VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Dokumentace k projektu do předmětů IFJ a IAL

Implementace interpretu imperativního jazyka IFJ16

Tým 051, varianta a/2/I

Tým řešitelů:

Petr Valenta	xvalen20 – 20% – vedoucí týmu
Eliska Kadlecova	xkadle34 – 20%
Daniel Doubek	xdoube03 – 20%
Martin Kovařík	xkovar79 – 20%
Petr Jareš	xjares00 – 20%

12. prosince 2016

Obsah

1 Úvod

Tato dokumentace popisuje vnitřní strukturu a celkovou implementaci interpreteru pro jazyk IFJ16, který je zjednodušenou podmnožinou jazyka JAVA SE 8, což je staticky typovaný objektově orientovaný jazyk. Dokument dále popisuje všechny důležité prvnky včetně dalších náležitostí, jakými jsou užité algoritmy či specifické metody, jež zajišťují celkový chod programu. Pro lepší orientaci je dokumentace členěna do jednotlivých kapitol a podkapitol. Je také doplněna o grafická schémata, která popisují sktrukturu konečného automatu.

2 Struktura projektu

Práci jsme si rozdělili do několika základních částí, kterými byly:

- Lexikální analyzátor
- Syntaktický analyzátor
- Sémantický analyzátor
- Interpret

2.1 Lexikální analyzátor

Lexikální analyzátor představuje vstupní část překladače a je implementován jako konečný automat viz příloha C), který načítá a poté také čte všechny znaky ze zdrojového souboru. Ty pak následně vyhodnocuje a vrací je v podobě tokenů, které jsou vázany v jednosměrném seznamu. Ten byl zvolen kvůli jednoduchosti přistupu k následujícím tokenům. Tokeny pak využívá syntaktická analýza. Každý takový token nese své typové číselné označení, které ho blíže specifikuje. Pokud neexistuje pro danou situaci žádný vhodný stav konečného automatu, je navrácena lexikální chyba – E_LEX.

2.2 Syntaktický analyzátor

Syntaktický analyzátor kontroluje smysl programu, respektive syntax jeho konstrukcí, z hlediska předem definovaných pravidel (gramatiky). Vstupem syntaktického analyzátoru je řetězec tokenů, které získává z lexikálního analyzátoru. Výstupem je informace, zda je zdrojový kód syntakticky správný. V našem projektu jsme na analýzu základních programovacích konstrukcí využili metodu rekurzivního sestupu shora dolů a nerekurzivního zdola nahoru. Tato metoda byla implementována s použitím LL gramatiky, která je popsána níže.

2.3 Sémantický analyzátor

Třídy, metody a globální proměnné jsou ukládány do příslušných tabulek symbolů, které jsou implementovány jako vyhledávací binární strom ve kterém je klíčem jejich název. V tabulkách symbolů jsou uloženy informace například o tom, zda byla proměnná inicializována, jaký je návratový typ metody, kolik má parametrů a jaké jsou jejich typy apod. Jednotlivé položky v tabulce tříd a tabulce metod dále mají svoji vlastní lokální tabulku symbolů. Hodnoty jsou do tabulky vkládány ve dvou běžích. V prvním se načtou třídy, deklarace globálních proměnných a prototypy metod, v druhém se

vkládají proměnné do lokálních tabulek a dochází k sémantickým kontrolám s následným generováním instrukcí pro interpret.

2.4 Precedenční syntaktická analýza

Precedenční analýza slouží k vyhodnocování výrazů. Jejím vstupem je globální tabulka, kde jsou informace o funkcích a aktuální tabulka symbolů, která obsahuje informace o proměnných. Mezi další vstupní prvky patří globální proměnná CurrentVar nebo CurrentMethod, pak také globální tabulka symbolů a odkaz na dvousměrný zřetězený odkaz. CurrentVar se využívá, když chceme daný výraz přiřadit do proměnné a CurrentMethod se používá při přiřazování úplného identifikátoru. Zřetězený dvousměrný odkaz se chová podobně jako zásobník, jen se s ním lépe manipuluje v situaci, kdy do něj chceme vložit nebo vyjmout prvek. Seznam funguje jako struktura, která má 3 ukazatele na první a poslední prvek a poslední terminál, který se aktualizuje po každé redukci.

