

Projektová dokumentace

**Implementace překladače imperativního jazyka IFJ18**

Tým 28, varianta II.

**Tomáš Dorda xdorda00 25%**

Jakub Konetzný xkonet01 25%

Jan Osolsobě xosols00 25%

Patrik Strnad xstrna11 25%

**Obsah**

[Úvod 3](#_Toc531822726)

[Návrh a implementace 3](#_Toc531822727)

[Lexikální analýza 3](#_Toc531822728)

[Syntaktická analýza 3](#_Toc531822729)

[Sémantická analýza 3](#_Toc531822730)

[Generátor cílového kódu 3](#_Toc531822731)

[Generování návěští 3](#_Toc531822732)

[Generování výrazů 4](#_Toc531822733)

[Překladový systém 4](#_Toc531822734)

[Speciální algoritmy a datové struktury 4](#_Toc531822735)

[Hashovací tabulka 4](#_Toc531822736)

[Dynamický string 4](#_Toc531822737)

[Zpracování výrazů pomocí precedenční tabulky 4](#_Toc531822738)

[Lineární seznam na ukládání tokenů ze vstupu 5](#_Toc531822739)

[Práce v týmu 5](#_Toc531822740)

[Způsob práce v týmu 5](#_Toc531822741)

[Komunikace 5](#_Toc531822742)

[Verzovací systém 5](#_Toc531822743)

[Rozdělení práce mezi členy 5](#_Toc531822744)

[Závěr 6](#_Toc531822745)

# Úvod

Cílem projektu bylo implementovat překladač z jazyka IFJ18 do tří-adresného kódu IFJcode18. Jazyk IFJ18 je podmnožina jazyku Ruby. Překladač funguje jako konzolová aplikace, vstup se bere z standardního vstupu, a výsledný mezikód se generuje na standardní výstup.

# Návrh a implementace

Projekt je sestaven z několika dílčích částí, které jsou všechny kódovány podle standartu UTF-8. Pro lepší srozumitelnost je kód v vhodných místech komentovaný.

## Lexikální analýza

Celá lexikální analýza tkví primárně ve funkci getToken, který si po jednom bere znaky ze zdrojového souboru, a postupně tvoří token specifického druhu a hodnoty. Ihned v procesu generování tokenu detekuje lexikální chyby, a zastaví běh programu s vypsáním chybové hlášky. V případě celého čísla, nebo floatu, či dokonce čísla s exponentem, se navíc do podstruktury tokenu uloží do pole value hodnota zadaného čísla. Pokud je na vstupu string, načítá se písmenko po písmenku do naši vytvořené struktury dynamic\_string, který se jak název napovídá, dynamicky alokuje/zvětšuje podle potřeby. Lexikální analýza je vytvořena ve formě deterministického konečného automatu, který je uveden níže. Analýza v rámci jednoho tokenu se opakuje v nekonečném while cyklu, který vlastně nikdy sám o sobě neskončí, ale navrhnutý automat vždy vrátí hodnotu a tím pádem skončí.

## Syntaktická analýza

Při syntaktické analýze jsme při prvním průchodu naplnily list tokenů funkcí getToken a vložili jsme definované funkce a proměnné do tabulky symbolů. Při druhém průchodu jsme pracovali již s naplněným listem tokenů a zjišťovali jsme, zda je posloupnost daných tokenů syntakticky správná. Na začátku každého řádku jsme na základě typu prvního tokenu rozhodli, kterou funkci zavoláme přičemž funkce následně podle pravidel rozhodla zda je daný řádek syntakticky správně a tato činnost se opakuje dokud nezkontroluje poslední řádek.

## Sémantická analýza

Při sémantické analýze jsme pracovali s tokeny v tabulce symbolů kde jsme kontrolovali zda jsou proměnné či funkce definované a jakého jsou datového typu.

