VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

# Projektová dokumentace

## Implementace překladače imperativního jazyka IFJ18

**Tým 012, Varianta II.**

5. prosince 2018

| | | |
|---|---|---|
| **Michael Kinc** | **(xkincm02)** | 35 % |
| Martin Litwora | (xlitwo00) | 35 % |
| Marek Mündl | (xmundl00) | 15 % |
| Lukáš Kadlec | (xkadle36) | 15 % |

# Obsah

# 1 Úvod

Cílem projektu bylo vytvořit program v jazyce C, který načte zdrojový kód zapsaný ve zdrojovém jazyce IFJ18 a přeloží jej do cílového jazyka IFJcode18 (mezikód). V tomto dokumentu dále popisujeme postupy při implementaci a práci v týmu.

# 2 Implementace

## 2.1 Lexikální analýza

Celá lexikální analýza vychází z námi navrženého konečného automatu. Základem celého scanneru je funkce `get_token`, která je přesným přepisem automatu. Ten je v jazyce C implementován pomocí opakujícího se přepínače `switch`, jehož návěští `case` představují stavy konečného automatu. Funkce tedy čte výchozí soubor znak po znaku a rozpozná token, který je uložen do struktury, kde je uchováván jeho typ a atribut. Ten je uložen jako typ `union`, kde je rozlišeno, zda se jedná o atribut integeru, doublu nebo stringu.

V jazyce IFJ18 rozlišujeme tokeny jako integer, double, string, operátory násobení, dělení, sčítání, odčítání, identifikátory, ale také znak konce řádku `EOL` a znak konce souboru `EOF` a další. Úkolem lexikálního analyzátoru je také odstranit mezery, tabulátory a ignorovat řádkové případně blokové komentáře.

Načtený identifikátor postupně ukládáme do dynamického pole. Pro vyhodnocení, zda se opravdu jedná o identifikátor nebo klíčové slovo, slouží funkce `check_if_keyword`, která porovná nalezený identifikátor s klíčovými slovy, které obsahuje jazyk IFJ18.

## 2.2 Syntaktická analýza

Pro implementaci syntaktické analýzy byl zvolen rekurzivní sestup řízený LL – tabulkou a pravidly LL – gramatiky. Hlavní funkcí je funkce `analyser`, která tvoří jádro celého programu. Pro většinu pravidel jsme si vytvořili odpovídající funkce, které provádí jednotlivé syntaktické kontroly a žádají lexikální analyzátor o další token prostřednictvím funkce `get_token`.

### 2.2.1 Precedenční syntaktická analýza a zpracování výrazů

Při implementaci syntaktické analýzy výrazů jsme se řídili precedenční tabulkou. Hlavní část tvoří funkce `expression`, která se řídila načteným tokenem (terminálem) a nejvyšším terminálem na zásobníku. Pomocní nich jsme zjistili konkrétní znak v precedenční tabulce symbolů, podle čehož jsme zjistili, jakým pravidlem se máme řídit.

Pro správné fungování bylo potřeba vytvořit zásobník, jenž byl implementován na základě studijních materiálů předmětu IAL. Navíc bylo třeba implementovat rozšiřující funkce pro precedenční syntaktickou analýzu, například funkci, která správně provede operaci shift před nejvyšší terminál na zásobníku.

## 2.3 Sémantická analýza

Kontroly sémantiky byly navrženy tak, aby se prováděly během činnosti syntaktického analyzátoru. Hashovací funkci včetně fungovaní tabulky jsme vytvořili na základě druhé domácí úlohy z předmětu IAL.

Sémantická analýza byla implementována pouze částečně, ale nebyla dotažena do odevzdatelné podoby, proto jsme se ji rozhodli odstranit z výsledného progamu.

### 2.4 Generování kódu

Generování kódu bychom prováděli za běhu syntaktické analýzy, kde bychom volali jednotlivé funkce pro generování výsledného mezikódu. Ten by měl formu jednotlivých instrukcí založených na tříadresném kódu.

## 3 Práce s dynamickým polem

Pracovali jsme se strukturou `Dynamic_array`, ve které je uloženo pole znaku, aktuální a alokovaná délka. Výhody dynamického pole byly využity při práci řetězci, protože předem nevíme, jak budou dlouhé.

