

Dokumentace projektu

# Implementace překladače imperativního jazyka IFJ18

Tým 083, varianta I

**Jakub Man (xmanja00) 25%**

Jiří Tykva (xtykva00) 25%

Jan Martinák (xmarti83) 25%

Adam Melichar (xmelic22) 25% 30. listopadu 2018

Obsah

[1. Úvod 3](#_Toc531789189)

[2. Návrh a implementace 3](#_Toc531789190)

[2.1 Lexikální analýza 3](#_Toc531789191)

[2.2. Syntaktická analýza 3](#_Toc531789192)

[2.3. Precedenční syntaktická analýza 3](#_Toc531789193)

[2.4. Sémantická analýza 3](#_Toc531789194)

[2.5. Generátor kódu 3](#_Toc531789195)

[3. Datové struktury 4](#_Toc531789196)

[3.1. Buffer 4](#_Toc531789197)

[3.2. Abstraktní syntaktický strom 4](#_Toc531789198)

[3.3. Tabulka symbolů 4](#_Toc531789199)

[4. Práce v týmu 5](#_Toc531789200)

[4.1. Způsob práce 5](#_Toc531789201)

[4.2. Rozdělení úkolů 5](#_Toc531789202)

[5. Závěr 5](#_Toc531789203)

[5.1. Metriky kódu 5](#_Toc531789204)

[6. Zdroje 5](#_Toc531789205)

[7. Přílohy 6](#_Toc531789206)

[A. LL Gramatika 6](#_Toc531789207)

[B. LL – Tabulka 7](#_Toc531789208)

[C. Precedenční tabulka – Pravidla 7](#_Toc531789209)

[D. Precedenční tabulka 8](#_Toc531789210)

[E. Schéma konečného automatu 9](#_Toc531789211)

# Úvod

Cílem tohoto dokumentu je popsat námi vytvořený překladač imperativního jazyka IFJ18 založeného na jazyce Ruby.

# Návrh a implementace

Námi naprogramovaný překladač načte jako program na příkazové řádce vstupní program zapsaný v jazyce IFJ18, který potom převede na cílový jazyk IFJcode18.

## 2.1 Lexikální analýza

Prvním modulem překladače je lexikální analyzátor (scanner), mající za úkol rozložení vstupního programu na jednotlivé lexikální symboly vstupního programu, které dostává parser ve formě tokenů voláním funkce *scanner(&currentToken****)***. Lexikální analyzátor musí také umět odfiltrovat komentáře, bílé znaky a rozpoznat lexikální chyby. Scanner jsme realizovali jako determinitsický konečný automat jehož schéma je obsaženo v příloze E. Každý token je definován strukturou, která se skládá z pole znaků a čísla značícího typ.

## 2.2. Syntaktická analýza

Syntaktický analyzátor (parser) je srdcem překladače. Jeho úkolem je sestavení derivačního stromu na základě tokenů získaných ze scanneru. Derivační strom se sestavuje shora dolů podle vytvořené LL gramatiky (kapitola 7.3). Pokud parser narazí na výraz, je nutné volat funkci *prec\_table(tToken \*token, tSymPtr sym)*, která má na starosti analýzu výrazů. Jestliže se podaří sestavit derivační strom, program je napsán syntakticky správně, když se sestavení nezdaří, překladač se ukončí s příznakem syntaktické chyby.

Po úspěšném vytvoření derivačního stromu, zpracování všech výrazů a kontrole sémantiky vytváří skrz modul generátoru (generator.c) jednotlivé příkazy cílového jazyka.

## 2.3. Precedenční syntaktická analýza

Využívá se pro zpracování výrazů například v podmínkách jednotlivých funkcí. Precedenční analýza zpracovává výraz zdola nahoru podle precedenčních pravidel (kapitola 7.3).

## 2.4. Sémantická analýza

Sémantický analyzátor má na starosti kontrolu definicí a deklarací funkcí a proměnných. Pro tyto kontroly využívá tabulku symbolů, do které vkládá všechny využité objekty, například funkce nebo proměnné.

## 2.5. Generátor kódu

Generátor cílového kódu (v našem případě IFJcode18) vytváří pomocí tříadresného kódu získaného ze syntaktického analyzátoru.

# Datové struktury

## Buffer

Pracujeme s ním při

## Abstraktní syntaktický strom

Syntaktický strom užíváme pro překlad a optimalizaci

## Tabulka symbolů

Náš cílový jazyk využívá tři paměťové rámce (globální, lokální a dočasné), pro jejich realizaci si však vystačíme s jednou tabulkou symbolů pro globální proměnné a následně generujeme tabulku symbolů pro každou definovanou funkci. Do tabulky symbolů se ukládají informace o všech využitých konstantách, proměnných a funkcích. Tyto informace jsou nezbytné pro generování kódu, říkají nám, jestli už byla dané proměnná či funkce využita, jakého je typu, nebo jestli je deklarována.

Tabulky symbolů máme zkonstruované metodou binárního vyhledávacího stromu.

