Projektová dokumentace

Implementace překladače jazyka IFJ21

Tým 064, varianta II

Mikhailov Kirill (xmikha00) - 40% **(vedoucí)**
Naumenko Maksim (xnaume01) - 60%
Mazurava Maryia (xmazur08) - 0%

1. Úvod

Hlavním úkolem je naimplementovat v jazyce C překladač jazyka IFJ21, který je zjednodušenou podmnožinou jazyka Teal, staticky typovaného imperativního jazyka.

2. Jednotlivé časti překladače

- Lexikální analyzátor
- Syntaktický analyzátor
- Sémantický analyzátor
- Generátor cílového kódu

2.1. Lexikální analyzátor

Jednotka překladače, která začne zpracovávat lexikální jednotky ze vstupního kódu a z nich bude vytvářet tokeny, které budou zařazeny do dalšího zpracování, je realizována v modulu ***scanner.c***
Zpracování a rozdělení tokenů na jednotlivé typy jsou řízeny konečným automatem (Obr.1) . Podle přečteného znaku rozhodneme, do kterého stavu toho automatu se musíme přesunout. V tom stavu budeme generovat data, kterými naplníme ten nový token.
Pro zpracování celého lexému a vygenerování dat, které se nám budou dále hodit, používáme ***dynamický řetězec(dynamic string)***. Ten dostává znak po znaku dokud se nerozhodneme, že dany lexém je úplně přečten. Pokud dostali jsme nějaké celé číslo, uložíme jej do proměnné s odpovídajícím typem dat uvnitř struktury tokenu a do proměnné ***tokenStringVal*** uložíme ID proměnné, pokud jedná se o proměnné.
Navíc máme funkcí pro detekci klíčových slov, takže každý přečtený identifikátor předáváme té funkci a ona rozhoduje, zda se jedná o ***keyword*** nebo ne.

2.2. Syntaktický analyzátor

Potom ten token je odeslán na syntaktickou analýzu, do modulu ***parser.c***. Parser se sestává ze dvou částí : ***Bottom-Up*** analýza (pro analýzu výrazů) na základě ***Precedenční tabulky*** (Obr.2) a ***Top-Down*** analýza s použitím ***LL-gramatiky*** a ***LL-tabulky***(Obr. 3 a 4), která je realizovaná principem ***rekurzivního sestupu***.
Současně s analýzou naplňujeme ***tabulku symbolů*** daty o všech proměnných, funkcích, jejich argumentech a návratových hodnotách, které získáváme v době analýzy.
Máme II variantu zadání, takže tu tabulku museli jsme implementovat jako ***hash-table***. Tu jsme měli díky tomu, že jeden z členů týmu měl předmět IJC a měl dobře projekt s tou tabulkou, takže měli jsme strukturu a základní funkce.
V té syntaktické analýze taky inicializujeme nove ***uzly AST-stromu***, který je hlavním cílem práce syntaktického analyzátoru. Uzly uchovávají informace o typech operací a jejich operandech(každý uzel má přiřazenou hashovací tabulku)

Pozn:Bottom-Up analýza je implementována v souboru ***bottom_up_analysis.c***

2.2.1 Precedenční tabulka

E → id
E → #E
E → E * E
E → E / E
E → E // E
E → E + E
E → E - E
E → E .. E

E → E < E
E → E <= E
E → E > E
E → E >= E
E → E == E
E → E ~= E
E → (E)

	#	*	/	//	+	-	..	<	<=	>	>=	==	~=	(	)	i	$
#	*-*	>	>	>	>	>	>	>	>	>	>	>	>	<	*-*	<	>
*	<	>	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
/	<	>	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
//	<	>	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
+	<	<	<	<	>	>	>	>	>	>	>	>	>	<	>	<	>
-	<	<	<	<	>	>	>	>	>	>	>	>	>	<	>	<	>
..	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
<	<	<	<	<	<	<	<	*-*	*-*	*-*	*-*	*-*	*-*	<	>	<	>
<=	<	<	<	<	<	<	<	*-*	*-*	*-*	*-*	*-*	*-*	<	>	<	>
>	<	<	<	<	<	<	<	*-*	*-*	*-*	*-*	*-*	*-*	<	>	<	>
>=	<	<	<	<	<	<	<	*-*	*-*	*-*	*-*	*-*	*-*	<	>	<	>
==	<	<	<	<	<	<	<	*-*	*-*	*-*	*-*	*-*	*-*	<	>	<	>
~=	<	<	<	<	<	<	<	*-*	*-*	*-*	*-*	*-*	*-*	<	>	<	>
(	<	<	<	<	<	<	<	<	<	<	<	<	<	<	=	<	*-*
)	>	>	>	>	>	>	>	>	>	>	>	>	>	*-*	>	*-*	>
i	>	>	>	>	>	>	>	>	>	>	>	>	>	*-*	>	*-*	>
$	<	<	<	<	<	<	<	<	<	<	<	<	<	<	*-*	<	*-*