## 4 Testování

Testování bylo poměrně náročné, protože jsme neustále naráželi na nové problémy a záležitosti, které byly potřeba řešit. Využili jsme standardní nástroj na hledání chyb v softwaru GDB, zejména při špatné práci s pamětí, a také bylo využito kontrolních výpisů během programu.

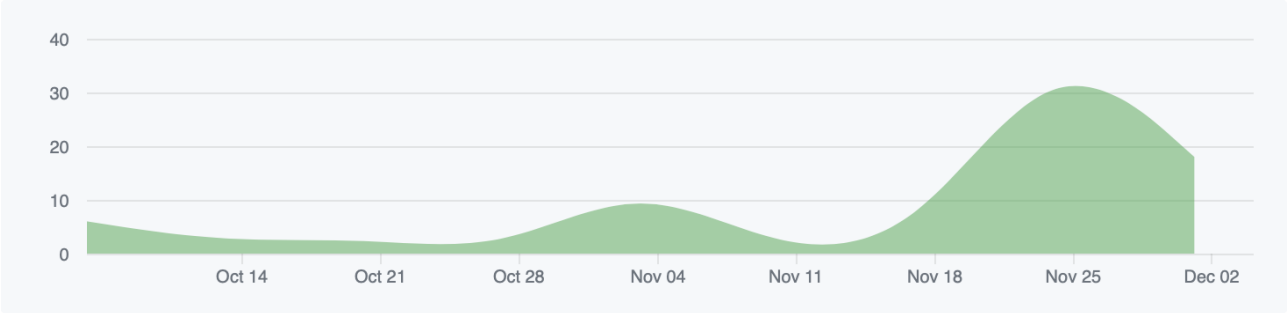
## 5 Práce v týmu

Náš tým byl díky předchozí skupinové práci v předešlých semestrech sestaven již od začátku září. Obecná komunikace probíhala především v privátní Facebookové skupině. Konverzace týkající se podrobností o implementaci pak probíhala na vlastním Discord kanále. Většina projektu byla vytvořena na osobních schůzkách. Detaily a testování pak byly řešeny prostřednictvým hovorů. Na první schůzce jsme se pouze snažili pochopit zadání, vytvořili jsme soukromý repositář na GitHubu a kanál na Discordu. Jelikož jsme neměli v tomto okamžiku znalost o všech částech tohoto projektu, tak nebylo možné práci mezi členy týmu rozdělit.

Spolupráce v týmu místy vázla. Ne všichni členové věnovali projektu stejné úsilí, což vedlo v kombinaci s povinnostmi týkajících se jiných předmětů k velké časové ztrátě a k prakticky nepřetržité práci zbytku týmu v konečné fázi projektu.

| | |
|---|---|
| **Michael Kinc** | organizace práce, lexikální analýza, syntaktická analýza, testování, tvorba dokumentace a prezentace |
| Martin Litwora | lexikální analýza, syntaktická analýza, syntaktická analýza výrazů, testování |
| Marek Mündl | tvorba konečného automatu a pomoc při tvorbě lexikálního analyzátoru |
| Lukáš Kadlec | tvorba konečného automatu a pomoc při tvorbě lexikálního analyzátoru |

