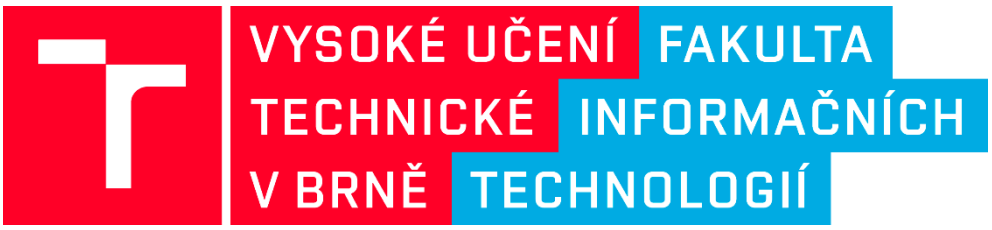


Dokumentace projektu

Interpret jazyka IFJ16

Tým 073, varianta b/1/I

Rozšíření FUNEXP

Petr Buchal	**xbucha02**	**20 %**	**(vedoucí)**
Petra Buchtová	xbucht25	20%	
Pavel Černý	xcerny66	20%	
Radim Červinka	xcervi21	20%	
Marek Doležel	xdolez67	20%	

11.12.2016, Brno

Obsah

Způsob řešení

Návrh

Náš interpret jazyka IFJ16 (který je podmnožinou jazyka Java) jsme rozdělili do 3 základních modulů:
- **lexikální analyzátor** – provádí lexikální analýzu kódu a získává tokeny ze vstupního souboru
- **parser** – provádí syntaktickou a sémantickou analýzu a obsahuje 2 submoduly, syntaktický analyzátor a analyzátor výrazů
- **interpret** – měl provádět instrukce na instrukční pásce generované parserem

Implementace

Lexikální analyzátor

Lexikální analyzátor funguje na jednoduchém principu konečného automatu. Jeho schéma naleznete v sekci Grafy a tabulky. Celý modul obsahuje jednu funkci *get_token,* které je předán token, do kterého nahraje aktuální token ze vstupního souboru a v návratové hodnotě vrací, zda došlo nebo nedošlo k lexikální chybě. Tuto funkci volá parser, když potřebuje nový token ze souboru.

Syntaktický analyzátor

Syntaktický analyzátor je implementován metodou shora-dolů. Naše implementace využívá tabulku pravidel, podle které se automat rozhodne pro použití pravidla a zásobník na kterém se uchovávají terminály a neterminály. Samotná hlavní rutina syntaktického analyzátoru je smyčka, která kontroluje přítomnost terminálů a neterminálů na vrcholu zásobníku. V případě že je na vrcholu zásobníku terminál, potom dojde k jeho porovnání s aktuálním tokenem na vstupu. Pokud je detekována odlišnost, jedná o syntatickou chybu. V opačném případě smyčka pokračuje. Naopak je-li na vrcholu zásobníku neterminál, provede se jeho rozvětvení na terminály a neterminály, a to podle tabulky pravidel.

Precedenční analýza výrazů

Výrazy zpracováváme pomocí precedenční syntaktické analýzy. Do gramatiky jsme zahrnuli také vyhodnocování volání funkcí. Hlavní funkce parse_exp pracuje ve smyčce a načítá si nové tokeny ze zdrojového souboru. Na základě terminálu nejvýše na zásobníku a aktuálního vstupu se rozhoduje, zda se má otevřít handle (<) a vložit aktuální vstup na zásobník, pouze vložit vstup (=), nebo redukovat (>) podle pravidel na neterminál (P pro argumenty funkce, E pro vše ostatní). Analýza pracuje, dokud na vstupu a zároveň jako nejvrchnější terminál na zásobníku není ($ - "konec") a nad ním pouze neterminál E. Pro jednoznačnou detekci korektního konce vstupu jsme do precedenční tabulky přidali znak 'e'. Při redukci se volá pomocná funkce do_exp_semantic, která provádí semantické akce (např. tvoří nové dočasné proměnné) a generuje instrukce.

