VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Dokumentace projektu do předmětů IFJ a IAL

Interpret jazyka IFJ16

Tým 90, varianta b/1/II

11. prosince 2016

Jan Kotas (vedoucí):	xkotas07	25%
Pavel Šiška:	xsiska12	25%
Michal Szekeres:	xszeke01	25%
Dominik Tran:	xtrand00	25%

Obsah

1. Úvod

Tato dokumentace se zabývá popisem tvorby projektu do předmětu Formální jazyky a překladače (IFJ). Cílem projektu bylo vytvořit interpret pro objektově orientovaný jazyk IFJ16, který je podmnožinou jazyka Java SE 8.

Dokumentace se skládá z několika částí. V první části je popsán způsob, jakým jsme v rámci týmu pracovali, jaké jsme si zvolili komunikační prostředky, nástroje a metodiku vývoje. V druhé části jsou popsány všechny podstatné prvky našeho interpretu. Závěr dokumentace je věnován hodnocení naší práce.

2. Organizace projektu

Správná organizace projektu je velmi důležitá. Pokud má být projekt úspěšně dokončen, je třeba především dobře naplánovat všechny činnosti a předem se dohodnout na komunikaci a používaných nástrojích.

2.1. Příprava

V rámci přípravy jsme domluvili první Kick-off schůzku. Na schůzce jsme vypracovali časový plán, kde jsme si stanovili přibližné termíny a jednotlivé fáze vývoje. Také jsme se dohodli na rozdělení práce, které vypadalo následovně:

Jan Kotas - Vedoucí, plánování, návrh, testování, dokumentace

Pavel Šiška - Syntaktická analýza, interpret

Michal Szekeres - Precedenční a sémantická analýza

Dominik Tran - Lexikální analyzátor, vestavěné funkce

2.2. Komunikace v týmu

Výhodou našeho týmu bylo, že jsme až na Dominika bydleli v jednom bytě. Díky tomu jsme mohli řešit většinu otázek osobně. Ale i přesto jsme si stanovili komunikační kanály. Ke komunikaci jsme používali především Facebook nebo Skype. Skype se díky funkci sdílení obrazovky stal užitečným nástrojem pro refaktoring a vzájemnou konzultaci kódu.

K tomu, abychom měli stále přehled o dění v týmu, jsme zavedli pondělní schůze, kde jsme vždy probrali, kdo na čem pracoval a jaký bude náš další postup. Na každé pondělní schůzi vznikl zápis, kde byly sepsány prioritní problémy, které bylo třeba vyřešit. Součástí zápisu bylo i rozdělení úkolů na následující týden.

2.3. Metodika vývoje

Původně jsme zamýšleli využít agilní metodiku SCRUM, která využívá několika-týdenní sprinty k dosažení požadované množiny úkolů. Tato metodika ovšem narazila v případě plánování a

odhadů časové náročnosti jednotlivých úkolů. To bylo zapříčiněno tím, že jsme díky nezkušenosti a nedostatečné znalosti problematiky plánovali nereálné termíny.

Časem jsme tedy přistoupili extrémnímu programování s prvky prototypování. Například při programování parseru se vytvořily celkem čtyři verze. To bylo způsobeno především změnami v LL gramatice, kde by bylo nutné udělat v kódu rozsáhlé změny, které by jistě ovlivnily jeho kvalitu a zavedly by do něj mnoho chyb. Proto bylo jednoduší vytvořit kód nový s již zakomponovanými změnami.

2.4. Nástroje

Pro správu kódu jsme využívali verzovací systém *Git*. Jako hosting pro náš repozitář jsme zvolili *GitHub*, kde jsme ukládali jak samotné zdrojové kódy, tak i plány a automatické testy. Pro plánování schůzek jsme využívali *Kalendář Google***.** K tvorbě diagramu konečného automatu jsme využili online editor *Draw.io*. Skripty pro automatické spouštění testů jsme vytvářeli pomocí jazyka *Bash*.

