Faculty of Information Technology, Brno University of Technology

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DOCUMENTATION IFJ PROJECT 2021

Tým 008, varianta II

Martin Pokorný xpokor85 25%
Júlia Mazáková xmazak02 25%
Matej Matuška xmatus36 25%
Matěj Schäfer xschaf00 25%

Contents

1 TEAMWORK

We are used to work as a team because we worked together on previous team projects, specifically on IVS team project last semester.So we were familiar with our strenghts and weaknesses and we were also aware of possible problems that may occur. Our plan was to start as soon as possible and have everything done before the trial commit. It did not work out as we planned and the results were not as positive as we thought they would be. After the trial we started to work very intensively to finish this project and fix all problems found in the trial.

1.1 COMMUNICATION

Our main communication channel was Discord. We had several meetings where we planned how to proceed. If someone had problem or question it was solved/answered almost immediately. Also we were able to delegate some tasks if someone did not have time or had other issues that interfered with work on this project.

1.2 DIVISION OF TASKS

First we talked about what has to be done and than we assigned these tasks inidividually. More complicated parts of this project had two solvers to make things easier and to be sure it is correctly implemented. We used code reviews as a way to improve our code quality. Our task division looked something like this:

Login	**Tasks**
xpokor85	syntax analysis,semantic analysis, leadership
xmatus36	LL grammar, precedence analysis, testing
xmazak02	lexical analysis,symbol table, documentation
xschaf00	code generation, lexical analysis

2 IMPLEMENTATION

This project is divided into four main parts: Lexical analyzer, Syntax analyzer, Semantic Analyzer and Code Generator.

2.1 LEXICAL ANALYSIS

First step was to create scanner that would detect tokens with whom we would work in later parts of our compiler. We created final state machine diagram (see in attachments) and then we declared token structure and functions in scanner.h which we have implemented in scanner.c. We forgot some states in our FSM diagram at first so we had to add them later. The main function in our scanner get_next_token reads input from specific file or stdin and creates token with its type and its attribute. We had to distinguish between identificators and keywords so we checked every identificator to find out if it is keyword or not. If it gets valid token it correctly assigns type and attribute and function returns with 0 , however if it detects incorrect token , function returns with value 1 which results in LEXICAL ERROR. In case of string attribute we worked with dynamic strings which we have declared in header file dynamic_string.h and implemented in dynamic_string.c.

2.2 SYNTAX ANALYSIS

We started with LL rules. We had to remake them and fix them several times. Then we created LL table (see attachments).With the help of LL table it was quite easy to create syntax analysis. We came across some minor issues but after some debugging it worked like a charm.

2.3 SEMANTIC ANALYSIS

Semantic analysis is closely connected with symbol table so we had to make sure that these two parts are compatible first. It was mainly about adding items to symbol table and dealing with different scopes. The critical part was checking of declaration and data types of expressions or assignment to functions. Then we had to make a lot of debugging which affected overall code complexity. And there were a lot of exceptions for example with write operation call. Last but not least we incorporated code generation.

2.4 CODE GENERATION

Code generation's interface is in code_gen.h and implementation in code_gen.c. Generation was made for individual constructions of IFJ21. We focused on checking if there are any invalid operations with nill and undefined divisions by zero. There are functions for generating functions, frames, function calls, generating if else statements, while loops, assigning to parameters, type checking before operations and before assignments and so on. Than we tried to sync it with parser and focus mainly on debugging.

3 ALGORITHMS AND DATA STRUCTURES

3.1 SYMBOL TABLE

This important data structure is declared in symtable.h and implemented in symtable.c. We needed symbol table to store information about variables and functions such as definition,declaration, number of parameters, data types of parameters and so on. For data types we agreed that linked list would be the best solution so we can store multiple data types of parameters or return values . We have chosen variant in which we had to use common method for implementing symbol table and that would be hash table. Implementation of hash table was inspired by implementation of our member's solution of second homework from IJC [**?**] .As our hash function we used djb2[1] first reported by Dan Bernstein.And for the size of our hash table we used prime number 65535 recommended by [2]IBM.

3.2 STACK OF SYMBOL TABLES

As a solution for local variables and changing scope's visibility we decided to implement stack of symbol tables. in ST_stack.h is declaration of stack and its functions and in ST_stack.c are these functions implemented.So everytime new block occurs we push a new scope with local symbol table and after we are done using it we just pop this scope. The base of this stack is our global scope which is visible all the time.

3.3 STACK OF SYMBOLS

Stack needed for precedence analysis.Interface can be found in sym_stack.h and implementation of stack functions in sym_stack.c. We added some special functions for easier usage.

