FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Projektová dokumentace
Implementace překladače imperativního jazyka IFJ20

Tým 039, varianta I

9. prosince 2020

Tomáš Hrúz	(xhruzt00)	25 %
Mirka Kolaříková	(xkolar76)	30 %
Aleš Řezáč	(xrezac21)	20 %
Martin Žovinec	**(xzovin00)**	25 %

Obsah

1 Úvod

Cílem tohoto projektu je vytvořit překladač z jazyka IFJ20, který je založen na programovacím jazyku Go, do cílového jazyka IFJcode20. Projekt je implementován v jazyce C.

2 Implementace

V této kapitole představíme implementaci jednotlivých částí, které spolu komunikují a tvoří tak výsledný překladač.

2.1 Lexikální analýza

Zásadní částí překladače je **scanner**, který načítá a analyzuje jednotlivé lexémy a předává je parseru v podobě tokenů. Scanner jsme implementovali podle vytvořeného diagramu deterministického konečného automatu bez epsilon přechodů. Diagram uutomatu je v příloze 1.
Scanner je implementován v souborech `scanner.c` a `scanner.h`. Hlavní funkcí tohoto modulu je `get_new_token(&tokenStr)`, kde se ve smyčce načítají znaky ze standardního vstupu funkcí `getc()`. Konečný automat je v této funkci implementován jako jeden `switch`, kde každý případ `case` je jeden stav konečného automatu. Všechny konečné stavy jsou definovány v hlavičce a každému je přiřazeno konstantní číslo pomocí makra `#define`.
Když načítaný lexém dojde do konečného stavu automatu, funkce se ukončí a vrátí lexému odpovídající definované číslo. V případě lexému typu `int`, `float64` nebo `string` je hodnota uložena a vrácena ve stringu `tokenStr` . Pokud načítaný znak neodpovídá konečnému automatu, celý program se ukončí s kódem `1` a na standardní chybový výstup vypíše odpovídající chybovou hlášku.
Hexadecimální sekvence jsou převáděny na odpovídající ASCII hodnoty pomocí vytvořené funkce `hex_to_int(char ch)`.

2.2 Syntaktická analýza

Syntaktická analýza je implementována v souborech `parser.c` a `parser.h`. **Parser** je nejdůležitější částí, protože řídí průběh celého překladu.
Syntaktická analýza se řídí LL-gramatikou a metodou rekurzivního sestupu. Každému pravidlu v LL-gramatice odpovídá jedna funkce v parseru. LL-gramatiku je možné si prohlédnout v příloze 2.
Výchozí funkcí parseru je funkce `parse()`, která načte první token, zahájí kontrolu prvního pravidla a volá funkce pro kontrolu dalších pravidel.
Tokeny jsou postupně načítány ze scanneru funkcí `get_new_token(&tokenStr)`. V případě, že aktuální token neodpovídá pravidlu v gramatice, je celý program ukončen s chybovým kódem `2`. Pokud celý překlad proběhne bez chyby, funkce `parse()` vrátí úspěch (návratová hodnota `0`).
Pro analýzu výrazů je volána funkce `prec_parse(varNode *treePtr, int new_token, string tokenStr)`. Této funkci je předán první token výrazu a ukazatel na tabulku symbolů.

2.3 Sémantická analýza

Sémantická analýza probíha souběžně se syntaktickou analýzou. Pomocí tabulky symbolů (tady referenci na implementaci), která obsahuje mnoho pomocných funkcí, je kontrolováno, zda byly

proměnné a funkce definovány, zda jsou proměnné správného datového typu a zda volaná funkce obsahuje správný počet a typ parametrů. Tyto funkce jsou volány z parseru a z parseru výrazů. Pro kontrolu proměnných se využívá struktura varNode a pro kontrolu funkcí se využívá struktura funNode. Další kontroly se nacházejí přímo v parseru.

2.4 Zpracování výrazů

Analýza výrazů je zpracována odděleně v souborech `prec_parse.c` a `prec_parse.h`. Hlavní funkcí je `prec_parse(varNode *treePtr, int new_token, string tokenStr)`, která je volaná z parseru. Tato funkce dostane první token výrazu a odkaz na tabulku symbolů, ve které má hledat ve výrazu se vyskytující proměnné. V případě úspěchu vrátí strukturu `prec_end_struct` obsahující výsledný datový typ výrazu a token, podle kterého se vyhodnotilo ukončení výrazu. Zpracování výrazů probíhá pomocí precedenční analýzy. Nový token je porovnán s tokenem na vrcholu zásobníku a podle precedenční tabulky je umístěn na zásobník, nebo je výraz zpracován podle algoritmu probraného na přednášce IFJ. Precedenční tabulka je v příloze 3. Hodnota `X` značí sémantickou chybu, hodnota `Y` značí syntaktickou chybu.

