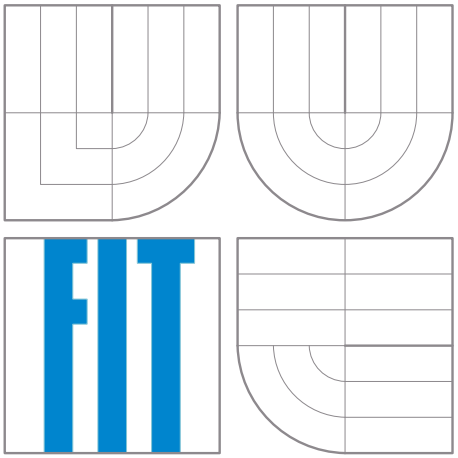
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



Dokumentace projektu do předmětů IFJ a IAL

# Interpret imperativního jazyka IFJ14

**varianta: b / 1 / II (tým 026)**

14. prosince 2014

**Řešitelé:**

| | |
|---|---|
| Motlík Matúš (vedoucí) | xmotli02 (20% bodového ohodnocení) |
| Marcin Juraj | xmarci05 (20% bodového ohodnocení) |
| Mour Lukáš | xmourl00 (20% bodového ohodnocení) |
| Měřínský Josef | xmerin02 (20% bodového ohodnocení) |
| Šikula Vojtěch | xsikul13 (20% bodového ohodnocení) |

**Rozšíření:**
REPEAT
ELSEIF

# Obsah

# 1 Úvod

Tato dokumentace popisuje implementaci projektu – interpretu imperativního jazyka IFJ14.

Projekt je součástí dvou předmětů – IFJ (Formální jazyky a překladače) a IAL (Algoritmy), Fakulty informačních technologií, Vysokého učení technického v Brně, zimní semestr 2014. Jedná se o skupinový projekt pro 4 – 5 lidí.

Úlohou interpretu je načtení zdrojového souboru (program v jazyce IFJ14) a jeho analýza. Pokud skončí úspěšně, načtený soubor by měl být správně interpretován.

# 2 Zadání

Jazyk IFJ14 je podmnožinou jazyka Pascal. Lze ho definovat jako staticky typovaný, procedurální jazyk. Podporuje definici proměnných a uživatelské funkce včetně rekurze. Přesná specifikace jazyka byla definována v zadání. Pro vypracování bylo vytvořeno několik variant, které se lišily algoritmy pro vyhledávání, řazení a typu tabulky symbolů.

## 2.1 Specifikované zadání

Náš tým si vybral variantu b / 1 / II:
b: vyhledávání podřetězce v řetězci podle libovolné heuristiky Boyer-Moorova algoritmu
1: řadící algoritmus Quick sort
II: tabulka symbolů implementována jako Hash table (tabulka s rozptýlenými položkami)

# 3 Struktura projektu

## 3.1 Lexikální analýza

Lexikální analýza je první fází procesu kompilace. Slouží pro čtení zdrojového programu a jeho přeložení na jednotlivé části nebo-li lexémy, což jsou například identifikátory, klíčová slova apod. V případě, že některý lexém je špatně zapsaný, tento analyzátor musí vrátit chybu v zapsaném lexému, v jiném případě vrací informace o tokenu a jeho atributy. Pro řešení lexikálního analyzátoru je velmi důležitý konečný automat, který má specifikovanou konečnou množinu stavů, ve které je obsažen počáteční stav a množina koncových stavů.

### 3.1.1 Řešení lexikálního analyzátoru

Hlavní částí lexikálního analyzátoru je funkce `getNextToken`, která vrací informace o úspěšném načtení a uložení tokenu, případně jeho chybné načtení, které může být způsobeno chybou v alokaci paměti, či špatnou posloupností znaků v souboru. Váhali jsme, jakou metodu využít pro vrácení informací o tokenu, ale nakonec jsme se rozhodli důležité informace o tokenu a atributy vracet pomocí ukazatele na strukturu, ve které je zapsáno právě o jaký token se jedná, respektive jeho hodnota, a v některých případech také jeho atribut (integer, double, string). Ten je řešený speciálním datovým typem union, ve kterém jsou tyto atributy již přiřazeny jednotlivému datovému typu.
A jak již bylo zmíněno, součástí lexikálního analyzátoru je konečný automat, jehož funkce je popsána grafem, viz Obrázek 1 v příloze.

