VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Dokumentace projektu

Interpret jazyka IFJ15

Tým 100, varianta a/2/II

Vedoucí: Tomáš Strnka (xstrnk01) 22%

Členové: Petr Žufan (xzufan00) 22%

Martin Wojaczek (xwojac00) 22%

Martin Kvapil (xkvapi13) 22%

Petr Gibiš (xgibis00) 12%

v Brně 28. listopadu 2015

Obsah

1 Úvod

Úkolem projektu bylo vytvořit funkční interpret jazyka IFJ15, který byl inspirován podmnožinou jazyka C++ verzí 2011. Dále v dokumentaci popisujeme základní principy použité k řešení.

Zvolili jsme si variantu a/2/II, což znamená vyhledávání podřetězce pomocí Knuth-Morris-Pratt algoritmu, řazení za pomoci Heap sort algoritmu a užití tabulky s rozptýlenými položkami.

2 Řešení projektu

Kapitola popisující jednotlivé části interpretu.

2.1 Lexikální analýza

Činnost, kterou provádí tzv. lexikální analyzátor (scanner) je první částí interpretu. Lexikální analýza se stará o odstranění bílých znaků a komentářů a rozdělení vstupního souboru na jednotky neboli tokeny. Identifikátory, klíčová slova, operátory, čísla, řetězce. Lexikální jednotky jsou dále poskytnuty k dalšímu zpracování syntaktickému analyzátoru.

Reprezentace „scanneru“ pomocí konečného automatu (viz příloha A). Nejdůležitější funkcí lexikální analýzy je *get_next_token()*, která opakovaným voláním navrací postupně všechny tokeny ze vstupního souboru. Když se ve vstupním souboru nachází lexikální chyba je navrácen token *LEX_ERROR* a načítání je ukončeno.

2.2 Syntaktická a sémantická analýza

Syntaktický analyzátor (parser) je nejdůležitějším členem celého kompilátoru. Je založený na rekurzivním sestupu. Postupně volá lexikální analyzátor pomocí funkce *GetNextToken()* a navrácené tokeny zpracovává. Aby syntaktická analýza fungovala správně, je potřeba vytvořit vhodně gramatická pravidla. Tyto pravidla reprezentuje LL gramatika (viz příloha C). Další částí parseru je precedenční syntaktická analýza, která kontroluje správnou syntax výrazů, podle precedenční syntaktické tabulky (viz příloha B). Pokud během kontroly parser zjistí, že program nedopovídá navržené LL gramatice, tak okamžitě skončí s chybou a dále již nepokračuje.

Během rekurzivního sestupu a precedenční syntaktické analýze se provádí kontrola sémantiky. Kontrola se zaměřuje na kontrolu deklarace proměnných. Další kontrola sémantiky probíhá až v interpretu. Zároveň s kontrolou syntaxe se generují instrukce a ukládají se symboly do globální a lokální tabulky symbolů.

2.3 Instrukce a interpret

Nezbytnou částí interpretu je generování a vykonávání instrukcí z instrukční pásky. Interpret pracuje se zásobníkem proměnných a s instrukcemi v tří-adresném kódu. Každý operand obsahuje jméno proměnné, po jejímž vyhledání získáme údaje jako například datový typ.

Interpret je závěrečnou fází zpracování zdrojového programu. Následuje po úspěšném skončení syntaktické analýzy, která již generovala posloupnost instrukcí tří-adresného kódu. Interpret vykonává

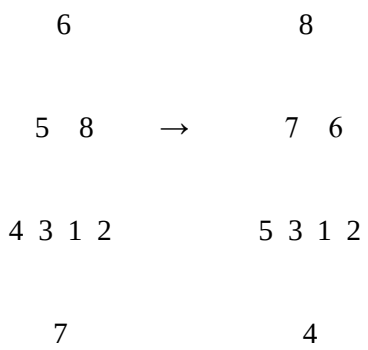
ještě některé sémantické kontroly inicializace a kompatibility datových typů. Instrukční páska je implementována jako lineárně zřetězeny seznam. Interpret vyjme z globální tabulky symbolů instrukční pásku funkce *main* a zpracovává instrukce jednu po druhé. Zároveň na zásobník vloží kopii lokální tabulky symbolů, do které ukládá hodnoty proměnných. Když dojde na volání funkce, vytvoří se nová kopie proměnných na vrcholu zásobníku a instrukční páska dané funkce se vloží do právě zpracovávané instrukční pásky. Návratová hodnota funkce je vždy uložena v proměnné #RETURN.

