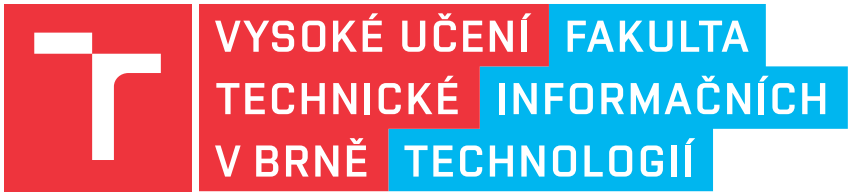



# Projektová dokumentacia

## Implementácia prekladača imperatívneho jazyka IFJ22

Tím xhorac20, Varianta BVS


| | | |
|---|---|---|
| **Andrej Horáček** | **(xhorac20)** | 23 % |
| Jakub Brčiak | (xbrcia00) | 23 % |
| Samuel Kentoš | (xkento00) | 31 % |
| Marek Pochop | (xpocho06) | 23 % |


7. decembra 2022

# Obsah

# 1 Úvod

Cieľom projektu bolo vytvoriť program v jazyku C, ktorý načíta zdrojový kód zapísaný v zdrojovom jazyku IFJ22, ktorý je zjednodušenou podmnožinou jazyka PHP a preložiť ho do cieľového jazyka IFJcode22 (medzikód).

Program funguje ako konzolová aplikácia, ktorá načíta zdrojový program zo štandardného vstupu a generuje výsledný medzikód na štandardný výstup alebo v prípade chyby vracia zodpovedajúci chybový kód.

# 2 Návrh a implementácia

Projekt sme zostavili z niekoľkých nami implementovaných čiastkových celkov, ktoré sú predstavené v tejto kapitole. Je tu tiež uvedené, akým spôsobom spolu jednotlivé čiastkové celky spolupracujú.

## 2.1 Lexikálna analýza

Pri tvorbe prekladača sme začali implementáciou lexikálnej analýzy. Hlavná funkcia tejto analýzy je `Scan`, pomocou ktorej sa číta znak po znaku zo zdrojového súboru a prevádza na štruktúru `Token`, ktorá sa skladá z typu a atribútu. Typy tokenu sú `EOF`, identifikátory, kľúčové slová, celé či desatinné číslo, reťazec a tiež porovnávacie a aritmetické operátory a ostatné znaky, ktoré môzu by použité v jazyku IFJ22. Do atribútu sa všetky hodnoty ukladaju v podobe `string`. S takto vytvoreným tokenom potom pracujú ďalšie analýzy.

Celý lexikálny analyzátor je implementovaný ako deterministický konečný automat podľa vopred vytvoreného diagramu 1. Analyzator načítava zo vstupu znak po znaku. Konečný automat je v jazyku C ako jeden `switch`, kde každý prípad `case` je ekvivalentný jednému stavu automatu. Ak načítaný znak nesúhlasí so žiadnym znakom, ktorý jazyk povoľuje, program je ukončený a vracia chybu 1. Inak sa prechádza do ďalšieho stavu a načíta sa ďalší znak, kým nemáme hotový jeden token, ktorý potom vraciame a ukončíme túto funkciu.

Pre spracovanie escape sekvencií v reťazci máme vytvorené pole o veľkosti 2 pre hexadecimalne zadanú a o velkosti 3 pre decimálne zadanú sekvenciu, ktoré je spočiatku vynulované a potom sa postupne napĺňa načítanými číslami. Nakoniec celé číslo prevedieme do ASCII podoby.

## 2.2 Syntaktická analýza

Najdôležitejšou časťou celého programu je syntaktická analýza.

Syntaktická analýza sa riadi LL – gramatikou a metódou rekurzívneho zostupu podľa pravidiel v LL – tabuľke 2. Ako prvá je volaná hlavná funkcia `Start`, ktorá rozhodne či ide o kľúčove slovo, výraz alebo volanie funkcie. Na základe toho zavolá príslušnú funkciu.