2.5 Generování cílového kódu

3 Abstraktní datové struktury

V následující kapitole představíme využité datové struktury.

3.1 Tabulka symbolů

Tabulku symbolů jsme řešili formou rekurzivního binárního vyhledávacího stromu. Rekurzivní implementaci jsme zvolili pro její jednoduchost, která nám umožnila rychlé programování základních operací nad binárním stromem, díky čemuž jsme se mohli dříve věnovat složitějším problémům.

3.1.1 Tabulka symbolů pro proměnné

Každá definice funkce v parseru si inicializuje svůj vlastní binární vyhledávací strom pro proměnné. Každý uzel stromu varNode obsahuje: Název proměnné Abstraktní datovou strukturu zásobníku, ve které je uložen typ proměnné a úroveň zanoření Ukazatele na levý a pravý podstrom

Proměnné se v parseru přidávají pomocí funkce BSTInsert (varNode *RootPtr, string Key, int Type, int scope), která zkontroluje, zda nedochází k redefinici proměnné ve stejné úrovni zanoření. Pokud tomu tak není, tak se pushne na vrchol zásobníku nový typ přepsané proměnné i s úrovní zanoření.

Při snížení zanoření parser volá funkci BSTScopeDelete(varNode *RootPtr, int newScope), která projde celý binární strom a popne všechny zásobníky proměnných, jejichž úroveň zanoření už není platná. V případě, že je vrchol zásobníku některé proměnné po popnutí prázdný, tak je uzel dané proměnné smazán a nahrazen nejpravějším uzlem z levého podstromu mazaného uzlu.

Pro jednoduchou integraci tabulky symbolů s parserem jsou k dispozici funkce: isDeclared (varNode RootPtr, string Key), pro snadnou kontrolu ukládání hodnot do nedeklarovaných proměnných getType(varNode RootPtr, string Key), pro kontrolu ukládání hodnot do proměnných se špatným datovým typem.

3.1.2 Tabulka symbolů pro funkce

Parser pro sématickou kontrolu funkcí využívá globální binární vyhledávací strom.

Každý uzel stromu funNode osahuje:

Název funkce Boolovské hodnoty isDeclared a isCalled pro kontrolu redeklarace funkce a volání bez deklarace funkce a listy Abstraktní datovou strukturu List s typy parametrů funkce Abstraktní datovou strukturu List s typy returnů funkce Ukazatele na levý a pravý podstrom

Zpracování funkce v parseru probíhá v několika krocích, které jsou z větší části shodné v případech volání i deklarace: 1. Funkcí addFunToTree(funNode *RootPtr, string Key) se přidá funkce s jejím názvem do stromu, pokud se v němu už nenachází. 2. Funkcí funDecCheck(funNode *RootPtr, string Key) se u deklarace funkce kontroluje případná redeklarace funkce. 3. Funkcemi addParam(funNode *RootPtr, string Key, int parameterType, int parameterOrder) a addReturn(funNode *RootPtr, string Key, int returnType, int returnOrder) se zkontroluje, zda funkce už byla deklarována nebo volána, pokud ne, tak se přidá nový parametr/return do odpovídajícího listu, pokud už volána byla, tak se parametr/return porovná s parametrem/returnem v listu uzlu funkce ve stejném pořadí 4. Na konci zpracování se využívají fukce: a. U deklarace addFunDec(funNode *RootPtr, string Key, int paramCount) zkontroluje počet parametrů a nastaví hodnotu isDeclared na true. b. U volání addFunCall(funNode *RootPtr,string Key,varNode varTree,int paramCount) zkontroluje počet parametrů a nastaví hodnotu isCalled na true. Tato funkce také kontroluje, zda se nevolá funkce se stejným názvem jako proměnná.

Zpracování funkce je děláno tímto univerzálním způsobem, protože volání a deklarace funkce může být v různém pořadí. Proto také probíhá kontrola volání funkce bez deklarace až na konci parseru funkcí isFunCallDec(funNode RootPtr), která rekurzivně projde celý binární strom a u každé funkce zkontroluje, že její funkce nemá hodnotu isDeclared true bez toho, aniž by hodnota isCalled byla také true, čímž zjistíme případnou chybu volání nedeklarované funkce.