Interpret

Interpret jsme chtěli založit na jednoduchém principu čtení instrukční pásky a provádění instrukcí, pomocí příkazu switch měl interpret jednoduše určit, co má provést za akci, dle instrukce. Největší problém, na kterém jsme se zasekli, bylo volání funkcí a korektní práce s parametry a lokálními proměnnými. Interpret měl pracovat se zásobníkem binárních stromů pro lokální proměnné a parametry pro daný rámec (pro danou funkci) a se zásobníkem pro parametry, návratovou hodnotu a adresu. Nestihli jsme vytvořit korektní instrukce a rozhraní mezi parserem a interpretem, jakožto i generování daných instrukcí.

Vývoj

Vývoj projektu probíhal modulárně, projekt jsme dělili na menší části, které jsme řešili nezávisle, následně jsme moduly a submoduly otestovali a poté jsme je mohli propojit s jiným, již otestovaným modulem, jak bylo potřeba.

Způsob práce v týmu

Vzhledem k modulárnímu přístupu k projektu jsme jednotlivé části projektu řešili nezávisle, v průběhu jsme se rozdělili na 2 skupiny řešící odlišné problémy. Členové těchto skupin se různě měnili dle potřeby a skupiny spolu komunikovali výhradně při potížích ve specifických oblastech a o společném rozhraní. Tým se scházel osobně jednou týdně, poslední měsíc projektu byly schůzky častější, dvakrát až třikrát do týdne. Ke komunikaci nám výrazně pomohl program *Slack,* který umožnuje velmi dobrou organizaci konverzací do tzv. chatovacích místností, které se lišily tématem, který se v nich probíral. K týmovým hovorům jsme rádi využívali službu *Skype* a pomohl nám místy i *TeamViewer*. Pro vývoj programu jsme využili verzovací systém *Git* a to konkrétně službu github.com.

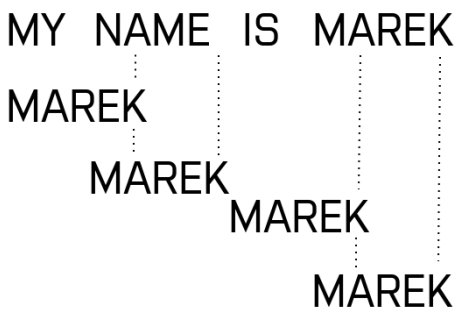
Rozšíření

Při tvorbě pravidel precedenční analýzy jsme se na volání funkce dívali jako na výraz. K podpoře rozšíření FUNEXP zbývalo již jen podporovat výrazy ve volání, což znamenalo jednoduchou úpravu LL pravidel a tedy jen pomocných struktur, nikoli zásah do již hotového kódu.

Algoritmy

Boyer-Mooreův algoritmus

Na hledání podřetězce jsme využili Boyer-Mooreův algoritmus. Ten přeskakuje prvky pole na základě znalosti, že na místě daných prvků se vzorek a text nemohou rovnat. Algoritmus má dvě heuristiky, my jsme využili heuristiku první s polem CharJump. Na následujícím obrázku lze vidět o kolik znaků se vzorek posouvá vůči textu, ve kterém se hledá shoda.



Pokud se první porovnávané znaky neshodují a znak z textu se nenachází ve vzorku, vzorek se posouvá o jeho celou délku. Pokud se daný znak ve vzorku nachází, posune se vzorek tak, aby se poloha znaku ve vzorku rovnala poloze znaku v textu. Pokud se první porovnávané znaky rovnají, porovnávají se znaky další, při shodě všech znaků text obsahuje vzorek. Při neshodě se vzorek opět posouvá o hodnotu pro daný symbol v poli CharJump. Toto pole se využívá jako zdroj informací o skocích (konkrétně o kolik symbolů se má vzorek posunout).

