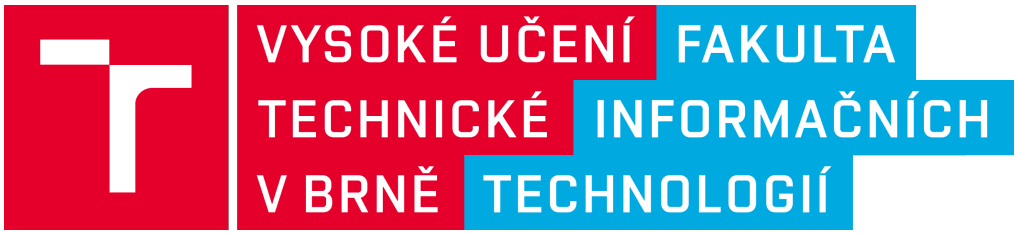
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



Dokumentace projektu do předmětů IFJ a IAL

INTERPRET JAZYKA IFJ16

Tým 22, varianta a/3/II

11. prosince 2016

Tomáš Aubrecht (vedoucí) – 33% xaubre02
Vít Ambrož – 33% xambro15
David Bednařík – 34% xbedna62
Andrej Čulen – 0% xculen02

Obsah

1 Úvod

Dokumentace je zaměřená na tvorbu a popis interpretu jazyka IFJ16. Tento jazyk je založen na objektově orientovaném jazyku **Java SE8** a pro naše účely je velmi zjednodušen.

Vypracování a následné hodnocení tohoto projektu je pro předmět *Formální jazyky a překladače*, na kterém stojí převážná část zadání, a *Algoritmy*, které specifikují typy algoritmů, které máme v implementaci použít.

2 Struktura projektu

Po týmové domluvě jsme projekt rozdělili do tří hlavních částí: **lexikální analyzátor**, **syntaxí řízený překlad** a **interpret**.

Jednotlivé části bylo možné vypracovávat téměř nezávisle, ale bylo třeba průběžně testovat jejich korektní napojení. Rozdělení práce tedy proběhlo tak, že si každý člen vybral jednu část. Na syntaktickém a sémantickém analyzátoru byla potřeba poněkud užší spolupráce, jelikož tuto část zpracovávali společně dva členové. Jádrem tohoto interpretu je právě syntaxí řízený překlad, který kromě kontroly gramatické správnosti programu v podstatě řídí všechny ostatní části.

Od lexikálního analyzátoru postupně přijímá vstupní data v podobě tokenů. Pokud se jedná o výraz, je využita precedenční analýza. Na příslušných místech během syntaktické analýzy jsou prováděny sémantické akce, které mohou generovat instrukce. Po jejich zpracování přichází na řadu interpret, který na základě instrukční pásky provede interpretaci daného programu.

2.1 Lexikální analyzátor

Aby mohl lexikální analyzátor fungovat, potřebuje na vstup dostat soubor se zdrojovým programem. Následně je jeho úkolem načítat data z tohoto souboru a na základě lexikálních pravidel jazyka IFJ16 tyto data rozdělit na tzv. lexémy (logicky oddělené lexikální jednotky).

Tyto lexémy reprezentuje jako tokeny, určuje jejich atributy a na žádost syntaktického analyzátoru mu je předává. Ten zároveň řídí jeho činnost, kdy posílá žádosti o další tokeny.

Mezi tyto atributy patří typ tokenu (např. identifikátor), pokud se jedná o klíčové slovo, tak o jaké, číslo řádku, na kterém se nachází, řetězec obsahující načítaná data a pomocné atributy pro práci s tímto řetězcem – jeho délka a kapacita.

Pokud se jedná o neplatný token, tak informuje syntaktický analyzátor pomocí chybového typu tokenu. Ten následně uvolní všechny dříve alokované zdroje, uzavře vstupní soubor a program ukončí s chybou lexikálního analyzátoru jako návratovou hodnotou.

Princip lexikálního analyzátoru je založen na konečném automatu (viz Příloha A). Uchovává si pouze informace o právě zpracovávaném tokenu a aktuální stav.

Podle načítaných dat se postupně dostane do koncového stavu, který určí o jaký token se jedná. Dalším jeho úkolem je odstranění veškerých bílých znaků ze zdrojového programu včetně komentářů.