Tabulka 1: Rozdělení práce v týmu



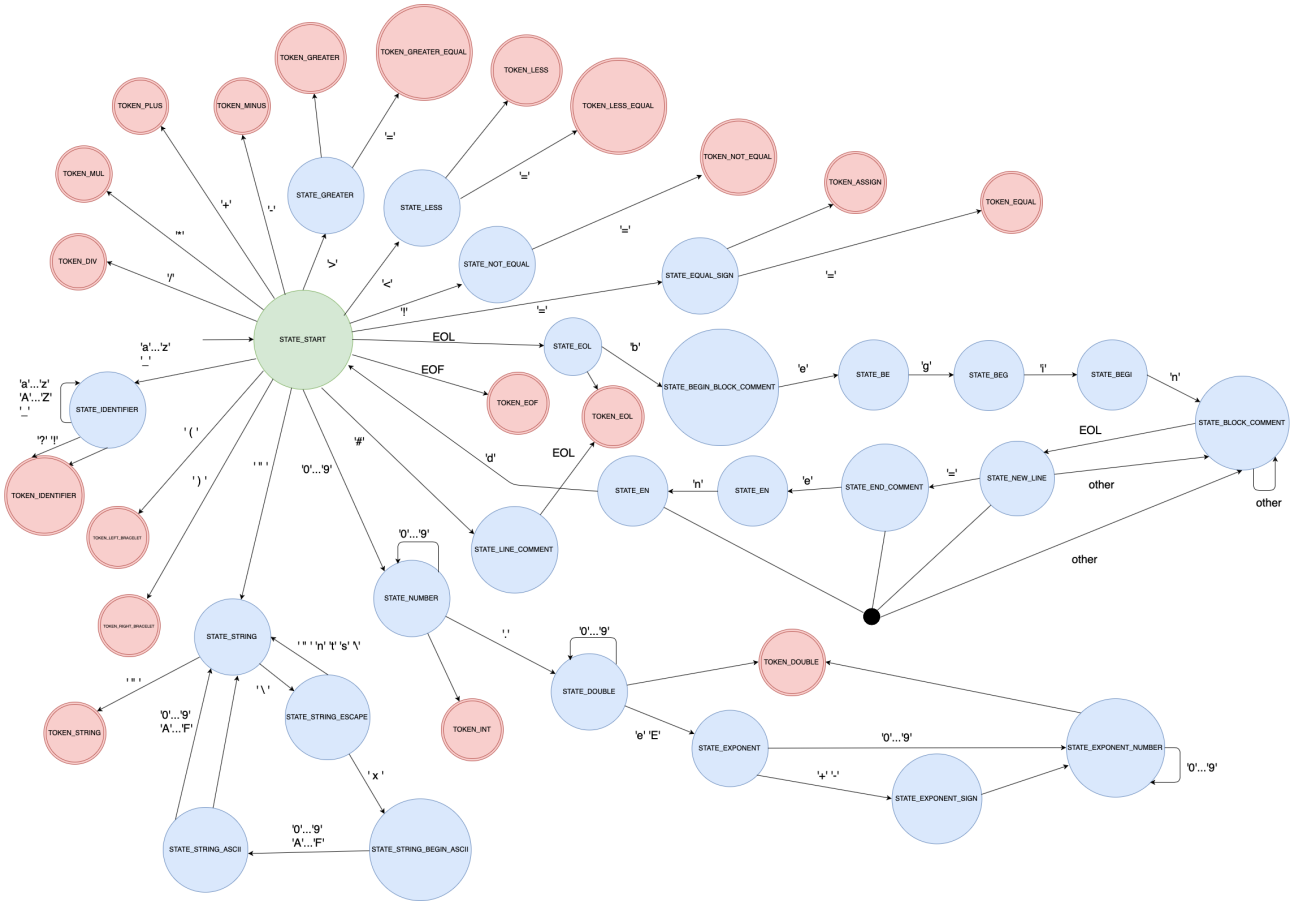

Obrázek 1: Množství vynaložené práce v průběhu semestru

# 6 Závěr

Projekt hodnotíme jako velice přínosný. Naučili jsme se, jak fungují překladače a rozšířili jsme si svoje obzory o tom, jak může probíhat práce v týmu a také jsme prohloubili svoje znalosti o nové konstrukce v programovacím jazyce C. I přes problémy se spoluprací se nám povedlo aspoň část projektu dotáhnout do úspěšného konce. Lexikální a syntaktická analýza s precedenční tabulkou pro vyhodnocování výrazů jsou zcela funkčí, bohužel jsme však z časových důvodů již nedokončili zbytek projektu.

# 7 Přílohy

## 7.1 Diagram konečně stavového automatu




Obrázek 2: Precedenční tabulka použitá při precedenční syntaktické analýze výrazů

