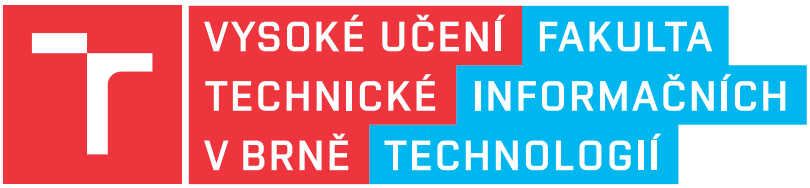



# Dokumentace projektu

## Implementace překladače imperativního jazyka IFJ23

Tým xsalta01, varianta vv-BVS

Autory:

Nadzeya Antsipenka xantsi00 - 23%
Lilit Movsesian xmovse00 - 23%
**Veranika Saltanava xsalta01** - 23%
Kirill Shchetiniuk xshche05 - 31%

6. prosince 2023

# Obsah


**1 Úvod** **2**

**2 Implementace** **2**
2.1 Lexikální analýza . . . . . 2
2.2 Syntaktická analýza . . . . . 2
2.2.1 Reseni nedeterminismu v LL-tabulce . . . . . 3
2.2.2 Rozhodovaní o ukončeni výrazu . . . . . 3
2.3 Sémantická analýza . . . . . 3
2.4 Generování kódu . . . . . 3
2.4.1 Začátek generování . . . . . 3
2.4.2 Generování funkce . . . . . 4
2.4.3 Generování výrazu . . . . . 4
2.4.4 Generování volání funkce . . . . . 4
2.5 Speciální algoritmy a datové struktury . . . . . 4
2.5.1 Zásobník . . . . . 4
2.5.2 Dynamické pole . . . . . 4
2.5.3 Dynamický řetězec . . . . . 5
2.5.4 Zdrojový soubor . . . . . 5
2.5.5 Výškově vyvážený binární vyhledávací strom . . . . . 5
2.6 Rozšíření na prémiové body . . . . . 5
2.6.1 BOOLTHEN . . . . . 5
2.6.2 CYCLES . . . . . 5
2.6.3 FUNEXP . . . . . 5

**3 Práce v týmu** **6**
3.1 Rozdělení práce mezi členy týmu . . . . . 6

**4 DKA** **7**

**5 LL-gramatika** **8**

**6 LL-tabulka** **10**

**7 Precedenční tabulka** **11**

**8 Použité zdroje** **12**

# 1 Úvod

Cílem tohoto projektu bylo vytvořit program v jazyce C, který načte zdrojový kód zapsaný ve zdrojovém jazyce IFJ23 a přeloží jej do cílového jazyka IFJcode23.

# 2 Implementace

Překladač je sestaven z několika dílcích částí, které jsou popsané dále.

## 2.1 Lexikální analýza

Vypracování projektu jsme začali tvorbou diagramu deterministického konečného automatu(viz DKA), popisujícího lexikální analýzu. Na základě společně vypracovaného a odsouhlaseného diagramu jsme zahájili práce na implementaci projektu. Implementace lexikálního analyzátoru je v souboru `scanner.c`, pomocný modul pro práce s tokeny je implementován v souboru `token.c`. Lexikální analýza v projektu je provedena v funkci `source_code_to_tokens`, kde je realizován konečný automat, který převádí zdrojový kód na pole tokenů. Jednotlivé tokeny ukládáme jako inicializovanou strukturu `token_t`. Automat je realizován pomocí příkazu `switch`. Načtené symboly přidáváme do lexému a v každém konečném stavu lexém resetujeme. Identifikátory od klíčových slov oddělujeme porovnáním načteného lexému s klíčovými slovy. Skript zpracovává řádkové a blokové komentáře na základě načtených symbolů '/' a '*', taktéž podporuje vnořené blokové komentáře. Pokud automat narazí na symbol odřádkováni, nastaví u předešlého tokenu položku `has_newline_after` na hodnotu `true`. Pokud automat vrátí řetězcový literál, zavolá se funkce `verify_str`, která zpracovává řetězce a převádí je do formátu, který akceptuje interpret. Pokud řetězcový literál je víceřádkový, funkce `verify_str` zpracovává odřádkovaní a případně odstraňuje zbytečné bílé symboly na začátku řádku. Případné escape sekvence jsou převedeny na desítkový ASCII kód. Výsledkem provedené lexikální analýzy jsou tokeny, se kterými dál pracujeme v rámci projektu. Komunikace lexikálního analyzátoru s polem tokenů je provedena pomocí interface `TokenArray`.