# Práce v týmu

## Způsob práce

Po několika úvodních schůzkách týkajících se rozvržení práce a způsobu programování jsme začali nejprve samostatně, později však kolektivně studovat látku potřebnou k správnému pochopení funkce překladače. V aplikaci Trello jsme si sepsali všechny potřebné úkoly a pomocí této aplikace jsme sledovali průběh práce. Komunikace v teamu probíhala především osobně nebo pomocí Facebooku.

Samotné programování překladače, jsme začali řešit koncem října a to téměř vždy při hromadném setkání v prostorech fakulty. Při implementaci jsme využívali metod agilního programování, především párové programování nebo sprint.

Pro tvorbu programu jsme využívali textový editor Atom s rozšířením Teletype, které umožňovalo pracovat zároveň více lidem na jednom souboru. Testování probíhalo buď s pomocí aplikace Cygwin nebo přímo na Ubuntu. Verzování souborů probíhalo pomocí softwaru Git a repozitáře GitHub.

Pro ověření správné funkce jednotlivých dílčích částí překladače jsme si vytvořili komplexní sadu testů.

## Rozdělení úkolů

* Jakub Man – Vedoucí týmu, testování, generátor
* Adam Melichar – Scanner, parser, sémantický analyzátor
* Jiří Tykva – Scanner, parser, sémantický analyzátor
* Jan Martinák – Scanner, dokumentace, prezentace

# Závěr

Celá naše skupina se shodla na tom, že největší přínos tohoto projektu byl v poznání skupinové práce na složitějším projektu.

## Metriky kódu

Počet řádků:

Počet souborů:

Velikost binárního souboru:

# Zdroje

* ČEŠKA, Milan, Tomáš HRUŠKA a Miroslav BENEŠ. Překladače. Brno: Vysoké učení technické, 1993. ISBN 80-214-0491-4.
* ČEŠKA, Milan a Zdena RÁBOVÁ. Gramatiky a jazyky. 3. vyd. Brno: Vysoké učení technické, 1988.
* ČEŠKA, Milan a Tomáš HRUŠKA. Gramatiky a jazyky: cvičení. 2. vyd. Brno: Vysoké učení technické, 1988.

# Přílohy

## LL Gramatika

<PROG> → def id (<PARAMS>) eol <STATEMENT\_N> end eol <PROG>

<PROG> → STATEMENT eol <PROG>

<PROG> → ε

<PARAMS> → id <PARAMS\_N>

<PARAMS> → ε

<PARAMS\_N> → , id <PARAMS\_N>

<PARAMS\_N> → ε

<STATEMENT> → if expression then eol <STATEMENT\_N> else eol <STATEMENT\_N> end

<STATEMENT> → while expression do eol <STATEMENT\_N> end

<STATEMENT> → id <DEF\_ARGS>

<STATEMENT> → ε

<STATEMENT\_N> → <STATEMENT> eol <PROG>

<STATEMENT\_N> → ε

<DEF\_ARGS> → = <DEFINE>

<DEF\_ARGS> → <ARGS>

<DEFINE> → id <ARGS>

<DEFINE> → expression

<DEFINE> → ε

<ARGS> → ε

<ARGS> → (<ARGS>)

<ARGS> → <VALUE> <ARGS\_N>

<ARGS\_N> → , <VALUE> <ARGS\_N>

<ARGS\_N> → ε

<VALUE> → int

<VALUE> → float

<VALUE> → id

<VALUE> → string

<VALUE> → nil

## LL – Tabulka

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | def | id | ( | ) | eol | end | , | if | expression | then | else | while | do | = | int | float | string | $ |
| PROG | 1 | 2 |  |  | 2 | 3 |  | 2 |  |  | 3 | 2 |  |  |  |  |  | 3 |
| PARAMS |  | 4 |  | 5 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| STATEMENT\_N |  | 12 |  |  | 12 | 13 |  | 12 |  |  | 13 | 12 |  |  |  |  |  |  |
| STATEMENT |  | 10 |  |  | 11 |  |  | 8 |  |  |  | 9 |  |  |  |  |  |  |
| PARAMS\_N |  |  |  | 7 |  |  | 6 |  |  |  |  |  |  |  |  |  |  |  |
| DEF\_ARGS |  | 15 | 15 |  | 15 |  |  |  |  |  |  |  |  | 14 | 15 | 15 | 15 |  |
| DEFINE |  | \* |  |  | 18 |  |  |  | 17 |  |  |  |  |  |  |  |  |  |
| ARGS |  | 21 | 20 | 19 | 19 |  |  |  |  |  |  |  |  |  | 21 | 21 | 21 |  |
| VALUE |  | 26 |  |  |  |  |  |  |  |  |  |  |  |  | 24 | 25 | 27 |  |
| ARGS\_N |  |  |  | 23 | 23 |  | 22 |  |  |  |  |  |  |  |  |  |  |  |

\*Poznámka: Pravidlo 16 platí pro uživatelem definované funkce, pravidlo 17 pro proměnné.

## Precedenční tabulka – Pravidla

E → E + E

E → E - E

E → E \* E

E → E / E

E → (E)

E → i

E → int

E → float

E → string

E → E < E

E → E <= E

E → E > E

E → E >= E

E → E == E

E → E != E

## Precedenční tabulka

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

## Schéma konečného automatu