Quick sort

Vyhledávání jsme implementovali pomocí quick sortu, to je algoritmus využívající mechanismus rozdělení (funkce partition). Tento mechanismus rozděluje prvky pole na dvě části, vlevo jsou prvky menší nebo rovny pseudomediánu (V našem případě je pseudomedián číslem ze středu intervalu. Při kontrole konce pole slouží pseudomedián rovněž zároveň jako zarážka.), vpravo jsou prvky větší. Na nově vzniklé dvě části se opět aplikuje mechanismus partition. Přestává se aplikovat teprve až u částí, které mají pouze dva prvky.

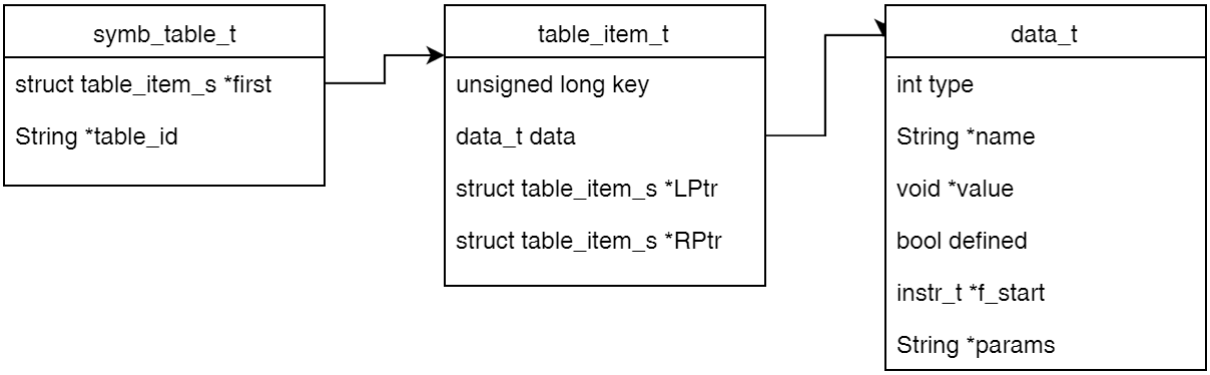
Binární vyhledávací strom

Pro implementaci tabulky symbolů jsme použili binární vyhledávací strom.

Binární vyhledávací strom se skládá z uzlů (tento uzel obsahuje klíč, pravý podstrom a levý podstrom). Uzly jsou uspořádány podle klíče, všechny uzly levého podstromu mají menší klíč než otcovský uzel a všechny uzly pravého podstromu mají větší klíč než otcovský uzel.

Jako klíč jsme použili zahashovaný název identifikátoru. Jako hashovací funkci jsme využili SDBM. Každý uzel binárního stromu obsahuje ukazatel na levý a pravý podstrom, klíč a strukturu data_t, ve které jsou uloženy informace o proměnných a funkcích. Vkládání uzlů a jejich hledání je implementováno nerekurzivně, mazání je implementováno rekurzivně. Tabulky symbolů pro

