

# FAKULTA INFORMAČNÍCH TECHNOLOGIÍ VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

## **Dokumentácia – IFJ 2018**

Tím 40, varianta II

Adam Hostin	xhosti02	25 %
Sabína Gregušová	xgregu02	25 %
<b>Dominik Peza</b>	<b>xpezad00</b>	<b>25 %</b>
Adrián Tulušák	xtulus00	25 %

# Obsah

<b>1</b>	<b>Úvod</b>	<b>2</b>
<b>2</b>	<b>Lexikálna analýza</b>	<b>2</b>
2.1	Štruktúra Token.t . . . . .	2
2.2	Spracovanie reťazcov . . . . .	2
<b>3</b>	<b>Syntaktická analýza</b>	<b>2</b>
3.1	Precedenčná analýza výrazov . . . . .	2
3.2	Sématická analýza . . . . .	3
3.3	Tabuľka symbolov . . . . .	3
<b>4</b>	<b>Generátor inštrukcií</b>	<b>3</b>
<b>5</b>	<b>Práca v tíme</b>	<b>3</b>
5.1	Komunikácia . . . . .	3
5.2	Verzovanie . . . . .	3
5.3	Hodnotenie . . . . .	3
<b>A</b>	<b>Deterministický konečný automat</b>	<b>5</b>
<b>B</b>	<b>LL-gramatika</b>	<b>5</b>
<b>C</b>	<b>Precedenčná tabuľka</b>	<b>5</b>

# 1 Úvod

Naším cieľom je implementovať prekladač imperatívneho jazyka IFJ18 do predmetov IFJ a IAL v jazyku C. Hlavnou náplňou práce bola implementácia: lexikálneho analyzátoru, parsera (syntaktická a sématická analýza) a generátora inštrukcií.

## 2 Lexikálna analýza

Na začiatku sme implementovali lexikálny analyzátor v súbore `lexer.c`, ktorého základom je deterministický konečný automat (ďalej iba DKA). Hlavnou funkciou v tomto súbore je `get_next_token`, ktorá číta jednotlivé znaky a pomocou príkazu `switch` prechádza do nasledujúcich stavov podľa DKA až kým nevyhodnotí lexikálne správny token, inak vracia `ER_LEX`. Lexikálny analyzátor musí mať nadstavený vstupný súbor, ktorý obsahuje program napísaný v jazyku IFJ18. Pri úspešnom vyhodnotení tokenu sa správne uvoľní všetká alokovaná pamäť. Matematické a relačné operátory sú vyhodnotené vcelku rýchlo a jednoducho, identifikátory, reťazce a čísla vyžadujú viacej prechodov a používajú dynamický reťazec, o ktorom ďalej pojednáva sekcia Spracovanie reťazcov. Pri identifikátore sa kontrolujú povolené znaky na základe pozície v reťazci a na záver sa identifikátor porovná so všetkými kľúčovými slovami, ak sa nájde zhoda, je to kľúčové slovo, inak je to identifikátor. Reťazce sú ohraničené dvojitémi úvodzovkami (") a môžu obsahovať escape sequence. Pre tento prípad existuje špeciálny stav `STATE_BACKSLASH_LITERAL`, do ktorého sa prechádza pri prečítaní znaku `\` a čaká sa na skratku escape sequence, ako napríklad `t`, `s` alebo `n`.

### 2.1 Štruktúra `Token_t`

Pre jednoduchšiu prácu s tokenmi sme použili štruktúru `Token_t`, ktorá obsahovala:

- `union Token_attr`
- `struct Token_type`

**Union `Token_attr`** obsahuje možné atribúty tokenu, konkrétne to sú: `string`, `integer`, `flt` a `keyword`.

**Struct `Token_type`** obsahuje typy tokenov, konkrétne to sú: `EOF`, `EOL`, identifikátor, kľúčové slovo, relačné a matematické operátory, ľava a pravá zátvorka, čiarka, komentár, `int`, `float` a `string`.

### 2.2 Spracovanie reťazcov

Pre jednoduchšie spracovanie reťazcov sme sa rozhodli implementovať súbor `dynamic_string.c`. Jeho súčasťou je aj štruktúra `string_t`, ktorá obsahuje samotný ukazateľ na dynamický reťazec, súčasnú veľkosť reťazca a celkovú veľkosť bufferu. Na začiatku je alokovaný reťazec s veľkosťou 10 a pri každom pridaní znaku sa kontroluje, či je ešte v reťazci miesto. Keď sa blížime k zaplneniu reťazca, funkcia `check_empty_bites` zväčší veľkosť buffera o 5, čím zaistí adekvátnu veľkosť pre reťazec. Všetky alokácie pamäte sú kontrolované a ich zlyhanie je adekvátne ošetrené vrátením internej chyby `ER_INTERNAL`.