## 7.2 LL – gramatika

1. <prog> -> def id ( <param> ) eol <statement> end <prog>
2. <prog> -> eol <prog>
3. <prog> -> <body>
4. <body> -> <statement> <end>
5. <end> -> eof
6. <param> -> id <param_n>
7. <param> -> $\varepsilon$
8. <param_n> -> , id <param_n>
9. <param_n> -> $\varepsilon$
10. <statement> -> if <expression> then eol <statement> else eol <statement> end eol <statement>
11. <statement> -> while <expression> do eol <statement> end eol <statement>
12. <statement> -> id <assignement> eol <statement>
13. <statement> -> <expression> eol <statement>
14. <statement> -> print <parameters>
15. <statement> -> $\varepsilon$
16. <assignment> -> = <right_side>
17. <assignment> -> <parameters>
18. <right_side> -> <func_call>
19. <right_side> -> <expression>
20. <func_call> -> inputs <parameters>
21. <func_call> -> inputi <parameters>
22. <func_call> -> inputf <parameters>
23. <func_call> -> length <parameters>
24. <func_call> -> substr <parameters>
25. <func_call> -> ord <parameters>
26. <func_call> -> chr <parameters>
27. <func_call> -> id <parameters>
28. <parameters> -> ( <arg> )
29. <parameters> -> <arg>
30. <arg> -> <value> <arg_n>
31. <arg> -> $\varepsilon$
32. <arg_n> -> , <value> <arg_n>
33. <arg_n> -> $\varepsilon$
34. <value> -> double
35. <value> -> integer
36. <value> -> string
37. <value> -> id

Tabulka 2: LL – gramatika, podle které jsme psali náš syntaktický analyzátor

## 7.3 LL – tabulka

| | def | id | ( | ) | eol | end | eof | , | if | then | else | while | do | print | = | inputs | inputi | inputf | length | substr | ord | chr | double | integer | string | $ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| <prog> | 1 | 3 | | | 2 | | 3 | | 3 | | | 3 | | 3 | | | | | | | | | | | | |
| <param> | | 6 | | 7 | | | | | | | | | | | | | | | | | | | | | | |
| <statement> | | 12 | | | | 15 | 15 | | 10 | | 15 | 11 | | 14 | | | | | | | | | | | | |
| <body> | | 4 | | | | | 4 | | 4 | | | 4 | | 4 | | | | | | | | | | | | |
| <end> | | | | | | | 5 | | | | | | | | | | | | | | | | | | | |
| <param_n> | | | | 9 | | | | 8 | | | | | | | | | | | | | | | | | | |
| <expression> | | | | | | | | | | | | | | | | | | | | | | | | | | |
| <assignment> | | 17 | 17 | | 17 | | | | | | | | | | 16 | | | | | | | | 17 | 17 | 17 | |
| <parameters> | | 29 | 28 | | 29 | 29 | 29 | | | | 29 | | | | | | | | | | | | 29 | 29 | 29 | |
| <right_side> | | 18 | | | | | | | | | | | | | | 18 | 18 | 18 | 18 | 18 | 18 | 18 | | | | |
| <func_call> | | 27 | | | | | | | | | | | | | | 20 | 21 | 22 | 23 | 24 | 25 | 26 | | | | |
| <arg> | | 30 | | 31 | 31 | 31 | 31 | | | | 31 | | | | | | | | | | | | 30 | 30 | 30 | |
| <value> | | 37 | | | | | | | | | | | | | | | | | | | | | 34 | 35 | 36 | |
| <arg_n> | | | | 33 | 33 | 33 | 33 | 32 | | | 33 | | | | | | | | | | | | | | | |

Tabulka 3: LL – tabulka použitá při implementaci syntaktické analýzy

## 7.4 Precedenční tabulka

| | * | / | + | - | < | <= | > | >= | = = | != | ( | ) | i | $ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| * | > | > | > | > | > | > | > | > | > | > | < | > | < | > |
| / | > | > | > | > | > | > | > | > | > | > | < | > | < | > |
| + | < | < | > | > | > | > | > | > | > | > | < | > | < | > |
| - | < | < | > | > | > | > | > | > | > | > | < | > | < | > |
| < | < | < | < | < | | | | | > | > | < | > | < | > |
| <= | < | < | < | < | | | | | > | > | < | > | < | > |
| > | < | < | < | < | | | | | > | > | < | > | < | > |
| >= | < | < | < | < | | | | | > | > | < | > | < | > |
| = = | < | < | < | < | < | < | < | < | | | < | > | < | > |
| != | < | < | < | < | < | < | < | < | | | < | > | < | > |
| ( | < | < | < | < | < | < | < | < | < | < | < | = | < | > |
| ) | > | > | > | > | > | > | > | > | > | > | | > | | |
| i | > | > | > | > | > | > | > | > | > | > | | > | | > |
| $ | < | < | < | < | < | < | < | < | < | < | < | | < | |

Tabulka 4: Precedenční tabulka pro syntaktickou analýzu výrazů