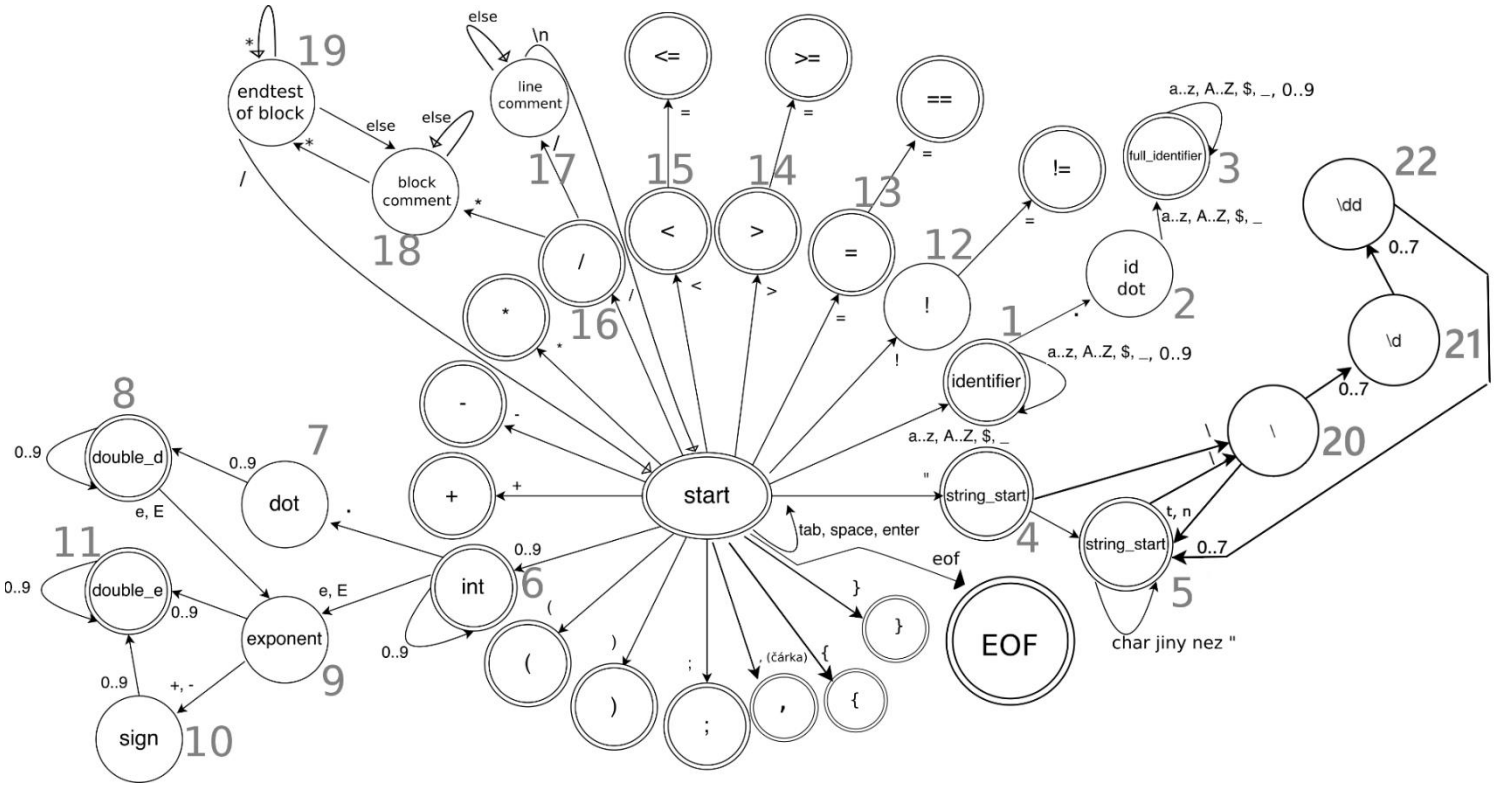
jednotlivé funkce jsou ukládány na zásobník tabulek symbolů, tabulka symbolů pro static proměnné a funkce stojí mimo tento zásobník.



Grafy a tabulky

Lexikální analýza

Konečný automat



Syntaktická a sémantická analýza

Pravidla pro precedenční tabulku

#	Expression
0	E → E * E
1	E → E / E
2	E → E + E
3	E → E - E
4	E → (E)
5	E → i
6	E → lit
7	E → f ( )
8	E → f (E)
9	E → f (P)
10	P → E , E
11	P → P , E
12	E → E > E
13	E → E < E
14	E → E >= E
15	E → E <= E
16	E → E == E
17	E → E != E