## 3.2 Syntaktická a sémantická analýza

Syntaktický analyzátor (SA) kontroluje na základě tokenů správnost syntaxe. Správná syntaxe je definována pomocí LL-gramatiky. Dále syntaktický analyzátor provádí sémantické kontroly a generuje tříadresný kód.

SA pracuje na principu rekurzivního sestupu založeného na LL-tabulce (viz Obrázek 2 v příloze), která je vytvořená z LL-gramatiky.

Výstup z lexikálního analyzátoru (token) je vstupem pro analyzátor syntaktický. Pro každý token se SA snaží sestrojit derivační strom. Pokud se to nepovede, znamená to chybu syntaxe. Dále se zaznamenávají deklarace proměnných, které jsou pak uloženy do příslušných tabulek symbolů. Do globální tabulky symbolů se ukladají také konstanty a funkce. Ke každé funkci je pak uložen odkaz na její lokální tabulku symbolů obsahující její lokální proměnné a parametry. Zpracování výrazů provádí precedenční analyzátor.

Průběžně se provádějí sémantické kontroly, jako např. typová kontrola, správný počet a typ parametrů při volání funkce atd. Následně se generují instrukce, které se ukládají do seznamu instrukcí.

## 3.3 Precedenční analýza výrazů

Precedenční analýza (dále jen PA) slouží k vyhodnocování výrazů. Kdykoliv syntaktický analyzátor (dále jen SA) narazí v pravidle na výraz, zavolá PA, která mu poté zpracovaný výraz vrátí.

Vstupem PA je aktuální tabulka symbolů obsahující informace o proměnných. Další vstupní prvky pak tvoří ukazatel na list instrukcí, jelikož PA už přímo vytváří a ukládá instrukce v tříadresném kódu, které se poté budou interpretovat. PA potřebuje také vstupní token.

Výstupem PA je pak chybový kód a vygenerované instrukce. Pokud zpracování výrazu proběhlo bez chyby, SA očekává výsledek výrazu v unikátní proměnné uložené v tabulce symbolů, která je vytvořena pouze pro tento účel předání (má unikátně vytvořený název, tudíž je zaručeno, že nemůže kolidovat s uživatelskými proměnnými).

Precedenční analýza využívá ke své činnosti lexikální analyzátor, od něhož si může vyžádat další token. Při redukování pravidel PA jsou přímo generovány instrukce. Základním stavebním kamenem PA je precedenční tabulka (viz Obrázek 3 v příloze). V této tabulce jsou definována syntaktická pravidla pro všechny výrazy. Samotný algoritmus je pak tvořen cyklem, ve kterém jsou postupně zpracovávané jednotlivé tokeny pomocí precedenční tabulky a redukované pomocí redukčních pravidel:

1: E -> i
2: E -> (E)
3: E -> E operátor E

Jako pomocná datová struktura pro ukládání zredukovaných a zpracovávaných výrazů zde slouží zásobník.

## 3.4 Interpret

Pokud lexikální, syntaktický a sémantický analyzátor nenašel ve zdrojovém souboru chybu, dostává interpret na vstupu lineární seznam instrukcí v tříadresném kódu a provádí se samotné vykonání programu (tzv. interpretace). Každý příkaz v tříadresném kódu je zapsán pomocí uspořádané čtveřice: operátor, operand1, operand2 a výsledek. V tomto případě se jedná o celočíselnou proměnnou, která

určuje typ instrukce a tři ukazatele většinou ukazující do tabulky symbolů, případně na instrukci, na kterou se má skočit.

Protože jazyk IFJ14 podporuje funkce, je třeba zajistit, aby se při případném rekurzivním volání jednotlivé lokální proměnné nepřekrývaly. Tento problém je řešen pomocí abstraktního datového typu zásobník, kdy prováděnou funkci reprezentuje vrchol zásobníku. Každá položka obsahuje adresu na lineární seznam lokálních proměnných a informace, kam skočit po skončení funkce, kam vložit návratovou hodnotu a samozřejmě adresu další položky zásobníku. Speciální lokální proměnná se shodným jménem jako funkce je umístěna implicitně na začátek lineárního seznamu lokálních proměnných, což vede k efektivnějšímu předání návratové hodnoty.

# 4 Vestavěné funkce

Pro práci s řetězci jsme využili již implementovaného řešení z jednoduchého interpretu v souboru `str.c`, který nám byl k dispozici.

## 4.1 Řazení

Součástí jazyka IFJ14 je vestavěná funkce `sort`, která využívá k tomu odpovídající algoritmus dle varianty zadání.