## 2.2 Syntaktická analýza

Dále jsme společně sestavili LL-gramatiku (viz LL-gramatika). Syntaktický analyzátor je implementován na základě metody rekurzivního sestupu. Pro implementaci jsme použili LL-tabulku (viz LL-tabulka). Implementaci syntaktického analyzátoru máme v souboru `parser.c`. Pro každý non-terminal jsme vytvořili funkci se `switch` příkazy, ve kterých se rekurzivně zpracovávají jednotlivá pravidla. K zpracování termninalu používáme funkci `match`, která porovnává současný token s požadovaným tokenem a nastaví následující token jako současný. Syntaktická analýza se provádí ve dvou průchodech. Během prvního průchodu se sbírají všechny definice a deklarace funkcí do tabulky symbolů. Ve druhém průchodu provádíme sémantické kontroly a generování kódu. Analýzu výrazu provádíme na základě precedenční analýzy. Implementace je založená na precedenční tabulce (viz Precedenční tabulka). Analýza výrazu

je implementována ve souboru `expr_parser.c`. Pro získání indexu v precedenční tabulce používáme funkci `map_token`. Poté podle dvou indexu vyhledáváme jednotlivou akci, kterou musíme provést. Akci 'shift' odpovídá hodnota '1', 'reduce' - '2', 'equal' - '3', 'error' - '4'. Pro kontrolu místa deklarace funkce máme proměnné `inside_func`, `inside_loop` a `inside_branch`. Deklarace funkcí je povolena pouze v hlavním tělu programu mimo cykly a podmíněné příkazy. Pro kontrolu, že příkaz `return` je uvnitř funkce, používáme proměnnou `inside_func`. Pro kontrolu, že non-void funkce obsahuje alespoň jeden příkaz `return` používáme proměnnou `has_return`.

### 2.2.1 Reseni nedeterminismu v LL-tabulce

Během návrhu gramatiky se objevil problém nedeterminismu při zpracování non-terminálu `<CALL_PARAM>`, tento problém jsme vyřešili tak, že nejprve zkusíme to zpracovat podle pravidla číslo `48`. Pokud se to nepodaří, zkoušíme to zpracovat podle pravidla číslo `49`.

### 2.2.2 Rozhodovaní o ukončeni výrazu

Kvůli tomu, že v `IFJ23` mohou byt výrazy na více řádcích a zároveň používáme konec řádku jako oddělovač jednotlivých přikážu, potřebovali jsme nějak rozhodnout zda je konec řádku zároveň i koncem výrazu. Na to používáme první token na následujícím řádku, přídáváme jej do výrazu a sledujeme, zda analyzátor výrazu vrátí chybu. Způsobí-li přidání tokenu chybu, ukončujeme analýzu výrazu, jinak pokračujeme.

## 2.3 Sémantická analýza

Pro sémantickou analýzu používáme tabulku symbolů a zásobník tabulek symbolů. Tabulka symbolu je implementována v souboru `symtable.c`. Jednotlivé tabulky symbolů jsou implementovány pomocí výškově vyváženého binárního vyhledávacího stromu. V rámci sémantické analýzy ověřujeme správnost použití proměnných, konkretně jestli jsou definovány a deklarovány. Pro uložení funkcí používáme vlastní tabulku symbolů, která je určena jenom pro funkce. Během inicializace tabulky symbolů přidáváme do ní vestavěné funkce. Pomocí tabulky symbolů pro funkce ověřujeme správnost volání funkce a správnost typu návratového příkazu. Správnost volání ověřujeme pomocí funkce `check_func_signature`, která porovnává požadovanou a získanou signatury volání.

