Vysoké učení technické v Brně

Fakulta informačních technologií

Dokumentace projektu z předmětů IFJ a IAL

Implementace interpretu jazyka IFJ14

Tým 017, varianta a/1/I

Autoři: **Michal Janoušek** [xjanou15] - 20 %
Zdeněk Studený [xstude21] - 20 %
Martin Nosek [xnosek10] - 20 %
Jan Pawlus [xpawlu00] - 20 %
Antonín Čala [xcalaa00] - 20 %

Rozšíření: FUNEXP
REPEAT
BOOLOP
MINUS
ELSEIF

Obsah

1 Úvod

Dokumentace se zaobírá vývojem interpretu imperativního jazyka IFJ14. Popisuje práci v týmu, klíčové prvky našeho interpretu, specifická řešení (správce paměti) a algoritmy určené našim zadáním - `quicksort` (řazení) a `Knuth-Morris-Prat` (hledání podřetězce v řetězci).

Příloha obsahuje LL gramatiku, precedenční tabulku a diagram konečného automatu popisující lexikální analyzátor.

2 Práce v týmu

2.1 Rozdělení práce

Zodpovědnost vedoucího padla automaticky na Michala Janouška, jenž je z nás nejlepší programátor. Ten si vzal také na starost pravděpodobně nejtěžší část projektu - syntaktickou analýzu. Martin Nosek se nabídl k pomoci při syntaktické analýze a měl na starost syntaktickou a sémantickou analýzu výrazů, stejně tak generování kódu výrazů. Zdeněk Studený se poté dobrovolně ujal lexikální analýzy, Jan Pawlus interpretace a vyhledávajícího algoritmu a nakonec Antonín Čala implementoval správce paměti a řazení.

2.2 Metodika

Ze začátku se každý snažil implementovat nezávisle svoji část, teprve později se projekt spojil a s tímto se začaly objevovat chyby ve větším měřítku. Celkově jsme ale neměli žádný problém většího rozměru, který by nás zastavil na delší časový úsek. Celou dobu se téměř každá část testovala malými testy. Pokud by se tedy měla vybrat metodika programování, která byla našemu přístupu nejblíž, bylo by to extrémní programování. Každopádně se ale nedá mluvit o nějaké ucelené metodice, neboť nikdo z nás neměl podobnou zkušenost s prací v týmu.

2.3 Git

Pro sdílení společného kódu jsme se rozhodli použít verzovací systém `Git` a volba hostingu pro náš kód padla na `BitBucket.org`. `Git` pro nás byl při vývoji klíčový, jelikož umožňuje snadné a rychlé sdílení veškerého kódu mezi všemi členy týmu. Pro zajímavost - celkový počet commitů byl před odevzdáním 421.

2.4 Schůzky a časové rozvržení

Protože jsme se všichni předem znali, panovala v týmu velmi přátelská a benevolentní nálada. Dost možná to bylo i trochu na škodu - interpret jsme chtěli mít hotový již před prvním pokusným odevzdáním, což se nám právě kvůli až přílišné vzájemné důvěře nepovedlo. Na druhou stranu jsou všichni členové týmu zodpovědní a schopní, takže dokončení projektu před finálním odevzdáním nebyl žádný problém. Schůzky byly nepravidelné. Ze začátku jsme řešili pouze teoretické otázky a celkový návrh, v pozdějších fázích jsme spolu trávili více času programováním.

3 Implementace

3.1 Lexikální analýza

Lexikální analýza je zajištěna pomocí implementace konečného automatu, jenž je přiložen jako příloha v kapitole 7.1. V implementaci je však nutné ošetřit také věci, které automat nezahrnuje. Je nutné před posláním kontrolovat hodnotu ID, jestli se nerovná předem definovaným klíčovým slovům (např. begin, true, atd.). Jazyk IFJ2014 je case insensitive, proto se hodnoty ID převáděli na malá písmena už v lexikální analýze.

3.2 Syntaktická a sémantická analýza

Syntaktický analyzátor jsme kvůli zjednodušení práce rozdělili na dvě části - pro implementaci hlavní části syntaktické analýzy jsme využli metodu rekurzivního sestupu. Ta se řídí podle LL gramatiky (7.2). Tabulka symbolů je implementována pomocí binárního vyhledávacího stromu. Celkem se používají v každé chvíli až tři stromy, jeden pro funkce, druhý pro globální proměnné a třetí pro lokální proměnné. U proměnných se ukládá mimo jejich jména také typ (slouží pro sémantickou analýzu) a index který slouží k adresování při interpretaci kódu. U funkcí se pomocí jednosměrného seznamu ukládají parametry a jejich typy, jako první je vždy uložen návratový typ funkce a za ním následují jednotlivé parametry. Kód se generuje přímo při syntaktické a sémantické analýze, které probíhají současně. Jako vnitřní kód jsme použili tříadresný kód.