3 Algoritmy a datové struktury

3.1 Heap sort

Neboli řazení hromadou. Podstatou řadící metody je implementace hromady polem. Je to implementace s implicitním zřetězením prvků binární stromové struktury hromady. Halda má rekurzivní vlastnost, že každý podstrom je taktéž haldou. Dále pro haldu platí, že kořenový prvek má vyšší hodnotu než jeho potomci. V reprezentaci je prvek označen indexem n a jeho potomci indexy 2n+1 a 2n+2. Funkce siftdown upravuje prvky haldy tak, že porovnává kořenový prvek haldy s jeho potomky a prohodí je v případě, že potomci mají vyšší hodnotu než kořenový prvek. Tato operace se opakuje i pro podhaldy které mají jako kořenový prvek potomky předchozí haldy. V případě výměny kořenového prvku je možné, že dojde k nerovnováze o úroveň méně, tehdy se siftdown se aplikuje i na danou úroveň. Takto program pokračuje, dokud nedojde ke kořenovému prvku bez potomků anebo konkrétní halda splňuje uspořádání. Program nakonec přečte prvky z uspořádané haldy.

Ukázka třídění haldy: 65843127



V ukázce došlo k 3 krokům výměny. Prvek 8 a 6 byly prohozeny, a dále 7 a 4, kdy došlo k nerovnováze na vyšší úrovni, které vede k dalšímu prohození s prvkem 5.

3.2 Knuth-Morris-Prattův algoritmus

Pro vyhledávání podřetězce v řetězci při použití vestavěné funkce je použit Knuth-Morris-Prattův algoritmus. Ten vytváří masku hledaného podřetězce a určuje posun vyhledávání v případě neshody v porovnávání určitého písmene podřetězce s písmenem v prohledávaném textu.

Ukázka masky podřetězce:

1	2	3	4	5	6	7	8
A	B	A	C	A	B	D	B
0	0	1	0	1	2	0	0

Maska je pole indexů, které určuje na kterou pozici se má vyhledávací funkce vrátit v případě neshody. V našem případě (viz ukázka) je vidět význam tohoto algoritmu kdy se uvnitř podřetězce nachází stejná sekvence jako na začátku. Písmena s indexy 1, 2 a 5, 6 jsou stejné. Pokud nastane neshoda na indexu 7, pak víme, že předchozí znaky byly zkontrolovány a je tedy možné posunout rovnou na index 2. Následuje kontrola indexu 3, případě neúspěchu i tohoto indexu se podle masky vracíme na index 0. Dochází ke zrychlení porovnávání ve smyslu, že díky přeskočení určitých částí textu, které jsou již zkontrolovány během předchozích kroků. Takto se podřetězec posunuje po délce celého textu, dokud nenarazí na konec řetězce. Časová složitost celého algoritmu je o (m+n) díky vytvoření masky.

3.3 Tabulka s rozptýlenými položkami

Datová struktura tabulka s rozptýlenými položkami je použita pro uchování symbolů funkcí v překládaném kódu. Použita je tabulka s označením globální, pro symboly funkce, která obsahuje ukazatele na další tabulky, s označením lokální, pro symboly vně funkcí. Výhodou této datové struktury je rychlost vyhledávání symbolů v tabulce. Tabulka je reprezentována polem ukazatelů na jednosměrné seznamy. Každá položka obsahuje ukazatel na symbol a na další položku seznamu.

Pomocí hashovací funkce je pro každý symbol mapován klíčový řetězec na index, kterým je určeno rozložení v poli ukazatelů. Při vyhledávání se podle indexu určí seznam a ten sekvenčně se prohledá pro požadovaný symbol.

4 Práce v týmu

Před samotným zadáním projektu jsme si podle zadání z minulých let smluvili a rozdělili práci podle schopností a praxe každého z nás. Práce na projektu započaly koncem října, kdy jsme si založili Git repositář a byl zaznamenán první „commit“. Během řešení jsme se scházeli každý týden, abychom diskutovali o problémech jednotlivých částí a jak je následovně dát dohromady. Problémem byla velká časová náročnost projektu. Bojovali jsme do poslední chvíle.

