



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

IFJ - DOKUMENTACE PROJEKTU
IFJ - PROJECT DOCUMENTATION

SEMESTRÁLNÍ PROJEKT
TERM PROJECT

AUTORI PRÁCE
AUTHORS

**Marek Tamaškovič, Martin Vaško,
Michal Vaško, Jiří Záleský, Jakub Zárybnický**

BRNO 2017

Abstrakt

Dokumentácia popisuje implementáciu interpretu jazyka IFJ16, ako podmnožinu jazyka java8, bez podpory objektového programovania. Projekt sa dá rozdeliť na štyri hlavné časti, z ktorých bude každej venovaná osobitná kapitola.

- Lexikálny analyzátor, ktorý zo zdrojového programu získava tokeny
- Syntaktický analyzátor, ktorý rozdeľujeme na dve podčasti- syntaktický analyzátor jazykových konštrukcií a na analyzátor výrazov.
- Sémantický analyzátor, ktorý v zdrojovom programe zisťuje, či konštrukcie s ktorými sa pracuje v programe nachádzajú v globálnom resp. lokálnom priestore.
- Interpret, ktorú ma za úlohu previesť interpretáciu programu.

Obsah

1	Štruktúra projektu	2
1.1	Lexikálna analýza	2
1.2	Syntaktická analýza	2
1.3	Sémantická analýza	3
1.4	Interpret	3
1.5	Garbage collector	3
1.6	Algoritmy	3
1.6.1	List-Merge sort	3
1.6.2	KMP Vyhľadání podřetězce	4
1.6.3	Binárny vyhľadávací strom - BVS	4
2	Vývoj	5
2.1	Rozdelenie práce	5
2.2	Komunikačné kanály	5
3	Záver	6
3.1	Metriky	6
4	Príloha	7
4.1	Diagram lexeru	7
4.2	LL gramatika	8
4.3	Precedenční tabulka	9

Kapitola 1

Štruktúra projektu

1.1 Lexikálna analýza

Lexikálna analýza alebo teda Scanner funguje na princípe konečného automatu. Postupne načítava znaky zo vstupného súboru a posiela ich syntaktickému analyzátoru vo forme tokenov. Poslany token obsahuje informácie o type, obsahu a pozícii (riadok, stĺpec) na ktorej sa v interpretovanom subore nachádza. Všetky tieto informácie su potrebné pre uľahčenie práce parseru. V prípade ,že načítame niečo čo nezapadá do pravidiel programovacieho jazyka IFJ16 nastane lexikálna chyba. Náš scanner podporuje aj všetky dostupne a povolené rozšírenia ako napríklad unárne operatory či rozšírenie BASE a ďalšie. Diagram konečného automatu si môžete prezrieť v priloženej prílohe. V diagrame došlo k upravám, kvôli prehľadnosti stavov, preto stavy v diagrame úplne nezodpovedajú stavom v kóde.

1.2 Syntaktická analýza

Syntaktická analýza byla původně psaná jako pokus o reimplementaci Parsecu (<https://hackage.haskell.org/package/parsec>) v C. Pak mi někdo připomněl, že se máme LL gramatiky držet docela blízko, takže hlavní parser je teď klasický *recursive descent parser*. (Jsou tam pozůstatky předchozí verze, která pomocí spojového seznamu a zásobníku implementovala *backtracking*.)

Výrazy jsou dle zadání zpracované pomocí *operator-precedence parseru*. Rozšíření UNARY vyžadovalo úpravu vstupu, na začátku vstupu, po otevírací závorce nebo po operátoru se — přepíše na *u—*, tj. unární mínus.

Výstup parseru je pak naplněná tabulka symbolů, s třídami (pro kontrolu redefinice), funkcemi a proměnnými (s jejich plně kvalifikovanými názvy, ať nemusíme pro každou třídu mít oddělenou tabulku). Funkce jsou implementované jako spojový seznam příkazů, ty jsou pak *tagged union* všech možných příkazů (for, funcall, blok apod.). Výrazy jsou reprezentované jejich AST, pomocí rekurzivní *tagged union*. Seznamy argumentů (při volání nebo deklaraci funkce) jsou opět implementované pomocí spojového seznamu.