Další části je syntaktický analyzátor výrazu implementováný jako syntaktický analyzátor pracující zdola nahoru. Postupně zpracovává tokeny, které přijímá od lexikalního analyzátoru, do té doby, než narazí na token, jehož hodnota neodpovídá výrazu. Ke své práci využívá jednoho zásobníku implementovaného jako jednosměrně vázaný seznam. Jádrem syntaktického analyzátoru pracujícího zdola nahoru je precedenční tabulka (7.3) porovnávající terminály na vrcholu zásobníku s příchozími tokeny. Při tvorbě precedenční tabulky bylo třeba brát ohled na prioritu a asociativitu operátorů, které jsou v našem případě totožné s jazykem Free Pascal. Dále bylo třeba brát v potaz identifikátory a jejich možné pozice, závorky a ukončovač zpracovávaného výrazu, kterým je jakýkoliv token, jehož hodnota neodpovída výrazu. Tabulka poté byla upravena pro individuální potřeby našeho řešení, jako například zpracovávání funkcí. Na základě tabulky poté probíhá vyhodnocení výrazu a případné aplikování pravidel za současného generování kódu. Paralelně se syntaktickou analýzou výrazu probíhá analýza sémantická k jejíž implementaci bylo zapotřebí vytvořit navíc jeden zásobník, do kterého jsou ukládány operandy.

Za zmínku stojí přidání rozšíření funkcionality překladače v podobě unárního mínus, které bylo řešeno doplněním podmíněného přikazu v hlavním cyklu zpracovávajícím jednotlivé tokeny. A následným přidáním pravidla do tabulky pravidel a generováním kódu.

3.3 Interpret

Interpret provádí požadované operace na základě aktuální instrukce (tříadresného kódu). Při spuštění interpretu se vytvoří tabulka globálních proměnných (informace o počtu a typu globálních proměnných jsou uloženy v tabulce funkcí, resp. v první položce této tabulky). Za popis stojí volání funkcí - při volání se aktuální instrukce uloží na vrchol zásobníku instrukcí a vytvoří se tabulka lokálních proměnných, kam se uloží mimo nově deklarované lokální proměnné i parametry funkce. Pokud se vracíme z funkce, pokračuje se instrukcí, která je na vrcholu zásobníku, a zruší se aktuální tabulka lokálních proměnných. Tabulky lokálních proměnných na sebe musí navazovat (při návratu z funkce se ukládá návratová hodnota do předchozí lokální tabulky), implementace tedy připomíná lineární jednosměrný seznam.

3.4 Správce paměti

Místo jednotlivého časově náročného alokování nového prostoru jsme se rozhodli v našem programu vytvořit vlastního správce paměti. Správce má na starost alokaci větších paměťových bloků najednou a jejich dílčí přerozdělení. Pomocí volání funkcí na přidělení části pole jsme se vyhnuli příliš častému užívání funkce malloc, která je tak omezena na pouhou inicializaci, nebo pokud je paměť zaplněna, a je třeba přidat další paměťový blok. K jeho funkci slouží několik globálních proměnných, které po inicializaci obsahují pole struktur příslušného datového typu. Správce paměti taktéž funguje dobře jako prevence nechtěných memory leaků, které by mohly nastat při nesprávné funkčnosti či při chybách překladu vstupního kódu.

4 Vestavěné funkce

4.1 Copy, Length

Tyto vestavěné funkce nestojí za popis, neboť jsou implementovány funkcemi `strncpy`, resp. `strlen` z knihovny `string` jazyka C.

4.2 Find - Morris-Knuth-Pratt

Jednoduché hledání podřetězce v řetězci je založeno na prostém posouváním podřetězce o jedno místo doprava, pokud nastane neshoda. Tato metoda je ale časově náročná - Morris-Knuth-Prattův algoritmus přináší určitou míru optimalizace. Nejprve se pomocí prefixové funkce vytvoří pole (označované jako π) o délce podřetězce, které bude obsahovat informaci o tom, jak moc se můžeme při neshodě posouvat. Index tohoto pole určuje délku úseku podřetězce, pro který se hledá nejdelší prefix, jenž je zároveň úplným suffixem tohoto úseku. Hodnota π na tomto indexu je právě délka prefixu. Znamená to tedy, že pokud se náš podřetězec shoduje například se třemi prvními znaky řetězce, ale poté nastane neshoda, podíváme se do pole π na index tři a o tolik podřetězec posuneme doprava.