## Generátor cílového kódu

Generování cílového mezikódu IFJcode18. probíhá v průběhu analýzy. Po zdárném dokončení všech analýz se vygenerovaný kód vypíše na standardní výstup. Spolu z kódem se generují i komentáře, pro větší přehlednost kódu

## Generování návěští

Generování návěští pro funkce je ve tvaru $jmeno\_funkce. Pro funkce se generuje návratová hodnota, která se ukládá do globální proměnné. Ostatní návěští se generují pomoci zásobníku, který zajištuje unikátnost jednotlivých návěští. Jsou ve tvaru $%identifikator\_navesti.

## Generování výrazů

Všechny výrazy jsou ukládány na datový zásobník, a následně provedeny operace. Výsledek se uloží do globální proměnné a následně, pokud existuje přiřazení tak do lokální proměnné.

## Překladový systém

Projekt jsme během vývoje překládali i s CMake, i s GNU Make. Pro zjednodušení a sjednocení ale finálně používáme jen klasický GNU Make, protože ten je dle zadání nutný. K překladu se používá přiložený soubor Makefile, následně se poté provádí příkaz make. Výchozí výsledek make je jeden spustitelný soubor, který se poté dá spustit společně s argumentem ve formě souboru s IFJ18 kódem. Makefile také obsahuje příkazy pro smazání aktuálně vygenerovaných .o a .zip souborů, a zazipování výsledku společně s Makefile, souborem “rozdeleni” a dokumentací.

# Speciální algoritmy a datové struktury

## Hashovací tabulka

Implementovali jsme tabulku s rozptýlenými položkami sloužící jako tabulka k uchování symbolů. Pro zřetězení synonym jsme použili lineárně vázané seznamy. Velikost mapovacího pole jsme nastavili tak, aby byla prvočíslo, tudíž aby se zaručil průchod všemi položkami a její naplnění nepřesáhlo 75 %. Zvolili jsme 2039.

Pro mapovací funkci jsme jsme použili sbdm funkci(<http://www.cse.yorku.ca/~oz/hash.html>), která poskytuje dobrý rozptyl a jde lze dobře obecně používat. Každá položka tabulky obsahuje ukazatel na další prvek a ukazatel token ukazatel na token v kterém je uložen unikátní klíč v podobě řetězce a další prvky tokenu, jako třeba bool, který nás informuje o tom, zda byla položka již definovaná. Dále také datový typ, i do jakého setu token patří (sign, variable…). Mimo deklaraci samotné tabulky jsme implementovali i funkce co operují nad ní, a těmi jsou například její inicializace, vyhledání položky, odstranění položky nebo uvolnění celé tabulky z paměti.

## Dynamický string

Pro práci se stringy jsme si vytvořili praktickou strukturu Dynamic\_string, abychom nemuseli řešit manuálně práci s ukazatelem na char. Vše, co se týče našeho dynamického stringu, je zahrnuto v odděleném souboru Dynamic\_string (.c/.h). K dispozici máme nad dynamickým stringem funkci inicializace, smazání obsahu, přidání charakteru i celého stringu k existujícímu dynamickému stringu, a možnost dynamického rozšíření stringu před vložením obsahu, dle potřeby.

## Zpracování výrazů pomocí precedenční tabulky

Pro zpracování výrazů pomocí precedenční syntaktické analýzy jsme implementovali zásobník. V precedenční tabulce bylo možné si zjednodušit do jednoho sloupce a řádku operátory + - a také operátory \* / a všechny relační operátory. Implementace vyhodnocování výrazů je v souborech expressions.c a expressions.h.

## Lineární seznam na ukládání tokenů ze vstupu

Využíváme pro ukládání tokenů ze vstupního kódu. Využíváme obousměrně vázaný seznam, abychom měli přístup pro aktuální i prvek před aktuálním prvkem.

# Práce v týmu

## Způsob práce v týmu

První společnou poradu jsme měli koncem září, ale na projektu jsme začali až v průběhu října. Na začátku jsme si rozdělili jednotlivé části, které každý budeme implementovat, ale v průběhu práce na projektu jsme rozdělení zrušili, a každý pracoval na části, která mu víc seděla. Na jednotlivých částech pracovali jednotlivci, popřípadě dvojice, dle velikosti a náročnosti dané části.

