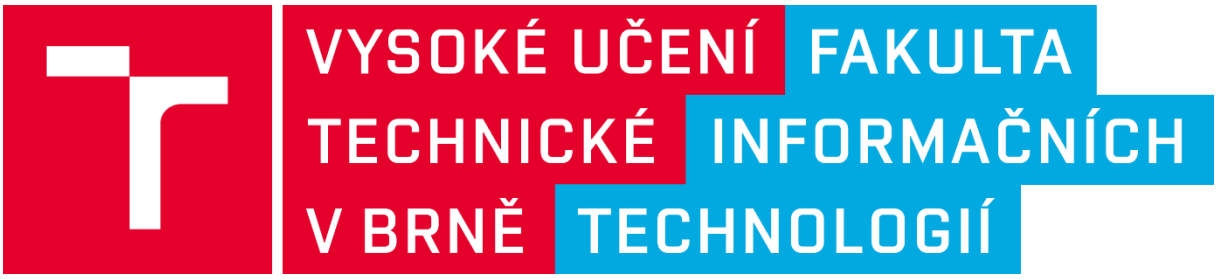



# Dokumentace k projektu z předmětů IFJ a IAL

## Implementace překladače imperativního jazyka IFJ23

Tým xstude27, varianta TRP-izp

**Jiří Studený (xstude27)** – 25%

Kryštof Samek (xsamek10) – 25%

Petr Čajánek (xcajan02) – 25%

Radek Libra (xlibra02) – 25%

# Obsah

# 1 Úvod

Tento dokument poskytuje komplexní přehled týmového projektu, jehož cílem je vývoj překladače pro imperativní programovací jazyk IFJ23, inspirovaného zjednodušenou podmnožinou jazyka Swift. Překladač načte zdrojový kód a vygeneruje výsledný mezikód v jazyce IFJcode23

# 2 Práce v týmu

Na úplném začátku práce na projektu jsme si domluvili osobní schůzku, na které jsme si stanovili způsob práce, komunikace, a prvotní rozdělení práce mezi členy týmu. Jednotlivé části projektu byli vytvořeny jednotlivcem nebo dvojicí členů týmu.

## 2.1 Komunikace

Jako hlavní nástroj naší komunikace jsme si zvolili aplikaci Discord, díky níž jsme mohli efektivně a pohodlně sdílet informace, diskutovat o úkolech a koordinovat naše aktivity. Discord jsme si vybrali především, protože každý člen tuto aplikaci již každodenně používal, tudíž zahájení komunikace týkající se projektu bylo velmi snadné.

## 2.2 Verzovací systém

Pro sdílení a správu našeho kódu jsme si vybrali Git a GitHub. Git nám umožnil snadné sledování změn a případné vracení se k předchozím verzím, kdykoliv bylo potřeba. Toto velice flexibilní řešení bylo klíčové pro efektivní práci v našem týmu, protože umožňovalo každému členovi práci na jednotlivých částech projektu nezávisle na ostatních členech. Github nám pak sloužil jako centralizované místo, kde jsme uchovávali všechen náš postup na tomto projektu.

## 2.3 Rozdělení práce

- **Jiří Studený (xstude27)** – Generování kódu, Řetězec dynamické délky
- Kryštof Samek (xsamek10) – Syntaktická analýza, Sémantická analýza
- Petr Čajánek (xcajanek02) – Lexikální analýza, Dokumentace
- Radek Libra (xlibra02) – Generování kódu, Dokumentace, Tabulka symbolů

