FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Projektová dokumentace
Implementace překladače imperativního jazyka IFJ21

Tým 081, varianta 1

03.prosince 2021

Gazizov Zhasdauren	(xgaziz00)	19 %
Vladislav Mikheda	**(xmikhe00)**	27 %
Anvar Kilybayev	(xkilyb00)	27 %
Vladislav Khrisanov	(xkhris00)	27 %

Obsah

1 Úvod

Cílem tohoto projektu bylo vytvoření překladače v jazyce C, který se postupně načítá zdrojový kód zapsaný ve IFJ21 a pak ho překládá do cílového jazyka IFJcode21(mezikódu).

2 Implementace

2.1 Lexikální analýza

Lexikální analýza je implementována jako konečný automat. Během lexikální analýzy jsou ve funkce `get_token` se postupně načítají jednotlivé znaky ze vstupního souboru, dokud se automat nenarazí na řetězec, který představuje identifikátor, řetězec, desetinný nebo celočíselný literál, operátor, podporovaný speciální znak, konec řádku nebo konec souboru. Tato data je předávána syntaktickou analýzou ve struktuře tToken, který obsahuje typ tokenu a Unii, ve které je podle typu tokenu uložen buď dynamický řetězec (pro literály řetězců a identifikátory) nebo hodnota celočíselného datového typu; nebo float pro číselné literály. Jak samotná struktura, tak všechny typy přijatých tokenu jsou definované v souboru `lexicalanalysis.h`

2.2 Sémantická a syntaktická analýza

Syntaktická analýza je srdcem programu. Ona požádá tokeny, odešle zpracované tokeny shromážděné v AST strom a pak do generatora kódu. Syntaktická analýza je postavena na gramatice LL1. Je v ní vnořený sémantický analyzátor, protože by mezi sebou měli velmi často spolupracovat. Úkolem syntaktického analyzátoru je identifikovat syntaktické chyby v programu. Například po `IF` nemůže chybět `else`. Sémantický analyzátor porovnává typy proměnných. Definovaná proměnná, deklarovaná funkce atd. Celý program je v souboru
`syntacticalanalyzer.c`.

2.2.1 Precedenční syntakticka analýza

Součastí syntaktické alalýzy je precedenční, která rozebírá výrazy pomocí gramatiky zdola nahoru.
Matematické výrazy jsou zpracovány voláním funkce `preced_expression`. Tato funkce pracuje se zásobníkem ve struktuře `t_stack`. Na základě tabulka rozhodne, kterou operaci provést pro zpracování výrazu. Operace sh je implementována ve funkci `preced_expression`. Tato funkce přidává stack zarážky a aktuální to-

ken, a načte další token pomocí makra `get_token`. Operace Red, implementovaná ve funkci `reduce_buy_rule`, používá pomocné vlastních pravidel.

$E \rightarrow i$	$E \rightarrow E + E$	$E \rightarrow E < E$
$E \rightarrow (E)$	$E \rightarrow E - E$	$E \rightarrow E > E$
I	$E \rightarrow E * E$	$E \rightarrow E <= E$
I	$E \rightarrow E \,/\, E$	$E \rightarrow E >= E$
I	$E \rightarrow E \,//\, E$	$E \rightarrow E \,..\, E$
$E \rightarrow E == E$	$E \rightarrow \# E$	I

2.3 Generování cílového kodu

Generator cíloveho kodu byl implementovan v modulu `codgen.c` .Úkolem tohoto modulu je vytvářret cílovy kod, tedy v našem případe kod IFJcode21.

