VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Projekt z předmětů IFJ & IAL
Překladač jazyka IFJ17

Švub Daniel (xsvubd00)

Poisl Daniel (xpoisl00)
Weigel Filip (xweige01)
Zwierz Jan (xzwier00)

6. prosince 2017 Tým 88, varianta II

Obsah

1 Úvod

Tato dokumentace popisuje návrh a způsob implementace projektu z projektů IFJ a IAL. Jedná se o analyzátor zdrojových kódů jazyka IFJ17, zjednodušené varianty jazyka FreeBasic. Druhá část projektu, interpret daného jazyka do mezikódu IFJcode17, není implementována. Program pouze ověří lexikální, syntaktickou a sémantickou správnost zdrojového kódu.

2 Práce v týmu

2.1 Rozdělení práce

- **Daniel Švub** – vedoucí týmu, gramatika, parser, zásobník, úpravy
  35% - *Odůvodnění:* implementace celého parseru včetně sémantické analýzy
- **Daniel Poisl** – scanner, tabulka symbolů
  15%
- **Filip Weigel** – scanner, GIT repozitář, funkce ke strukturám
  35% - *Odůvodnění:* debugging projektu, administrace repozitáře
- **Jan Zwierz** – gramatika, dokumentace, makefile
  15%

2.2 Komunikace mezi členy

Naše schůzky probíhaly na kolejích, kde bydlí 3 ze 4 členů týmu. Na první schůzce byly rozděleny úkoly a části projektu jednotlivým členům. Tři na této koleji se scházeli, jak uznali za vhodné, čtvrtý člen docházel, kdykoli to bylo potřeba, popř. komunikace probíhala přes internet.

2.3 Sdílení výsledků práce

Ke sdílení práce jednotlivců byl členem týmu Filipem Weigelem vytvořen repozitář na serveru GitHub. Ten byl využíván ke sdílení aktuálních verzí souborů, na kterých jednotliví členové provedli jakékoli úpravy. Nejdříve byly na repozitář umístěny soubory scanner.c a parser.c společně s hlavičkovými soubory. Následovaly další zdrojové soubory s tabulkami vypracovanými v excelu a nakonec i dokumentace.

2.4 Problémy

Jelikož jsme špatně odhadli rozsah projektu a měli jsme problém se zahájením implementace, nedokázali jsme dokončit celý projekt. Problém spočíval v přílišném odkládání zahájení prací na projektu s tím, že při samotné implementaci jsme naráželi na víc problémů než jsme očekávali. Například problémy s kompatibilitou zdrojových souborů tvořených jednotlivými členy, přičemž každý očekával něco jiného od zdrojového kódu druhé strany. To způsobovalo nečekané chyby při překladu programu a vyžádalo si zdlouhavé debuggování.

3 Implementace

3.1 Scanner

Scanner je v programu využíván na čtení symbolů ze vstupu a následným předáním dat parseru, ale provádí i lexikální analýzu. Realizován je pomocí podmínek, které rozpoznají jednoduché znaky (+, -, . . .) a pro složitější výrazy jako např. klíčová slova je využito jednotlivých case. Celkem je ve scanneru použito 10x case. Každý case slouží k jinému rozpoznávání, ale některý case se skládá i z více case.

Scanner je volán Parserem. Scanner následně začne číst vstupní kód a zapisuje data na pole charu. Pokud je zjištěna lexikální chyba v zadávaném kódu, je Scanner ukončen a posílá Parseru hodnotu 1(chyba v programu v rámci lexikální analýzy). Data se poté posílají zpět Parseru na syntaktickou a sémantickou analýzu.

- **Case 1**: Spustí se pokud je zadán *backslash* a určí zda se jedná o celočíselné dělení, nebo o komentář
- **Case 2**: Rozpoznávání zda se jedná o klíčové slovo jazyka, nebo zda je zadáván identifikátor proměnné, funkce.
- **Case 3**: Zapisování a určení zda se jedná o celé číslo. Pokud je zadána tečka tak se spouští *Case 7*.
- **Case 4**: zapisování vstupního řetězce. Zapisování se ukončí pokud je zadána *uvozovka*.
- **Case 5**: Čte řádkový komentář a hledá jeho ukončení pomocí odřádkování v kódu.
- **Case 6**: Čte blokový komentář. Po přečtení apostrofu je spuštěn *Case 8*.

- **Case 7**: Čte čísla a zapisuje si jejich hodnotu. Po zadání operátoru je příslušná hodnota čísla odeslána Parseru.
- **Case 8**: Kontroluje zda se po *apostrofu* nachází *backslash*. Pokud ano tak je ukončen blokový komentář, pokud ne tak se opět spouští *Case 6*.
- **Case 9**: Ošetření vícenásobného odřádkování v zadávaném kódu.
- **Case 10**: Kontrola jestli se po vykřičníku zadává vstupní řetězec. Pokud je za vykřičníkem cokoliv jiného než uvozovka tak je vrácena lexikální chyba kódu.