Obr.2

Pozn: *-* - žadné pravidlo

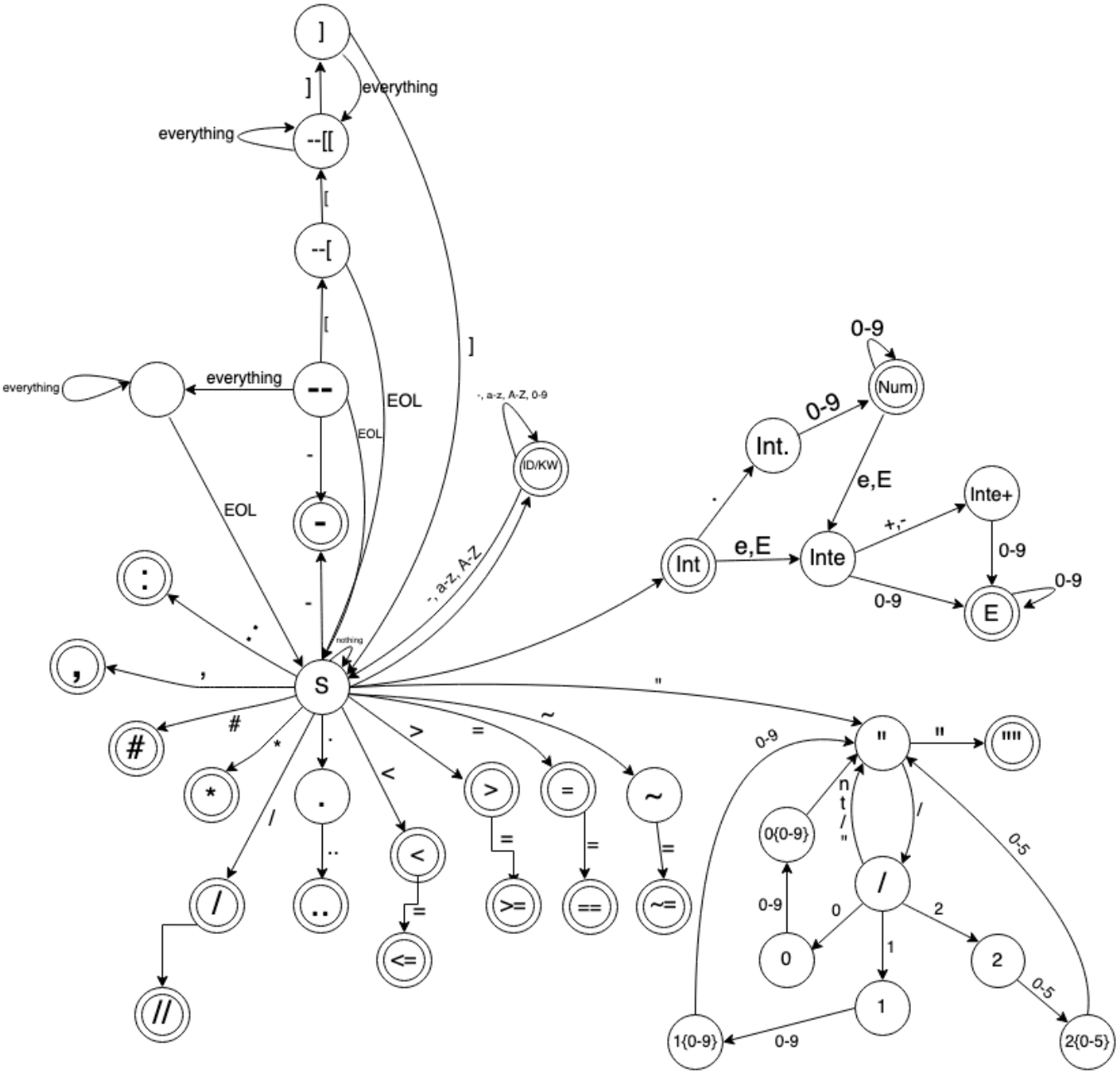
2.2.2 LL-gramatika

1. <prog> → <prolog> <func> eof
2. <prolog> → require “ifj21”
3. <func> → ε
4. <func> → global id : function (<list_of_datatypes>) : <list_of_return_datatypes>
5. <func> → function id (<list_of_parameters>) : <list_of_return_datatypes> <list_of_statements> end
6. <func> → id (<list_of_parameters>)
7. <list_of_return_datatypes> → datatype <more_datatypes>
8. <list_of_datatypes> → ε
9. <list_of_datatypes> → datatype <more_datatypes>
10. <more_datatypes> → ε
11. <more_datatypes> → , datatype <more_datatypes>
12. <list_of_parameters> → ε
13. <list_of_parameters> → <parameter> <more_parameters>
14. <more_parameters> → ε
15. <more_parameters> → , <parameter>
16. <parameter> → id : datatype
17. <list_of_statements> → ε
18. <list_of_statements> → <statement> <list_of_statements>
19. <statement> → local id : datatype = <expression_or_func_call>
20. <statement> → id <more_ids> = <expression_or_func_call>
21. <statement> → if expression then <list_of_statements> else <list_of_statements> end
22. <statement> → while expression do <list_of_statements> end
23. <statement> → id (<list_of_parameters>)
24. <statement> → return <list_of_expressions>
25. <more_ids> → ε
26. <more_ids> → , id <more_ids>
27. <list_of_expressions> → ε
28. <list_of_expressions> → expression <more_expressions>
29. <more_expressions> → ε
30. <more_expressions> → , expression <more_expressions>
31. <expression_or_func_call> → <list_of_expressions>
32. <expression_or_func_call> → id (<list_of_parameters>)

2.2.1 LL - tabulka

	require	global	function	id	datatype	local	if	while	return	,
<prog>	1									
<prolog>	2									
<func>		4	5	6						
<list_of_return_datatypes>					7					
<more_datatypes>										11
<list_of_parameters>				13						
<more_parameters>										15
<parameter>				16						
<list_of_statements>				18		18	18	18	18	
<statement>				21		19	21	22	24	
<more_ids>										26
<list_of_expressions>										
<more_expressions>										
<expression_or_func_call>			32							

FSM pro lexikální analýzu



Obr.1

2.3. Sémantický analyzátor

