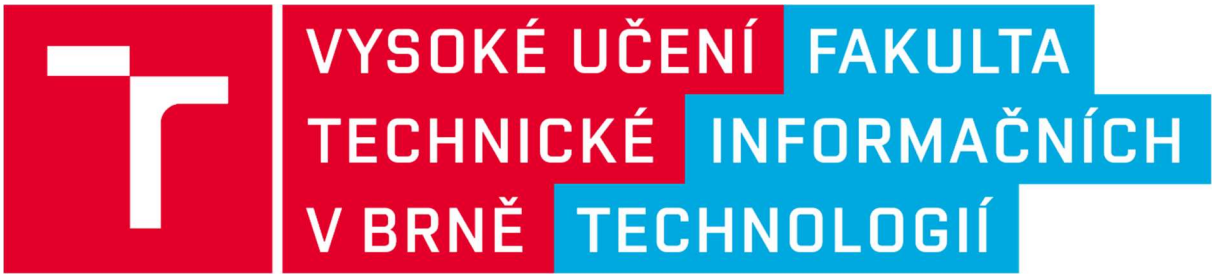



# IFJ – Dokumentace

## Týmový projekt – Překladač jazyka IFJ22.

Tým Xonder05, varianta TRP

**Daniel Onderka xonder05 - 25%**

Ondřej Bahounek - xbahou00 - 25%

Aleksandr Kasianov - xkasia01 - 25%

Tomáš Prokop - xproko49 - 25%

Brno - 7.12.2022

# Obsah

# 1. Úvod

Cílem projektu bylo vytvořit program v jazyce C, který načte zdrojový kód naspaný v jazyce IFJ22 a přeloží jej do cílového jazyka IFJcode22 (mezikód).

# 2. Návrh a implementace

## 2.1. Lexikální analýza (Scanner)

Lexikální analýza je prvním modulem překladače, jejím úkolem je zpracovat zdrojový kód načítaný ze standartního vstupu a vytvořit z něj tokeny, jednoznačné části zdrojového programu, se kterými pracuje zbytek překladače. Implementace lexikální analýzy je provedena pomocí konečného automatu. Postupně načítá jednotlivé vstupní znaky a na základě jejich hodnoty a aktuálního stavu automatu rozhoduje, jakou akci bude následně vykonávat. Tímto způsobem načítá znaky až do té doby, něž se dostane do koncového stavu a pro právě načtený znak už nevede z tohoto stavu další cesta, v tom případě byl úspěšně načten token. Druhou možností je, že skončí ve stavu, který není označený jako koncový, což značí chybně zadaný vstupní program.

Modul scanneru také převádí řetězce na řetězce cílového jazyka tzn. bíle znaky a speciální symboly na escape sekvence. Díky statické proměnné ví, kdy byl zavolán poprvé a má přijímat prolog programu. Tyto problémy nejsou pro jejich složitost zakresleny do digramu konečného automatu. Kvůli načítání libovolně dlouhého vstupu a řetězců, byl vytvořen modul pro práci s řetězci dynamické délky.

## 2.2. Syntaktická analýza shora dolů (Parser)

V první části překladu, při které se kontroluje, zda je zadaný vstupní kód zapsán lexikálně a syntakticky správně řídí veškerou činnost překladače modul syntaktického analyzátoru. Protože parser neumí pracovat přímo s načítaným programem, ale s již zpracovanými tokeny, musí vždy když potřebuje nový vstup zavolat lexikální analýzu, která mu vrátí nový token. Načtené tokeny pak porovnává s tím, co by očekával, že se může na vstupu vyskytovat. Pro zjednodušení implementace byla vytvořena LL-Gramatika popisující funkčnost analyzátoru. Samotná implementace je pak provedena metodou rekurzivního sestupu. Každému pravidlu gramatiky odpovídá jedna funkce programu. V nich se kontroluje, zda jsou terminály na správných místech a pro nonterminály se volají další funkce rekurzivního sestupu. Kromě samotné kontroly správnosti kódu, vytváří parser také výsledný syntaktický strom a záznamy do tabulky symbolů.