# 3 Návrh a implementace

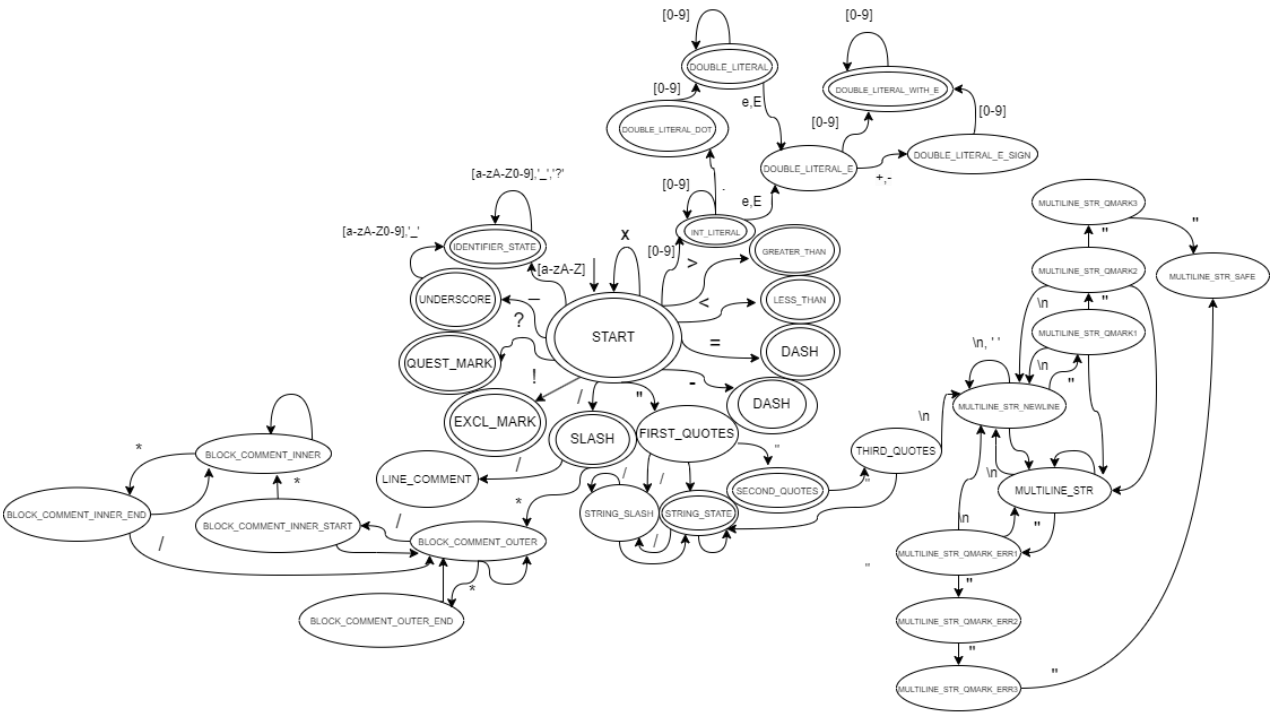
## 3.1 Lexikální analýza

Lexikální analýza jazyka IFJ23 je v programu implementována pomocí deterministického konečného automatu (DKA), který postupně čte znaky ze standardního vstupu a na základě současného stavu a dostupných přechodů, tak určí validitu tokenu (lexému), jeho typ a jeho hodnotu. DKA je implementován jako tzv. žravý, což znamená, že znaky čte a zpracovává, dokud hodnota tokenu odpovídá nějakému typu.

Analýza je v programu implementována ve dvou souborech: `scanner.h` a `scanner.c`. Hlavičkový soubor `scanner.h` obsahuje definice struktury Token a výčtových typů `STATE`, `TYPE` a deklarace funkcí použitých při provádění lexikální analýzy. Datový typ STATE definuje množinu jednotlivých stavů DKA použitého k řízení procesu lexikální analýzy, datový typ `TYPE` definuje množinu možných typů lexému, struktura Token slouží pro uložení informace o hodnotě a typu (`TYPE`) lexému. Zdrojový soubor `scanner.c` obsahuje definice funkcí deklarovaných v hlavičkovém souboru.

Principiální funkcí analýzy je funkce `get_token`, která vykonává výše zmíněné čtení a zpracovávání znaků, na základě hodnoty proměnné start, datového typu STATE, je řízena abstrakce DKA. Po přečtení znaku, po kterém by token již neodpovídal žádnému typu, tak se daný znak vrátí do proudu znaků funkcí `ungetc()` a následně zavoláním funkce Token `create_token` je vytvořena proměnná typu Token, reprezentující daný lexém.

