

Projektová dokumentace

Překladač jazyka IFJ19

Tým 040, Varianta II

**Boris Burkalo** (xburka00) 33%

Jiří Herrgott (xherrg00) 33%

Jan Klusáček (xklusa14) 33%

Prosinec 2019

Obsah

[1. Úvod 3](#_Toc26726348)

[2. Tým – rozdělení 3](#_Toc26726349)

[2.1. Týmová práce 3](#_Toc26726350)

[2.2. Vývojové prostředí + verzování 3](#_Toc26726351)

[2.3. Tabulka rozdělení práce 4](#_Toc26726352)

[3. Implementace 4](#_Toc26726353)

[3.1. Lexikální analýza 4](#_Toc26726354)

[3.2. Syntaktická a sémantická analýza 4](#_Toc26726355)

[3.3. Generování kódu 5](#_Toc26726356)

[4. Závěr????? Možná nedávat 5](#_Toc26726357)

# 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 nějak rozdělit 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, Ubunutu), 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 setkání ve škole 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, výrazy |
| 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.). 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????//doplnit co vrací asi(Sterv). Do struktury Token předává typ daného lexému a případně jeho data. V případě, že se jedná o typ identifikátor (u nás variable myslím), 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 (možná napsat něco víc Sterv).

## 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 je řízen pravidly LL(1) gramatiky(Boris). Z modulu main je volána funkce prog, která se postará o vše potřebné pro zahájení parsování, jako například spuštění 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(Boris)

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



asi o ní něco napsat+ doplnit legendu(Boris)

## 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 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????? Možná nedávat

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.

SPEC. ALG A DAT STRUKTURY (dyn řetězec, hashovací tabulka, zásobník?)????

🡪 Bude doplněno

Grafy a obrázky

• Diagram konečného automatu, který specifikuje lexikální analyzátor.

• LL-gramatiku, LL-tabulku a precedenční tabulku, podle kterých jste implementovali

váš syntaktický analyzátor.

🡨

POUŽITÁ LITERATURA -> NĚCO BY TAM MĚLO BÝT