## 2.4 Generování kódu

Za cílový kód považujeme IFJcode23. Kód je generován na standardní výstup pří druhém průchodu syntaktického analyzátoru. Jednotlivé funkce pro generování kódu jsou implementovány ve souboru `codegen.c`. Generování kódu provádíme hned na standardní výstup programu pomoci funkce `gen_line`.

### 2.4.1 Začátek generování

Na začátku generování kódu generujeme header a pomocné globální proměnné. Hned poté generujeme nepodmíněný skok na hlavní tělo programu. Mezi skokem a hlavním

tělem programu generujeme vestavěné funkce. Pro generace začátku programu voláme funkci `gen_header` a `gen_std_functions`.

### 2.4.2 Generování funkce

Před začátkem každé funkce generujeme nepodmíněný skok až za konec funkce pro obcházení funkce. Poté generujeme návěští, které odpovídá jménu funkci a vytváříme nový rámec. Dál generujeme nepodmíněný skok na definice lokálních proměnných, použitých během vykonání funkce. Hned za tímto skokem máme návěští pro návrat z definice lokálních proměnných a generujeme načtení parametrů v opačném pořadí z vrcholu zásobníku. Poté generujeme příkazy uvnitř funkce. Pří naražení na příkaz `return`, v případě non-void funkce přirazíme do proměnné `GF@$RET` výsledek návratového výrazu, generujeme zrušení rámce a návrat z funkce. V případě void funkce a naražení na poslední příkaz, generujeme hned za nim zrušení rámce a návrat z funkce. Za koncem funkce generujeme návěští pro definice lokálních proměnných. Generujeme definice proměnných a zpáteční skok do načteni parametru. Hned za tím generujeme návěští pro obcházení funkce.

### 2.4.3 Generování výrazu

Během syntaktické analýzy výrazu ukládáme všechny použité ve výrazech proměnné na zásobník. V případě funkce jako součásti výrazů generujeme voláni funkce a dáváme návratovou hodnotu funkce (hodnotu proměnné `GF@$RET`) na zásobník.

### 2.4.4 Generování volání funkce

Rozhodli jsme, že budeme předávat všechny parametry funkcí přes zásobník v původním pořadí. Dále generujeme volání podle uvedeného ve zdrojovém kódu jména.

## 2.5 Speciální algoritmy a datové struktury

### 2.5.1 Zásobník

Jako pomocnou strukturu pro implementaci tabulky symbolů a syntaktického analyzátoru výrazu jsme použili zásobník. Zásobník je implementován ve souboru `stack.c` jako spojový seznam, kde vrchol zásobníku je první prvek seznamu. Zásobník umožňuje ukládaní libovolného ukazatele na cokoliv. Operace nad zásobníkem umožňuje interface `Stack`.

### 2.5.2 Dynamické pole

Jako pomocnou strukturu pro různé případy jsme v souboru `list.c` implementovali dynamické pole, které umožňuje ukládaní ukazatelů na jakékoliv datové typy. Dynamické pole je použito pro ukládaní alokovaných ukazatelů pro další jednoduchou správu paměti. Operace nad polem umoznuje interface `DynamicArray`.

### 2.5.3 Dynamický řetězec

Jako alternativu poli znaku jsme implementovali dynamické řetězce pro jednoduchou práci s řetězci. Implementace dynamických řetězců je ve souboru `string_util.c`. Operace nad řetězcem umožňuje interface `String`.

### 2.5.4 Zdrojový soubor

Pro jednoduchou práci se zdrojovým kódem jsme implementovali datovou strukturu pro zdrojový kód. Implementace struktury je ve souboru `source_file.c`. Komunikace se zdrojovým kódem je provedena pomoci interface `SourceCode`.

### 2.5.5 Výškově vyvážený binární vyhledávací strom

Tabulky symbolů použité v rámci sémantické analýzy jsou implementovány pomocí výškově vyváženého binárního vyhledávacího stromu.