## 3.2 Diagram konečného automatu lexikální analýzy




**X**={'(', ')', '{', '}', '+', ':', '*', '.', ','}

## 3.3 Sémantická a syntaktická analýza

Syntaktická a sémantická analýza jsou zásadní součástí procesu překladu jazyka IFJ23. V našem projektu je tento proces implementován jako analýza rekurzivním sestupem a to v souborech `parser.c` a `parser.h`. Tento parser na základě tokenů vytvořených Lexikální analýzou zajišťuje, že zdrojový kód splňuje pravidla gramatiky jazyka. V sémantické analýze se zaměřuje na význam, logiku a zda je kód sémanticky smysluplný. Dále má na starosti převod zdrojového kódu do strukturované formy, v našem případě do abstraktního syntaktického stromu (ATS).

Parser je v našem projektu pečlivě navržen tak, aby vyhovoval specifickým potřebám jazyka IFJ23. Mezi klíčové funkce parseru patří `parseExpr`, `parseStmt`, `parseTerm`, `preProcess`, `parseParams`, `parseCondition`, `parseCallParams`. Tyto funkce spolupracují s mnoha dalšími pomocnými funkcemi na správném fungování procesu, který tento parser vykonává.

### 3.3.1 Klíčové funkce

- `parseStmt` a `parseExpr`:
  - Jsou klíčové pro analýzu příkazů a výrazů v kódu. `parseStmt` rozkládá příkazy na jednotlivé uzly v AST, zatímco `parseExpr` se zabývá analýzou výrazů a správnému zpracování operátoru a priorit.
- `parseTerm`:
  - Tato funkce slouží k zpracovávání termů (základních prvků) zdrojového kódu.
- `preProcess`:
  - Tato funkce je zodpovědná za kontrolu uživatelsky definovaných funkcí, analýzu hlavního těla programu a následně těl uživatelsky definovaných funkcí.
- `parseParams`:
  - Pro analýzu parametrů uživatelsky definovaných funkcí.
- `parseCondition`:
  - Je navržena k analýze podmínek v strukturách if a while.
- `parseCallParams`:
  - Analyzuje paramenty při volání funkcí a vytváří pro ně uzly v ATS.

### 3.3.2 Pomocné funkce

- `initDet` a `getTypeExpre`:
  - Tyto funkce se starají o inicializaci symbolů a určení typu výrazů, což je velmi důležité pro správné rozpoznání a klasifikaci prvků v kódu
- `assertConsume`, `Consume` a `TryConsume`:
  - Zajišťují správné zpracování tokenů během analýzy kódu. `assertConsume` ověřuje, zda aktuální token odpovídá očekávanému typu, `Consume` posouvá analýzu vpřed a `TryConsume` zkouší spotřebovat token specifického typu.
- `freeTree`:
  - Umožňují čištění a uvolnění paměti alokované pro ATS

- `GetArg`:
    - Tato funkce zpracovává argumenty volání funkcí

- `skipFnc`:
    - Pomocná funkce pro přeskočení definice funkce. Užitečná ve fázích analýzy, kdy je potřeba se zaměřit pouze na určité části kódu.
- `parseReturn a parseReturn2`:
    - Analyzuje návratové hodnoty funkcí.
- `parseWhile`:
    - Zpracovává while cykly. Tato funkce rozkládá while cykly na AST uzly, což umožňuje další analýzu a zpracování těchto cyklů.
- `parseIf`:
    - Analyzuje if podmínky. Zajišťuje rozklad if větví a jejich else částí do struktury AST, což umožňuje další logické zpracování těchto podmínek.
- `parseFuncCall`:
    - Zajišťuje analýzu volání funkcí. Tato funkce kontroluje správnost syntaxe a sémantiky při volání funkcí a generuje příslušné uzly AST.
- `parseStmt`:
    - Univerzální funkce pro zpracování různých typů příkazů v programu. Rozkládá příkazy na jednotlivé uzly AST.
- `parseCallParams`:
    - Analyzuje parametry při volání funkcí. Tato funkce je zodpovědná za správné rozložení a zpracování parametrů funkcí do uzlů AST.
- `bin_prec`:
    - Funkce pro určení precedence binárních operátorů.
- `ChackFuncParams`:
    - Provádí kontrolu správnosti a typů parametrů funkcí. Tato funkce zajišťuje, že parametry při volání funkcí odpovídají očekávaným typům a formátům.