2.5 Interpret

Interpret je hlavní vykonavatel samotného programu. Jeho funkčnost závisí na tzv. listu instrukcí, který je implementován jednosměrně vázaným lineárním seznamem. Tento seznam byl převzat z jednoduchého interpretu a z prvního domácího úkolu předmětu IAL. Interpret seznam postupně prochází a v závislosti na aktuální instrukci, která je předávána pomocí tříadresného kódu, provádí příslušné kroky a operace. Většina instrukcí vychází z jazyka symbolických adres. Hodnoty jsou ukládány na zásobník, kde jsou prováděny veškeré výkonné operace, jako například operace relační, aritmetické nebo také přesuny proměnných. U numerických operací se provádí i kontrola typů operandů, pokud je to nutné. Interpret prochází seznamem dokud nenarazí na chybu, pak se ukončí s danou návratovou hodnotu, případně dokud nenarazí na instrukci I_STOP, pak interpret skončí úspěšně.

3 Algoritmy

3.1 Metoda Heap sort

Heap sort patří mezi nejrychlejší řadící metody. Algoritmus si vytvoří 2 části, neseřazenou část a seřazenou, a postupně z neseřazené části do seřazené přesouvá největší prvek dokud se nedostane nakonec. Algoritmus využívá strukturu nazývanou halda, která uspořádá prvky. Největší prvek je vyměněn s posledním prvkem a přesunut. Halda se poté znovu seřadí. Výsledkem je seřazené pole prvků podle ordinální hodnoty. Algoritmus má složitost O(n log n).

3.2 Knuth-Morris-Prattův algoritmus

Knuth-Morris-Pratt algoritmus se používá při vyhledávání substringu ve stringu. Substring je analyzován a sestaví se tabulka záchytných bodů na základě prefixů, která nám udává kolik předchozích porovnání můžeme přeskočit v případě, že aktuální selže. Algoritmus poté začne porovnávat. Při neshodě znaků se posune na další záchytný bod. Knuth-Morris-Pratt algoritmus využívá informace z minulých porovnání a neporovnává už znaky které porovnal dříve. Vrací pozici substringu ve stringu, nebo -1 v případě že nebyla shoda, a celkový proces má složitost O(n+k).

4 Práce v týmu

4.1 Obecné informace o této kapitole

V této kapitole Vám popíšeme, jakým způsobem probíhal vznik projektu.

4.2 Time management

Hned na první schůzce jsme si vytvořili přibližný časový plán, včetně prvního rozdělení práce a určení si termínu, kdy je potřeba mít nejpozději tuto část hotovou. Poté jsme však postupně zjišťovali, že náš prvotní časový plán není jednoduché dodržovat. Během řešení totiž přišly první půlsemestrální zkoušky, a proto bylo potřeba věnovat čas také jim. Toto nás zdrželo asi nejvíce, nicméně stále se to dalo zvládnout. Nakonec jsme stihli odevzdat projekt v předtermínu k pokusnému hodnocení. Od této chvíle jsme se již soustředili na poslední testování a tvorbu dokumentace včetně prezentace.

4.3 Komunikace v týmu a uchovávání zdrojových kódů

Jak už jsme již výše zmiňovali, časové možnosti jednotlivých členů se značně lišily a četnost našich schůzek se měnila kvadraticky. Nejdříve jsme se všichni sešli, abychom si rozdělili práci. Na této schůzce jsme si také určili naše komunikační medium. Nakonec jsme se všichni shodli na internetovém portálu Slack.com, který umožnuje vkládání jednotlivých bloků kódu, plánování schůzek či jen prostou komunikaci mezi členy, která může probíhat buď globálně mezi všemi členy týmu nebo pouze mezi konkrétními členy týmu. Zdrojové kódy jsme ukládali na GitHub, který jsme měli propojený přímo s naším vývojovým prostředím, což znamenalo to, že jsme mohli pohodlně nahrávat změny na web přímo ze zdroje. Od této chvíle se naše časové možnosti postupně zmenšovaly, a proto bylo za potřebí pracovat samostatně na jednotlivých částech projektu. Toto fungovalo celkem bez problému. Všichni členové našeho týmu byli svědomití, a když se někdo začal opožďovat příliš, tak většinou stačilo malé pokárání ze strany vedoucího týmu.