## 2.6 Rozšíření na prémiové body

### 2.6.1 BOOLTHEN

Realizovali jsme podporu typu `Bool` a `Bool?`, booleovských literálů `true` a `false` a základních booleovských operátorů `!`, `&&` a `||`. Dále jsme podpořili rozšířený podmíněný příkaz `if - else if - else` a udělali `else if` a `else` nepovinnými.

### 2.6.2 CYCLES

Podpořili jsme cyklus `for`, příkazy `break` a `continue`. Rozsah cyklu `for` musí být celočíselnou hodnotou.

### 2.6.3 FUNEXP

Podpořili jsme volání funkce jako součást libovolných výrazu, výrazy mohou být v parametrech volání funkce.

# 3 Práce v týmu

Hned na začátku jsme rozhodli, že nad všemi součástmi projektu budeme pracovat společně, aby každý pochopil, co se děje v každé části.

Jako verzovácí systém jsme použili Git, jako vzdálený repozitář GitHub. Pro spoluprací jsme použili technologie `Code With Me` od `JetBrains`. Jako vývojové prostředí jsme použili `IDE CLion`.

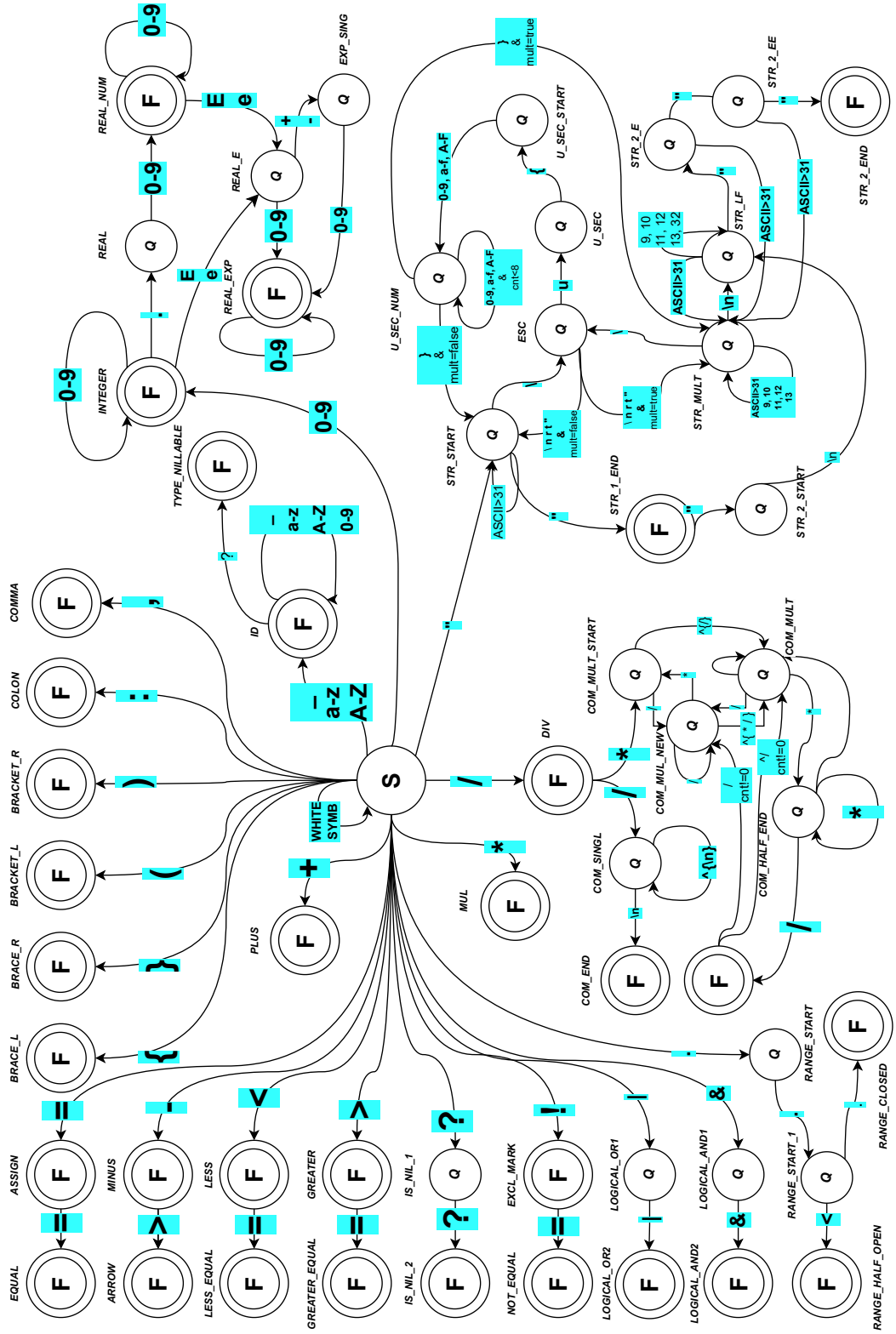
## 3.1 Rozdělení práce mezi členy týmu

Tabulka 1 shrnuje rozdělení práce v týmu mezi jednotlivými členy.

| Člen týmu | Přidělená práce |
|---|---|
| Nadzeya Antsipenka | lexikální analýza, syntaktická analýza, LL-gramatika, LL-tabulka, precedenční tabulka dokumentace, prezentace |
| Lilit Movsesian | lexikální analýza, syntaktická analýza, LL-gramatika, LL-tabulka, precedenční tabulka, tabulka symbolů (bvs), dokumentace, prezentace |
| **Veranika Saltanava** | vedení týmu, organizace práce, dohlížení na provádění práce, LL-gramatika, LL-tabulka, precedenční tabulka, syntaktická analýza, dokumentace |