3.2 Zásobník pro zpracování výrazů

Tento zásobník je implementován v souborech `precedence_stack.c` a `precedence_stack.h`. Do zásobníku se ukládá struktura `struc_tokens`, obsahující typ tokenu a jeho obsah. Nad zásobníkem jsou implementovány základní funkce jako `push_precStack`, `pop_precStack`, `isEmpty_precStack`, `isFull_precStack`, `peek1_precStack` a nějaké další.

3.3 Vázaný seznam na cílový kód

3.4 Dynamický string

Využili jsme dynamický string implementovaný v souborech str.c a str.h z ukázkového interpretu.

4 Testování

Testování probíhalo ze začátku manuálně a v pozdější části vývoje jsme přešli na automatické testy.

5 Práce v týmu

Tým jsme vytvořili již před začátkem semestru. Na komunikaci jsme si vytvořili Discord server, kde probíhaly online schůzky a měli jsme zde i sdílený TODO list. Na řešení menších problémů jsme používali Facebook Messenger. Schůzky probíhaly minimálně jednou týdně a podle potřeby i vícekrát.

Na projektu jsme začali pracovat ihned po zveřejnění zadání. Původní plán byl využít metodiku Scrum, ta se bohužel neuplatnila a přešli jsme na rozdělení práce podle jednotlivých částí projektu. Každý člen týmu se věnoval své části a ostatní členové týmu pomáhali když nastal nějaký problém. Některé části překladače vyžadovaly spolupráci členů týmu, tyto části jsme tedy řešili společně. Díky pravidelným schůzkám jsme mohli sledovat pokrok a měli jsme dobrou představu o tom, jak se projekt postupně vyvíjí.

5.1 Sdílení kódu

Použili jsme verzovací systém Git a servery GitHub. Další software, skterý jsme použili je GitKraken a funkci Source Control pro Visual Studio Code.

5.2 Rozdělení práce

Tabulka 1 obsahuje rozdělení práce mezi členy týmu.

Člen týmu	Práce na projektu
Tomáš Hrúz	LL-Gramatika, Parser, Dokumentace
Mirka Kolaříková	Scanner, Precedenční analýza výrazů, Parser, Dokumentace
Aleš Řezáč	Generování kódu
Martin Žovinec	Vedoucí týmu, Tabulka symbolů, Testování, Generování kódu