U parsování výrazu jsem si nejdřív špatně přečetl zadání a implementoval taky *recursive descent parser* (s backtrackováním). Při reimplementaci jsem si ale nemohl nevšimnout, že výsledný kód je mnohem méně čitelný, tři- až čtyřikrát delší a algoritmus se ladil o dost hůř.

1.3 Sémantická analýza

Sémantická analýza kódu prebieha samostatne po syntaktickej analýze, z dôvodu jazyka IFJ16(Podmnožina java8), ktorý podporuje používanie staticky typovaných premenných aj v iných triedach. Dochádzalo by preto ku konfliktom(nevyplneným údajom v tabuľke symbolov) v prvom prechode syntaktickej analýzy. Rozšírenia ako CYCLES a BOOLOP museli rozšíriť sémantické kontroly o príkazy - break,continue, a rekurzívne spracovanie booleovských podmienok.

1.4 Interpret

Interpret má za úlohu vykonať to, čo sa nachádza v zdrojovom kóde interpretovaného programu. Náš interpret interpretuje abstraktný syntaktický strom, ktorý vytvorila syntaktická analýza. Ten lineárne prechádza a vyhodnocuje výrazy pokým nenastane volanie funkcie. V tom momente si vyhladá v tabuľke symbolov abstraktný syntaktický strom danej funkcie, vytvorí lokálnu tabuľku symbolov, ktorú bude používať volaná funkcia, vloží do nej argumenty funkcie a začne vykonávať telo funkcie. Pri ukončovaní funkcie interpret vloží návratovú hodnotu z funkcie na zásobník a ukončí interpretáciu funkcie. Následne si interpret danú hodnotu vyberie zo zásobníka a použije ju v interpretácii pôvodného abstraktného syntaktického stromu. Vstavané funkcie sú riešené obdobne. Taktiež si pri interpretácii interpret kontroluje behové chyby ako napríklad delenie nulou alebo práca s neinicializovanými premennými. Ak taká situácia nastane interpret sa ukončuje s chybovou hláškou a príslušným návratovým kódom pre danú chybu.

1.5 Garbage collector

Pro lepší správu pamäti jsme implementovali garbage collector. Je implementován jako statické pole struktury obsahující void ukazatel a ukazatel na další prvek stejného typu. Což umožňuje za sebou prvky řadit jako lineární spojový seznam. Po naalokování paměti je volána funkce, která vytvoří nový prvek. Na základě adresy je posléze pomocí hashovací funkce určeno nové umístění prvku. Je-li již na daném umístění nějaký prvek, je umístěn za poslední prvek v pořadí. Při mazání prvku je použita hashovací funkce a následným porovnáváním se najde daný prvek ve spojovém seznamu a po přepojení prvků je paměť, a následně i prvek, uvolněna.

Součástí jsou i modifikované funkce pro alokaci, realokaci a dealokaci paměti. Ve funkcích pro alokaci a realokaci je také kontrola toho, zda alokace proběhla úspěšně.

1.6 Algoritmy

1.6.1 List-Merge sort

Metoda řazení pole využívající princip slučování (mergování) seznamů. V první fázi je vytvořeno pomocné pole indexů, které je následně využito pro najití začátků neklesajících posloupností. Konec posloupnosti je v pomocném poli vyznačen pomocí 0. Začátek každé posloupnosti je uložen do pomocné datové struktury typu seznam a přidán do fronty.

Ve fázi druhé, pak vždy vezmeme první dva prvky fronty a seřadíme je. Z těchto dvou prvků vznikne jeden větší, který umístíme na konec fronty. Takto postupujeme dokud

nezůstane ve frontě pouze jeden seznam, který je výsledkem řazení. Pro větší jednoduchost kódu, je před spojováním a řazením zaručeno pořadí tak, aby seznam obsahující nižší index byl vždy první z dvojice pro seřazení.