3. Základní struktura projektu

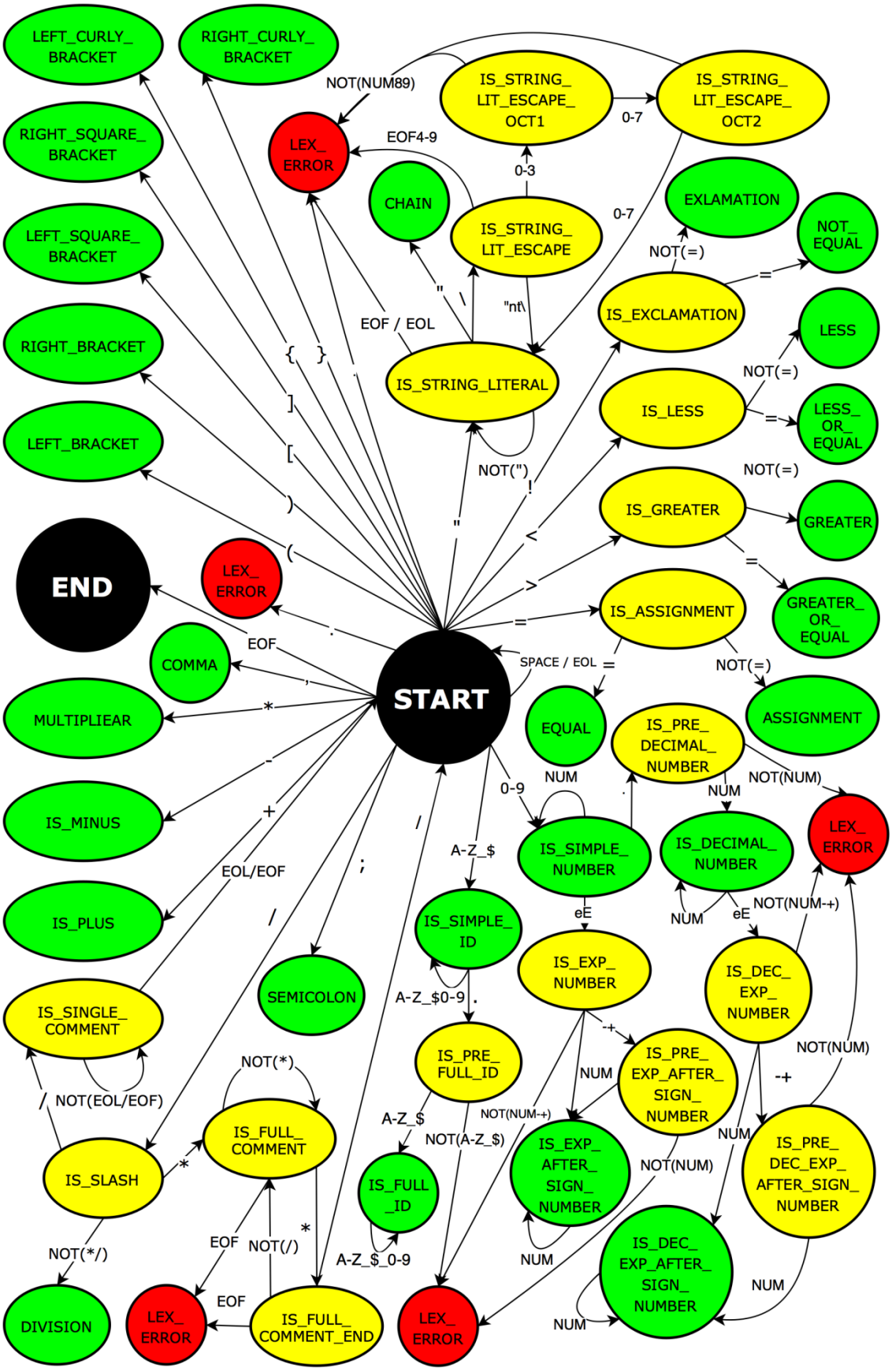
Samotný interpret se skládá z několika důležitých částí. Tyto části jsou: lexikální analyzátor, syntaktický analyzátor, precedenční analyzátor výrazů a interpret. Jednotlivé části jsou navzájem provázané a komunikují spolu přes předem připravené rozhraní.

Hlavní součástí interpretu je syntaktický analyzátor, z kterého se řídí ostatní části programu. Syntaktický analyzátor volá lexikální analyzátor a požaduje po něm tokeny, díky nimž dokáže postupně projít celý vstupní kód a postupně vyhodnocovat, zda je syntakticky a popřípadě i sémanticky správně. Pokud vše proběhne v pořádku, spustí se interpretace kódu.