## Komunikace

Jako hlavní komunikační kanál jsme používali Discord, ve kterém jsme měli kanál pro každou důležitou část, a zde komentovali průběh vývoje. Měli jsme také týmový disk, na který jsme ukládali soubory, a komentovali zadání. Vždy jednou týdně jsme se sešli na živo, a zhodnotili proběhlou práci, a vytyčili cíle, které chceme do dalšího sezení udělat. Během celého vývoje probíhali večerní hovory, na kterých jsme diskutovali aktuální problémy.

## Verzovací systém

Jako verzovací systém pro náš projekt jsme zvolili git. Projekt jsme založili na privátním GitLabu. 3 členové týmu využívali direktně CLion, tedy prostředí ve kterém jsme převážně projekt dělali, a čtvrtý používal SourceTree. V podstatě jsme všichni používali stejné git příkazy, šlo jen o uživatelské prostředí.

Konflikty v souborech, když už nějaké nastaly, jsme řešili jednoduše mergem. Jelikož šlo o textové soubory, git si s tím poradil a převážně nám správně spojil soubory. Pokud ne, tak jsme si mezi sebou řekli, co se změnilo - Jeden danou změnu provedl, druhý přijal stav souboru od druhého.

Použili jsme okrajově i soubor .gitignore, který sděluje, které soubory se nemají verzovat. U nás to byl CMakeLists, a .idea.

## Rozdělení práce mezi členy

Tomáš Dorda- vedení týmu, tabulka symbolů, lineární seznam, vyhodnocování výrazů

Jakub Konetzný- syntaktická analýza, sémantická analýza, testování

Jan Osolsobě- generování kódu, dokumentace, testování,

Patrik Strnad- lexikální analýza, testování, dokumentace

# Závěr

Projekt nám zabral mnoho času, ale nakonec se nám povedlo ho dovést do funkčního stavu. Bohužel s projektem jsme začali poměrně pozdě, a proto jsme byli většinu času ve stresu, zda-li to stihneme. Díky projektu jsme si osvojili práci v týmu, v praxi použili nové programovací techniky a algoritmy a naučili se využívat verzovací systém Git. Podařilo se nám stihnout obě pokusné odevzdání, díky kterým jsme zjistili nedostatky v našem projektu.

obrz 1


Legenda:

START = START MBE = MIGHT\_BE\_EQUAL

GT = GREATER\_THAN EQ = EQUALS

LT = LESS\_THAN SCH = STARTCHUNKCOMMENTARY

GEQ = GREATER\_OR\_EQUAL SCH = STARTCHUNKCOMMENTARYCONTINUE

LEQ = LESS\_OR\_EQUAL EX = EXCLAMATION

EOL = END\_OF\_LINE NEQ = NOT\_EQUALS

KW = KEYWORD CMT = COMMENT

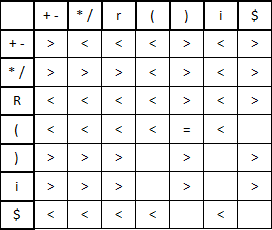
STR = STRING N = NUMBER

N\_D = NUMBER\_DEC N\_F = NUMBER\_FLOAT

N\_E = NUMBER\_EXP NES = NUMBER\_EXP\_SIGN

NED = NUMBER\_EXP\_DONE

Precedenční tabulka:



LL tabulka:

|  |  |
| --- | --- |
| <program> | ε |
| <program> | <statement> |
| <program> | def id ( <parameters> ) EOL <statement> END <program> |
|  |  |
| <statement> | ε |
| <statement> | if <expression> then <statement> END |
| <statement> | if <expression> then <statement> else <statement> END |
| <statement> | while <expression> do <statement> END |
| <statement> | id |
| <statement> | id <parameters> |
| <statement> | id ( <parameters> ) |
| <statement> | id = <expression> |
| <statement> | <statement> |
|  |  |
| <parameters> | ε |
| <parameters> | id <parameter\_n> |
|  |  |
| <parameter\_n> | , <parameter\_n> |
| <parameter\_n> | ε |