### 3.3.3 Pravidla gramatiky

1) Program → Function_definition,
2) Program → Statement,
3) Program → EOF
4) Function_definition → "func", Identifier, "(", List_of_parameters, ")", "->", Type, "{", Statement, "}", "end"
5) List_of_parameters → Parameter
6) List_of_parameters → Parameter, Next_parameter
7) Parameter → Parameter_Name, Identifier, ":", Type
8) Parameter → , "_", Identifier, ":", Type
9) Next_parameter → ",", Parameter
10) Type → "Double" | "Double?"
11) Type → "String" | "String?"
12) Type → "Int" | "Int?"
13) Condition → Expression, Cond, Expression
14) Statement → If_Statement, Statement
15) Statement → While_Statement, Statement
16) Statement → Definition_Statement, Statement
17) Statement → Assignment_Statement, Statement
18) Statement → Return_Statement, Statement
19) Statement → FuncCall_Statement, Statement
20) Statement → ε
21) If_Statement → "if",( "(", Condition, ")" | "let", Identifier), "{", Statement, "}", "else", "{", Statement, "}", "end"
22) While_Statement → "while", "(", Condition, ")", "{", Statement, "}", "end"
23) FuncCall_Statement → Identifier, "(", Call_parameter, ")"
24) Call_parameter → ( Parameter_Name, ":", Identifier | Identifier )
25) Definition_Statement → ("var" | "let"), Identifier, ( ":", Type, "=", Expression | ":", Type | "=", Expression )
26) Assignment_Statement → Identifier, "=", Expression
27) Return_Statement → "return", Expression
28) Identifier → ( Letter | "_" ), Identifier_Suffix
29) Identifier_Suffix → "_" | Letter | Digit
30) Parameter_Name → ( "_" | Letter ), Parameter_Name_Suffix
31) Parameter_Name_Suffix → "_" | Letter | Digit
32) Expression → (FunCall_Statement | Identifier | IntTerm | DoubleTerm | StringTerm ) , Oper, (FunCall_Statement | Identifier | IntTerm | DoubleTerm | StringTerm )
33) Letter → "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"
34) Digit → "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
35) Cond → "= =" | "!=" | "<=" | ">=" | "<" | ">"
36) Oper → "+" | "-" | "/" | "*" | "??"

### 3.3.4 LL Tabulka

| | Program | Function_definition | List_of_parameters | Parameter | Next_parameeter | Type | Condition | Statement | If_statement | While_statement | FuncCall_Statement | Call_parameter | Definition_Statement | Assigment_Statement | Return_Statement | Identifier | Identifier_suffix | Expression | Parameter_Name | Parameter_Name_Suffix | Cond | Oper |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| func | 1 | 4 | | | | | | | | | | | | | | | | | | | | |
| var | 2 | | | | | | | 16 | | | | | 25 | | | | | | | | | |
| let | 2 | | | | | | | 16 | | | | | 25 | | | | | | | | | |
| integer | | | | | | 12 | 12 | | | | | | | | | | | 32 | | | | |
| double | | | | | | 10 | 12 | | | | | | | | | | | 32 | | | | |
| string | | | | | | 11 | 12 | | | | | | | | | | | 32 | | | | |
| EOF | 3 | | | | | | | | | | | | | | | | | | | | | |
| nil | | | | | | | | 20 | | | | | | | | | | | | | | |
| return | | | | | | | | 18 | | | | | | | 27 | | | | | | | |
| if | 2 | | | | | | | 14 | 22 | | | | | | | | | | | | | |
| while | 2 | | | | | | | 15 | | 23 | | | | | | | | | | | | |
| , | | | | | 9 | | | | | | | | | | | | | | | | | |
| _ | | | 5 | 8 | | | 12 | 17 | | | 23 | 24 | | 26 | | 28 | 29 | 32 | 30 | 31 | | |
| == | | | | | | | | | | | | | | | | | | | | 41 | | |
| != | | | | | | | | | | | | | | | | | | | | 41 | | |
| <= | | | | | | | | | | | | | | | | | | | | 41 | | |
| >= | | | | | | | | | | | | | | | | | | | | 41 | | |
| < | | | | | | | | | | | | | | | | | | | | 41 | | |
| > | | | | | | | | | | | | | | | | | | | | 41 | | |
| + | | | | | | | | | | | | | | | | | | | | | | 42 |
| - | | | | | | | | | | | | | | | | | | | | | | 42 |
| / | | | | | | | | | | | | | | | | | | | | | | 42 |
| * | | | | | | | | | | | | | | | | | | | | | | 42 |
| ?? | | | | | | | | | | | | | | | | | | | | | | 42 |
| Letter | | | 5 | 7 | | | 12 | 17 | | | 23 | 24 | | 26 | | 28 | 29 | 32 | 30 | 31 | | |
| Digit | | | | | | | | | | | | | | | | | 29 | | | 31 | | |