### 2.2.1 Spracovánie výrazov pomocou precedenčnej syntaktickej analýzy

Precedenčná analýza je v syntaktickej analýze definovaná a volaná ako ostatné pravidlá v LL – gramatike, ale je zvlášť implementovaná v súbore `analzyator.c`, a jej rozhranie je v súbore `analzyator.h`.

Pri spracovávaní výrazov je použitá precedenčná tabuľka 3. Keďže operátory + a – majú rovnakú asociativitu a prioritu, mohli sme ich zjednodušiť do jedného stĺpca a riadku tabuľky `+ -`. To isté sme mohli urobiť s operátormi `*` a `/` (v tabuľke stĺpec a riadok `* /`) a tiež všetkými relačnými operátormi (v tabuľke stĺpec a riadok `r`). Riadok a stĺpec `i` symbolizuje identifikátor, číselnú hodnotu alebo reťazec. Medzi povolené symboly vo výrazoch patria všetky operátory, literály, ktoré sú v tabuľke zastúpené už predtým popísaným `i`, a zátvorky. Tieto symboly sú terminály. Všetky ostatné symboly, ktoré výraz obsahovať nemôže, sú zastúpené symbolom `$`. Riadky tabuľky označujú vrchný terminál v zásobníku symbolov a stĺpce symbol v aktuálnom tokene.

Po spustení analýzy sa podľa vrchného terminálu v zásobníku symbolov a symbolu aktuálneho tokenu vykonávajú rôzne operácie. Pokiaľ je aktuálny terminál > ako posledný terminál zásobniku, zavoláme funkciu `StackPush`, ktorá daný terminál vloží do zásobnkíku. Ak je aktuálny terminál <= zavolá sa funkcia

`StackProcess`, ktorá posledné 3 tokeny zo zásobníku nahradí znakom `$`, čo znamá, že daná časť výrazu bola spracovaná. Počas vykonávania redukcie podľa stanovených pravidiel sa otestuje sémantika ľavého a pravého operandu. Analýza sa ukončuje ak na zásobníku nieje žiaden terminál.

## 2.3 Sémantická analýza

V štruktúre `ItemF` sú uložené tabuľky funkcií, ktoré obsahujú odkaz na lokálne tabuľky symbolov. Tabuľky symbolov sú implementované pomocou `BVS` a slúžia na kontrolu, či daný identifikátor existuje a či súhlasí jeho dátový typ, prípadne návratová hodnota.

## 2.4 Generovanie cieľového kódu

Generovanie cieľového kódu pre nás znamená generovanie medzikódu IFJcode22. Kód je generovaný na štandardný výstup počas všetkých analýz.

### 2.4.1 Rozhranie generátoru kódu

Generovanie kódu je implementované ako samostatný modul v súbore `generator.c`, ktorého rozhranie sa nachádza v súbore `generator.h` Rozhranie ponúka 3 základné funkcie. Ide o funkciu, ktorá pripraví generovanie kódu, táto funkcia sa volá na začiatku syntaktickej analýzy. Funkcia, ktorá uvoľní z pamäti všetky rezervované prostriedky generátorom kódu sa volá po dokončení syntaktickej analýzy. Posledná dôležitá funkcia generuje vytvorený kód na patričný výstup. Rozhranie ponúka množstvo ďalších funkcií pre generovanie jednotlivých častí programu, napr. generovanie začiatku funkcie, volanie funkcie, deklarácia premennej, funkcií na spracovanie výrazov, podmienok, cyklov, atd. Všetky tieto funkcie sa volajú v správnej chvíli a so správnymi parametrami v priebehu syntaktickej analýzy a syntaktickej analýzy výrazov.

### 2.4.2 Začiatok generovánia

Na začiatku generovania je vygenerovaná hlavička medzikódu, ktorá zahŕňa potrebné náležitosti pre korektnú interpretáciu medzikódu a skok do hlavného tela programu. Potom sú vygenerované vstavané funkcie, ktoré sú zapísané priamo v jazyku IFJcode22.

