

Projektová dokumentace  
**Implementace překladače imperativního jazyka IFJ22**  
Tým xmoise01, varianta BVS

24. listopadu 2022

<b>Nikita Moiseev</b>	<b>(xmoise01)</b>	25 %
Maksim Kalutski	(xkalut00)	25 %
Elena Marochkina	(xmaroc00)	25 %
Nikita Pasynkov	(xpasyn00)	25 %

# Obsah

<b>1 Úvod</b>	<b>1</b>
<b>2 Návrh a implementace</b>	<b>1</b>
2.1 Lexikální analýza . . . . .	1
2.2 Syntaktická analýza . . . . .	1
2.3 Sémantická analýza . . . . .	2
2.4 Generování cílového kódu . . . . .	2
<b>3 Práce v týmu</b>	<b>2</b>
3.1 Způsob práce v týmu . . . . .	2
3.1.1 Vývoj . . . . .	3
3.1.2 Komunikace . . . . .	3
3.2 Rozdělení práce mezi členy týmu . . . . .	3
<b>4 Závěr</b>	<b>3</b>
<b>A Diagram konečného automatu specifikující lexikální analyzátor</b>	<b>4</b>
<b>B LL – gramatika</b>	<b>5</b>
<b>C LL – tabulka</b>	<b>6</b>
<b>D Precedenční tabulka</b>	<b>7</b>

# 1 Úvod

Cílem projektu bylo vytvořit program v jazyce C, který načte zdrojový kód zapsaný ve zdrojovém jazyce IFJ22, jenž je zjednodušenou podmnožinou jazyka PHP a přeloží jej do cílového jazyka IFJcode22 (mezikód).

Program funguje jako konzolová aplikace, které načítá zdrojový program ze standardního vstupu a generuje výsledný mezikód na standardní výstup nebo v případě chyby vrací odpovídající chybový kód.

## 2 Návrh a implementace

Projekt jsme sestavili z několika námi implementovaných dílčích částí, které jsou představeny v této kapitole. Je zde také uvedeno, jakým způsobem spolu jednotlivé dílčí části spolupracují.

### 2.1 Lexikální analýza

Při tvorbě překladače jsme začali implementací lexikální analýzy. Hlavní funkce této analýzy je `get_next_token`, pomocí níž se čte znak po znaku ze zdrojového souboru a převádí na strukturu `token`, která se skládá z typu a hodnoty. Typy tokenů jsou `EOF`, speciální znaky, speciální závorky PHP, identifikátory, klíčová slova, datové typy a také aritmetické, relační a logické operátory a operátor přiřazení a ostatní znaky, které mohou být použity v jazyce IFJ2022. Hodnota atributu je `value`. Pokud je typ tokenu identifikátor, pak bude atribut daný identifikátor, když by byl typ tokenu klíčové slovo, přiřadí atributu dané klíčové slovo, pokud číslo, atribut bude ono číslo. S takto vytvořeným tokenem poté pracují další analýzy.