| Kirill Shchetiniuk | generování cílového kódu, sémantická analýza, syntaktická analýza, syntaktická analýza výrazů, implementace pomocných datových struktur, LL-gramatika, LL-tabulka, precedenční tabulka, testování, rozšíření BOOLTHEN, CYCLES, FUNEXP |

Tabulka 1: Rozdělení práce v týmu mezi jednotlivými členy

# 4 DKA




Obrázek 1: DKA

# 5 LL-gramatika

Jednotlivé non-terminály jsou označené jako <NON-TERMINAL>, terminály jsou označené jako "terminal", konec souboru je označen jako $, počáteční non-terminál je označen jako <S>.

1. <S> -> <CODE> $

2. <CODE> -> <VAR_DECL> "eol" <CODE>
3. <CODE> -> <LET_DECL> "eol" <CODE>
4. <CODE> -> <FUNC_DECL> "eol" <CODE>
5. <CODE> -> <WHILE_LOOP> "eol" <CODE>
6. <CODE> -> <FO_LOOP> "eol" <CODE>
7. <CODE> -> <BRANCH> "eol" <CODE>
8. <CODE> -> <ID_CALL_OR_ASSIGN> "eol" <CODE>
9. <CODE> -> <RETURN>
10. <CODE> -> "break"
11. <CODE> -> "continue"
12. <CODE> -> $\varepsilon$

13. <RETURN> -> "return" <RET_EXPR>
14. <RET_EXPR> -> <EXPR>
15. <RET_EXPR> -> $\varepsilon$

16. <VAR_DECL> -> "var" "id" <VAR_LET_TYPE> <VAR_LET_EXP>
17. <LET_DECL> -> "let" "id" <VAR_LET_TYPE> <VAR_LET_EXP>
18. <VAR_LET_TYPE> -> ":" <TYPE>
19. <VAR_LET_TYPE> -> $\varepsilon$
20. <VAR_LET_EXP> -> "=" <EXPR>
21. <VAR_LET_EXP> -> $\varepsilon$

22. <FUNC_DECL> -> "func" "id" "(" <PARAM_LIST> ")" <RET_TYPE> "{" <CODE>
"}"
23. <RET_TYPE> -> "->" <TYPE>
24. <RET_TYPE> -> $\varepsilon$
25. <PARAM_LIST> -> <PARAM> <NEXT_PARAM>
26. <PARAM_LIST> -> $\varepsilon$
27. <PARAM> -> <PARAM_NAME> "id" ":" <TYPE>
28. <PARAM_NAME> -> "id"
29. <PARAM_NAME> -> "_"
30. <NEXT_PARAM> -> "," <PARAM> <NEXT_PARAM>
31. <NEXT_PARAM> -> $\varepsilon$