4.1 Rozdělení práce

Tomáš Strnka	Vedoucí týmu, parser, dokumentace
Martin Wojaczek	Vyhodnocování výrazů, testování, parser
Petr Žufan	Interpret, struktury, testovaní
Petr Gibiš	Algoritmy, dokumentace
Martin Kvapil	Scanner, dokumentace

5 Závěr

Projekt se podařilo úspěšně dokončit a otestovat vzorovými zdrojovými kódy a několika vlastními. Nebylo to snadné a často jsme zažívali perné chvilky, obzvláště když ubíhaly poslední hodiny. Všichni to ale bereme jako velkou zkušenost pro naši budoucí kariéru.

6 Literatura

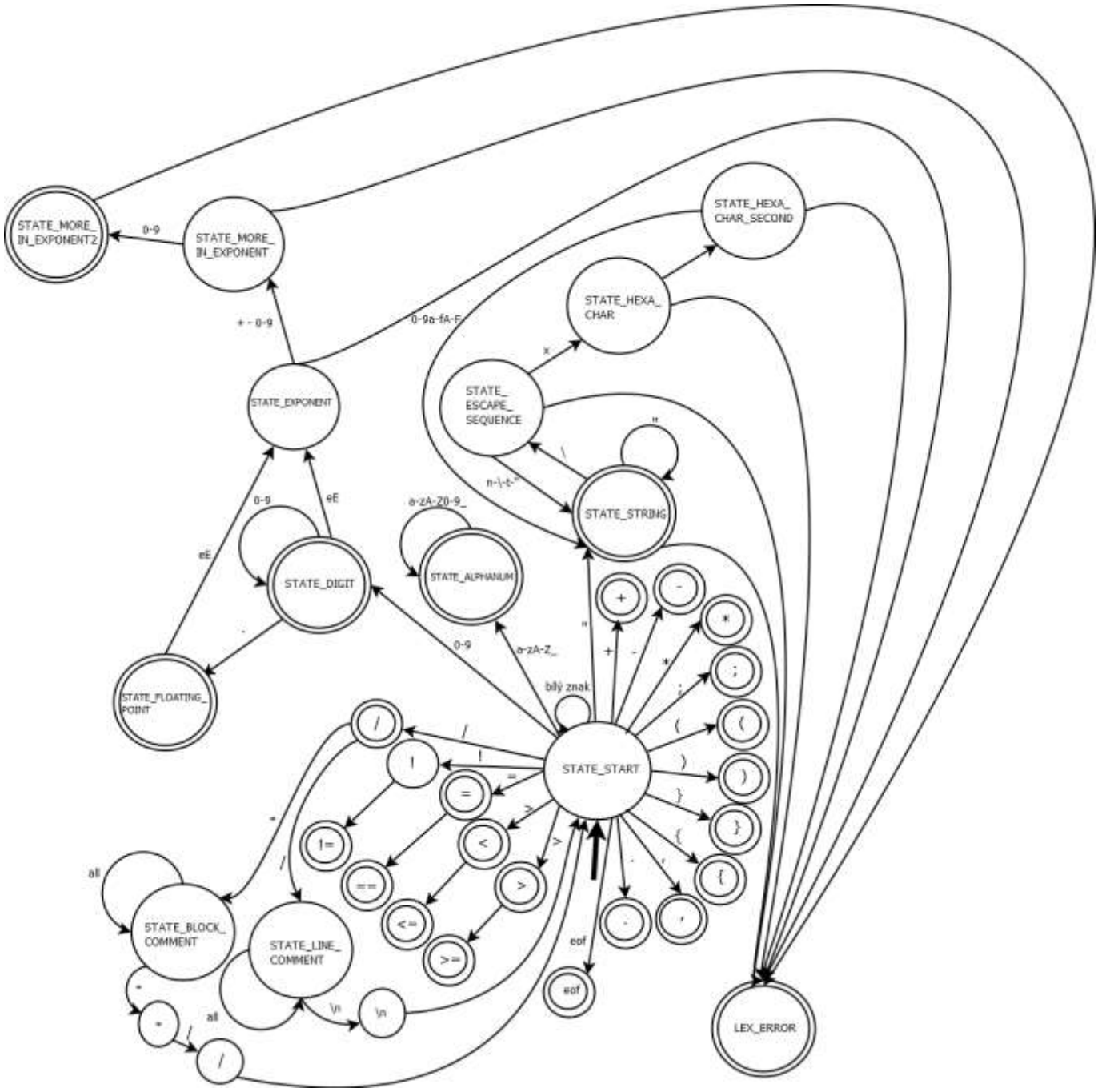
HONZÍK, Jan. Algoritmy – studijní opora. Brno: Vysoké učení technické, 2015.