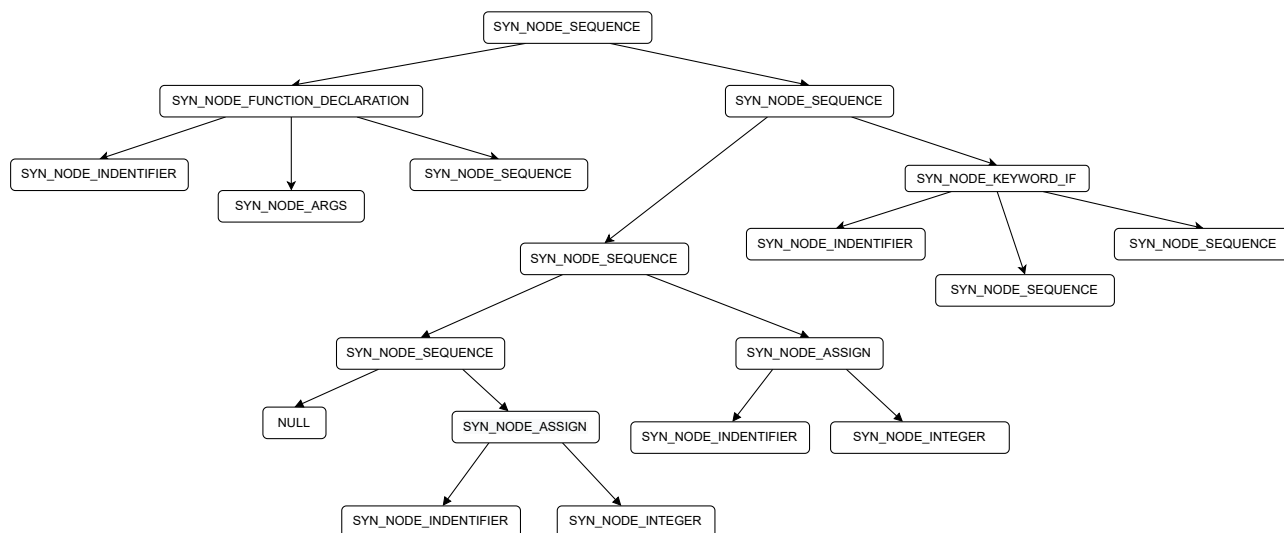
Celý lexikální analyzátor je implementován jako deterministický konečný automat podle předem vytvořeného diagramu 3. Konečný automat je v jazyce C jako jeden nekonečně opakující se `switch`, kde každý případ `case` je ekvivalentní k jednomu stavu automatu. Pokud načtený znak nesouhlasí s žádným znakem, který jazyk povoluje, program je ukončen a vrací chybu `1 LEXICAL ERROR CODE 1`. Jinak se přechází do dalších stavů a načítají se další znaky, dokud nemáme hotový jeden token, který potom vrátíme a ukončíme tuto funkci.

### 2.2 Syntaktická analýza

Nejdůležitější částí celého programu je syntaktická analýza. Syntaktická analýza je implementována v souboru `syntax_analyzer.c`, a její rozhraní pro implementaci v `syntax_analyzer.h`.

Syntaktická analýza je implementována pomocí rekurzivního sestupu na základě LL gramatiky 2. Syntaktický analyzátor zpracovává všechny části LL gramatiky na výrazy podle pravidel v LL - tabulce 3. Úkolem syntetického analyzátoru je sestavit abstraktní syntaktický strom na základě seznamu tokenů. Strom je postaven tak, že je nastavena priorita operací a ve výsledku můžeme načtením tohoto stromu spustit kód v požadovaném pořadí. Pokud je nalezena chyba, překladač dokončí kontrolu, vymaže alokovanou paměť a zobrazí chybovou hlášku na standardní chybový vstup. Syntaktický analyzátor nijak neupravuje tokeny a všechny chyby jsou detekovány až v průběhu sémantické analýzy. Po vytvoření abstraktního syntaktického stromu syntaktický analyzátor přejde do další fáze sémantické analýzy.

Příklad stromu pro syntaktickou analýzu je uveden na obrázku 1.