	*	/	+	-	(	)	i	lit	f	,	>	<	>=	<=	==	!=	$
*	>	>	>	>	<	>	<	<	<	>	>	>	>	>	>	>	>
/	>	>	>	>	<	>	<	<	<	>	>	>	>	>	>	>	>
+	<	<	>	>	<	>	<	<	<	>	>	>	>	>	>	>	>
-	<	<	>	>	<	>	<	<	<	>	>	>	>	>	>	>	>
(	<	<	<	<	<	=	<	<	<	<	<	<	<	<	<	<	
)	>	>	>	>		>				>	>	>	>	>	>	>	>
i	>	>	>	>		>				>	>	>	>	>	>	>	>
lit	>	>	>	>		>				>	>	>	>	>	>	>	>
f					=												
,	<	<	<	<	<	>	<	<	<	>	<	<	<	<	<	<	
>	<	<	<	<	<	>	<	<	<	>							>
<	<	<	<	<	<	>	<	<	<	>							>
>=	<	<	<	<	<	>	<	<	<	>							>
<=	<	<	<	<	<	>	<	<	<	>							>
==	<	<	<	<	<	>	<	<	<	>							>
!=	<	<	<	<	<	>	<	<	<	>							>
$	<	<	<	<	<		<	<	<		<	<	<	<	<	<	e

Precedenční tabulka

LL-gramatika

#	Expression	Predict
1	CLASS_LIST → class id { STATIC_LIST } CLASS_LIST	class
2	CLASS_LIST → EPSILON	$
3	STATIC_LIST → static DEF STATIC_LIST	static
4	STATIC_LIST → EPSILON	}
5	DEF → TYPE_NVOID id DEF_NVOID	int, double, string
6	DEF → void id (PARAM_LIST) { FUNC_BODY }	void
7	TYPE_NVOID → int	int
8	TYPE_NVOID → double	double
9	TYPE_NVOID → string	string
10	DEF_NVOID → ASSIGN	=
11	DEF_NVOID → (PARAM_LIST) { FUNC_BODY }	(
12	DEF_NVOID → ;	;
13	PARAM_LIST → EPSILON	)
14	PARAM_LIST → TYPE_NVOID id PARAMS	int, double, string
15	PARAMS → , TYPE_NVOID id PARAMS	,
16	PARAMS → EPSILON	)
17	FUNC_BODY → TYPE_NVOID id LOC_VARDEF FUNC_BODY	int, double, string
18	FUNC_BODY → COMP_STAT FUNC_BODY	if, while, return
19	FUNC_BODY → EPSILON	}
20	COMP_STAT → return EXP ;	return
21	COMP_STAT → if (EXP) { COMP_STAT } else { COMP_STAT }	if
22	COMP_STAT → while (EXP) { COMP_STAT }	while
23	COMP_STAT → id ID_STAT COMP_STAT	id
24	COMP_STAT → EPSILON	}
25	ID_STAT → ASSIGN	=
26	ID_STAT → CALL	(
27	ASSIGN → = EXP ;	=
28	CALL → (ARG_LIST) ;	(
29	ARG_LIST → EPSILON	)
30	ARG_LIST → EXP ARGS	int_imm, double_imm, string_imm, id, (
31	ARGS → , EXP ARGS	,
32	ARGS → EPSILON	)
33	LOC_VARDEF → ASSIGN	=
34	LOC_VARDEF → ;	;
35	FUNC_BODY → id ID_STAT FUNC_BODY	id

Rozdělení práce

20 % - Petr Buchal – vedoucí týmu, scanner, tabulka symbolů, algoritmy, dokumentace
20 % - Petra Buchtová – LL gramatika, precedenční tabulka, parser
20 % - Pavel Černý - LL gramatika, precedenční tabulka, parser
20 % - Radim Červinka – scanner, interpret, dokumentace
20 % - Marek Doležel – supervize, LL gramatika, precedenční tabulka, parser, testování

Metriky kódu

Počet řádků: 5227
Počet zdrojových souborů: 34
Velikost spustitelného souboru: 130 kB