### 2.4.3 Generovanie funkcií

Každá funkcia je tvorená návestím v tvare `$function$nazov` a svojím lokálnym rámcom. Pre funkcie sa vždy generuje východisková návratová hodnota, ktorá sa ukladá na lokálny rámec, ktorý je po odchode z funkcie dostupný ako dočasný rámec. Pred zavolaním funkcie sú hodnoty parametrov uložené do dočasného rámca s číslom, ktoré zodpovedá poradiu daného parametra a po vstupe do funkcie sú všetky odovzdané parametre uložené na lokálne rámce so svojim názvom. Pri generovaní návratu z funkcie je odovzdaný výsledok spracovaného výrazu do premennej s návratovou hodnotou funkcie a vykonaný skok na koniec funkcie.

### 2.4.4 Generovanie výrazov

Všetky výrazy sa počas syntaktickej analýzy výrazov ukladajú na dátový zásobník a v pravú chvíľu sú vykonané patričné operácie s hodnotami na vrchole zásobníka. Hodnota spracovaného výrazu sa ukladá do lokálnej premennej na lokálnom rámci.

### 2.4.5 Generovanie návestia

Všetky návestia, pokiaľ neuvažujeme návestia pre funkcie a iné špeciálne návestia, sú generované v tvare `$identifikator$deep%label$`, kde `identifikator` zaistí unikátnosť návestia medzi jednotlivými

funkciami, `deep` je hĺbka zanorenia návestia (napríklad u podmienok a cyklov) a `label` určuje začiatok, koniec, prípadne štart

### 2.5 Prekladový systém

Projekt je možné preložiť, nástrojom GNU Make. ktorý vytvorý spustiteľný súbor `compiler`.

#### 2.5.1 GNU Make

Jednou z požiadaviek na odovzdanie projektu bolo priložiť k odovzdanému projektu i súbor `Makefile` a preloženie projektu príkazom `make`. Z tohto dôvodu sme vytvorili prekladový systém pomocou nástroja GNU Make.

V súbore `Makefile` sú nastavené pravidlá pre preklad. Je tu nastavený prekladač `gcc` a všetky potrebné parametre pre preklad. Vytvorili sme pravidlá, ktoré najskôr vytvoria objektové súbory zo všetkých súborov s príponou `.c` v danom adresári pomocou vytvorených automatických pravidiel a prekladača `gcc`, ktorý dokáže automaticky generovať tieto pravidlá. Následne je zo všetkých objektových súborov vytvorený jeden spustiteľný súbor.

## 3 Špeciálne algoritmy a datové štruktúry

Pri tvorbe prekladača sme implementovali niekoľko špeciálnych dátových štruktúr, ktoré sú v tejto kapitole predstavené.

### 3.1 Binárny vyhľadávací strom

Implementovali sme binárny vyhľadávací strom, ktorý slúži ako strom symbolov, čo bol nami zvolený typ zadania, ktorý súvisí s predmetom IAL. Strom sme implementovali ako štruktúra.

Každá položka BVS obsahuje unikátny kľúč v podobe reťazca, ktorým je identifikátor funkcie či premennej. Ďalej obsahuje odkaz na ľavého a pravého potomka, na koreň a dátovú časť. Dátová časť obsahuje informáciu o tom, akého dátového typu je daný symbol a zanorenie. Posledným prvkom údajovej časti, ktorý sa používa iba v prípade, že ide o funkciu, sú parametre uložené v štrúktúre `Param`, ktorá je implementovaná v podobe jednosmerne viazaného zoznamu.

Implementovali sme aj niekoľko potrebných funkcií, ktoré tvoria rozhranie pre prácu s BVS. Ide o nasledujúce funkcie: inicializácia, pridanie novej položky, pridanie určitého parametra pre konkrétnu položku stromu, vyhľadanie položky a uvoľnenie stromu z pamäte. Pri implementácii sme sa inšpirovali študijnou literatúrou predmetu IAL.

### 3.2 Zásobník symbolov pre precedenčnú syntaktickú analýzu

Implementovali sme zásobník symbolov v súbore `stack.c`, ktorý používame pri precedenčnej syntaktickej analýze. Jeho rozhranie je v súbore `stack.h`. Má implementované základné zásobníkové operácie ako`StackPush`, `StackPop`.