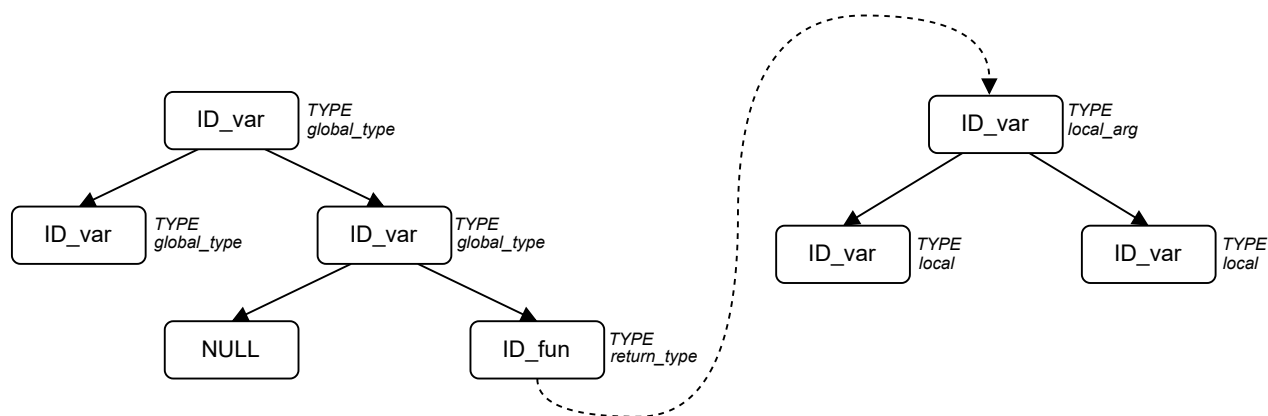


Obrázek 1: Abstraktní syntaktický strom přiřazení proměnných

## 2.3 Sémantická analýza

Sémantická analýza je implementována v souboru `semantic_analyzer.c` a její rozhraní pro implementaci v `semantic_analyzer.h`. Sémantický analyzátor pracuje s abstraktním syntaktickým stromem, který vytvořil syntaktický analyzátor. Úkolem sémantického analyzátoru je kontrola sémantickou správnost zdrojového programu: kontrola deklarací, datových typů, seznamů parametrů apod. a to pomocí kontroly datových typu. K tomu využívá datové struktury a funkce pro sémantické kontroly. Pokud při kontrole je nalezena chyba, překladač dokončí kontrolu, vymaže alokovanou paměť a zobrazí chybovou hlášku na standardní chybový vstup. Po dokončení sémantické analýzy se překladač přesune do fáze generování kódu.

Příklad stromu pro sémantickou analýzu je uveden na obrázku 2.



Obrázek 2: Derivační strom pro sémantickou analýzu

## 2.4 Generování cílového kódu

# 3 Práce v týmu

## 3.1 Způsob práce v týmu

Na projektu jsme začali pracovat na začátku října. Práci jsme si dělili postupně, tj. neměli jsme od začátku stanovený kompletní plán rozdělení práce. Na dílčích částech projektu pracovali většinou dvojice členů týmu.

### 3.1.1 Vývoj

Veškerá naše práce byla rozdělena do sprintů, každý trval týden. Během sprintu musel každý jednotlivý člen týmu splnit určitý objem úkolu.

Pro správu souborů projektu jsme používali verzovací systém Git. Jako vzdálený repozitář jsme používali repozitář na *GitHubu*.

Git nám umožnil pracovat na více úkolech na projektu současně v tzv. větvích. Většinu úkolů jsme nejdříve připravili do větve a až po otestování a schválení úprav ostatními členy týmu jsme tyto úpravy začlenili do hlavní vývojové větve.

Pro testování jsme použili knihovnu pro testování jednotek *googletest*.

### 3.1.2 Komunikace

Komunikace mezi členy týmů probíhala převážně osobně nebo prostřednictvím aplikace Telegram.

V průběhu řešení projektu jsme měli i osobní setkání každý týden, kde jsme probírali a řešili problémy týkající se různých částí projektu. Pro plánování úkolů jsme použili webovou aplikaci *Notion*.