4.3 Sort - Quicksort

Funkci na řazení sort v našem programu obstarává dle zadání algoritmus quicksort, který patří mezi časově méně náročné řadící algoritmy. Jeho ideální funkčnost se odvíjí od volby pivotu, který slouží jako hranice v poli a rozděluje jej na dvě dílčí části, kde jedna strana pole obsahuje hodnotu menší než je hodnota pivotu a druhá naopak větší. Volba pivotu stejně jako quicksort samotný lze vyřešit více způsoby, jako například matematickým průměrem hraničních hodnot pole, či jinou metodou. My jsme se rozhodli vyřešit ji napevno volbou jednoho prvku, neboť v případě datového typu string není zaručena žádná zjevná matematická funkce, která by jej zvolila lépe. Quicksort samotný je dále implementován s pomocí rekurze, která může být při dlouhých řetězcích možná náročnější než nerekurzivní implementace, ale zato nám program připadá srozumitelnější a přehlednější.

5 Závěr

Zní to určitě jako klišé, ale tento projekt byl prvním týmovým počinem všech členů a tím pádem se neobešel bez chyb v komunikaci a pomalejším postupem, než bylo původně v plánu. Vzhledem k úspěšnému dokončení to je ale zajímavá a hlavně velmi cenná zkušenost, jelikož interpret už není úplně triviální věc. Práci hodně ulehčil verzovací systém `Git` a klíčový byl také přístup vedoucího, který všechny členy týmu vždy správně nasměroval. Nakonec lze i říct, že nás všechny vývoj bavil, protože ke každé části je nutný velmi individuální přístup a nikde není přesně zadáno, jak máme při implementaci postupovat.

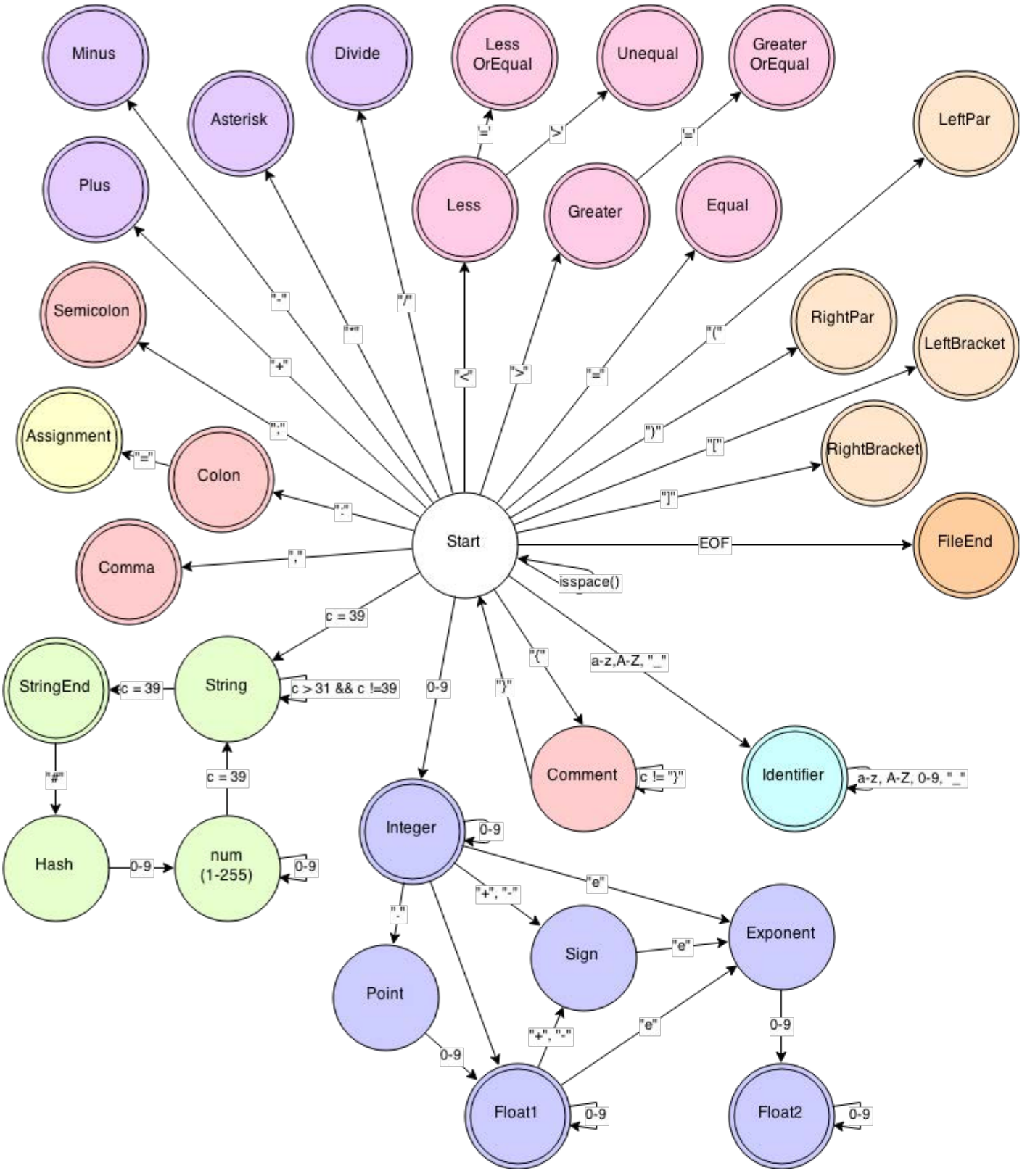
V průběhu vývoje se testovaly jednotlivé celky projektu našimi dílčími testy, v závěrečné fázi se interpret jako celek testoval rozsáhlými skripty.