5 Testování

Testování našeho projektu probíhalo na základě námi vytvořeného automatizovaného skriptu, který porovnával výstupní data s hodnotami výstupu. Když bylo testování úspěšné, zkusili jsme požádat i ostatní skupiny, abychom si i na jejich testech ověřili, zda program vrací správné hodnoty. Pakliže se objevily jakékoliv nesrovnalosti, byla předána chyba tomu členovi z týmu, který měl danou část interpretu na starost.

6 Závěr

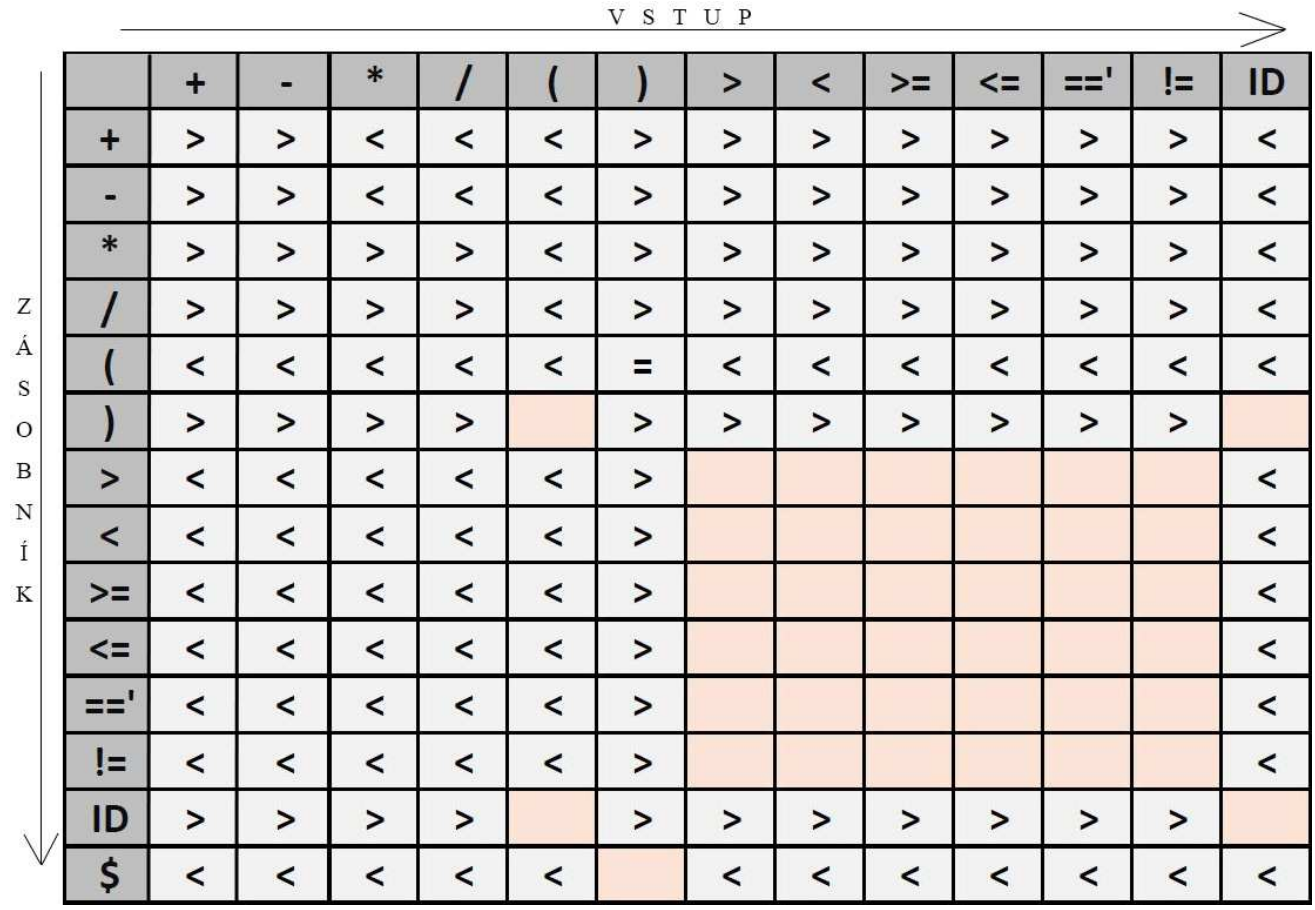
Projekt IFJ16 byl pro nás velmi cennou zkušeností. Nikdo z členů týmu doposud neměl zkušenost s projektem podobného rozměru, a proto pro nás práce představovala velikou výzvu. Rozvrhnout si práci tak, aby se dala stihnout v termínu či jen sesynchronizovat čas tak, aby vyhovoval všem v týmu, bylo celkem obtížné. Myslíme si však, že jsme se úkolu zhostili zodpovědně.