## 3.2 Rozdělení práce mezi členy týmu

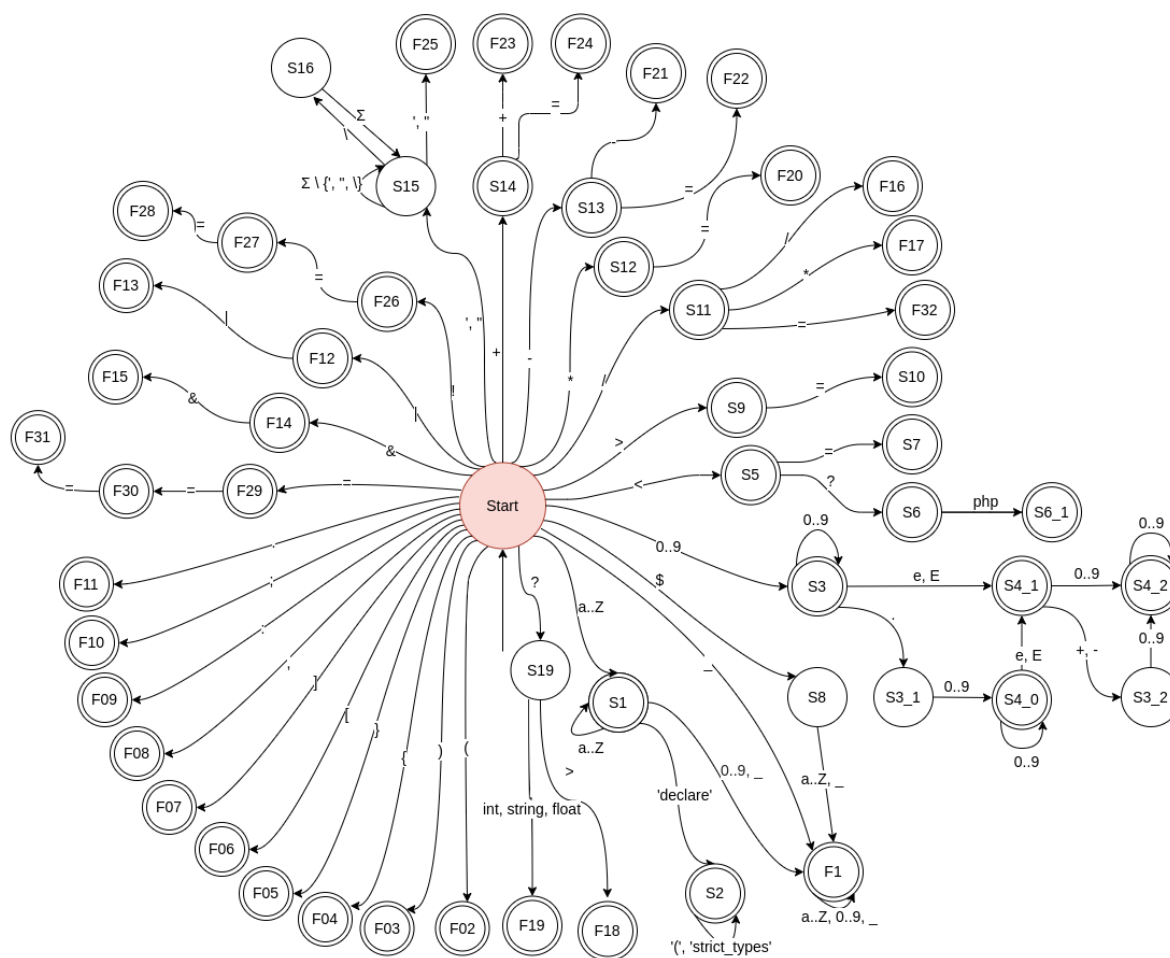
Práci na projektu jsme si rozdělili rovnoměrně s ohledem na její složitost a časovou náročnost. Každý tedy dostal procentuální hodnocení 25 %. Tabulka 1 shrnuje rozdělení práce v týmu mezi jednotlivými členy.

Člen týmu	Přidělená práce
<b>Nikita Moiseev</b>	vedení týmu, organizace práce, dohlížení na provádění práce, konzultace, kontrola, lexikální analýza, syntaktická analýza
Maksim Kalutski	generování cílového kódu, testování, dokumentace
Elena Marochkina	implementace tabulky symbolů, syntaktická analýza, testování, dokumentace
Nikita Pasynkov	lexikální analýza, syntaktická analýza, testování, dokumentace