### 4.1.1 Quick sort

Dle zadání jsme pro řazení využili řadící algoritmus Quick sort, nebo-li řazení rozdělováním. Tento algoritmus je považován za nejrychlejší a nejúčinnější metodu řazení. Jeho principem je stanovení jednoho prvku, tzv. pivota, což je jeden prvek, který stojí mezi dvěma podmnožinami, na které se rozdělí zbývající řazené prvky. Pivot, jako prvek v poli, by měl být optimálně mediánem. Stanovení mediánu je ovšem náročná operace, proto je využit tzv. pseudomedián, hodnota ze středu intervalu, určená jako `((první prvek + poslední prvek) / 2)`. Je prokázáno, že pseudomedián dokáže funkci mediánu účelně zastat a jeho použití je ekvivalentní. Pokud je pivot stanoven, rozdělení na dvě podmnožiny využívá algoritmus, který musí znát dva indexy – první prvek (levý index) a poslední prvek řazeného pole (pravý index). Poté se od levého indexu směrem doprava vyhledává první hodnota větší než je pivot a směrem doleva se ve druhé části vyhledává hodnota, která je první menší a je provedena jejich záměna dle jejich velikosti. Pokud dojde k překřížení indexů, toto rozdělování končí a dostáváme první podmnožinu, kde jsou čísla menší než je pivot, pivot, jako prvek stojící samostatně a druhou podmnožinu, která obsahuje čísla větší než je daný pivot.

Samotný hlavní algoritmus Quick sortu je již velice jednoduchý, obzvláště ten rekurzivní, který byl využit a implementován v našem projektu. Existuje však také nerekurzivní varianta.

Pro implementaci byly využity znalosti získané z předmětu IAL a inspirací byly též materiály k tomuto předmětu.

## 4.2 Vyhledávání

Pro vyhledávání podřetězce v řetězci v jazyce IFJ14 slouží vestavěná funkce `find`. Opět byl použit algoritmus dle varianty zadání.

### 4.2.1 Boyer-Moore algoritmus

Boyer-Moore algoritmus patří společně s Knutt-Morris-Pratt algoritmem k nejvýznamějším v oblasti vyhledávání, avšak první zmíněný je považován za dokonalejší, při vhodné implementaci dokonce několikrát rychlejší. Boyer-Moore algoritmus je možné využít pomocí dvou heuristik. Dle zadání bylo možné vybrat heuristiku dle libosti, zvolena byla heuristika první.

Principem první heuristiky je porovnávání vzorku (to, co vyhledáváme) s řetězcem (ve kterém vyhledáváme). Začíná se porovnáním znaků ve vzorku zprava doleva s daným řetězcem. Pokud se žádný ze znaků ve vzorku nerovná dané části řetězce, dojde k posunu vzorku o celou jeho délku a opět nastává porovnání. Pokud nastane shoda, porovnává se další znak vlevo a pokud se ten nerovná, dojde k posunu vzorku, avšak ne o celou délku, ale za poslední znak, který se shodoval. Takto probíhá porovnání v celém řetězci, dokud nenajde shodu celého vzorku v řetězci. Pokud taková shoda není nalezena, řetězec tento podřetězec neobsahuje.
Algoritmus byl doplňen, aby fungoval case insensitive a bylo ošetřeno zacyklení v některých případech.

Pro implementaci byly využity znalosti získané z předmětu IAL a inspirací byly též materiály k tomuto předmětu.

# 5 Tabulka symbolů

Pro tabulku symbolů jsme opět dle varianty zadání využili tabulku s rozptýlenými položkami (Hash table). Hashovací tabulka byla převzata z druhého projektu předmětu IJC (autor Motlík Matúš), samotná implementace funkcí se jen upravila pro potřeby projektu.

Při vyhledávání v tabulce se počítá hash z klíče, udělá se modulo podle velikosti tabulky a vrátí se index pole, kde se daná položka nachází. V případě kolize se iteruje přes odkaz a porovnávají se klíče. Velikost tabulky je dána prvočíslem 193, protože na naší testovací sadě programů většinou počet klíčů nepřesáhl 75% velikosti tabulky.

V projektu je použito několik tabulek symbolů, jedna globální, pro globální proměnné, konstanty, pomocné proměnné potřebné při řešení výrazů a pro funkce. Samotné položky pro funkce dále obsahují také odkazy na své lokální tabulky, kde jsou uloženy jejich lokální proměnné a parametry. Ukládání hodnot proměnných a také položek pro proměnné a funkce do globální tabulky symbolů je řešeno pomocí unionu.

