

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

# Úvod

Cílem našeho projektu byl 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.

# Lexikální analyzátor

Námi implementovaná lexikální analýza (soubory *lex.c* a *lex.h*) má za úkol projít celý vstupní program. Lexikální analyzátor přečte celý program a rozdělí je na tokeny, se kterými 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 párkrát předělat, 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 FSM se udělá token nebo neudělá a vyvolá se 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.

# 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, dokud jsme se nedostali do finální podoby, jelikož jsme nacházeli pořád 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 rekurzivní sestup. V rekurzivním sestupu se volají funkce mezi neterminály a následně se jde postupně podle gramatiky.

# 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 jednom přijde expression od syntaktické analýzy a druhý zásobník se využívá pro vyhodnocení výrazu. Na druhý zásobník se postupně pushují části expressionu a pomocí precedenční tabulky se vyhodnocuje co se má pushnout 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.

# 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.

# 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 IFJ22.

# 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.

# Práce v týmu

Na projektu jsme začali pracovat na začátku října a nejdříve jsme se drželi každotýdenního setkání a případná krátká setkání na Discordu. Od listopadu jsme se již nescházeli pravidelně, ale namísto toho jsme se častěji scházeli na Discordu 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 |

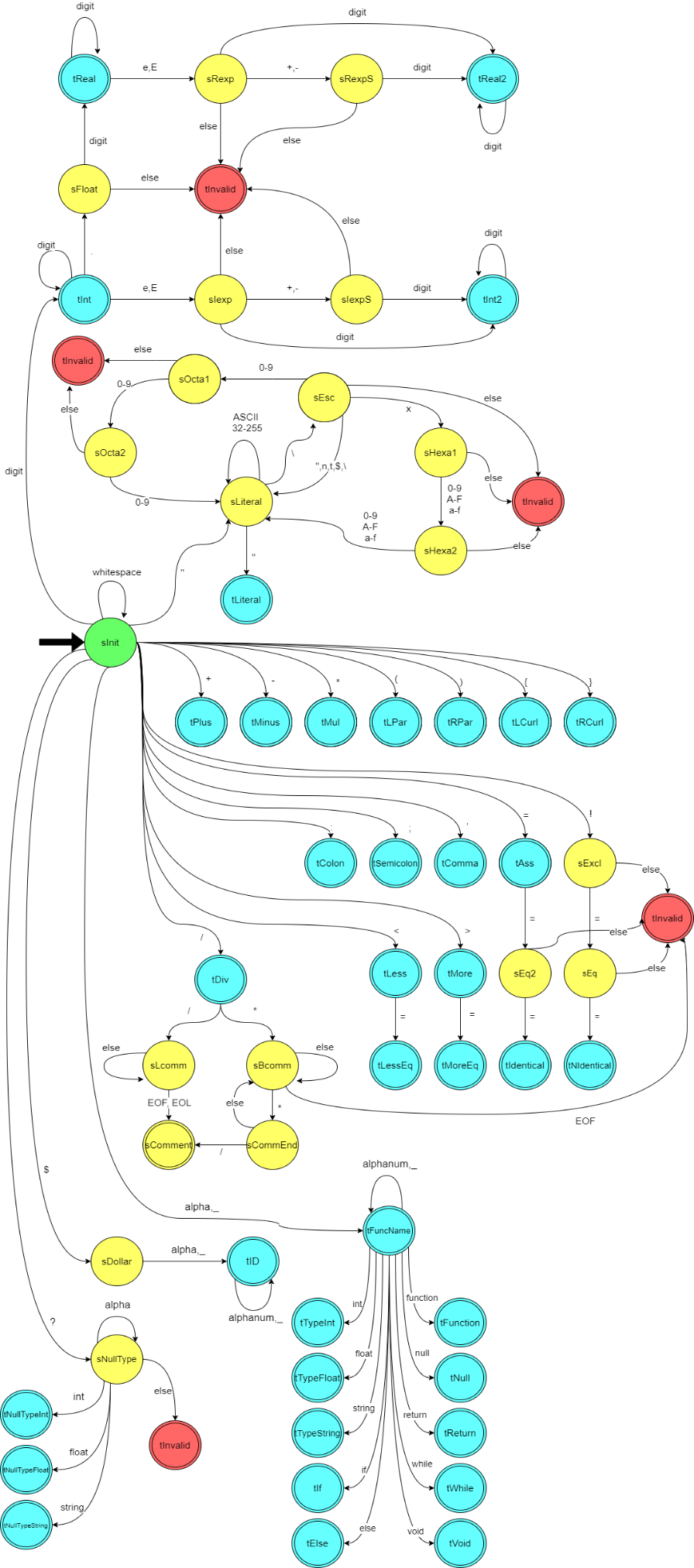
# Speciální použité techniky a algoritmy

Pro snazší přístup k pomocným funkcím jsme se 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 standartní output jsme si vytvořili funkci *dbgMsg*, která zastává funkci *prinft*. Abychom nemuseli před odevzdáním mazat nepotřebné řádky, tak *dbgMsg* využívá námi definovanou globální proměnnou, která když se rovná 1, tak nám funkce bude vypisovat hlášky a pokud ne, 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)

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

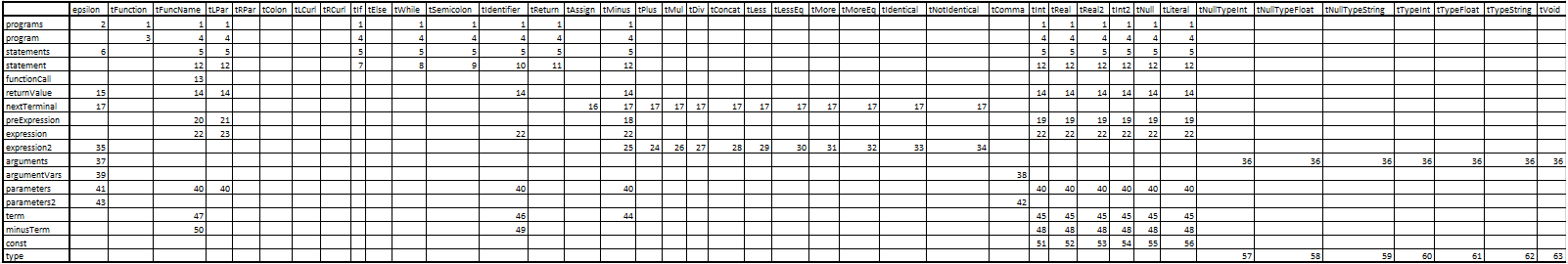
parameters2 -> tComma expression parameters2  
parameters2 -> ε

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



Obsah obrázku stůl

Popis byl vytvořen automaticky