7 Metriky kódu

Počet řádků zdrojového textu: **6065**

Velikost spustitelného programu: **161 KiB**

A Konečný automat - Lexikální analyzátor



B Precedenční tabulka

	+	-	*	/	<	>	<=	>=	==	!=	(	)	i	$	s	d	f
+	>	>	<	<	>	>	>	>	>	>	<	>	<	>	<	<	<
-	>	>	<	<	>	>	>	>	>	>	<	>	<	>	0	<	<
*****	>	>	>	>	>	>	>	>	>	>	<	>	<	>	0	<	<
/	>	>	>	>	>	>	>	>	>	>	<	>	<	>	0	<	<
<	<	<	<	<	>	>	>	>	>	>	<	>	<	>	<	<	<
>	<	<	<	<	>	>	>	>	>	>	<	>	<	>	<	<	<
<=	<	<	<	<	>	>	>	>	>	>	<	>	<	>	<	<	<
>=	<	<	<	<	>	>	>	>	>	>	<	>	<	>	<	<	<
==	<	<	<	<	>	>	>	>	>	>	<	>	<	>	<	<	<
!=	<	<	<	<	>	>	>	>	>	>	<	>	<	>	<	<	<
(	<	<	<	<	<	<	<	<	<	<	<	=	<	0	<	<	<
)	>	>	>	>	>	>	>	>	>	>	0	>	0	>	0	0	0
i	>	>	>	>	>	>	>	>	>	>	0	>	0	>	0	0	0
$	<	<	<	<	<	<	<	<	<	<	<	0	<	0	<	<	<
s	>	0	0	0	>	>	>	>	>	>	0	>	0	>	0	0	0
d	>	>	>	>	>	>	>	>	>	>	0	>	0	>	0	0	0
f	>	>	>	>	>	>	>	>	>	>	0	>	0	>	0	0	0

> – redukce, = – výjimka, < – handle, 0 – nedefinováno

C LL Gramatika

1. PROGRAM → FUNC_DCLR
2. FUNC_DCLR → TYPE id (PARAM) SELECT FUNC_DCLR

3. PARAM → TYPE id PARAM_N
4. PARAM → ε
5. PARAM_N → , TYPE id PARAM_N
6. PARAM_N → ε

7. SELECT → BODY
8. SELECT → ;

9. BODY → { STMNT }
10. STMNT → BODY STMNT

11. STMNT → IF STMNT
12. STMNT → FOR STMNT
13. STMNT → CIN STMNT
14. STMNT → COUT STMNT
15. STMNT → RETURN STMNT
16. STMNT → PROM STMNT

17. FOR → for (TYPE id I_PROM ; EXPR ; id = EXPR) STMNT
18. IF → if (EXPR) STMNT else STMNT
19. CIN → cin >> id ID_N ;
20. COUT → cout << TERM TERM_N ;
21. RETURN → return EXPR ;

22. ID_N → >> id ID_N
23. ID_N → ε

24. TERM_N → << TERM TERM_N
25. TERM_N → ε

26. I_PROM → = EXPR
27. I_PROM → ε

28. PROM → id = CALL_DEC ;
29. PROM → TYPE id I_PROM ;

30. CALL_DEC → id (LIST_PAR)

31. CALL_DEC → EXPR

32. LIST_PAR → TERM LIST_PAR_N
33. LIST_PAR_N → , TERM LIST_PAR_N
34. LIST_PAR_N → ε
35. LIST_PAR → ε

36. TYPE → int
37. TYPE → string
38. TYPE → double
39. TYPE → auto
40. TERM → term