32. <BRANCH> -> "if" <BR_EXPR> "{" <CODE> "}" <ELSE>
33. <BR_EXPR> -> <EXPR>
34. <BR_EXPR> -> "let" "id"
35. <ELSE> -> "else" <ELSE_IF>

36. <ELSE> -> $\varepsilon$
37. <ELSE_IF> -> "if" <BR_EXPR> "{" <CODE> "}" <ELSE>
38. <ELSE_IF> -> "{" <CODE> "}"
39. <ELSE_IF> -> $\varepsilon$

40. <WHILE_LOOP> -> "while" <EXPR> "{" <CODE> "}"

41. <FOR_LOOP> -> "for" <FOR_ID> "in" <EXPR> <RANGE> "{" <CODE> "}"
42. <FOR_ID> -> "id"
43. <FOR_ID> -> "_"
44. <RANGE> -> "..." <EXPR>
45. <RANGE> -> "..<" <EXPR>

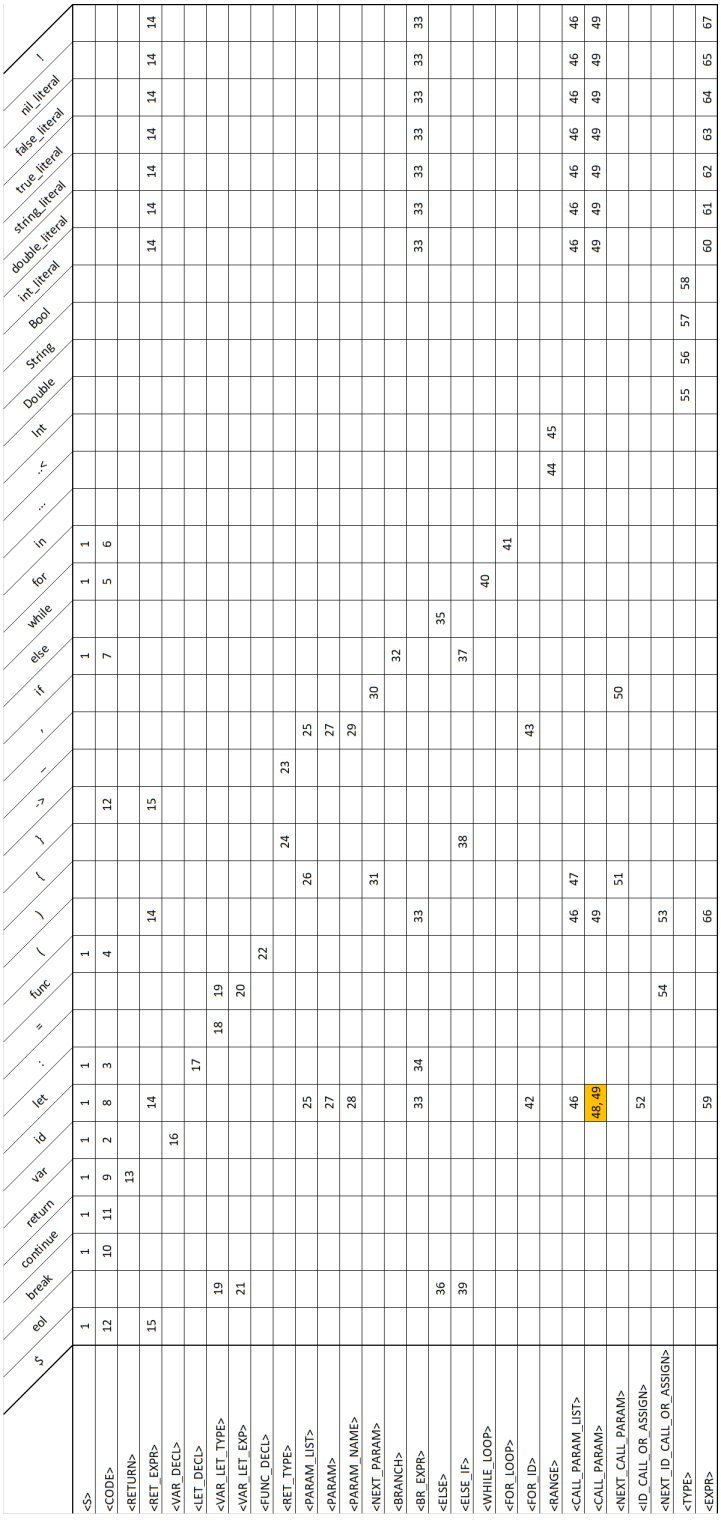
46. <CALL_PARAM_LIST> -> <CALL_PARAM> <NEXT_CALL_PARAM>
47. <CALL_PARAM_LIST> -> $\varepsilon$
48. <CALL_PARAM> -> "id" ":" <EXPR>
49. <CALL_PARAM> -> <EXPR>
50. <NEXT_CALL_PARAM> -> "," <CALL_PARAM> <NEXT_CALL_PARAM>
51. <NEXT_CALL_PARAM> -> $\varepsilon$