### 3.3.5 Precedenční tabulka

| | ! | * | / | + | - | == | != | < | > | <= | >= | ?? |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ! | = | < | < | < | < | < | < | < | < | < | < | < |
| * | > | = | = | < | < | < | < | < | < | < | < | < |
| / | > | = | = | < | < | < | < | < | < | < | < | < |
| + | > | > | > | = | = | < | < | < | < | < | < | < |
| - | > | > | > | = | = | < | < | < | < | < | < | < |
| == | > | > | > | > | > | = | = | < | < | < | < | < |
| != | > | > | > | > | > | = | = | < | < | < | < | < |
| < | > | > | > | > | > | = | = | = | < | < | < | < |
| > | > | > | > | > | > | = | = | < | = | < | < | < |
| <= | > | > | > | > | > | = | = | < | < | = | < | < |
| >= | > | > | > | > | > | = | = | < | < | < | = | < |
| ?? | > | > | > | > | > | > | > | > | > | > | > | = |

## 3.4 Generování cílového kódu cílového kódu

Jako poslední část procesu překladače je pak generování výsledného kódu v jazyce IFJcode23. Tento podproces překladače je implementován v souboru `codegen.c` a zabývá se transformací abstraktního syntaktického stromu (ATS) do spustitelného kódu v zadaném jazyce. Hlavní funkce zajišťující tuto transformaci je funkce `generateCode.`

Funkce `generateCode` postupně prochází ATS a na základě typu každého uzlu generuje příslušný kód. Toto je pak implementováno pomocí switche, který rozlišuje jednotlivé uzly, mezi které patří například EXPR_NODE, RETURN_NODE, DEF_NODE a mnoho dalších. Každý z těchto uzlů vyžaduje implementování specifické logiky. U složitějších uzlu je tato logika zajištěna příslušnými funkcemi.

### 3.4.1 Funkce pro zpracování uzlů

- `generateFunction`:
  - Zabývá se generováním kódu pro definici funkcí, včetně nastavení rámce pro lokální proměnně a generování kódu pro tělo funkce.
- `generateReturn`:
  - Generuje kód pro návratové příkazy funkcí.
- `generateDefinition` a `generateAssign`:
  - Tyto funkce se zabývají generováním kódu pro definice a přiřazení hodnot proměnných.
- `generateExpression`:
  - Funkce pro generování kódu pro výrazy, podporuje různé typy operací a výrazů.
- `generateLogicalExpression`:
  - Zajišťuje generování logických výrazů, jako jsou porovnání a logické operace.
- `generateWhileStatement`:
  - Zabývá se generováním kódu pro while struktury.
- `generateIfStatement`:
  - Generuje kód pro if-else struktury.
- `generateCall`:
  - Tato funkce je určena pro generování kódu pro volání funkce.
- `generateParameter`:
  - Zaměřuje se na generování kódu pro parametry funkcí.

### 3.4.2 Vestavěné funkce

Jako jeden z podmínka zadání toho projektu bylo implementování vestavěných funkcí. V naší implementaci jsme toto vyřešili tak, že definice těchto vestavěných funkcí je vytisknuta na začátek výsledného kódu v jazyce IFJcode23 i v případě, že se tyto funkce v daném programu nepoužívají. Tyto vestavěné funkce jsou implementovány v souboru `codegen.c`.