# 6 Rozšíření

## 6.1 REPEAT

Cyklus repeat je obdobou cyklu while s tím rozdílem, že proběhne vždy minimálně jednou, protože podmínka cyklu se vyhodnocuje až na konci. Tělo cyklu obsahuje vždy alespoň jeden příkaz. Vzhledem k značné podobnosti cyklů while a repeat byla implementace poměrně jednoduchá.

## 6.2 ELSEIF

Také jsme implementovali rozšíření pro podmíněný příkaz if-then bez části else.

# 7 Vývoj

## 7.1 Řešitelský tým

Náš tým byl sestaven na základě vzájemné známosti v době zadání projektu. Na prvním setkání byl stanoven vedoucí týmu. Dále jsme si rozdělili jednotlivé části, na kterých bude kdo pracovat.

## 7.2 Rozdělení práce

**Matúš Motlík** - Vedoucí projektu, syntaktický analyzátor, tabulka symbolů, testování
**Juraj Marcin** - Precedenční analýza výrazů
**Lukáš Mour** - Algoritmy, testování, dokumentace
**Josef Měřínský** - Lexikální analyzátor
**Vojtěch Šikula** - Interpret, testování

## 7.3 Komunikace v týmu

Hlavním komunikačním prostředkem byly týmové schůzky, na kterých se prezentovala odvedená práce ostatním členům a plánovala další práce do budoucna. Tyto schůzky se konaly přibližně 1x týdně.
Mezi další komunikační prostředky byly též využívány sociální sítě.

## 7.4 Správa souborů

Využit byl soukromý repozitář verzovacího systému GitHub. Žádný z členů týmu s tímto systémem neměl zkušenosti, proto nám tvorba projektu přinesla i znalost a zkušenost s jeho využíváním.

# 8 Závěr

Pro ověření funkčnosti projektu jsme využili obě pokusná odevzdání, první s celkovým výsledkem 67%, druhé 75%. Funkčnost projektu byla důkladně otestována na serveru Merlin i na 64-bitové platformě Linux Ubuntu.

Tato dokumentace byla vytvořena v textovém sázecím systému LaTeX.