52. <ID_CALL_OR_ASSIGN> -> "id" <NEXT_ID_CALL_OR_ASSIGN>
53. <NEXT_ID_CALL_OR_ASSIGN> -> "(" <CALL_PARAM_LIST> ")"
54. <NEXT_ID_CALL_OR_ASSIGN> -> "=" <EXPR>

55. <TYPE> -> "Int"
56. <TYPE> -> "Double"
57. <TYPE> -> "String"
58. <TYPE> -> "Bool"

59. <EXPR> -> "id"
60. <EXPR> -> "int_literal"
61. <EXPR> -> "double_literal"
62. <EXPR> -> "string_literal"
63. <EXPR> -> "true_literal"
64. <EXPR> -> "false_literal"
65. <EXPR> -> "nil_literal"
66. <EXPR> -> "("
67. <EXPR> -> "!"

# 6 LL-tabulka



| | $ | eol | break | continue | return | var | id | let | : | = | func | ( | ) | { | } | -> | _ | , | if | else | while | for | in | ... | ..< | Int | Double | String | Bool | int_literal | double_literal | string_literal | true_literal | false_literal | nil_literal | ! |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| <S> | 1 | | | 1 | 1 | 1 | 1 | 1 | 1 | | | 1 | | | | | | | | 1 | | 1 | 1 | | | | | | | | | | | | | |
| <CODE> | 12 | | | 10 | 11 | 9 | 2 | 8 | 3 | | | 4 | | | | 12 | | | | 7 | | 5 | 6 | | | | | | | | | | | | | |
| <RETURN> | | | | | | 13 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| <RET_EXPR> | 15 | | | | | | | 14 | | | | | 14 | | | 15 | | | | | | | | | | | | | | 14 | 14 | 14 | 14 | 14 | 14 | 14 |
| <VAR_DECL> | | | | | | | 16 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| <LET_DECL> | | | | | | | | | 17 | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| <VAR_LET_TYPE> | | 19 | | | | | | | | 18 | 19 | | | | | | | | | | | | | | | | | | | | | | | | | | |
| <VAR_LET_EXP> | | 21 | | | | | | | | | 20 | | | | | | | | | | | | | | | | | | | | | | | | | | |
| <FUNC_DECL> | | | | | | | | | | | | 22 | | | | | | | | | | | | | | | | | | | | | | | | |
| <RET_TYPE> | | | | | | | | | | | | | | 24 | | | 23 | | | | | | | | | | | | | | | | | | | | |
| <PARAM_LIST> | | | | | | | | 25 | | | | | | | | | | 25 | | | | | | | | | | | | | | | | | | | |
| <PARAM> | | | | | | | | 27 | | | | | | 26 | | | | 27 | | | | | | | | | | | | | | | | | | | |
| <PARAM_NAME> | | | | | | | | 28 | | | | | | | | | | 29 | | | | | | | | | | | | | | | | | | | |
| <NEXT_PARAM> | | | | | | | | | | | | | 31 | | | | | | 30 | | | | | | | | | | | | | | | | | | |
| <BRANCH> | | | | | | | | | | | | | | | | | | | | 32 | | | | | | | | | | | | | | | | | |
| <BR_EXPR> | | | | | | | | 33 | 34 | | | | 33 | | | | | | | | | | | | | | | | | 33 | 33 | 33 | 33 | 33 | 33 | 33 | 33 |
| <ELSE> | | 36 | | | | | | | | | | | | | | | | | | | 35 | | | | | | | | | | | | | | | | |
| <ELSE_IF> | | 39 | | | | | | | | | | | | 38 | | | | | | 37 | | | | | | | | | | | | | | | | | |