3.1. Lexikální analýza

Lexikální analyzátor má za úkol postupně číst vstupní kód a rozdělit jej na malé jednotky – tzv. lexémy. Jediným vstupem lexikálního analyzátoru je vstupní soubor. Aby mohl lexikální analyzátor správně fungovat, musí být deterministický a musí přesně popisovat tzv. lexikální pravidla. Lexikální pravidla jsou předem dána programovacím jazykem, který popisují.

K návrhu a grafickému znázornění lexikálního analyzátoru výborně poslouží konečný automat (viz obrázek 1). Konečný automat funguje na jednoduchém principu postupného načítání jednotlivých znaků, díky nimž se rozhoduje, do jakého stavu vstoupí. Za předpokladu, že načtený lexém splnil lexikální pravidla a dostal se do konečného stavu, je činnost lexikálního analyzátoru ukončena a syntaktickému analyzátoru se předá token s potřebnými informacemi o daném lexému. Pokud ovšem načtený znak nesplňuje lexikální pravidla, tak je syntaktickému analyzátoru nahlášena lexikální chyba.



Obrázek 1: Konečný automat

3.2. Syntaktická a sémantická analýza

Syntaktický analyzátor slouží k vyhodnocení syntaktické správnosti programu. Každý programovací jazyk definuje povolené konstrukce, které smí programátor při psaní kódu využívat. Tyto konstrukce jsou zapsány v LL gramatice (viz obrázek 2). LL gramatika je tedy soubor pravidel definující syntaktickou správnost kódu.

<PROG>	->	class ID { <PRVKY_TŘÍDY> } <PROG>
<PROG>	->	Ɛ
<PRVKY_TŘÍDY>	->	static <TYP> ID <ZÁV_STŘED_ROV>
<PRVKY_TŘÍDY>	->	Ɛ
<ZÁV_STŘED_ROV>	->	= EXPRESSION ; <PRVKY_TŘÍDY>
<ZÁV_STŘED_ROV>	->	; <PRVKY_TŘÍDY>
<ZÁV_STŘED_ROV>	->	(<PARAMETR>) { <VE_FUNKCI> }<PRVKY_TŘÍDY>
<TYP>	->	INT
<TYP>	->	DOUBLE
<TYP>	->	STRING
<TYP>	->	VOID
<PARAMETR>	->	<TYP> ID <DALŠÍ_PARAMETR>
<PARAMETR>	->	Ɛ
<DALŠÍ_PARAMETR>	->	, <TYP> ID <DALŠÍ_PARAMETR>
<DALŠÍ_PARAMETR>	->	Ɛ
<PARAMETR_VOLANI>	->	<ID_FULLID> <DALŠÍ_PARAMETR_VOLÁNÍ>
<PARAMETR_VOLANI>	->	Ɛ
<DALŠÍ_PARAMETR_VOLÁNÍ>	->	, <ID_FULLID> <DALŠÍ_PARAMETR_VOLÁNÍ>
<DALŠÍ_PARAMETR_VOLÁNÍ>	->	Ɛ
<STŘ_ROVN>	->	; <VE_FUNKCI>
<STŘ_ROVN>	->	= <ITEM> <VE_FUNKCI>
<ITEM>	->	<ID_FULLID> (<PARAMETR_VOLÁNÍ>) ;
<ITEM>	->	EXPRESSION ;
<ID_FULLID>	->	ID
<ID_FULLID>	->	FULL ID
<ROVN_FUN>	->	= <ITEM>
<ROVN_FUN>	->	(<PARAMETR_VOLÁNÍ>) ;
<VE_FUNKCI>	->	<TYP> ID <STŘ_ROVN>
<VE_FUNKCI>	->	<ID_FULLID> <ROVN_FUN> <VE_FUNKCI>
<VE_FUNKCI>	->	IF (EXPRESSION) { <BLOK> } ELSE { <BLOK> } <VE_FUNKCI>
<VE_FUNKCI>	->	WHILE (EXPRESSION) { <BLOK> } <VE_FUNKCI>
<VE_FUNKCI>	->	RETURN EXPRESSION ;
<VE_FUNKCI>	->	Ɛ
<BLOK>	->	IF (EXPRESSION) { <BLOK> } ELSE { <BLOK> } <BLOK>
<BLOK>	->	WHILE (EXPRESSION) { <BLOK> } <BLOK>
<BLOK>	->	ID <ROVN_FUN> <BLOK>
<BLOK>	->	FULL ID <ROVN_FUN> <BLOK>
<BLOK>	->	Ɛ

