionCall

const -> tInt
const -> tReal
const -> tReal2
const -> tInt2
const -> tNull
const -> tLiteral

type -> tNullTypeInt
type -> tNullTypeFloat
type -> tNullTypeString
type -> tTypeInt
type -> tTypeFloat
type -> tTypeString
type -> tVoid

	epsilon	tFunction	tFuncName	tLPar	tRPar	tColon	tLCurl	tRCurl	tIf	tElse	tWhile	tSemicolon	tIdentifier	tReturn	tAssign	tMinus	tPlus	tMul	tDiv	tConcat	tLess	tLessEq	tMore	tMoreEq	tIdentical	tNotIdentical	tComma	tInt	tReal	tReal2	tInt2	tNull	tLiteral	tNullTypeInt	tNullTypeFloat	tNullTypeString	tTypeInt	tTypeFloat	tTypeString	tVoid
programs	2	1	1	1					1		1	1	1	1		1												1	1	1	1	1	1							
program		3	4	4					4		4	4	4	4		4												4	4	4	4	4	4							
statements	6		5	5					5		5	5	5	5		5												5	5	5	5	5	5							
statement			12	12					7		8	9	10	11		12												12	12	12	12	12	12							
functionCall			13																																					
returnValue	15		14	14									14			14												14	14	14	14	14	14							
nextTerminal	17														16	17	17	17	17	17	17	17	17	17	17	17														
preExpression			20	21												18												19	19	19	19	19	19							
expression			22	23									22			22												22	22	22	22	22	22							
expression2	35															25	24	26	27	28	29	30	31	32	33	34														
arguments	37																																	36	36	36	36	36	36	36
argumentVars	39																										38													
parameters	41		40	40									40			40												40	40	40	40	40	40							
parameters2	43																										42													
term			47										46			44												45	45	45	45	45	45							
minusTerm			50										49															48	48	48	48	48	48							
const																												51	52	53	54	55	56							
type																																	57	58	59	60	61	62	63

LL Tabulka

Precedenční tabulka		Vstup														
		*	/	+	-	.	<	>	<=	>=	===	!==	(	)	id	$
Zásobník	*	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
	/	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
	+	<	<	>	>	>	>	>	>	>	>	>	<	>	<	>
	-	<	<	>	>	>	>	>	>	>	>	>	<	>	<	>
	.	<	<	>	>	>	>	>	>	>	>	>	<	>	<	>
	<	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
	>	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
	<=	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
	>=	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
	===	<	<	<	<	<	<	<	<	<	>	>	<	>	<	>
	!==	<	<	<	<	<	<	<	<	<	>	>	<	>	<	>
	(	<	<	<	<	<	<	<	<	<	<	<	<	=	<	x
	)	>	>	>	>	>	>	>	>	>	>	>	x	>	x	>
	id	>	>	>	>	>	>	>	>	>	>	>	x	>	x	>
	$	<	<	<	<	<	<	<	<	<	<	<	<	x	<	x

Precedenční tabulka