Reference

[1] *Zjednodušená implementace interpretu jednoduchého jazyka*. Dostupné na: <https://www.fit.vutbr.cz/study/courses/IFJ/private/projekt/jednoduchy_interpret.zip>.

[2] HONZÍK, J. *Algoritmy – studijní opora* [online].

A Tabulka precedenčního analyzátoru



	+	-	*	/	(	)	>	<	>=	<=	=='	!=	ID
+	>	>	<	<	<	>	>	>	>	>	>	>	<
-	>	>	<	<	<	>	>	>	>	>	>	>	<
*	>	>	>	>	<	>	>	>	>	>	>	>	<
/	>	>	>	>	<	>	>	>	>	>	>	>	<
(	<	<	<	<	<	=	<	<	<	<	<	<	<
)	>	>	>	>		>	>	>	>	>	>	>	
>	<	<	<	<	<	>							<
<	<	<	<	<	<	>							<
>=	<	<	<	<	<	>							<
<=	<	<	<	<	<	>							<
=='	<	<	<	<	<	>							<
!=	<	<	<	<	<	>							<
ID	>	>	>	>		>	>	>	>	>	>	>	
$	<	<	<	<	<		<	<	<	<	<	<	<

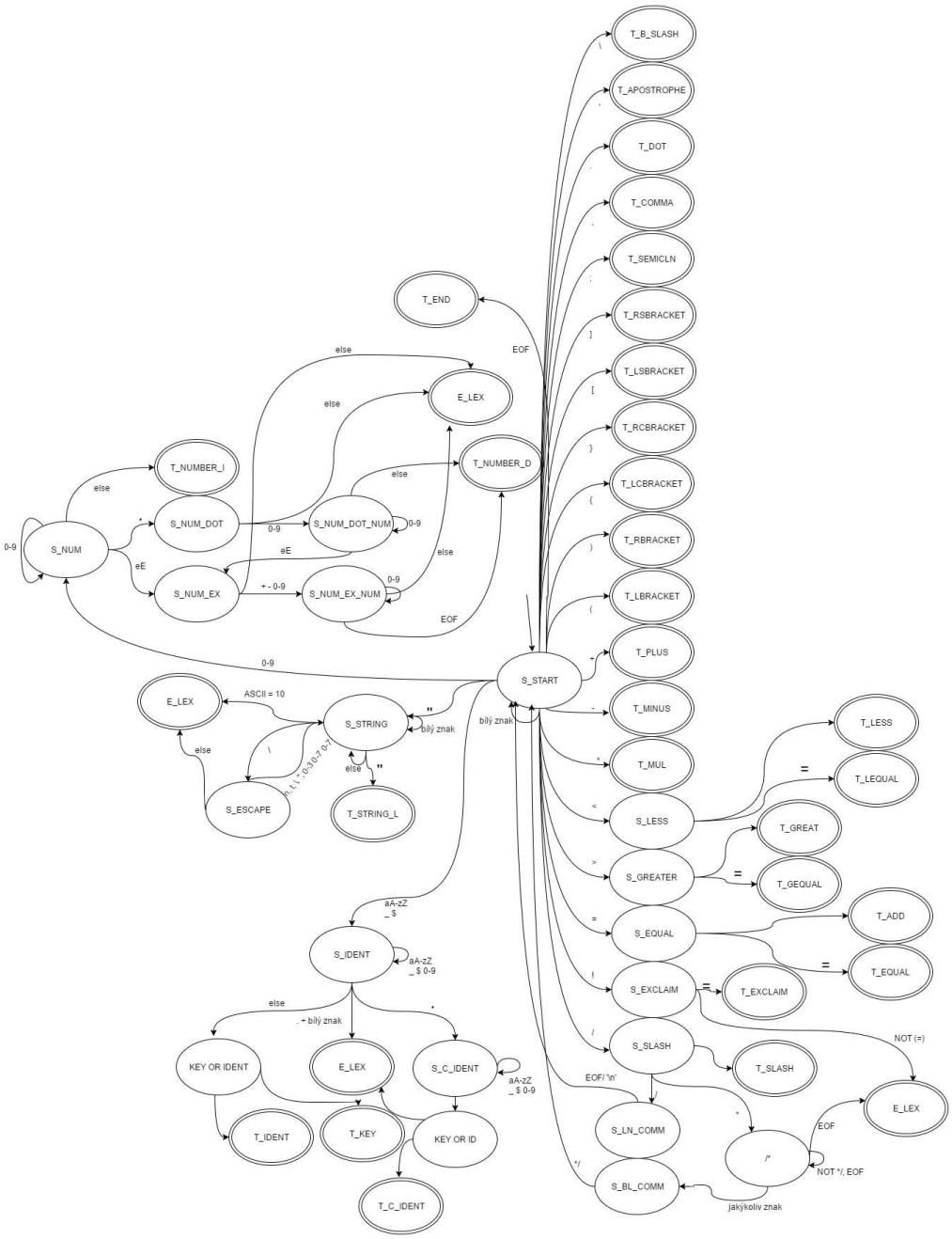
Obrázek A: Precedenční analýza

B LL Gramatika

```
Grammar
METHOD →    static type id ( PARAM ) lb STATEMENT_LIST rb.
PARAM → .