Obrázek 2: LL gramatika

Součástí syntaktického analyzátoru je i sémantická kontrola, kde se kontroluje například typová kompatibilita, návratový typ nebo počet parametrů při volání funkce. Sémantická kontrola se provádí až při druhém průchodu kódu, jelikož je nejprve potřeba načíst všechny informace pro vyhodnocení sémantické správnosti. Potřebné informace se ukládají do tabulek symbolů, kde se nachází seznam všech tříd, funkcí a proměnných použitých ve vstupním kódu.

Syntaktický analyzátor si postupně žádá tokeny od lexikálního analyzátoru. Podle typu získaného tokenu následně simuluje vytváření derivačního stromu. Derivační strom se vytváří podle syntaktických pravidel daných gramatikou jazyka. Selhání při tvorbě derivačního stromu indikuje syntaktickou chybu programu.

3.3. Precedenční analýza

K vyhodnocování výrazů se využívá tzv. precedenční analýza. Precedenční analýza se realizuje pomocí precedenční tabulky (viz obrázek 3), ve které jsou vyhodnoceny všechny možné stavy, které ve výrazech mohou nastat. V precedenční tabulce se vždy porovnává nejbližší terminál, který je uložený na zásobníku (řádek), s tokenem, který je aktuálně načtený (sloupec).

Precedenční analýza je volána v případě, že syntaktický analyzátor narazí na výraz. Syntaktický analyzátor při volání precedenčního analyzátoru předává ukazatel na aktuální tabulku symbolů, kterou precedenční analyzátor využívá k sémantické kontrole.

Při zpracovávání výrazu rozlišujeme, zda má být výsledkem výrazu pravdivostní hodnota (v případě podmínek a cyklů) nebo zda je výsledkem číselná či řetězcová hodnota (výrazy přiřazení). Výraz, jehož výsledkem je pravdivostní hodnota, je rozšířením výrazu přiřazovacího. Precedenční analyzátor, stejně jako syntaktický analyzátor, žádá od lexikálního analyzátoru tokeny, pomocí kterých postupně vyhodnocuje výraz.

Výstupem precedenčního analyzátoru je zpracovaný výraz v podobě instrukční pásky, popřípadě odpovídající chybový kód.

	+	-	*	/	<	>	<=	>=	==	!=	int	doub	str	i	(	)	$
+	>	>	<	<	>	>	>	>	>	>	<	<	<	<	<	>	>
-	>	>	<	<	>	>	>	>	>	>	<	<	-	<	<	>	>
*	>	>	>	>	>	>	>	>	>	>	<	<	-	<	<	>	>
/	>	>	>	>	>	>	>	>	>	>	<	<	-	<	<	>	>
<	<	<	<	<	-	-	-	-	-	-	<	<	-	<	<	>	>
>	<	<	<	<	-	-	-	-	-	-	<	<	-	<	<	>	>
<=	<	<	<	<	-	-	-	-	-	-	<	<	-	<	<	>	>
>=	<	<	<	<	-	-	-	-	-	-	<	<	-	<	<	>	>
==	<	<	<	<	-	-	-	-	-	-	<	<	-	<	<	>	>
!=	<	<	<	<	-	-	-	-	-	-	<	<	-	<	<	>	>
int	>	>	>	>	>	>	>	>	>	>	-	-	-	-	-	>	>
doub	>	>	>	>	>	>	>	>	>	>	-	-	-	-	-	>	>
str	>	>	>	>	>	>	>	>	>	>	-	-	-	-	-	>	>
i	>	>	>	>	>	>	>	>	>	>	-	-	-	>	<	>	>
(	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	=	>
)	>	>	>	>	>	>	>	>	>	>	-	<	-	>	>	>	>
$	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	-	p

Obrázek 3: Precedenční tabulka

3.4. Interpret

Po přeložení vstupního kódu a vygenerování instrukcí přichází na řadu samotný interpret. Interpretace kódu se provede pouze v případě, že je program syntakticky i sémanticky správně.