2.2 Syntaktický analyzátor

Jedna část syntaxí řízeného překladu je právě syntaktický analyzátor. Jeho úkolem je provést kontrolu, zda interpretovaný zdrojový kód je syntakticky správně. Implementace je založena na LL gramatice (viz Příloha B) a je použita prediktivní metoda syntaktické analýzy, tudíž metoda shora dolů. Pro simulaci vytváření derivačního stromu je zde implemenotván zásobník pro ukládání terminálů a neterminálů.

Operace, které vykonává jsou následující, požádá lexikální analýzu o token a simuluje vytvoření části derivačního stromu. Tento proces se opakuje dokud je vše zpracováno nebo nastane, že není žádné pravidlo pro zpracování tokenu, tudíž syntaktická chyba. Pro výrazy je zvlášť implementován syntaktický analyzátor a to

metodou precedenční analýzy, tudíž zdola nahoru. Tato metoda je založena na precedenční tabulce (viz Příloha C), která určuje jaká operace se má provést. V případě výskytu výrazu, analýza shora dolů začne ukládat příchozí tokeny do fronty, která se po ukončení výrazu odešlou do syntaktického analyzátoru výrazů.

2.3 Sémantický analyzátor

Sémantický analyzátor pracuje na stejné úrovni jako syntaktický analyzátor. Na příslušných terminálech a neterminálech jsou vytvořeny potřebné údaje. Aby bylo možné veškeré sémantické akce dobře realizovat, využíváme především pomocných tokenů, ve kterých můžeme uchovat data z předešlých tokenů, které úspěšně prošly přes syntaktickou analýzu.

Jelikož v jazyce IFJ16 je podporováno volání uživatelských funkcí, které lexikálně nemusí předcházet jejich definicím a podobně i přístupu ke statickým proměnným, rozhodli jsme se využít dvouprůchodovou analýzu.

V prvním průchodu analýzy získáme data o proměnných a funkcích, které vkládáme do tabulky symbolů. Zároveň v prvním průchodu provádíme sémantické kontroly redefinice proměnných, tříd a funkcí.

V druhém průchodu pak provádíme veškeré ostatní sémantické kontroly. Pokud v konkrétním případě není nalezena sémantická chyba, na příslušném místě můžeme generovat tříadresný kód do instrukční tabulky (viz Interpret).

2.4 Interpret

Pro generování kódu jsme si zvolili tři adresný kód a vytvořili si vlastní instrukční sadu. Generované instrukce se ukládají v tabulce, kde prvek obsahuje název návěští, kde se instrukce nachází a seznam uchovávající posloupnost instrukcí. Rozlišujeme návěští na název funkce, větev podmínky if, else a cyklus while. Také můžeme rozlišit instrukce do dvou kategorií. Obyčejné instrukce a instrukce skoku.

Interpretace začne na návěští *Main.run* a postupně vykonává dané instrukce. Aby jsme zjistili v jakém návěští se nacházíme máme zde zásobník, který nám toto zajistí. Při skoku se nám uloží na vrchol zásobníku návěští, na které skáčeme a při vykonání všech instrukcí na daném návěští, splnění podmínky pro cyklus while nebo návrat z funkce se odebere prvek na vrcholu zásobníku a pokračuje se vykonávání instrukcí na návěští, které je nově na vrcholu zásobníku. Interpret také vykonává sémantické kontroly zda se nepracuje s neinicializovanou proměnou nebo zda datové typy proměnných jsou korektní pro danou operaci.

3 Řešení algoritmů do předmětu IAL

Následující algoritmy byly požadovány předmětem *Algoritmy*.

3.1 Knuth-Morris-Prattův algoritmus

Jedná se o algoritmus, který slouží k vyhledávání podřetězců v řetězci. V případě neshody vzoru podřetězce s daným řetězcem se hledaný vzor posune tak, aby se v řetězci nemusel znovu vracet k částem, které již byly zkontrolovány. Svým posunem tak přeskočí část, která se neúplně shodovala. Přesněji se posune o počet shodných znaků. Toto je jeho velkou výhodou, protože minimalizuje časové ztráty.

V našem interpretu tuto metodu využívá vestavěná funkce *ifj16.find*.

3.2 Shell-Sort

Tento řadící algoritmus slouží k seřezení symbolů v řetězci podle jejich ordinální hodnoty. Tato metoda pracuje na opakovaných průchodech, kdy vyměňuje prvky vzdálené o stejný krok. Počáteční krok má hodnotu povoliny počtu symbolů v řetězci.