3 Algoritmy a datové struktury

3.1 Tabulka symbolů

Tabulky symbolů jsou implementovány jako binární vyhledávací stromy. Každý uzel stromu obsahuje kromě identifikátoru a ukazatelů na jeho dva podstromy také data. V nich je uložen typ identifikátoru, informace, jestli byl identifikátor již definován, ukazatel na lokální tabulku symbolů funkce a počet parametrů funkce. V jednotlivých binárních stromech pak vyhledáváme za pomocí klíče, kterým je pro nás identifikátor. Funkce pro práci s tabulkou symbolů jsou implementovány v souboru `symboltable.c`

4 Práce v týmu

4.1 Způsob práce v týmu a komunikace

Na projektu jsme začali pracovat ihed kdýž ho zadání bylo pošleno do WIS.Práci jsme podělili rovnoměrně pro každého člena týmu. Některý úlohy potřebovali několik lidí a proto jsme dělali je ve dvojicich. Komunikace v týmu probíhala osobně prostřednictvím aplikace Discord.

4.2 Verzovací systém

Pro správu souborů jsme používali verzovací systém Github. Zdrojové kody jsme měli uložené na vzdálenem repozitáře Github. Díky Github se nám usnadnila spo-

lupráce s projektem.Staré a nové verze projektu jsme ukladali do něho aby každý z nás mohl se na ně podívat a vyjádřit se k němu.

4.3 Rozdělení práce

Práci jsme rovnoměrně rozdělili mezi členy týmu tak, aby každý z nás mohl dokončit svou část projektu.
Práci jsme rozdělili tak:
Gazizov Zhasdauren : dokumentace, tabulka symbolů, testování.
Vladislav Mikheda (Vedoucí) : vedení týmu, lexikální analýza,syntaktická analýza, semantická analýza,testování.
Anvar Kilybayev : precendeční syntaktická analýza, tabulka symbolů, testování.
Vladislav Khrisanov : generátor kodu,testování

5 Závěr

Tento projekt pro nás byl překvapením. Zpočátku jsme nevěděli, co máme dělat a jak rozdělit práci. Ale po několika dnech jsme to vyřešili a začali pracovat. Také tento projekt nás naučil pracovat velmi dobře v týmu, během práce jsme se naučili mnoho nového nejen v předmětech IFJ a IAL, ale také v programování.