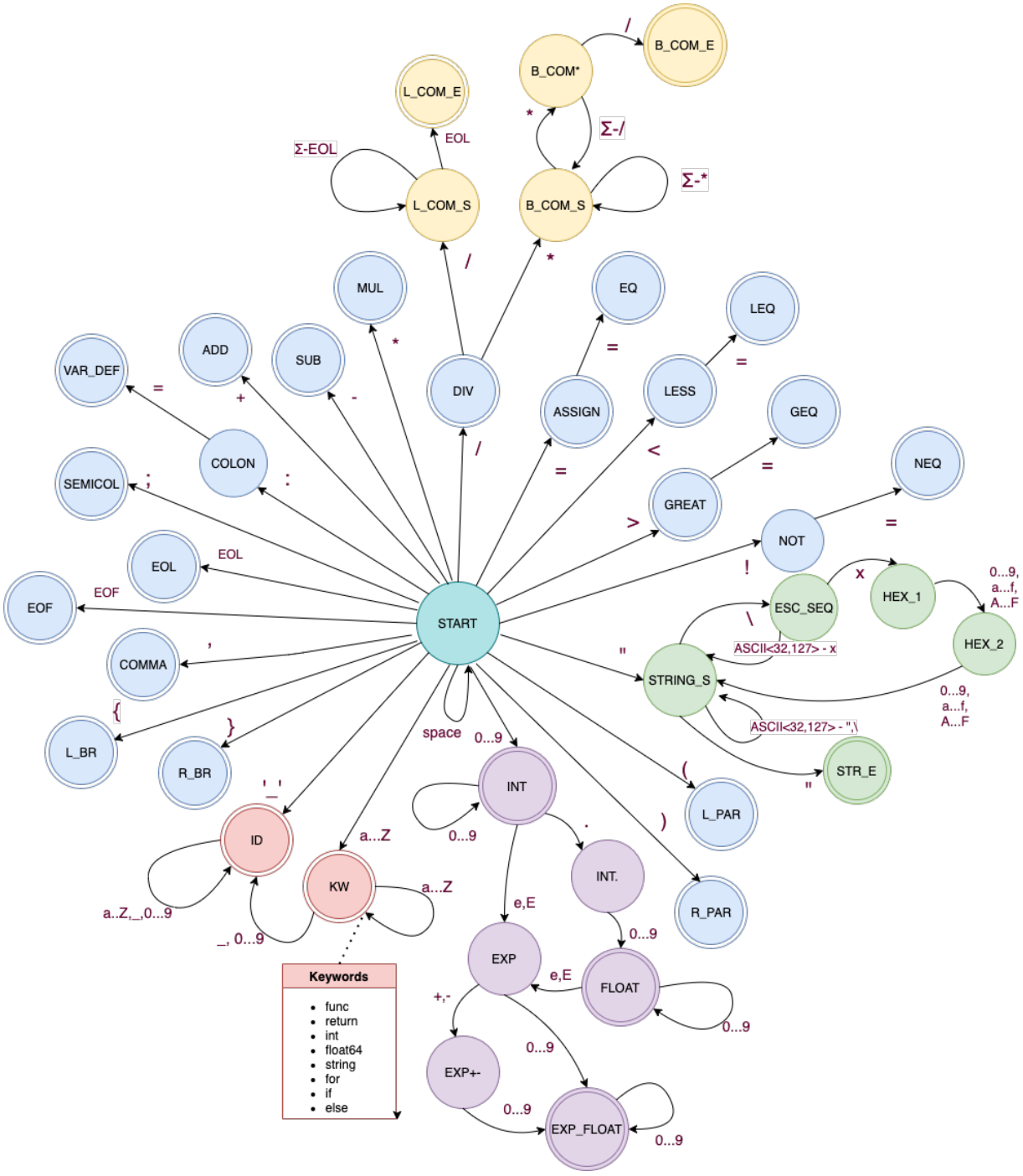
Tabulka 1: Rozdělení práce mezi členy týmu

6 Závěr

Projekt byl rozsáhlý ale díky našim zkušenostem jsme si s projektem poradili mnohem lépe než minulý rok. Scanner byl hotový již před registrací týmů. Syntaktická analýza byla hotová pár týdnů po scanneru a v čase nultého pokusného odevzdání jsme odevzdali i precedenční analýzu výrazů. Do dalšího pokusného odevzdání byla hotová i tabulka symbolů, s ní i velká část sémantické analýzy a základní generování kódu. Do finálního odevzdání jsme pracovali převážně na generování kódu a dělaly se drobné úpravy.

Projekt nám dal cenné zkušenosti s praktickou tvorbou překladačů, prací v týmu na rozsáhlém projektu a obohatil naše dovednosti s Gitem.

A Diagram konečného automatu



Obrázek 1: Diagram konečného automatu pro lexikální analýzu

B LL-gramatika

1	<prog>	<opt_eol>	<prolog>	<fun_def_list>	EOF							
2	<prolog>	package	main	EOL								
3	<fun_def_list>	<opt_eol>	<fun_def>	<fun_def_list>								
4	<fun_def>	func	main	(	)	{	EOL	<stat_list>	}			
5	<fun_def>	func	ID	(	<fun_params>	)	(	<fun_returns>	)	{	EOL	<stat_list>	}
6	<fun_params>	<par>	<par_next>									
7	<fun_params>	e										
8	<par_next>	,	<par>	<par_next>								
9	<par_next>	e										
10	<par>	ID	<type>									
11	<type>	FLOAT64										
12	<type>	INT										
13	<type>	STRING										
14	<fun_returns>	<ret>	<ret_next>									
15	<fun_returns>	e										
16	<ret_next>	,	<ret>	<ret_next>								
17	<ret_next>	e										
18	<ret>	<type>										
19	<stat_list>	<opt_eol>	<stat>	EOL	<stat_list>							
20	<stat_list>	e										
21	<stat>	if	<exp>	{	<stat_list>	}	else	{	<stat_list>	}		
22	<stat>	for	<var_def>	;	<exp>	;	<opt_exp>	<opt_eol>	{	EOL	<stat_list>	}
23	<stat>	<var_def>										
24	<stat>	return	<return_value>									
25	<stat>	<ass_stat>										
26	<stat>	print	(	<print_param>	)							
27	<stat>	<fun>										
28	<var_def>	ID	:=	<exp>								
29	<var_def>	e										
30	<ass_stat>	<ass_id>	=	<ass_exp>								
31	<ass_id>	ID	<ass_ids>									
32	<ass_ids>	,	ID	<ass_ids>								
33	<ass_ids>	e										
34	<ass_exp>	<fun>										
35	<ass_exp>	<exp>	<ass_exps>									
36	<ass_exps>	,	<exp>	<ass_exps>								
37	<ass_exps>	e										
38	<opt_exp>	<exp>										
39	<opt_exp>	e										
40	<opt_eol>	EOL	<opt_eol>									
41	<opt_eol>	e										
42	<fun>	ID	(	<fun_call_param>	)							
43	<fun_call_param>	<id_or_type>	<fun_call_params>									
44	<fun_call_params>	,	<id_or_type>	<fun_call_params>								
45	<fun_call_params>	e										
46	<print_param>	<id_or_type>	<print_params>									
47	<print_params>	,	<id_or_type>	<print_params>								
48	<print_params>	e										
49	<return_value>	<id_or_type>	<return_values>									
50	<return_values>	,	<id_or_type>	<return_values>								
51	<return_values>	e										
52	<id_or_type>	ID										
53	<id_or_type>	<type>										
54	<exp> Handled in prec_parser											