| | TOKEN_PROLOG | KEYWORD_FUNCTION | TOKEN_COMMA | KEYWORD_INT | KEYWORD_FLOAT | KEYWORD_STRING | KEYWORD_VOID | TOKEN_VAR_ID |
|---|---|---|---|---|---|---|---|---|
| programm | 1 | | | | | | | |
| command_or_declare_function | | 3 | | | | | | 3 |
| delcare_function | | 4 | | | | | | |
| parameters | | | 7 | 5 | 5 | 5 | 5 | |
| data_type | | | | 8 | 9 | 10 | 11 | |
| command | | | | | | | | 12 |
| call_function_or_expresion | | | | | | | | |
| term | | | 21 | | | | | |

| | TOKEN_FUNC_ID | KEYWORD_NULL | KEYWORD_IF | KEYWORD_WHILE | KEYWORD_RETURN | INT | FLOAT | STRING | VAR_ID | EPS |
|---|---|---|---|---|---|---|---|---|---|---|
| programm | | | | | | | | | | |
| command_or_declare_function | 3 | | 3 | 3 | 3 | | | | | |
| delcare_function | | | | | | | | | | |
| parameters | | | | | | | | | | |
| data_type | | | | | | | | | | |
| command | 24 | | 22 | 23 | 25 | | | | | |
| call_function_or_expresion | 14 | | | | | | | | | |
| term | | 19 | | | | 15 | 16 | 17 | 18 | 20 |

# LL – Gramatika:

program => TOKEN_PROLOG command_or_declare_function TOKEN_END_TAG
command_or_declare_function => command
command_or_declare_function => declare_function
delcare_function => KEYWORD_FUNCTION TOKEN_FUNC_ID TOKEN_L_PAR parameters TOKEN_R_PAR TOKEN_COLON
data_type TOKEN_L_BRAC command TOKEN_R_BRAC
parameters => data_type TOKEN_VAR_ID parameters
parameters => epsilon
parameters => TOKEN_COMMA parameters
data_type => KEYWORD_INT / KEYWORD_Q_INT
data_type => KEYWORD_FLOAT / KEYWORD_Q_FLOAT
data_type => KEYWORD_STRING / KEYWORD_Q_STRING
data_type => KEYWORD_VOID / KEYWORD_Q_VOID
command => command_variable => TOKEN_VAR_ID TOKEN_EQUAL call_function_or_expresion TOKEN_SEMICOLON
call_function_or_expresion => expression
call_function_or_expresion => call_function => TOKEN_FUNC_ID TOKEN_L_PAR term TOKEN_R_PAR
term => TOKEN_INT term
term => TOKEN_FLOAT term
term => TOKEN_STRING term
term => TOKEN_VAR_ID term
term => KEYWORD_NULL term
term => epsilon

term => TOKEN_COLON term
command => command_if => KEYWORD_IF TOKEN_L_PAR EXPRESSION TOKEN_R_PAR TOKEN_L_BRAC COMMAND TOKEN_R_BRAC KEYWORD_ELSE TOKEN_L_BRAC COMMAND TOKEN_R_BRAC
command => command_while => KEYWORD_WHILE TOKEN_L_PAR EXPRESSION TOKEN_R_PAR TOKEN_L_BRAC COMMAND TOKEN_R_BRAC
command => command_call_function => call_function TOKEN_SEMICOLON
command => command_return => KEYWORD_RETURN EXPRESSION TOKEN_SEMICOLON