## 4 Práca v tíme

### 4.1 Spôsob práce v tíme

Na projekte sme začali pracovať začiatkom októbra. Prácu sme mali rozdelenú už z minulého roku (skoro). Na častiach projektu pracovali väčšinou jednotlivci alebo dvojice členov tímu. V prípade potreby sme si navzájom pomáhaly.

### 4.1.1 Verzovací systém

Na správu súborov projektu sme používali verziovací systém Git. Ako vzdialený repositár sme používali GitHub.

Git nám umožnil pracovať na viacerých úlohách na projekte súčasne v tzv. vetvách. Väčšinu úloh sme najskôr pripravili do vetvy až po otestovaní a schválení úprav ostatnými členmi tímu sme tieto úpravy začlenili do hlavnej vývojové vetvy.

### 4.1.2 Komunikácia

Komunikácia medzi členmi tímov prebiehala prevažne osobne alebo prostredníctvom aplikácie Discord, kde sme si vytvorili skupinovú konverzáciu.

V priebehu riešenia projektu sme mali i niekoľko osobných stretnutí, kde sme párovo programovali a riešili problémy týkajúce sa rôznych častí projektu.

## 4.2 Rozdelenie práce medzi členov tímu

Prácu na projekte sme si rozdelili rovnomerne s ohľadom na zložitosť a časovú náročnosť. Tabuľka 1 obsahuje rozdelenie práce v tímu mezi jednotlivými členmi.

| Člen týmu | Pridelená práca |
|---|---|
| **Andrej Horáček** | generovanie cielového kódu, dokumentácia, prezentácia, testovanie |
| Samuel Kentoš | syntaktická analýza výrazov, LL gramatika, štruktúra projektu, dokumentácia, testovanie |
| Jakub Brčiak | tabuľka symbolov, testovanie |
| Marek Pochop | sémantická analýza, testovanie |

Tabuľka 1: Rozdelenie práce v tíme medzi jednotlivými členmi

# 5 Záver

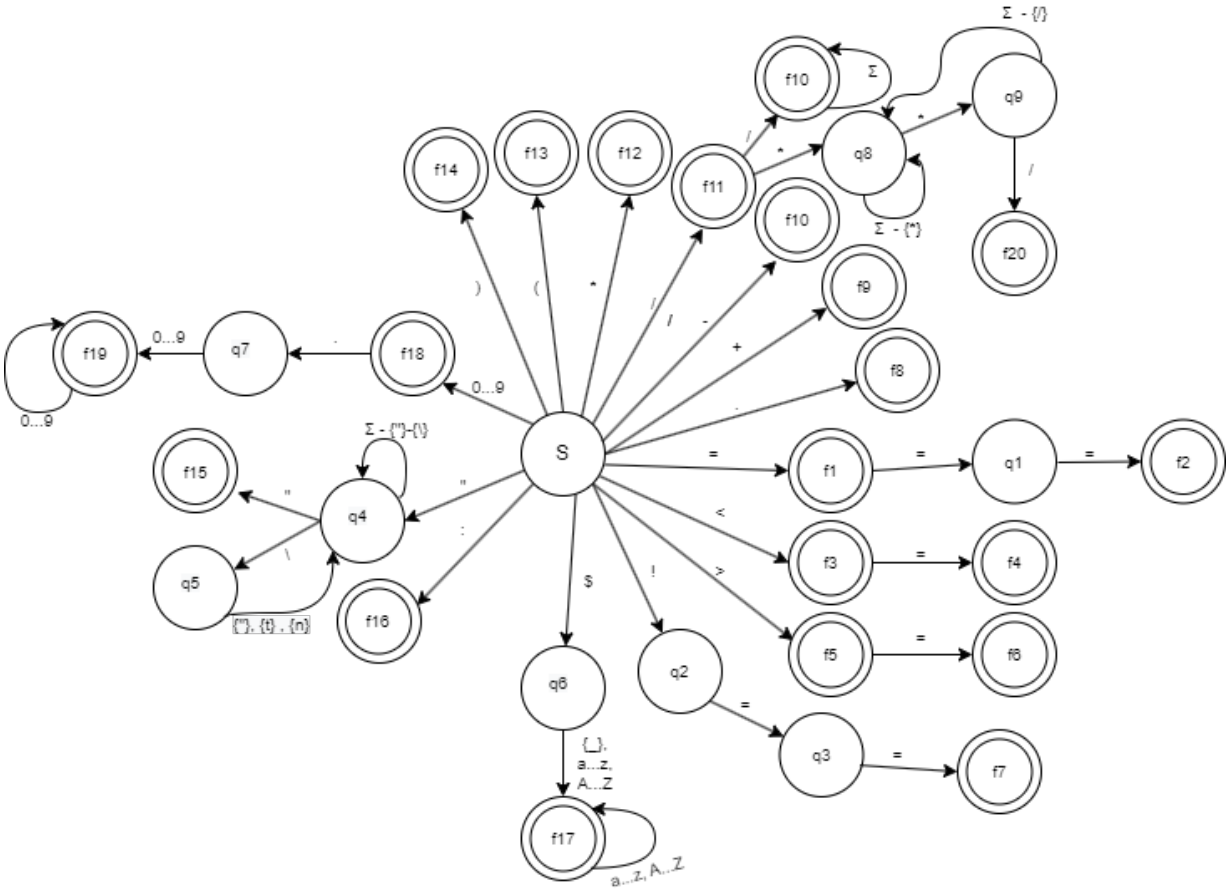
S Projektom sme boli oboznámený už z minulého roku. Takže sme sa ho snažili začať riešiť čo najskôr. Čo sa nám viac menej podarilo.