PARAM → type id PARAM.
PARAM → cm type id PARAM.
STATEMENT_LIST →    .
STATEMENT_LIST →    STATEMENT STATEMENT_LIST.
STATEMENT → id ID.
STATEMENT → return EXPRESSION sc.
STATEMENT → break sc.
STATEMENT → continue sc.
STATEMENT → if ( EXPRESSION ) lb STATEMENT_LIST rb ELSEIF_STATEMENT ELSE_STATEMENT.
STATEMENT → while ( EXPRESSION ) lb STATEMENT_LIST rb.
STATEMENT → type TYP.
STATEMENT → static type id sc.
TYP →   id ID1.
ID →    eq EQ.
ID →    ( PARAM ) sc.
ID1 →   sc.
ID1 →   eq EXPRESSION sc.
EQ →    EXPRESSION sc.
EQ →    id (PARAM ) sc.
ELSEIF_STATEMENT →  .
ELSEIF_STATEMENT →  elseif ( EXPRESSION ) lb STATEMENT_LIST rb ELSEIF_STATEMENT.
ELSE_STATEMENT →    .
ELSE_STATEMENT →    else lb STATEMENT_LIST rb.
```

Obrázek B: LL gramatika

C Struktura konečného automatu lexikálního analyzátoru



Obrázek C: Konečný automat