1.6.2 KMP Vyhledání podřetězce

Knuth-Morris-Prattův algoritmus se využívá pro urychlení hledání výskytu stejných podřetězců v hledaném řetězci. Algoritmus využívá pomocného pole obsahující informace o hledaném podřetězci. Toto pole musí být vytvořeno ještě předtím než je zahájeno hledání. Složitost algoritmu je $\mathcal{O}(n)$.

V první fázi je zavolána pomocná funkce Prefixcreator, která do pomocného pole uchovávajícího pozice od kterých se má pokračovat ve porovnávání, umístí hodnoty na základě nalezených podřetězců. Není-li podřetězec větší než 1 pak je pomocná tabulka vyplněna -1. Ve fázi druhé již probíhá porovnávání hledaného řetězce s řetězcem vstupním. Výsledkem je pozice na, které začíná hledaný řetězec.

1.6.3 Binárny vyhľadávací strom - BVS

Pre vkladanie premenných, funkcií a tried sme v zadaní využili binárny vyhľadávací strom AVL. Základom stromu je vloženie a nájdenie uzlu. V rámci vloženia navyše strom vyvažujeme čo uľahčuje následne vyhľadanie uzlu v strome. Pre vkladanie máme 3 typy uzlov- uzly pre vstavané funkcie, uzly pre jednoduchú sémantickú kontrolu a uzly s konkrétnymi hodnotami. Pre jazyk IFJ16 sme zaviedli aj vyhľadávanie v dvoch stromoch naraz (lokálny a globálny) spolu so zistením mena triedy, z dôvodu rovnakých názvov funkcií v rámci rôznych tried.

Vkladanie je založené na porovnávaní kľúčov. Ak je kľúč ktorý vkladáme menší (v abecede viac naľavo) vkladáme do ľavého uzlu podstromu, inak vkladáme do pravého uzlu podstromu. Pri zhode kľúčov nastáva sémantická chyba z dôvodu možného pretypovania už existujúceho kľúča (názvu funkcie, premennej alebo triedy)

Kapitola 2

Vývoj

2.1 Rozdelenie práce

- **Martin Vaško** - Algoritmy(BVS),sémantické kontroly,testovanie,správa, návrh a výpomoc vrámci interpretu.
- **Marek Tamaškovič** - Interpretácia AST, dokumentácia, garbage-collector, testovanie.
- **Michal Vaško** - Lexikálny analyzátor,dokumentácia,prezentácia.
- **Jiří Záleský** - Algoritmy(List-Merge sort,KMP), sémantické kontroly, vstavané funkcie, testovanie.
- **Jakub Zárybnický** - Parser, precedenčná a syntaktická analýza, návrh, rozdelenie práce a štýl kódu.

2.2 Komunikačné kanály

Online komunikácia prebiehala v 3 typoch - pre rozdelenie práce sme používali **Trello**¹ kde sme si rozdeľovali nové úlohy, rozšírenia a detaily. Pre menej formálnu komunikáciu a výpomoc navzájom sme sa stretávali na komunikačnom kanály **Slack**² - kde sme si spisovali blížiac sa termíny a do osobitnej zložky #announcements sa nám zasielal výpis z prekladu **Travis-CI**³.

Ako verzovací systém sme si zvolili **Git**⁴.

Ako tím sme mali naplánované stretnutia pravidelne každé 2 až 3 týždne, kde sme preberali náš aktuálny stav a prognózu do ďalších dní, riešili rozhrania jednotlivých modulov a implementáciu rozšírení.