Tabulka 1: Rozdělení práce v týmu mezi jednotlivými členy

## 4 Závěr

## A Diagram konečného automatu specifikující lexikální analyzátor



S1: keyword and ID state  
S2: declare state  
S3: integer state  
S3\_1, S3\_2: float middle state  
S4\_0, S4\_1, S4\_2: float state  
S5: less  
S6: open php bracket  
S6\_1: open php bracket  
S7: less or equal  
S8: identifier state  
S9: greater  
S10: greater or equal  
S11: divide  
S12: multiply  
S13: minus  
S14: plus  
S15: string state  
S16: string escape state  
S17: not  
S18: not equal  
S19: optional question mark

F01: identifier  
F02: left parenthesis  
F03: right parenthesis  
F04: left curly brackets  
F05: right curly brackets  
F06: left square brackets  
F07: right square brackets  
F08: comma  
F09: colon  
F10: semicolon  
F11: concatenation  
F12: bitwise or  
F13: logical or  
F14: bitwise and  
F15: logical and  
F16: comment

F17: multiline comment  
F18: close php bracket  
F19: optional data type  
F20: multiply assign  
F21: decrement  
F22: minus assign  
F23: increment  
F24: plus assign  
F25: string  
F26: not  
F27: not equal  
F28: typed not equal  
F29: assign  
F30: equal  
F31: typed equal  
F32: divide assign

Obrázek 3: Diagram konečného automatu specifikující lexikální analyzátor

## B LL – gramatika

1. `<prog> -> <?php <declare> <f-dec-stats> <stat-list> ?>`
2. `<prog> -> <? <declare> <f-dec-stats> <stat-list> ?>`
3. `<f-dec-stats> -> <f-dec-stat>`
4. `<f-dec-stats> -> <f-dec-stat> <f-dec-stats>`
5. `<f-dec-stat> -> function ID ( <f-args> ) : <f-type> { <stat-list> }`
6. `<f-dec-stat> -> function ID ( <f-args> ) { <stat-list> }`
7. `<f-type> -> int`
8. `<f-type> -> float`
9. `<f-type> -> string`
10. `<f-type> -> void`
11. `<f-args> -> <f-arg>`
12. `<f-args> -> <f-arg>, <f-args>`
13. `<f-args> ->  $\epsilon$`
14. `<f-arg> -> <f-type> ID`
15. `<stat> -> ID = <expr> ;`
16. `<stat> -> ID = ID ( <args> ) ;`
17. `<stat> -> ID ( <args> ) ;`
18. `<stat> -> return <expr> ;`
19. `<stat> -> if ( <expr> )`
20. `<stat> -> if ( <expr> ) <stat> else <stat>`
21. `<stat> -> while ( <expr> ) <stat>`
22. `<stat> -> { <stat-list> }`
23. `<args> -> <arg>`
24. `<args> -> <arg>, <args>`
25. `<arg> -> <term>`
26. `<arg> ->  $\epsilon$`
27. `<term> -> int`
28. `<term> -> float`
29. `<term> -> string`
30. `<term> -> NULL`
31. `<term> -> ID`
32. `<stat-list> -> <stat> <stat-list>`
33. `<stat-list> ->  $\epsilon$`
34. `<expr> -> EXPR <expr>`
35. `<expr> -> ID ( <args> )`
36. `<expr> ->  $\epsilon$`
37. `<declare> -> declare(strict-types = 0) ;`
38. `<declare> -> declare(strict-types = 1) ;`

Tabulka 2: LL – gramatika řídící syntaktickou analýzu

## **C LL – tabulka**

Tabulka 3: LL – tabulka použitá při syntaktické analýze



## D Precedenční tabulka

	+ -	* /	<>	(	)	id	\$
+ -	>	<	>	<	>	<	>
* /	>	>	>	<	>	<	>
<>	<	<	0	<	>	<	>
(	<	<	<	<	=	<	0
)	>	>	>	0	>	0	>
id	>	>	>	0	>	0	>
\$	<	<	<	<	0	<	End

+ - – aritmetické operátory + a -

\* / – aritmetické operátory \* a /

<> – relační operátory ==, !=, <, >, <=, >=

( – levá závorka

) – pravá závorka

id – identifikátor

\$ – konec vstupu

Tabulka 4: Precedenční tabulka použitá při syntaktické analýze