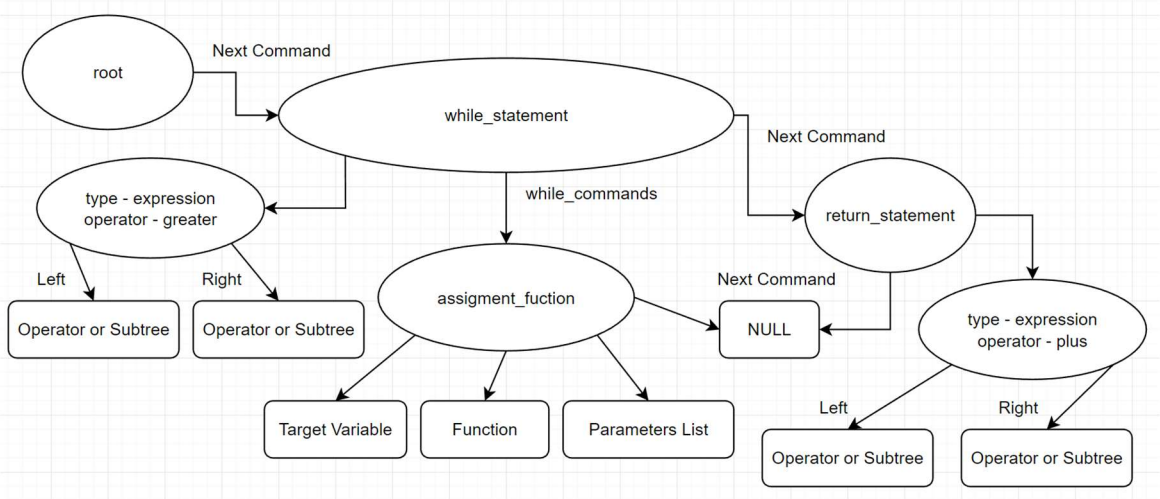
## 2.3. Precedenční syntaktická analýza

Precedenční analýza je v rámci překladače využita pro kontrolu správnosti matematických a logických výrazů. Precedenční analyzátor je volán analyzátorem shora-dolu vždy, když v kódu narazí na místo, kde by se měl vyskytovat výraz. Implementace je založena na precedenční tabulce, podle ní, vstupního symbolu a terminálu na zásobníku se určuje jaký bude další krok. Jak už tedy bylo řečeno, implementace je založena na zásobníku. Ten pracuje s datovými strukturami, které se skládají ze dvou hlavních částí, identifikátoru typu položky a dat. Identifikátory reprezentují všechny terminální znaky, které se můžou ve výrazech vyskytovat. A také několik neterminálních znaků, které slouží k reprezentaci již částečně zpracovaných výrazů, či k řízení toku programu. Na datovou sekci se poté ukládají hodnoty z načtených tokenů a postupně se zde vytváří výsledný syntaktický strom.
Jako výsledek své činnosti vrací precedenční analyzátor ukazatel na strukturu ast_t. V případě, že byl kontrolovaný výraz zapsán správně obsahuje tato struktura ukazatel na vrchol abstraktního syntaktického stromu popisujícího tento výraz. V případě neúspěchu hodnotu NULL.

| | + | - | * | / | . | === | !== | > | < | >= | <= | i | ( | ) | $ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| + | > | > | < | < | > | > | > | > | > | > | > | < | < | > | > |
| - | > | > | < | < | > | > | > | > | > | > | > | < | < | > | > |
| * | > | > | > | > | > | > | > | > | > | > | > | < | < | > | > |
| / | > | > | > | > | > | > | > | > | > | > | > | < | < | > | > |
| . | > | > | < | < | > | > | > | > | > | > | > | < | < | > | > |
| === | < | < | < | < | < | > | > | < | < | < | < | < | < | > | > |
| !== | < | < | < | < | < | > | > | < | < | < | < | < | < | > | > |
| > | < | < | < | < | < | > | > | > | > | > | > | < | < | > | > |
| < | < | < | < | < | < | > | > | > | > | > | > | < | < | > | > |
| >= | < | < | < | < | < | > | > | > | > | > | > | < | < | > | > |
| <= | < | < | < | < | < | > | > | > | > | > | > | < | < | > | > |
| i | > | > | > | > | > | > | > | > | > | > | > | | | > | > |
| ( | < | < | < | < | < | < | < | < | < | < | < | < | < | = | |
| ) | > | > | > | > | > | > | > | > | > | > | > | | | > | > |
| $ | < | < | < | < | < | < | < | < | < | < | < | < | < | | |

## 2.4. Syntaktický strom

Výsledkem činnosti syntaktické analýzy je struktura abstraktního syntaktického stromu. Každý uzel stromu v sobě nejprve uchovává informace o svém typu. Jednotlivé typy odpovídají základním konstrukcím zdrojového kódu např: přiřazení do proměnné, podmínky, cykly a funkce. Na základě této informace se určuje, jaká data bude konkrétní uzel v sobě uchovávat. V datové části pak mohou být uloženy ukazatele do tabulky symbolů, ukazatele na další uzly (příkazy) a v určitých případech jsou data uložena přímo v uzlu (bezprostřední operandy). Poslední informací ukládanou v uzlech, je odkaz na následující příkaz v bloku příkazů. Tento odkaz může nabývat také hodnotu NULL, což na nejvyšší úrovni značí konec programu.