| <WHILE_LOOP> | | | | | | | | | | | | | | | | | | | | | | 40 | | | | | | | | | | | | | | | |
| <FOR_LOOP> | | | | | | | | | | | | | | | | | | | | | | | 41 | | | | | | | | | | | | | | |
| <FOR_ID> | | | | | | | | 42 | | | | | | | | | | 43 | | | | | | | | | | | | | | | | | | | |
| <RANGE> | | | | | | | | | | | | | | | | | | | | | | | | | 44 | 45 | | | | | | | | | | | |
| <CALL_PARAM_LIST> | | | | | | | | 46 | | | | | 46 | 47 | | | | | | | | | | | | | | | | 46 | 46 | 46 | 46 | 46 | 46 | 46 | 46 |
| <CALL_PARAM> | | | | | | | | 48, 49 | | | | | 49 | | | | | | | | | | | | | | | | | 49 | 49 | 49 | 49 | 49 | 49 | 49 | 49 |
| <NEXT_CALL_PARAM> | | | | | | | | | | | | | | 51 | | | | | 50 | | | | | | | | | | | | | | | | | | |
| <ID_CALL_OR_ASSIGN> | | | | | | | | 52 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| <NEXT_ID_CALL_OR_ASSIGN> | | | | | | | | | | | | 54 | 53 | | | | | | | | | | | | | | | | | | | | | | | | |
| <TYPE> | | | | | | | | | | | | | | | | | | | | | | | | | | 55 | 56 | 57 | 58 | | | | | | | |
| <EXPR> | | | | | | | | 59 | | | | | 66 | | | | | | | | | | | | | | | | | 60 | 61 | 62 | 63 | 64 | 65 | 67 |

Obrázek 2: LL(1)-tabulka

# 7 Precedenční tabulka

| | unwrap | not | mul | add | ?? | rel | && | \|\| | ( | i | ) | $ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| unwrap | | > | > | > | > | > | > | > | | | > | > |
| not | < | < | > | > | > | > | > | > | < | < | > | > |
| mul | < | < | > | > | > | > | > | > | < | < | > | > |
| add | < | < | < | > | > | > | > | > | < | < | > | > |
| ?? | < | < | < | < | < | > | > | > | < | < | > | > |
| rel | < | < | < | < | < | | > | > | < | < | > | > |
| && | < | < | < | < | < | < | > | > | < | < | > | > |
| \|\| | < | < | < | < | < | < | < | > | < | < | > | > |
| ( | < | < | < | < | < | < | < | < | < | < | = | |
| i | > | | > | > | > | > | > | > | | | > | > |
| ) | | | > | > | > | > | > | > | | | > | > |
| $ | < | < | < | < | < | < | < | < | < | < | | |

Tabulka 2: Precedenční tabulka

Operátory v tabulce jsou v pořadí od nejvyšší priority do nejnižší (`unwrap` - nejvyšší, `||` - nejnižší). Operátory se stejnou prioritou a asociativitou jsme zahrnuli do následujících skupin:

```
unwrap = {!}
not = {!}
mul = {*, /}
add = {+, -}
rel = {==, !=, <, >, <=, >=}
```

# 8 Použité zdroje

1. Přednášky, materiály a záznamy z demonstračních cvičení z předmětu IFJ
2. https://www.cs.princeton.edu/courses/archive/spring22/cos320/LL1/
3. https://www.fit.vutbr.cz/~ikocman/llkptg/
4. https://developer.apple.com/documentation/swift/