## 8.1 Metriky

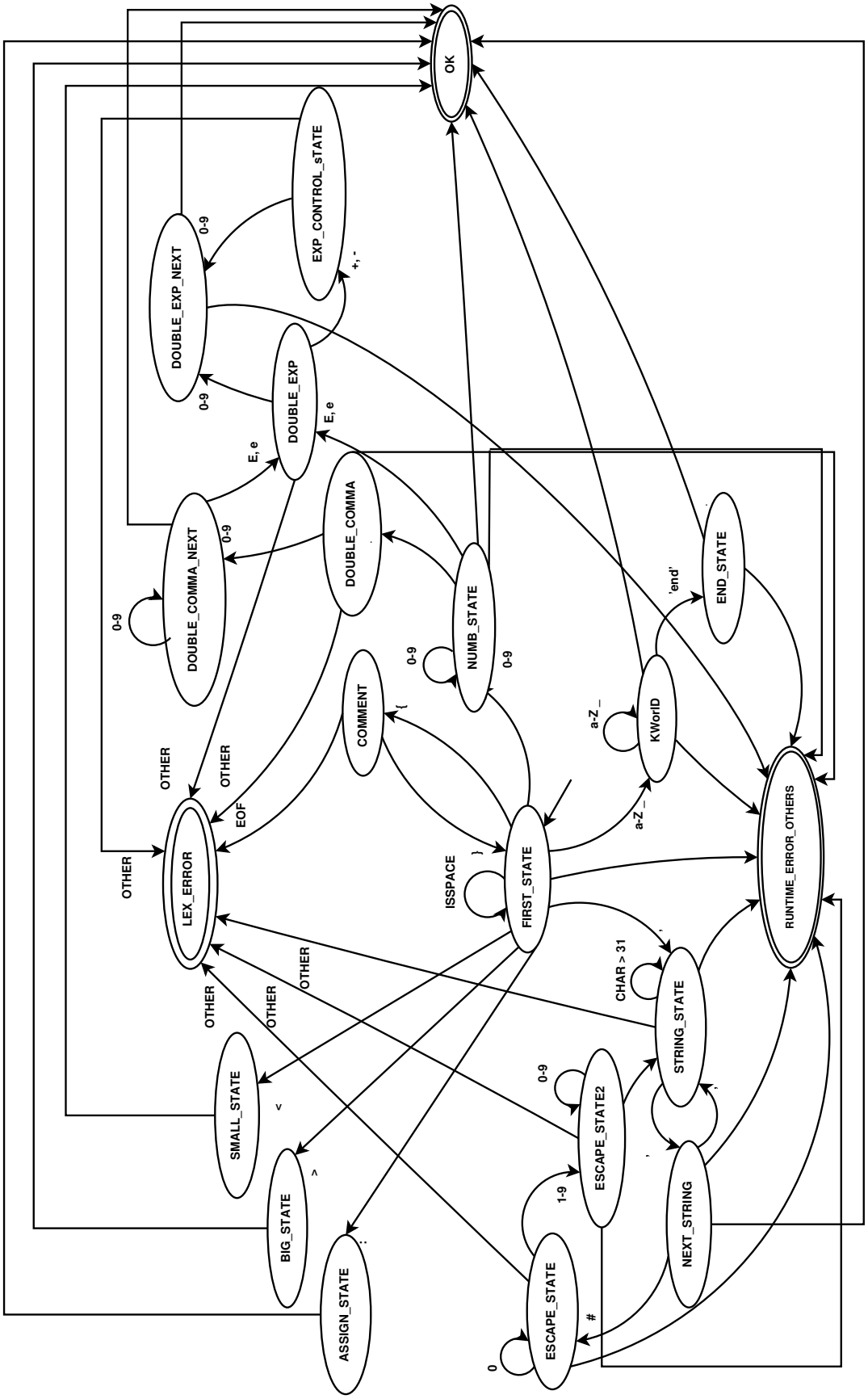
1) Počet zdrojových souborů: 18

2) Počet řádků kódu včetně komentářů: 6773

3) Velikost spustitelného souboru: 139,6 kB (Linux Ubuntu 64-bit)

## 8.2 Použité zdroje

1) Studijní opora IAL (Algoritmy)
2) Přednášky předmětu IAL (Algoritmy)
3) Přednášky předmětu IFJ (Formální jazyky a překladače)
4) Konzultace s odborným asistentem




**Obrázek 1:** Konečný automat lexikálního analyzátoru

1. parse → var id:<typ>;<premenne>EOF
2. parse → begin<telo>end. EOF
3. parse → function idF(<param_list>):<typ>;<funkcia><decllist>EOF
4. premenne → id:<typ>; <premenne>
5. premenne → <decllist>
6. decllist → begin <telo>end.
7. decllist → function idF(<param_list>):<typ>;<funkcia><decllist>
8. param_list → id:<typ><param_list2>
9. param_list → $\epsilon$
10. param_list2 → ; id:<typ><param_list2>
11. param_list2 → $\epsilon$
12. funkcia → forvard ;
13. funkcia → var id:<typ>; <decllist2>
14. funkcia → begin <telo>end;
15. decllist2 → id:<typ>; <decllist2>
16. decllist2 → begin <telo>end;
17. telo → <prikaz><telo3>
18. telo → $\epsilon$
19. telo3 → ; <telo>
20. telo3 → $\epsilon$
21. prikaz → while PA do <zloz_prikaz>
22. prikaz → if PA then <zloz_prikaz>else <zloz_prikaz>
23. prikaz → begin <telo>end
24. prikaz → id := <vyraz>
25. prikaz → readln (id)
26. prikaz → write ("term"<term_list>)
27. vyraz → PA
28. vyraz → idF ( <param>)
29. vyraz → find( id, id)
30. vyraz → copy (id, id, id)
31. vyraz → sort (id)
32. vyraz → length (id)
33. term_list → ,"term"<term_list>
34. term_list → $\epsilon$
35. zloz_prikaz → begin <telo>end
36. param → id <param2>
37. param → $\epsilon$
38. param2 → , id <param2>
39. param2 → $\epsilon$

**Obrázek 2:** LL gramatika

| Tabulka PA | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | Vstupní token | | | | | | |
| | | ( | ) | + - | * / | compare | i | $ |
| vrcholu zásobníku | ( | > | = | > | > | > | > | ERROR |
| | ) | ERROR | < | < | < | < | ERROR | < |
| | + - | > | < | < | > | < | > | < |
| | * / | > | < | < | < | < | > | < |
| | compare | > | < | > | > | < | > | < |
| | i | ERROR | < | < | < | < | ERROR | < |
| | $ | > | ERROR | > | > | > | > | ERROR |

**Obrázek 3:** Precedenční tabulka pro výrazy