Daný krok se pro následující etapu půlí až po etapu poslední, kde se prochází celým polem s velikostí kroku jedna, tedy se řadí symboly nacházející se vedle sebe.

Tuto metodu využívá vestavěná funkce *ifj16.sort*.

3.3 Tabulka symbolů

Implementace pomocí tabulky s rozptýlenými položkami, též hashovací tabulky. Postupuje se tak, že pomocí rozptylovací funkce se získá index pole, na který se následně uloží položka s daným klíčem. Nejhorší časová složitost může být lineární *O(n)* v závislosti na rozptylovací funkci.

Pokud má více různých klíčů stejnou rozptylovací hodnotu, začnou se vytvářet lineární seznamy synonym, které jsou explicitně zřetězeny. Nejefektivnější je vyhledávání v takovéto tabulce tehdy, pokud je počet seznamů synonym co největší a jejich délka co nejkratší.

Tento algoritmus používáme pro ukládání dat o proměnných a funkcích do globální tabulky symbolů, lokální proměnné do lokální tabulky symbolů a také dat o proměnných, které je potřeba vytvořit pro správnou funkčnost našeho tříadresného kódu.

4 Vývoj

Vše o tom, kdo na čem pracoval, jak jsme komunikovali a co jsme používali.

4.1 Rozdělení práce

Vít Ambrož – Sémantický analyzátor, tabulka symbolů, generování kódu
Tomáš Aubrecht – Lexikální analyzátor, Shell-Sort, testovací skript, dokumentace
David Bednařík – Syntaktický analyzátor, KMP algoritmus, interpret
Andrej Čulen – bohužel nic

4.2 Komunikace a použité nástroje

Jako členové týmu jsme se znali osobně již z dřívější doby, což byl předpoklad pro dobrou komunikaci. Buď jsme se osobně vídali na přednáškách, cvičeních a plánovaných schůzkách, nebo jsme pro běžnou komunikaci používali sociální síť **Facebook**, kde jsme vytvořili soukromý chat.

Vývoj probíhal na platformě Ubuntu. Pro ukládání naší práce jsme používali verzovací systém **Git**, konkrétně jsme si vytvořili soukromý repozitář **GitHubu** přístupný pouze členům týmu.

O konkrétních změnách v repozitáři jsme se pravidelně informovali, aby nedocházelo ke komplikacím při úpravě jednotlivých souborů mezi členy týmu.

4.3 Testování

Jelikož bylo nutné provádět větší množství testů, vytvořili jsme testovací skript, který tyto testy automaticky spustí a vyhodnotí. Každý člen poté mohl přidávat svoje testy, které kontrolovali určité aspekty programu. V tutu chvíli bylo možné provést kontrolu jedním příkazem místo spouštění jednotlivých testů, což velmi ulehčilo práci.

5 Závěr

Informace o tom, jak máme postupovat a implementovat náš interpret jsme z převážné většiny získali ze samostudia. To bylo nutné, protože pokud bychom čekali na danou látku, než se probere na přednáškách, odevzdat funkční projekt v daném termínu by pro nás bylo velmi náročné až nereálné.

Velmi jsme ocenili možnost pokusného odevzdání, kde jsme se dozvěděli, jak na tom procentuálně jsme. Bohužel asi týden před pokusným odevzdáním od nás odešel jeden člen týmu, který měl za úkol vytvořit interpret.

To jsme možná mohli vytušit už z dřívější doby, kdy se moc neúčastnil komunikace a celkově byl velmi pasivní.

Tímto způsobil nemalé komplikace, protože jsme museli přerozdělit jeho práci mezi zbylé členy týmu. Náš interpret nefungoval a termín pokusného odevzdání se neodkladně blížil. Do tohoto termínu se nám nepodařilo zprovoznit interpretaci, ale naštěstí se uskutečnilo i druhé kolo pokusného odevzdání, kde už náš interpret z větší části fungoval.