- `readString`, `readInt`, `readDouble`:
  - Tyto funkce slouží k načtení hodnot různých typů.
- `Int2Double, Double2Int`:
  - Tyto funkce zajišťují konverzi mezi celými čísly a desetinnými čísly.
- `chr`:
  - Tato funkce konvertuje celé číslo na odpovídající ASCII znak
- `lenght`:
  - Tato funkce vrací délku řetězce
- `ord`:
  - Vrací ASCII hodnotu prvního znaku řetězce
- `substring`:
  - Vytváří podřetězec ze zadaného řetězce na základně počátečního a konečného indexu
- `write`:
  - Tato funkce zajišťuje výpis hodnot

## 3.5 Tabulka symbolů

V rámci projektu IFJ23 byla implementována symbolická tabulka, která je zásadní pro správu identifikátorů a atributů symbolů během procesu překladu. Tabulka a operace nad ni jsou definovány v souboru `symtab.h` a implementace jednotlivých operací v `symtab.c`. Tabulka je strukturována jako hash tabulka o velikosti `TABLE_SIZE`, což umožňuje efektivní vyhledávání a minimalizaci kolizí.

Každý symbol v tabulce je reprezentován jako uzel `SymbolNode`, který obsahuje klíč symbolu, podrobnosti o symbolu a ukazatel na další uzel v případě kolizí hash hodnot. Podrobnosti o symbolu `SymbolDetails` zahrnují datový typ symbolu (např. `INTEGER_TYPE, DOUBLE_TYPE`, atd.), pravdivostní hodnotu indikující, zda byl symbol definován, parametry funkce, pokud je symbol funkce, jméno symbolu a úroveň oboru platnosti, která se dědí z tabulky.

Tabulka symbolů (`SymbolTableStruct`) obsahuje úroveň oboru platnosti, pole ukazatelů na symboly a ukazatel na nadřazenou tabulku, což umožňuje správu oblastí viditelnosti.

Hlavní funkce tabulky zahrnují `computeIndex` pro výpočet hash indexu z klíče, `initSymbolTable` pro inicializaci prázdné tabulky, `deleteSymbolTable` pro odstranění, `addSymbol` pro přidávání nových symbolů, `retrieveSymbol` pro vyhledávání symbolů a `removeSymbol` pro odstranění symbolů z tabulky.

Tato struktura a funkce poskytují robustní a efektivní způsob správy symbolů v překladači, což je nezbytné pro správnou funkcionalitu lexikálních, syntaktických a sémantických analýz vstupního programového kódu. Implementace symbolické tabulky tedy hraje klíčovou roli v celkovém návrhu a výkonu překladače IFJ23.

### 3.6 Řetězec dynamické délky

V rámci projektu IFJ23 byla vyvinuta struktura `InfStr`, která je určena pro práci s nekonečnými řetězci. Tato struktura, definovaná v hlavičkovém souboru `infStr.h` a implementovaná v souboru `infStr.c`, je to klíčová struktura pro flexibilní manipulaci s textovými řetězci, které se mohou dynamicky měnit a rozšiřovat během zpracování kódu. Struktura `InfStr` se skládá z ukazatele na znakové pole a celočíselné proměnné určující délku řetězce, což umožňuje flexibilní a efektivní manipulaci s textem.

Hlavní funkce poskytované touto strukturou zahrnují vytváření nové instance `InfStr`, uvolňování paměti, čištění obsahu a další operace jako přidávání znaků, spojování a kopírování řetězců. Tyto funkce poskytují širokou škálu nástrojů pro práci s textovými řetězci, což je zásadní pro účinné zpracování zdrojového kódu.

# 4 Závěr

V rámci tohoto týmového projektu jsme prokázali schopnost efektivní spolupráce, inovativního myšlení a technické dovednosti. Podařilo se nám úspěšně splnit všechny stanovené cíle a úkoly, přičemž každý člen týmu přispěl unikátními znalostmi a perspektivou. Tato zkušenost nejen že posílila naše technické a organizační dovednosti, ale také nás naučila cenné lekce o důležitosti týmové práce, adaptability a vzájemné podpory v dynamickém prostředí.