## 3 Syntaktická analýza

### 3.1 Precedenčná analýza výrazov

Syntaktická analýza výrazov je implementovaná v súbore `expression.c` pomocou precedenčnej tabuľky. Hlavné telo `expression.c` pozostáva z funkcie `handle_expression`, ktorá postupne spracováva jednotlivé tokeny alebo symboly a vyhodnocuje ich syntaktickú správnosť. Keďže nie je vždy úplne jednoznačne možné určiť, či nasledujúci token bude súčasťou výrazu alebo nie, rozhodli sme sa implementovať `buffer`, kam sa postupne ukladajú tokeny vo forme jendosmerne viazaného zoznamu. Ak parser vyhodnotí, že dané tokeny nie sú súčasťou výrazu, vyčistí `buffer` a syntaktická analýza pokračuje v parseri. V opačnom prípade je zavolaná funkcia `handle_expression`, ktorá pri spracovaní používa tokeny v `bufferi`, až kým nie je `buffer` prázdny a ďalej si žiada tokeny pomocou funkcie `get_next_token`. Z každého tokenu si vytvoríme symbol na základe jeho typu, ktorý si ďalej ukladáme na zásobník.

Spracovávanie symbolov je naprogramované na základe algoritmu uvedeného v prezentácii [1]. Symboly majú svoj status, ktorý sa kopíruje aj pri redukcii danej časti výrazu. Časti výrazu sa postupne dávajú na zásobník generátora a akonáhle už raz bol daný výraz daný do generátora, jeho status sa mení na `ON_GENERATOR_STACK` a pri ďalšej redukcii už nebude znova pridaný na zásobník generátora.

Pri spracovaní výrazov máme flag `return_code`, ktorý môže nadobúdať hodnotu:

- **EXPRESSION\_OK** - výraz má správnu syntax aj sématicku
- **SYNTACTICAL\_ERRORS** - výraz má syntaktickú chybu
- **UNDEFINED\_ID\_EXPRESSION** - výraz má sématickú chybu
- **ER\_INTERNAL** - interná chyba (neúspešný malloc, neúspešné pridanie do zásobníku a pod.)

## 3.2 Sématická analýza

## 3.3 Tabuľka symbolov

Tabuľku symbolov sme implementovali ako tabuľku s rozptýlenými položkami. Synonymá sú v tabuľke zoradené explicitne, čo zabezpečuje tereticky neobmedzený počet položiek uchovávateľných tabuľkou symbolov. Synonymá sú zretazené v jednosmerne viazaných zoznamoch. Veľkosť mapovacieho poľa sme vyberali tak, aby bola rovná prvočíslu. Naša tabuľka má veľkosť 6421. Očakávame, že naplnenie tabuľky nepresiahne 75%. Mapovaciu funkciu sme prevzali z druhej domácej úlohy z IAL, lebo nám pripadala efektívna a ľahko pochopiteľná. Funkcia spočítava ASCII hodnotu jednotlivých znakov kľúča a nakoniec vracia modulo veľkosti tabuľky z daného súčtu. Každá položka tabuľky obsahuje svoj vlastný unikátny kľúč, ktorý sa ukladá v podobe reťazca. Kľúče značia identifikátory funkcií a premenných. Každá položka taktiež obsahuje svoj typ, značiaci či sa jedná o premennú alebo funkciu, boolovskú hodnotu, určujúcu či bola položka definovaná, ukazateľ na ďalší prvok v zozname a integer značiaci počet parametrov v prípade, že sa jedná o funkciu. Položky sa ukladajú do 2 idetických tabuliek s rozptýlenými položkami v závislosti od toho, či sa jedná o lokálne alebo globálne premenné. Všetky funkcie sa ukladajú do globálnej tabuľky symbolov. Implementovali sme taktiež niekoľko funkcií zabezpečujúce pohodlnú prácu s tabuľkou. Funkcie zabezpečujú inicializáciu, výpis chybových hlásení, pridávanie prvkov, vyhľadávanie v tabuľke, kontrolu jednotlivých atribútov položiek a čistenie tabuľky.

# 4 Generátor inštrukcií

# 5 Práca v tíme

Náš tím sme si zostavili pomerne skoro. Po rozdelení práce na menšie celky sme začali každý pracovať na časti pridelenej vedúcim tímu. Približne 3 týždne pred pokusným odovzdaním sme začali jednotlivé časti spájať do celku.

## 5.1 Komunikácia

Už na začiatku sme sa dohodli na pravidelných týždenných stretnutiach, kde sme diskutovali o našej ďalšej práci na nadchádzajúci týždeň.

## 5.2 Verzovanie

Pre správu projektu sme používali verzovací systém Git a vzdialený repozitár GitHub. Tento spôsob správy projektu nám umožňoval pracovať na viacerých častiach projektu súčasne. Po otestovaní jednotlivých podčastí sme súbory začali spájať do celku.

## 5.3 Hodnotenie

## **Použitá literatúra**

- [1] MEDUNA, A.; LUKÁŠ, R.: Kapitola VIII. Syntaktická analýza zdola nahoru, rev. 1. septembra 2018, [vid. 3. novembra 2018].

**A Deterministický konečný automat**

**B LL-gramatika**

**C Precedenční tabulka**

	+	-	*	/	(	)	i	R	\$
+	>	<	<	<	<	>	<	>	>
-	>	>	<	<	<	>	<	>	>
*	>	>	>	>	<	>	<	>	>
/	<	>	>	>	<	>	<	>	>
(	<	<	<	<	<	=	<	<	
)	>	>	>	>		>		>	>
i	>	>	>	>		>		>	>
R	<	<	<	<	<	>	<		>
\$	<	<	<	<	<		<	<	