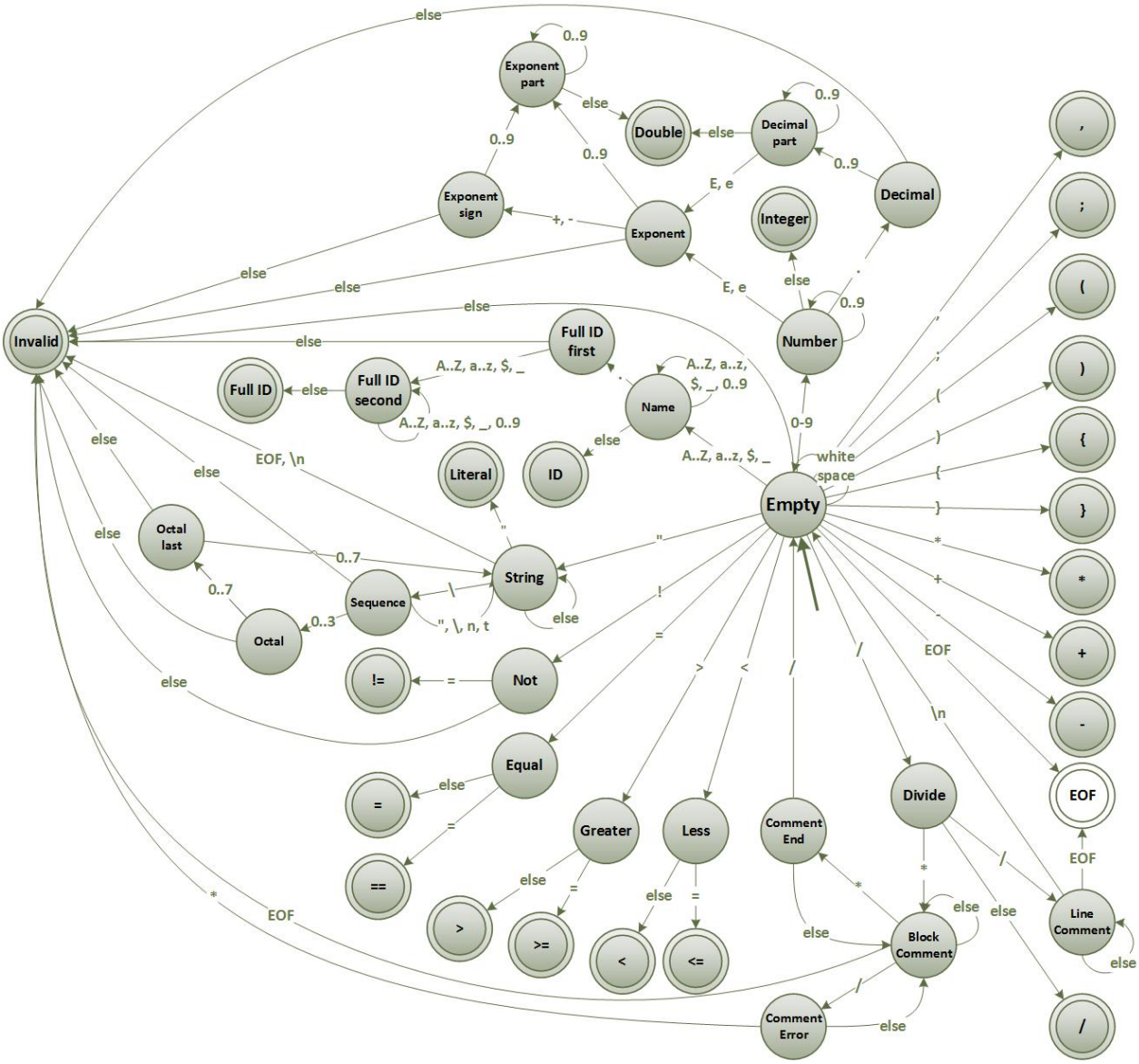
Interpret jazyka IFJ16 je doposud nejsložitějším a nejrozsáhlejším projektem, se kterým jsme se všichni setkali. Začali jsme na něm pracovat téměř ode dne jeho zadání až do termínu odevzdání.

I přes jeho časovou náročnost je práce na tomto projektu určitě velmi cennou zkušeností. Vyzkoušeli jsme si hned několik nových algoritmů, ale především jsme si mohli vyzkoušet týmovou práci. Týmová práce se totiž nemusí obejít bez komplikací, jak tomu bylo právě v našem případě. Důležité je naučit se takové situace pro budoucí praxi řešit a zároveň se naučit vzájemné spolupráci.