Interpret je implementován pomocí tří-adresných instrukcí. Interpret kontroluje návratové typy funkcí, inicializaci proměnných a provádí samotný běh programu. Nejdříve se inicializují globální proměnné a následně interpret pokračuje ve funkci *Main.run()*. V případě volání funkce se přeskočí na instrukční pásku dané funkce, ve které se provedou instrukce a následně se vrátí zpět za volání funkce, kde se zkontroluje návratový typ.

4. Algoritmy

V rámci projektu jsme měli také za úkol implementovat některé algoritmy v závislosti na volbě zadání. V našem případě jsme měli implementovat Boyer-Mooreův algoritmus, řadící algoritmus Quicksort a algoritmus pro tvorbu tabulky symbolů s rozptýlenými položkami (tzv. Hashovací tabulka).

4.1. Boyer-Mooreův algoritmus

Jedná se o algoritmus, který vyhledává podřetězec v řetězci. Hlavní výhodou algoritmu je rychlejší nalezení podřetězce, než je tomu u klasického intuitivního algoritmu. Algoritmus prochází řetězec a porovnává ho s hledaným vzorkem zprava doleva - tedy od koncových znaků. Při neshodě znaků se v řetězci kurzor posune o určitý počet míst, který je určen pomocí 2 heuristik - vybere se ta, která doporučí větší posun.

V projektu je Boyer-Mooreův algoritmus implementován rekurzivně. Pro výpočet posunu byla použita pouze první heuristika - tzv. bad character rule. Při volání algoritmu se nejprve spočítá posun pro každý možný znak pro případ neúspěchu porovnání. Poté se v cyklu porovnávají jednotlivé znaky textu a vzorku dokud se nedosáhne konce textu (značící neúspěch) nebo dokud se vzorek v textu nenalezne.

4.2. Quicksort

Quicksort je velmi rychlý řadící algoritmus. Ke své funkci využívá mechanismus rozdělení (partition). Ten pole rozdělí na levou a pravou stranu a seřadí jej tak, že se na levé straně nacházejí prvky menší nebo rovny tzv. pseudomediánu (dále pouze PM), oproti tomu na pravé straně se nacházejí prvky větší nebo rovny PM. Algoritmus se aplikuje na vzniklé části pole, dokud se nedosáhne celkového seřazení.

V projektu je Quick sort implementován rekurzivně. Jako PM se vybere hodnota uprostřed daného pole. Hledá se prvek zleva, který je větší nebo roven PM. Poté se hledá prvek zprava, který je menší nebo roven PM. Tyto prvky se prohodí. Algoritmus pokračuje do té doby, dokud se indexy hledání zleva (i) a zprava (j) nepřekříží. Dokud index i nedosáhne konce pole a index j nedosáhne začátku pole, tak se volá algoritmus znova pro rozsah pole 0 - j resp. i - N.

4.3. Tabulky symbolů s rozptýlenými položkami

Tabulky symbolů byly implementovány pomocí tabulky s rozptýlenými položkami (tzv. hash table). Hlavní tabulka symbolů v sobě obsahuje všechny třídy programu. Všechny třídy i funkce mají vlastní tabulku symbolů. V třídní tabulce symbolů jsou uloženy proměnné a funkce, které se nacházejí uvnitř mateřské třídy. Tabulky symbolů, které jsou určeny pro funkce, obsahují jejich lokální proměnné a parametry funkce. Do tabulek symbolů jsme ukládali klíč, ukazatel na data, boolean hodnotu (v případě, že je to funkce) a ukazatel na další položku. Z toho vyplývá, že byly položky ukládány jako jednosměrně vázaný seznam.

5. Závěr

Projekt pro nás byl velmi přínosnou zkušeností. Vyzkoušeli jsme si, co obnáší práce v týmu i způsob, jakým řešit náročnější projekty. Tento projekt nám taktéž poodhalil problematiku tvorby překladačů a pomohl nám k lepšímu pochopení probírané látky. Celkově tedy hodnotíme projekt jako velmi užitečný i přes velkou časovou náročnost.

5.1. Metriky

Počet Git commitů: 133

5.2. Použité zdroje

- Přednášky a materiály předmětu IFJ
- Skripta předmětu IAL
- WIRTH, Nikolaus. *Algoritmy a štruktúry údajov*. Bratislava: Alfa, 1988. Edícia výpočtovej techniky. ISBN 063-030-87.