6 LL – gramatika

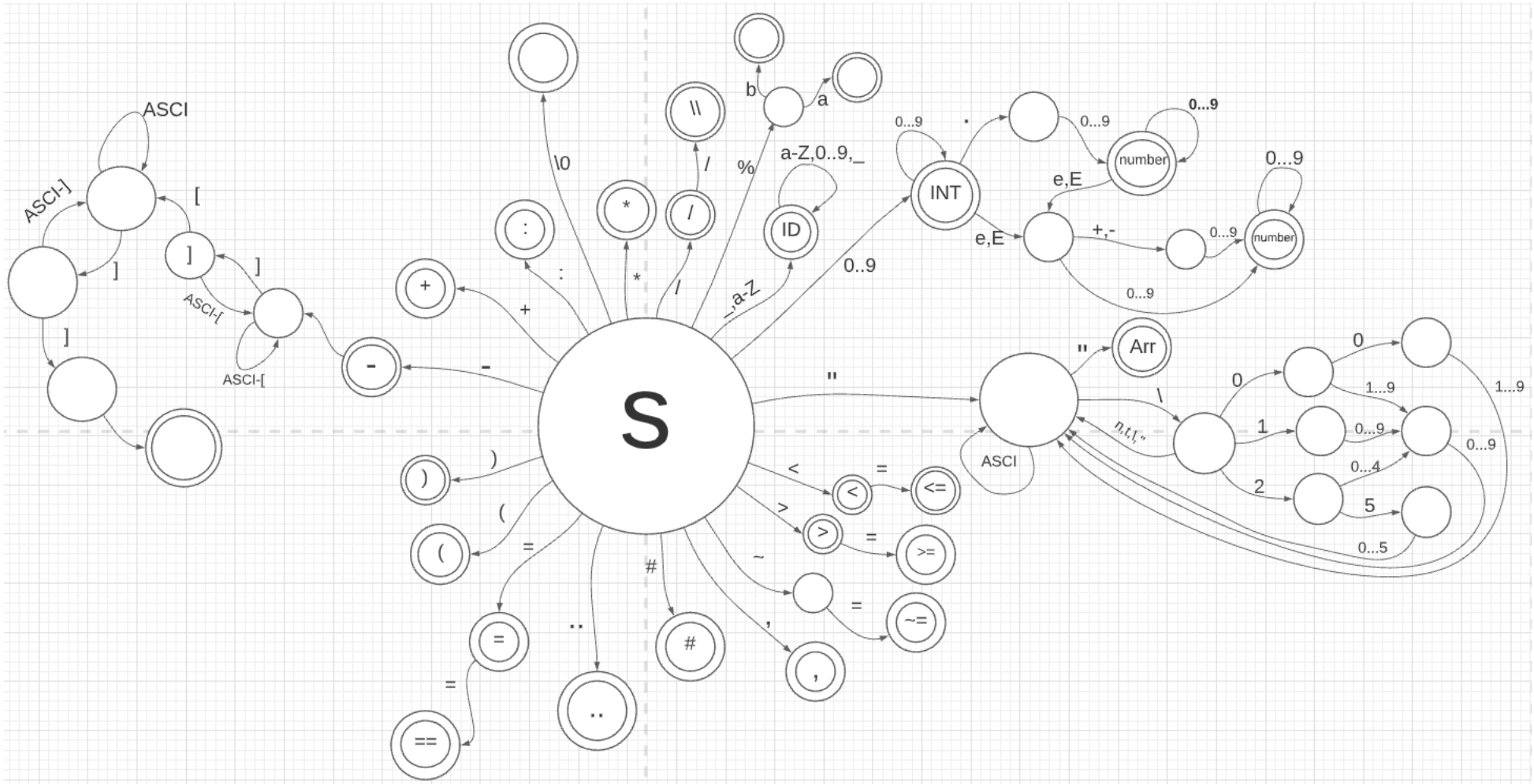
1. <prog> -> require «ifj21» <chunk>
2. <chunk> -> <function>
3. <chunk> -> EOF
4. <function> -> global fid : function (<global_params>) <return_types> <chunk>
5. <global_params> -> <global_par>
6. <global_params> -> ε
7. <global_par> -> id : <data_type> <next_global_par>
8. <global_par> -> <data_type> <next_global_par>
9. <next_global_par> -> , id : <data_type> <next_global_par>
10. <next_global_par> -> , <data_type> <next_global_par>
11. <next_global_par> -> ε
12. <function> -> function fid (<params>) <return_types> <statement> end <chunk>
13. <function> -> <function_call> <chunk>
14. <args> -> <expression> <new_expression>
15. <args> -> ε
16. <function_call> -> fid(<args>)
17. <function_call> -> write(<args>)
18. <function_call> -> reads()
19. <function_call> -> readi()
20. <function_call> -> readn()
21. <function_call> -> tointeger(<args>)
22. <function_call> -> substr(<args>)
23. <function_call> -> ord(<args>)
24. <function_call> -> chr(<args>)
25. <return_types> -> : <data_type> <next_data_type>
26. <return_types> -> ε
27. <next_data_type> -> , <data_type> <next_data_type>
28. <next_data_type> -> ε
29. <params> -> id : <data_type> <next_param>
30. <params> -> ε
31. <next_param> -> , id : <data_type> <next_param>
32. <next_param> -> ε
33. <data_type> -> integer
34. <data_type> -> string
35. <data_type> -> number
36. <value> -> <expression> <new_expression>
37. <value> -> <function_call>
38. <statement> -> local id : <data_type> = <value> <statement>
39. <statement> -> <function_call> <statement>
40. <statement> -> id <next_id> = <value> <statement>
41. <next_id> -> , id <next_id>
42. <next_id> -> ε
43. <new_expression> -> , <expression> <new_expression>
44. <new_expression> -> ε
45. <statement> -> if <expression> then
46. <statement> else <statement> end <statement>
47. <statement> -> while <expression> do <statement> end <statement>
48. <statement> -> return <expression> <statement>
49. <statement> -> ε

7 Precedenční tabulka

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

Obrázek 1: Precedenční tabulka

8 Celý automat



Obrázek 2: Celý automat