Scanner byl vyvíjen postupným přidáváním podmínek pro rozpoznávání jednotlivých znaků. Začínalo se jednoduchým rozpoznáváním operátorů (+, -, *, /). Další fáze bylo rozpoznávání klíčových slov a identifikátorů. Závěrečným krokem bylo oddělení komentářů ze vstupu, aby se překládal pouze kód a zapisování vstupních řetězců a čísel.

3.2 Parser

Parser je hlavní částí programu. Hlavními úkoly parseru jsou syntaktická a sémantická analýza kódu. Syntaktická analýza je dále rozdělena na rekurzívní sestup a precedenční syntaktickou analýzu.

Při vypracování Parseru se nejdříve vyřešila komunikace se Scannerem. Dalším krokem byla implementace rekurzívního sestupu a precedenční analýzy. Po předchozí implementaci se vyřešily chyby v gramatice a vypisování chybových hlášení. Následovala sémantická analýza. Vytvořil se soubor s funkcemi vztahující se k hashovací tabulce.

Analýza začíná kontrolou, pomocí Scanneru, zda se v kódu nenachází lexikální chyba. Pokud Scanner vrátí hodnotu 1 je Parser ukončen a vrací hodnotu *01: Lexikální chyba*. Chyba nebyla nalezena, a tak Parser přechází k samotné analýze. Parser volá Scanner funkcí getNextToken() a získává od něj číselný kód, který určuje co je nalezeno (identifikátor, celé číslo, desetinné atd.) a pole charu s příslušnou hodnotou nebo řetězcem. Pomocí analýzy kontroluje syntaktickou správnost kódu. Při nalezení syntaktické chyby je Parser ukončen a vypíše *02: Syntaktická chyba*.

Rekurzívní sestup je rozdělen do několika funkcí, které jsou rozlišeny pomocí předpony *des_* , např. *des_TTYPE*. Analýza shora dolů je používána ve většině kontrol analytické správnosti kódu. Precedenční analýza je použita jen ke kontrole správnosti výrazů v kódu. Precedenční analýza má proto jen jednu funkci s názvem *des_exp* a využívá externí funkci *pre_next*.

Sémantická analýza využívá ke své funkčnosti dvě hashovací tabulky. Obě tyto tabulky jsou stejného typu, ale do jedné se ukládají proměnné a do druhé funkce. S Hashovací tabulka je řešena v samostatném souboru *symtable.c*. Tento soubor má v sobě funkce na vytvoření tabulky, vložení prvku do tabulky, vyhledávání v tabulce, smazání prvku a smazání tabulky. Tabulka je tvořena strukturou. Parser vytvoří dvě tabulky a po nalezení funkce nebo proměnné zapisuje do příslušné tabulky. Pokud se zapisuje do proměnné tak se hledá, jestli proměnná existuje, a pokud ano tak se přiřadí určitá hodnota.

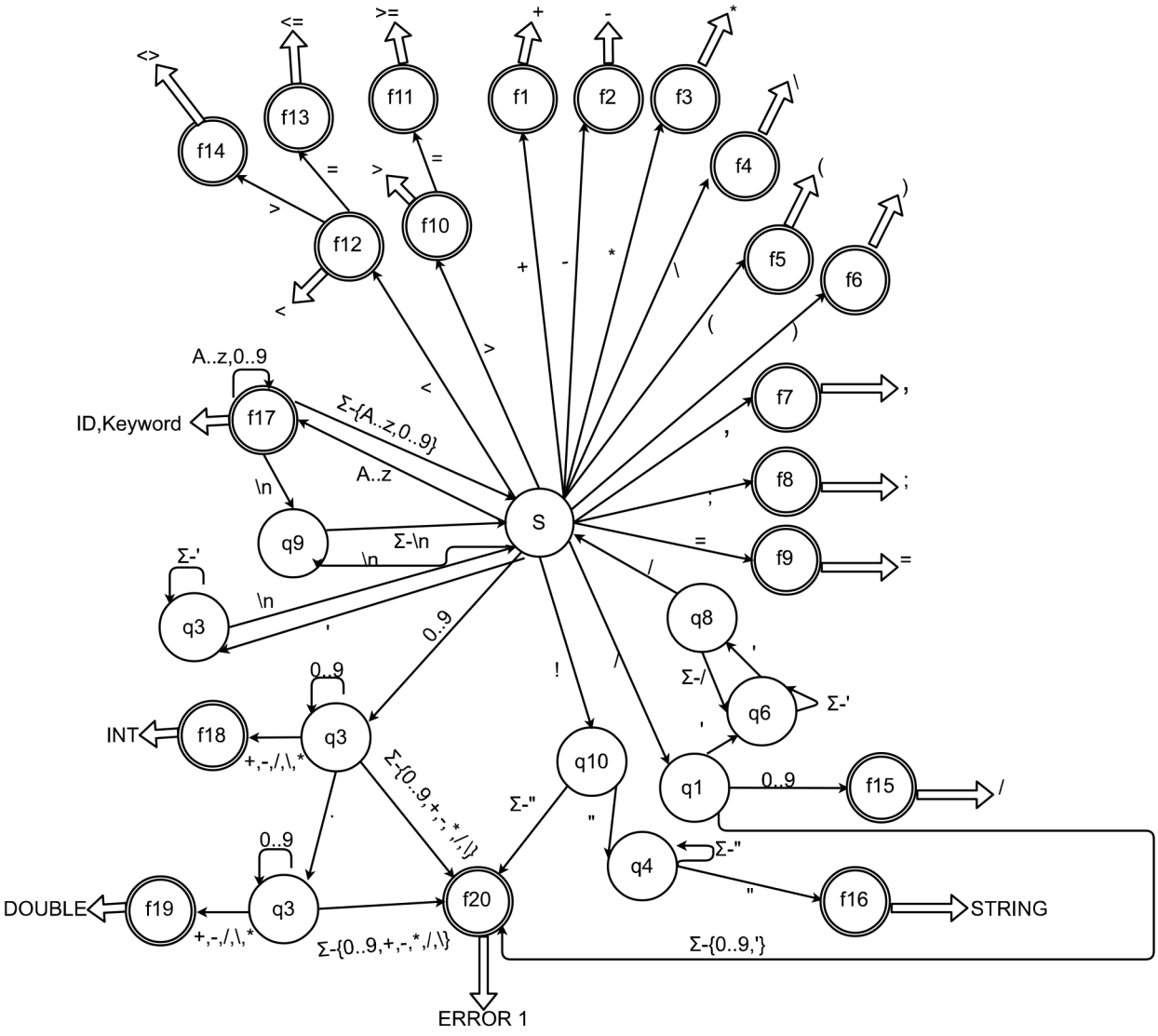
Problémy: Při implementaci parseru představovala největší problém přecedenční syntaktická analýza a její spolupráce s rekurzivním sestupem, kontrola datových typů a ovládání tabulek symbolů a mnohá omezení jazyka C. Některé funkce se nepodařilo vypracovat.

