

Projektová dokumentace

Překladač jazyka IFJ19

Tým 040, Varianta II

**Boris Burkalo** (xburka00) 34%

Jiří Herrgott (xherrg00) 33%

Jan Klusáček (xklusa14) 33%

Prosinec 2019

**Obsah**

[**Obsah** 2](#_Toc26888686)

[**1.** **Úvod** 3](#_Toc26888687)

[**2.** **Tým – rozdělení** 3](#_Toc26888688)

[**2.1.** **Týmová práce** 3](#_Toc26888689)

[**2.2.** **Vývojové prostředí + verzování** 3](#_Toc26888690)

[**2.3.** **Tabulka rozdělení práce** 4](#_Toc26888691)

[**3.** **Implementace** 4](#_Toc26888692)

[**3.1.** **Lexikální analýza** 4](#_Toc26888693)

[**3.2.** **Syntaktická a sémantická analýza** 5](#_Toc26888694)

[**3.2.1.** **Kontrola definicí funkcí a proměnných** 5](#_Toc26888695)

[**3.2.2.** **Výrazy** 6](#_Toc26888696)

[**3.3.** **Generování kódu** 6](#_Toc26888697)

[**4.** **Závěr** 6](#_Toc26888698)

[**5.** **Použitá literatura** 7](#_Toc26888699)

[**6.** **Diagramy a tabulky** 8](#_Toc26888700)

[Diagram konečného automatu pro lexikální analýzu (obr. 1) 8](#_Toc26888701)

[LL gramatika a příslušná LL tabulka 9](#_Toc26888704)

# **Úvod**

Zadáním projektu bylo vytvořit program v jazyce C, který načte zdrojový kód napsaný ve zdrojovém jazyce IFJ19 a přeloží jej do cílového jazyka IFJcode19 (mezikód). Jazyk IFJ19 je zjednodušenou podmnožinou jazyka Python3, a jedná se o dynamicky typovaný imperativní jazyk s funkcionálními prvky.

# **Tým – rozdělení**

## **Týmová práce**

V polovině října jsme se začali jako tým pravidelně scházet v prostorách školy. Nejprve jsme si společně pročítali zadání a snažili jsme se rozdělit si práci. Zpočátku se jednalo jen o malý úsek, a potom jsme si práci dělili podle aktuálních potřeb. Mimo rozdělených úkolů jsme si navzájem pomáhali s různými problémy.

Náš tým zvolil variantu II, která k implementaci použila tabulku s rozptýlenými položkami. Ke zhotovení projektu jsme využili několik dílčích částí, které budou popsané v následujících kapitolách. K řešení jsme využívali informace získané na přednáškách a stránce projektu, ale také jsme si sami vyhledávali potřebné informace na internetu.

## **Vývojové prostředí + verzování**

Všichni členové týmu byli zvyklí používat operační systém Linux (PopOS, Ubuntu), a tak nikde nenastával problém s kompatibilitou.

K práci jsme využívali verzovací systém GitHub, kde nám stačilo využít jednu větev, jelikož na daném souboru pracovali vždy maximálně dva lidé, a tak díky vzájemné komunikaci (hlavně pomocí aplikace Discord) nebyl problém s mergováním. Využívali jsme editor Atom, který má skvělé propojení s GitHubem, a také má vlastní nástroj pro mergování. Nicméně občas bylo nutné osobní setkání, např. kvůli propojení některých modulů.

Funkčnost jsme testovali vzdáleně na CentOS Merlin, a paměťové úniky pomocí programu Valgrind.

## **Tabulka rozdělení práce**

|  |  |
| --- | --- |
| Boris Burkalo | Vedení týmu, parser |
| Jiří Herrgott | Lexikální analyzátor, generování kódu |
| Jan Klusáček | Parser, dokumentace |

# **Implementace**

## **Lexikální analýza**

Hlavním modulem pro lexikální analýzu je modul scanner. Stavebním kamenem scanneru je funkce int get\_next\_token(Token \*token), která čte znak po znaku vstupní soubor, a ve switch case přechází do jednotlivých stavů konečného automatu (viz obr. 1). V případě načtení znaku patřícího dalšímu tokenu se volá funkce ungetc, a na vyžádání parseru se začíná načítat další token. Funkce get\_next\_token vrací příslušnou int hodnotu v případě neúspěchu, jinak vrací 0. Do struktury Token předává typ daného lexému, a případně jeho data. V případě, že se jedná o typ identifikátor, kontroluje se, jestli se nejedná o klíčové slovo.

Kromě posílání tokenů do parseru má scanner za úkol také odstranit z kódu nepotřebné části, jako jsou například komentáře.

Dalším modulem, který se využívá pro scanner, je modul strings, ve kterém se nachází struktura a funkce pro práci s dynamickým stringem. Modul obsahuje funkce například pro přidání a odebrání znaku, přidání celého řetězce, nebo dokonce vložení řetězce na určitou pozici v dynamickém stringu.

## **Syntaktická a sémantická analýza**

Tělem syntaktické a sémantické analýzy je modul parser, který však využívá další moduly popsané níže. Funguje na základě analýzy shora dolů a řídí se pravidly LL(1) gramatiky. Z modulu main je volána funkce prog, která se postará o vše potřebné pro zahájení parsování – inicializace scanneru, vytvoření tabulky s rozptýlenými položkami atd. Funkce body si žádá o tokeny ze vstupního souboru pomocí funkce get\_next\_token, které jsou předány ze scanneru a již prošly lexikální analýzou. Následně volá další funkce, pomocí kterých se zpracovávají celé příkazy.