6 Použité zdroje

[1] HONZÍK, Jan M. Algoritmy: Studijní opora. 2014. verze 14-Q.

7 Přílohy

7.1 Konečný automat lexikálního analyzátoru



7.2 LL gramatika

<PROGRAM>	<VARIABLES> <FUNCTIONS> <SEQUENCE> . eof
<VARIABLES>	var <VARIABLE>
<VARIABLES>	eps
<VARIABLE>	id : <TYPE> ; <NEXT_VARIABLE>
<NEXT_VARIABLE>	<VARIABLE>
<NEXT_VARIABLE>	eps
<TYPE>	integer
<TYPE>	real
<TYPE>	string
<TYPE>	boolean
<FUNCTIONS>	id (<PARAMS>) : <TYPE> ; <FUNCTION_BODY> ; <FUNCTIONS>
<FUNCTIONS>	eps
<PARAMS>	id : <TYPE> <NEXT_PARAM>
<PARAMS>	eps
<NEXT_PARAM>	; id : <TYPE> <NEXT_PARAM>
<NEXT_PARAM>	eps
<FUNCTION_BODY>	forward
<FUNCTION_BODY>	<VARIABLES> <SEQUENCE>
<SEQUENCE>	begin <INSTRUCTIONS> end
<INSTRUCTIONS>	<INSTRUCTION> <NEXT_INSTRUCTION>
<INSTRUCTIONS>	eps
<INSTRUCTION>	id := <EXPRESSION>
<INSTRUCTION>	<SEQUENCE>
<INSTRUCTION>	if <EXPRESSION> then <SEQUENCE> <ELSEIF>
<INSTRUCTION>	while <EXPRESSION> do <SEQUENCE>
<INSTRUCTION>	repeat <INSTRUCTION> until <EXPRESSION>
<INSTRUCTION>	readln (id)
<INSTRUCTION>	readln (id)
<NEXT_INSTRUCTION>	; <INSTRUCTION> <NEXT_INSTRUCTION>
<NEXT_INSTRUCTION>	eps
<ELSEIF>	else <SEQUNCE>
<ELSEIF>	eps
<WRITE_PARAM>	<EXPRESSION>
<NEXT_WRITE_PARAMS>	, <WRITE_PARAM> <NEXT_WRITE_PARAMS>
<NEXT_WRITE_PARAMS>	eps

7.3 Precendenční tabulka

	NOT	*	/	AND	+	-	OR	XOR	=	<	>	<>	<=	>=	(	)	ID	$
NOT	<	>	>	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
*****	<	>	>	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
/	<	>	>	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
AND	<	>	>	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
+	<	<	<	<	>	>	>	>	>	>	>	>	>	>	<	>	<	>
-	<	<	<	<	>	>	>	>	>	>	>	>	>	>	<	>	<	>
OR	<	<	<	<	>	>	>	>	>	>	>	>	>	>	<	>	<	>
XOR	<	<	<	<	>	>	>	>	>	>	>	>	>	>	<	>	<	>
=	<	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
<	<	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
>	<	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
<>	<	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
<=	<	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
>=	<	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
(	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	=	<	BLANK
)	BLANK	>	>	>	>	>	>	>	>	>	>	>	>	>	BLANK	>	BLANK	>
ID	BLANK	>	>	>	>	>	>	>	>	>	>	>	>	>	BLANK	>	BLANK	>
$	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	EXIT	<	BLANK