Vopred sme boli dohodnutý na komunikačných kanáloch, osobných stretnutiach a na používaní verzovacieho systému, teda sme s tímovou prácou nemali žiadny problém a pracovalo sa nám spoločne veľmi dobre.

Prácu na projekte chvíľkovo spomalili povinnosti z iných predmetov. No po všetkých polsemestrálkach a zápočtových písomkách sme boli motyvovaný a pripravený dotiahnuť projekt do zdarného konca. Jednotlivé časti projektu sme riešili individuálne a následne ich spoločne konzultovali. Vedomosti z prednášok a materiálov do predmetov IFJ a IAL boli na riešenie dostačujúce.

Tento projekt nám celkovo priniesol veľa znalostí ohľadom fungovania prekladačov, prakticky nám objasnil preberanú látku v predmetoch IFJ a IAL a priniesol nám skúsenosti s projektmi tohto rozsahu a tímovou prácou.

# A Diagram konečného automatu špecifikující lexikálny analyzátor




Obr. 1: Diagram konečného automatu špecifikující lexikálny analyzátor

# B LL – gramatika

1. <prog>-><prolog> <body>
2. <prog>  ->EOF
3. <prolog>->require <expression>
4. <body> -> $\epsilon$
5. <body> -> <def_fnc><body>
6. <body> -> <dec_fnc><body>
7. <body> -> <fnc><body>
8. <body>  -> <if>  <body>
9. <body>  -> <while>  <body>
10. <body> -> <expression><body>
11. <body> -> <dec_var><body>
12. <dec_var> -> <var_name>: <type>
13. <dec_var> -> = <var>
14. <<var> -> <expression>
15. <var> -> <fnc>
16. <var_name> -> <expression>
17. <def_fnc> -> <fnc_name> : function(<type>) : <type>
18. <dec_fnc> -> function <fnc_name>(<parameters>):<type> <body> end
19. <fnc> -> <fnc_name>(<var_name>)
20. <parameters> -> <para_name> : <type>
21. <para_name> -> <expression>
22. <type> -> integer
23. <type> -> float
24. <type> -> string
25. <type> -> null
26. <if> -> if<expression> then <body> else <body> end
27. <while> -> while <expression> do <body> end

Tabuľka 2: LL – gramatika riadiaca syntaktickú analýzu

# C Precedenčná tabuľka

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

Tabuľka 3: Precedenčná tabuľka použitá při precedenčnej syntaktickej analýze výrazu

# D Precedenčná tabuľka - skrátená

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

Tabuľka 4: Skrátená precedenčná tabuľka