¹<https://trello.com/>

²<https://slack.com/>

³<https://travis-ci.org/>

⁴<https://git-scm.com/>

Kapitola 3

Záver

Tento projekt, implementácia interpretu IFJ16, bol veľkou výzvou hlavne po kooperatívnej stránke. Najväčší problém pri kooperácii bolo zdieľanie zdrojových kódov, cez verziovací systém, ktorý sme sa ku koncu projektu naučili lepšie používať. Náš interpret bol testovaný viacerými testami z vlastnej dielne a taktiež na viacerých operačných systémoch AntergOS, Fedora 22, Ubuntu 16.04.

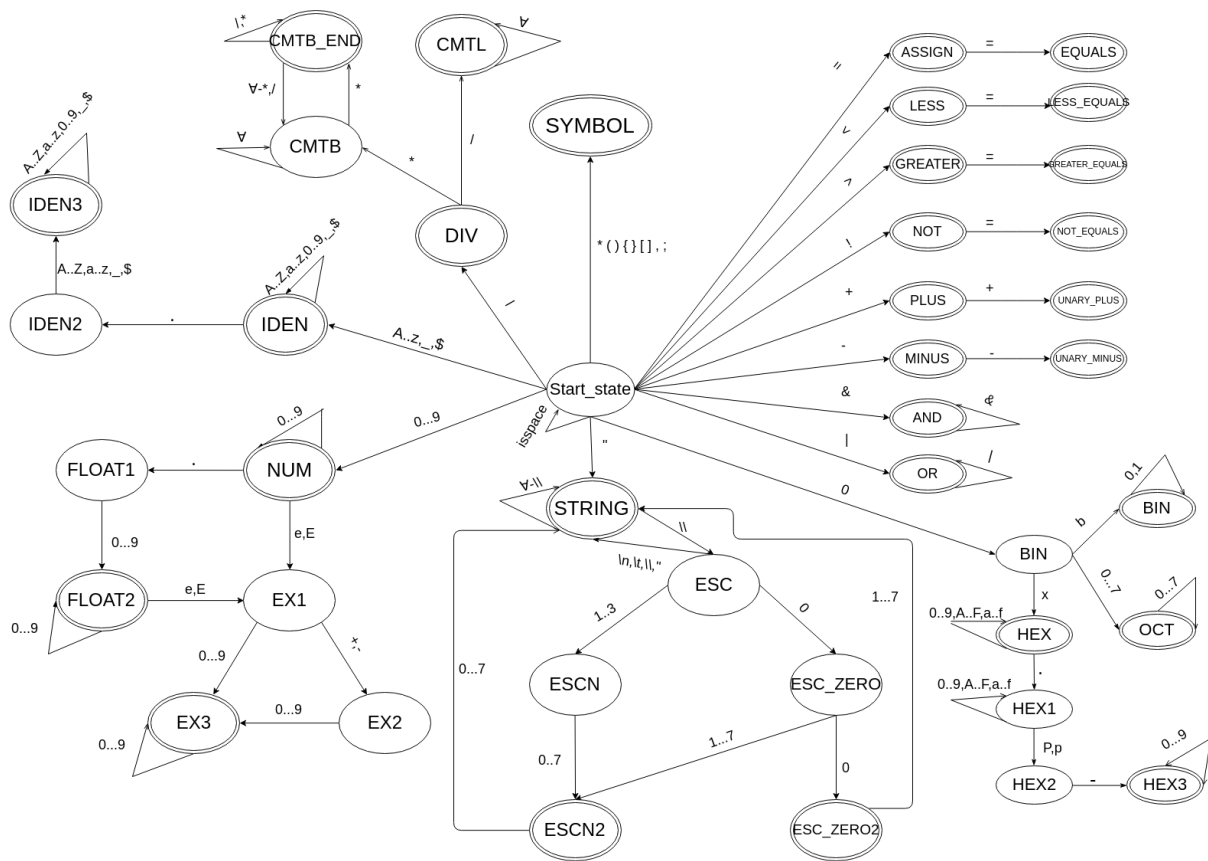
3.1 Metriky

- Počet súborov: **97** (**27** v C)
- celkový počet riadkov zdrojového kódu: **6622** (**5272** v C)
- Počet Git commitov: **392**