6 Zdroje

Logo VUT FIT Brno: `https://www.vutbr.cz/data_storage/multimedia/jvs/loga/02_fakulty/FIT/1-zakladni/CZ/PNG/FIT_barevne_RGB_CZ.png`
Literatura: Přednášky a studijní opory předmětů *IAL* a *IFJ*

A Konečný automat



B LL gramatika

<operator> → *
<operator> → /
<operator> → +
<operator> → -
<operator> → <
<operator> → >
<operator> → <=
<operator> → >=
<operator> → ==
<operator> → !=

<typ> → int
<typ> → double
<typ> → String

<hodnota> → <vyraz> }
<hodnota> → ID_FULL_ID <hodnota2>
<hodnota2> → <vyraz2> }
<hodnota2> → (<arg>)
<hodnota2> → eps

<trida> → class ID
<trida> → eps

<telo_tridy> → static <telo_tridy2>
<telo_tridy2> → <typ> ID <telo_tridy3> }
<telo_tridy2> → void ID (<param>) { <sekvence_prikazu> } <telo_tridy>
<telo_tridy3> → = <hodnota> ; <telo_tridy>
<telo_tridy3> → ; <telo_tridy>
<telo_tridy3> → (<param>) { <sekvence_prikazu> } <telo_tridy>
<telo_tridy> → eps

<param> → <typ> ID <param>
<param> → eps
<param2> → , <typ> ID <param2>
<param2> → eps

<sekvence_prikazu> → <podminka> <sekvence_prikazu>
<sekvence_prikazu> → <while> <sekvence_prikazu>
<sekvence_prikazu> → { <slozeny_prikaz> } <sekvence_prikazu>
<sekvence_prikazu> → <deklarace_promene> <sekvence_prikazu>
<sekvence_prikazu> → <prirazeni_volani_funkce> <sekvence_prikazu>
<sekvence_prikazu> → <narvat> <sekvence_prikazu>
<sekvence_prikazu> → eps

<slozeny_prikaz> → <podminka> <slozeny_prikaz>
<slozeny_prikaz> → <while> <slozeny_prikaz>
<slozeny_prikaz> → <prirazeni_volani_funkce> <slozeny_prikaz>
<slozeny_prikaz> → <navrat> <slozeny_prikaz>
<slozeny_prikaz> → eps

<podminka> → if (<vyraz>) { <slozeny_prikaz> } <else>
<else> → else { <slozeny_prikaz> }

<while> → while (<vyraz>) { <slozeny_prikaz>

<deklarace_promene> → static <typ> ID <inicializace>
<deklarace_promene> → <typ> ID <inicializace>
<inicializace> → ;
<inicializace> → = <hodnota> ;

<prirazeni_volani_funkce> → ID_FULL_ID <prirazeni_volani_funkce2>
<prirazeni_volani_funkce2> → = <hodnota> ;
<prirazeni_volani_funkce2> → (<arg>) ;

<arg> → TERM <arg2>
<arg> → eps
<arg2> → eps
<arg2> → , TERM <arg2>
<arg2> → + TERM <arg2>

<navrat> → return <navrat2>
<navrat2> → ;
<navrat2> → <hodnota> ;

<vyraz> → (<vyraz> <vyraz3>
<vyraz> → TERM <vyraz2>
<vyraz3> →) <vyraz2>
<vyraz2> → <operation> <vyraz>
<vyraz2> → eps

C Precedenční tabulka

	*	/	+	-	<	>	<=	>=	==	!=	(	)	term	$
*	>	>	>	>	>	>	>	>	<	>	<	>	<	>
/	<	<	>	>	>	>	>	>	<	>	<	>	<	>
+	<	<	>	>	>	>	>	>	<	>	<	>	<	>
-	<	<	>	>	>	>	>	>	<	>	<	>	<	>
<	<	<	<	<	>	>	>	>	<	>	<	>	<	>
>	<	<	<	<	>	>	>	>	<	>	<	>	<	>
<=	<	<	<	<	>	>	>	>	<	>	<	>	<	>
>=	<	<	<	<	>	>	>	>	<	>	<	>	<	>
==	<	<	<	<	>	>	>	>	<	>	<	>	<	>
!=	<	<	<	<	>	>	>	>	<	>	<	>	<	>
(	<	<	<	<	<	<	<	<	<	<	<	=	<	
)	>	>	>	>	>	>	>	>	>	>		>		>
term	>	>	>	>	>	>	>	>	>	>		>		>
$	<	<	<	<	<	<	<	<	<	<	<		<	