## 2.5. Tabulka symbolů

Tabulka symbolů je implementována na základě hashovací tabulky. Pro mapovaní pozívám algoritmus GNU ELF. Každá položka tabulky obsahuje klíč v podobě řetězce, kterým je identifikátor funkce nebo proměnný. Dále obsahuje ukazatel na další prvek a data, Každý symbol má jméno, typ, context. Context udává, v jaké funkci byl symbol deklarován. Existují dva typy symbolů: proměnná a funkce. Tabulka nemůže mít více znaků se stejnou sadou parametrů. Funkce mají několik dalších parametrů: počet a typ argumentů funkce, návratový typ funkce, má return ve těle funkce a má definici. Všechny funkce musí být deklarovány, a to pouze jednou

## 2.6. Generování kódu

Generování kódu se provádí průchodem syntaktickým stromem. Pro účely ukládání generovaného kódu jsme zvolili strukturu dvojně zřetězeného spojitého seznamu, kde každý prvek nese generovaný kód a informaci o svém typu. Tato struktura byla zvolena pro možnost řešení problému, kdy se proměnná poprvé definuje v cyklu. V cílovém kódu IFJcode22 by volání instrukce DEFVAR na stejnou proměnou vícekrát vedlo na chybu. Dvojně spojitý seznam nám bez větších problémů umožní vložit definice proměnné před začátek cyklu. Generovaný kód používá instrukce pro práci se zásobníkem na vyhodnocování výrazů a proměnné pro uchovávání hodnot. Generátor kódu provádí část sémantické analýzy, tak že generovaný kód za běhu kontroluje, jestli argumenty funkce a návratová hodnota funkce odpovídá definovanému typu. Taktéž provádí dynamické přetypování a kontrolu typů u aritmetických výrazů. Kód je po dokončení vypsán na standartní výstup.

## 2.7. Členění implementačního řešení

- Lexikální Analýza – scanner.*
- Sémantická Analýza – shora-dolů parser.*
- - precedenční expression.*
- Generování Kódu - code_gen.* code_gen_build.*
- Pomocné datové struktury - dyn_string.*, stack.*, stack_ast.*, abstract_syntax_tree.* symtable.*

# 3. Vypracovávání Projektu

## 3.1. Systém pro společnou práci

Pro správu souborů jsme si zvolili systém Git. Na vzdálený repositář GitHub jsme postupně nahrávali naše řešení překladače. GitHub umožnil pracovat všem členům týmu na projektu současně. Díky tomuto repositáři nikdo z nás nemusel čekat, než někdo jiný dokončí a odevzdá svůj kód, práce mohla probíhat paralelně.

## 3.2. Komunikace týmu

V týmu jsme komunikovali přes platformu Discord. Vytvořili jsme si vlastní skupinu, ve které probíhali veškerá diskuze ohledně způsobu řešení, opravy chyb a případných nápadů, aby každý člen týmu měl možnost vidět co se právě děje a řeší. Během práce na projektu jsem se také několikrát sešli na videohovoru, při kterém jsme vyřešili důležité věci, které nešli vyřešit pouhými zprávami.

## 3.3. Rozdělení práce mezi členy

| Člen týmu | Zpracovaná část projektu |
|---|---|
| Ondřej Bahounek | lexikální analýza, generování kódu, sémantika |
| Aleksandr Kasianov | syntaktická analýza shora-dolů, tabulka symbolů |
| Daniel Onderka | syntaktická analýza precedenční, abstract syntax tree |
| Tomáš Prokop | Dokumentace, Prezentace, opravy kódu, Makefile |

Tabulka 1: Tabulka rozdělení práce

# 4. Závěr

Projekt byl velmi rozsáhlý, proto jsme na jeho řešení strávili spoustu hodin. Bylo potřeba se učit v průběhu řešení a nabývat nových z předmětů IFJ a IAL. Během tvorby projektu jsme získali spoustu nových znalostí a zkušeností. Ať už v rámci řešení rozsáhlého projektu nebo fungování v týmu.