Sémantický analyzátor je implementován v modulu ***parser.c***. stejně jako syntaktický. Udělali jsme to tak, protože pří naší struktuře bylo to tím nejlepším řešením. Zjistili jsme, že musíme tu tabulku trochu zlepšit a udělat strukturu ***hash-table scope***(Jednosměrně vázaný seznám tabulek s rozptýlenými položkami), abychom mohli realizovat «viditelnost» uvnitř té analýzy. Každá další položka tohoto seznamu obsahuje tabulku symbolů "urovní výš" (hashTableList->first->symtable reprezentuje tabulku symbolů pro "nejvnořenější" uroveň) .Takovéto řešení umožnilo nám provádět kontroly nedefinovaných proměnných, kompatibilitu datových typů pří definicích/voláních funkcí atd.

Kontroly datových typů pří aritmetických/porovnávacích operacích provádíme pomocí dat, která jsou uložená v uzlech

Všechna chybová hlášení jsou implementována pomoci modulu ***error.c***

3. Týmová spolupráce

Týmová spolupráce tentokrát se stala naším hlavním problémem. Prvních pár týdnů všechno bylo super, dělali jsme schůzky atd. , ale potom jsme zjistili že jeden člen tymu neděla vůbec nic, kromě toho, že chodí na schůzky a občas se ptá, jak to s projektem. Takže dělali jsme ten překladač úplně ve dvou, proto nám chyběl čas na implementaci toho generátory cílového kódu.
Na komunikovaní používali jsme aplikaci Discord, tamto jsme udělali svůj kanál a všechno, co se tykalo projektu bylo probráno tam.
Jako verzovací systém používali jsme GitHub.

3.1 Rozdělení práce

Mikhailov Kirill – Implementace tabulky symbolů, Implementace hash-table-scope, Implementace syntaxického analyzátoru, Implementace sémantického analyzátoru,
Implementace lexikálního analyzátoru, dokumentace, testovaní

Naumenko Maksim – Implementace AST-stromu, Implementace DynamicString, Implementace tokenStack, Implementace syntaxického analyzátoru, Implementace sémantického analyzátoru,
Implementace lexikálního analyzátoru, testovaní

Mazurava Maryia - -

4. Závěr

Zjistili jsme, jak funguji překladače a jejich jednotlivé části, jaké struktury jsou potřebné pro implementací překladače. Vyzkoušeli jsme v praxi metodiky a teorie, které jsou přednášené rámci předmětů IFJ a IAL.
Bohužel, se nám nepodařila práce v týmu tentokrát, ale příště se získanými znalostmi budeme mnohem více připraveni na tak velký a komplexní projekt