### **Kontrola definicí funkcí a proměnných**

V případě, že se v zdrojovém souboru vyskytne volání funkce, nebo dojde k použití libovolné proměnné, je nutno zkontrolovat, zda funkce, nebo proměnná již byla definována. Tato kontrola je realizována vždy před použitím proměnné, nebo voláním funkce, pomocí funkce nad tabulkou symbolů – hSymtab\_it\* symtab\_it\_position(char \*searched\_for, hSymtab \*sym\_tab). Tato funkce prohledá celou tabulku symbolů, pokud hledanou položku najde, vrátí ji, v opačném případě vrátí NULL. Pro hledání proměnných je zde také pomocná funkce int is\_item\_var\_defined(char \*desired, hSymtab \*sym\_tab), která se podívá, zda u  proměnných je nastavená hodnota defined na true.

Pokud dojde k volání funkce v definici jiné funkce, před samotnou definicí volané funkce, volaná funkce a počet parametrů, se kterými byla zavolána, se uloží na zásobník, který se na konci parsování celý vyprázdní a zkontroluje (ve funkci int sym\_stack\_pop\_all(hSym\_fct\_node\* f\_top, hSymtab\* act\_table)), zda všechny volané funkce byly také definovány.

### **Výrazy**

Na zpracování výrazů se používá precedenční tabulka, která je implementována ve zvláštním modulu expression.



Legenda

< push vstupního symbolu na zásobník

> pop ze zásobníku (v případě ID, hned následuje push na zásobník ID)

E pravá závorka – pop ze zásobníku až po levou závorku (včetně)

N nedefinovaný stav (chyba)

## **Generování kódu**

V modulu generator se nacházejí funkce, které jsou volány za běhu parsování a postupně vytvářejí dynamický string, obsahující výsledný kód. String je poslán na standartní výstup v případě, že nenastala žádná chyba. Pomocný modul generator\_functions slouží pro generování vestavěných funkcí.

Při generování kódu jsme narazili na menší problém, a to, že definice funkce může být v jazyce IFJ19 kdekoli v těle programu. V našem výsledném kódu se tedy nacházela na stejném místě jako ve zdrojovém kódu, a interpret do ní vešel při procházení programem.

Vyřešili jsme to tím, že jsme vytvořili dva dynamické stringy, jeden pro funkce a druhý pro zbytek programu. Tyto stringy se na závěr konkatenují, a tak jsou funkce vždy nad tělem programu.

# **Závěr**

Před odevzdáním nám velmi pomohlo zkušební odevzdání. Vyděsilo nás, jak málo procent jsme dostali, a tak jsme si domluvili konzultace s Ing. Zbyňkem Křivkou Ph.D. Pomohl nám a navedl nás na chybu, kvůli které většina testů neprošla. Mimo to jsme ale objevili i další chyby, které jsme poté měli možnost opravit. Celkově nás projekt překvapil svým rozsahem, ale mnoho nás naučil.

# **Použitá literatura**

1. Meduna, A., Lukáš, R.: IFJ, přednášky [online]. Brno, URL:

<http://www.fit.vutbr.cz/study/courses/IFJ/public/materials/index.php>

# **Diagramy a tabulky**

## Diagram konečného automatu pro lexikální analýzu (obr. 1)

## 

## 

## LL gramatika a příslušná LL tabulka

1. <prog> -> <body>
2. <body> -> ε
3. <body> -> <command> EOL <body>
4. <command> -> ID = <assignment>
5. <command> -> FUNCTION ( <fct\_call>
6. <command> -> <predef\_fct>
7. <command> -> def <fction\_start>
8. <command> -> expression
9. <command> -> if <if\_statement>
10. <command> -> while <while\_statement>
11. <command> -> ε
12. <assignment> -> FUNCTION ( <fct\_call>
13. <assignment> -> expression
14. <assignment> -> <predef\_fct>
15. <if\_statement> -> expression : EOL INDENT <statement\_body> DEDENT <else\_statement>
16. <else\_statement> -> else : EOL INDENT <statement\_body>
17. <else\_statement> -> ε
18. <while\_statement> -> expression : EOL INDENT <statement\_body> DEDENT
19. <statement\_body> -> <command> EOL <statement\_body>
20. <statement\_body> -> DEDENT
21. <fct\_call> -> <type> <arg\_n> )
22. <arg\_n> -> , <type> <arg\_n>
23. <arg\_n> -> ε
24. <fction\_start> -> FUNCTION ( ID <arg\_n> ): EOL <fction\_body> INDENT
25. <fction\_body> -> <command> EOL <fction\_body>
26. <fction\_body> -> ε
27. <predef\_fct> -> inputs ( )
28. <predef\_fct> -> inputf ( )
29. <predef\_fct> -> inputi ( )
30. <predef\_fct> -> print ( <type> <arg\_n­> )
31. <predef\_fct> -> len ( <type> )
32. <predef\_fct> -> substr ( <type>, <type>, <type> )
33. <predef\_fct> -> ord ( <type>, <type> )
34. <predef\_fct> -> chr ( <type> )
35. <type> -> STR
36. <type> -> ID
37. <type> -> INT
38. <type> -> FLOAT