4 Závěr

Zpočátku se práce na projektu vyvíjela poměrně dobře, při vytváření rekurzívního sestupu a lexikálního analyzátoru všichni členové odvedli kvalitní práci. Následně ovšem došlo k špatnému rozdělení další práce, problémům se špatnou kompatibilitou souborů a nedostatečným obeznámením jednotlivých členů s kódy ostatních. Pod narůstajícím tlakem v časové tísni se projekt nepodařilo dokončit.

5 Přílohy

5.1 Konečný automat lexikálního analyzátoru



5.2 LL tabulka

	scope	declare	function	end	EOL	dim	ID	print	input	if	else	do	loop	)	TERM	,	return	EOF	[pa]
<prog>	1	25	30															33	
<sc-list>				3		2	2	2	2	2		2					2		
<func-list>				32		31	31	31	31	31		31					31		
<if-list>				10		9	9	9	9	9	11	9					9		
<else-list>				13		12	12	12	12	12		12					12		
<loop-list>						15	15	15	15	15		15	16				15		
<stat>						4	17	6	7	8		14					24		
<def>							5												
<ass>							19												18
<par-list>							27							26					
<par>														29		28			
<in-list>														20	21				
<in>														23		22			
<exp>																			34

5.3 Precedenční tabulka

	+	-	*	/	\	=	<>	<	<=	>=	(	)	i	$
+	>	>	<	<	<	>	>	>	>	>	<	>	<	>
-	>	>	<	<	<	>	>	>	>	>	<	>	<	>
*	>	>	>	>	>	>	>	>	>	>	<	>	<	>
/	>	>	>	>	>	>	>	>	>	>	<	>	<	>
\	>	>	<	<	>	>	>	>	>	>	<	>	<	>
=	<	<	<	<	<	>	>	>	>	>	<	>	<	>
<>	<	<	<	<	<	>	>	>	>	>	<	>	<	>
<	<	<	<	<	<	>	>	>	>	>	<	>	<	>
<=	<	<	<	<	<	>	>	>	>	>	<	>	<	>
>	<	<	<	<	<	>	>	>	>	>	<	>	<	>
>=	<	<	<	<	<	>	>	>	>	>	<	>	<	>
(	<	<	<	<	<	<	<	<	<	<	<	=	<	
)	>	>	>	>	>	>	>	>	>	>		>		>
i	>	>	>	>	>	>	>	>	>	>		>		>
$	<	<	<	<	<	<	<	<	<	<	<		<	

5.4 Zjednodušená precedenční tabulka

	+, -	*, /	\	=, <, >, <>, <=, >=	(	)	i	$
+, -	>	<	<	>	<	>	<	>
*, /	>	>	>	>	<	>	<	>
\	>	<	>	>	<	>	<	>
=, <, >, <>, <=, >=	<	<	<	>	<	>	<	>
(	<	<	<	<	<	=	<	
)	>	>	>	>		>		>
i	>	>	>	>		>		>
$	<	<	<	<	<		<	