Tabulka 2: LL-gramatika pro syntaktickou analýzu

C Precedenční tabulka

```
/*            +    -    *    /    <   <=    >   >=   ==   !=    (    )   ID  INT FLOAT STR   $     */
/* + */     { '>', '>', '<', '<', '>', '>', '>', '>', '>', '>', '<', '>', '<', '<', '<', '<', '>' },
/* - */     { '>', '>', '<', '<', '>', '>', '>', '>', '>', '>', '<', '>', '<', '<', '<', 'X', '>' },
/* * */     { '>', '>', '>', '>', '>', '>', '>', '>', '>', '>', '<', '>', '<', '<', '<', 'X', '>' },
/* / */     { '>', '>', '>', '>', '>', '>', '>', '>', '>', '>', '<', '>', '<', '<', '<', 'X', '>' },
/* < */     { '<', '<', '<', '<', 'X', 'X', 'X', 'X', 'X', 'X', '<', '>', '<', '<', '<', '<', '>' },
/* <= */    { '<', '<', '<', '<', 'X', 'X', 'X', 'X', 'X', 'X', '<', '>', '<', '<', '<', '<', '>' },
/* > */     { '<', '<', '<', '<', 'X', 'X', 'X', 'X', 'X', 'X', '<', '>', '<', '<', '<', '<', '>' },
/* >= */    { '<', '<', '<', '<', 'X', 'X', 'X', 'X', 'X', 'X', '<', '>', '<', '<', '<', '<', '>' },
/* == */    { '<', '<', '<', '<', 'X', 'X', 'X', 'X', 'X', 'X', '<', '>', '<', '<', '<', '<', '>' },
/* != */    { '<', '<', '<', '<', 'X', 'X', 'X', 'X', 'X', 'X', '<', '>', '<', '<', '<', '<', '>' },
/* ( */     { '<', '<', '<', '<', '<', '<', '<', '<', '<', '<', '<', '=', '<', '<', '<', '<', 'Y' },
/* ) */     { '>', '>', '>', '>', '>', '>', '>', '>', '>', '>', 'X', '>', 'X', 'X', 'X', 'X', '>' },
/* ID */    { '>', '>', '>', '>', '>', '>', '>', '>', '>', '>', 'Y', '>', 'X', 'X', 'X', 'X', '>' },
/* INT */   { '>', '>', '>', '>', '>', '>', '>', '>', '>', '>', 'Y', '>', 'X', 'X', 'X', 'X', '>' },
/* FLOAT */ { '>', '>', '>', '>', '>', '>', '>', '>', '>', '>', 'Y', '>', 'X', 'X', 'X', 'X', '>' },
/* STR */   { '>', 'X', 'X', 'X', '>', '>', '>', '>', '>', '>', 'Y', '>', 'X', 'X', 'X', 'X', '>' },
/* $ */     { '<', '<', '<', '<', '<', '<', '<', '<', '<', '<', '<', 'Y', '<', '<', '<', '<', '#' }
```

Tabulka 3: Precedenční tabulka pro analýzu výrazů