Kapitola 4

Príloha

4.1 Diagram lexeru



4.2 LL gramatika

```
ifj16 = \$ | class ifj16
class = "class" "{" classBody
classBody = "}" | "static" type simpleId classBody '
classBody ' = ";"
           | "=" expression ";"
           | "(" declarationList "{" functionBody
functionBody = "}"
           | type simpleId(q) functionBody '
           | command
functionBody ' = ";" | "=" expression ";"
command = "if" "(" expression ")" command
        | "while" "(" expression ")" command
        | "do" command "while" "(" expression ")" ";"
        | "for" "(" type simpleId for '
        | "return" return '
        | "{" commandList
        | "break" ";"
        | "continue" ";"
        | anyId command '
for ' = ";" for ' ' | "=" expression ";" for ' '
for ' ' = expression ";" anyId "=" expression ")" command
return ' = ";" | expression ";"
commandList = "}" | command commandList
command ' = "(" argumentList ";" | "=" expression ";"
declarationList = ")" | type simpleId declarationList '
declarationList ' = ")" | "," type simpleId declarationList '
argumentList = ")" | expression argumentList '
argumentList ' = ")" | "," expression argumentList '
type = "int" | "double" | "boolean" | "String" | "void"
anyId = simpleId | compoundId

expression = [see operator-precedence grammar]
```

4.3 Precedenční tabulka

	()	++	-	u-	!	*	/	+	-	i	¿	i=	¿=	=	!=	&&		id	li	tf	,	\$
(L	E	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	E	O
)	O	G	O	O	O	O	G	G	G	G	G	G	G	G	G	G	G	G	O	O	O	G	G
++	O	G	G	G	O	O	G	G	G	G	G	G	G	G	G	G	G	G	L	O	O	G	G
-	O	G	G	G	O	O	G	G	G	G	G	G	G	G	G	G	G	G	L	O	O	G	G
u-	L	G	O	O	G	G	G	G	G	G	G	G	G	G	G	G	G	G	L	G	L	O	G
!	L	G	O	O	G	G	G	G	G	G	G	G	G	G	G	G	G	G	L	G	L	O	G
*	L	G	L	L	L	L	G	G	G	G	G	G	G	G	G	G	G	G	L	L	L	G	G
/	L	G	L	L	L	L	G	G	G	G	G	G	G	G	G	G	G	G	L	L	L	G	G
+	L	G	L	L	L	L	L	L	G	G	G	G	G	G	G	G	G	G	L	L	L	G	G
-	L	G	L	L	L	L	L	L	G	G	G	G	G	G	G	G	G	G	L	L	L	G	G
i	L	G	L	L	L	L	L	L	L	L	L	G	G	G	G	G	G	G	L	L	L	G	G
¿	L	G	L	L	L	L	L	L	L	L	L	G	G	G	G	G	G	G	L	L	L	G	G
i=	L	G	L	L	L	L	L	L	L	L	L	G	G	G	G	G	G	G	L	L	L	G	G
¿=	L	G	L	L	L	L	L	L	L	L	L	G	G	G	G	G	G	G	L	L	L	G	G
=	L	G	L	L	L	L	L	L	L	L	L	L	L	L	G	G	G	G	L	L	L	G	G
!=	L	G	L	L	L	L	L	L	L	L	L	L	L	L	G	G	G	G	L	L	L	G	G
&&	L	G	L	L	L	L	L	L	L	L	L	L	L	L	L	L	G	G	L	L	L	G	G
	L	G	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	G	L	L	L	G	G
id	E	G	G	G	O	O	G	G	G	G	G	G	G	G	G	G	G	G	O	O	O	G	G
literal	O	G	O	O	O	O	G	G	G	G	G	G	G	G	G	G	G	G	O	O	O	G	G
true/false	O	G	O	O	O	O	G	G	G	G	G	G	G	G	G	G	G	G	O	O	O	G	G
,	L	E	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	E	O
\$	L	O	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	O	O