4 CONCLUSION

We learned a lot from this project. We managed to get it done on time but we should not underestimate time needed for completing this project. We have gotten better at bug detecting, debugging ,using different data structures and algorithms useful for our solution ,code reviewing , communication and coding in general. Also we learned that commenting is important especially when others are working with your code.

References

IJC IJC DU2 2021,`www.fit.vutbr.cz/study/courses/IJC/public/DU2.html`

[0] [1] Hash functions,`http://www.cse.yorku.ca/~oz/hash.html#:~:text=If%20you%20just%20want%20to,Also%20see%20tpop%20pp.`

[2] IBM`https://www.ibm.com/docs/en/iirfz/11.3.0?topic=analysis-considerations-sizing-hash-tables`

5 ATTACHMENTS

5.1 FSM OF LEXICAL ANALYZER

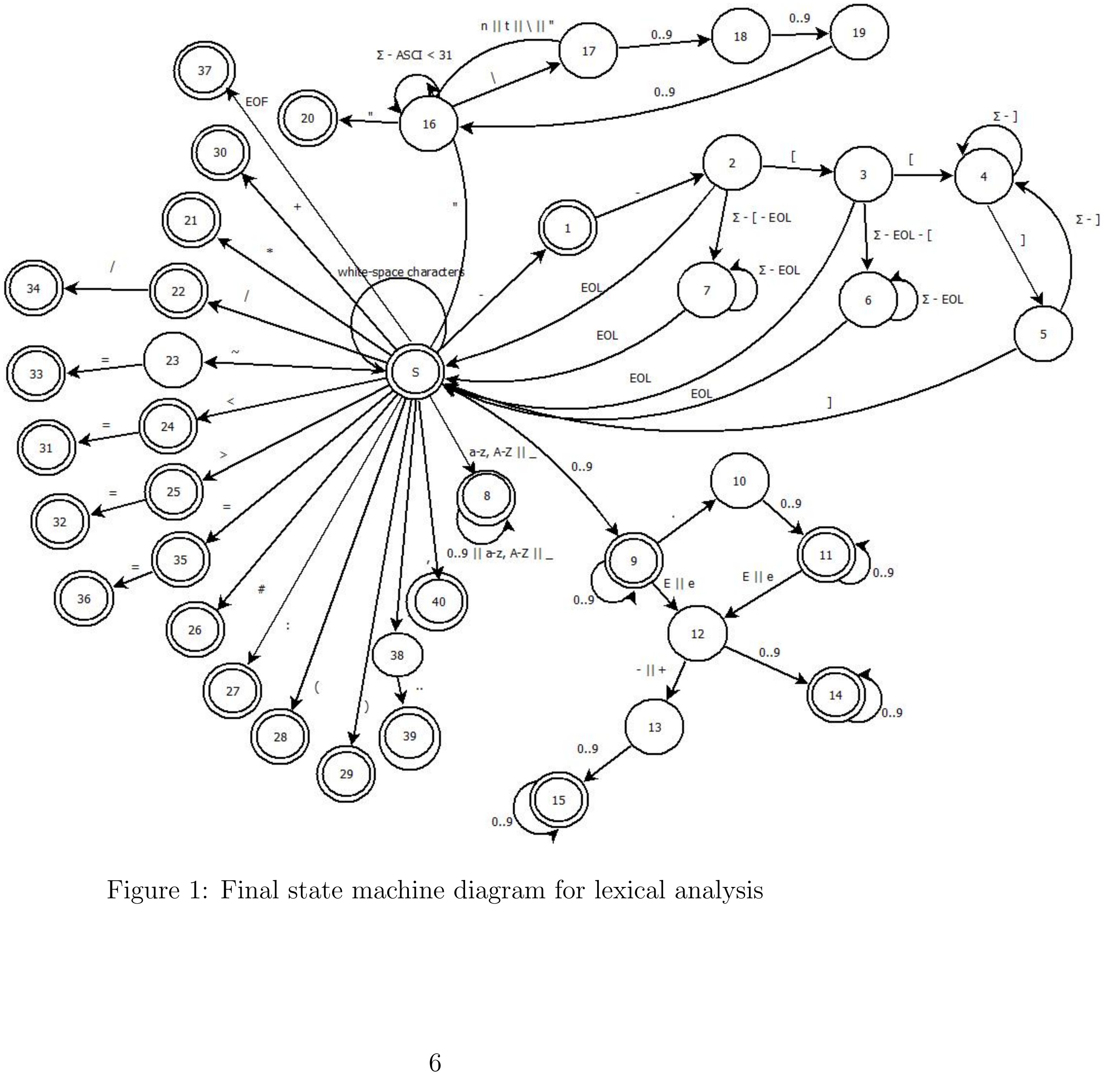
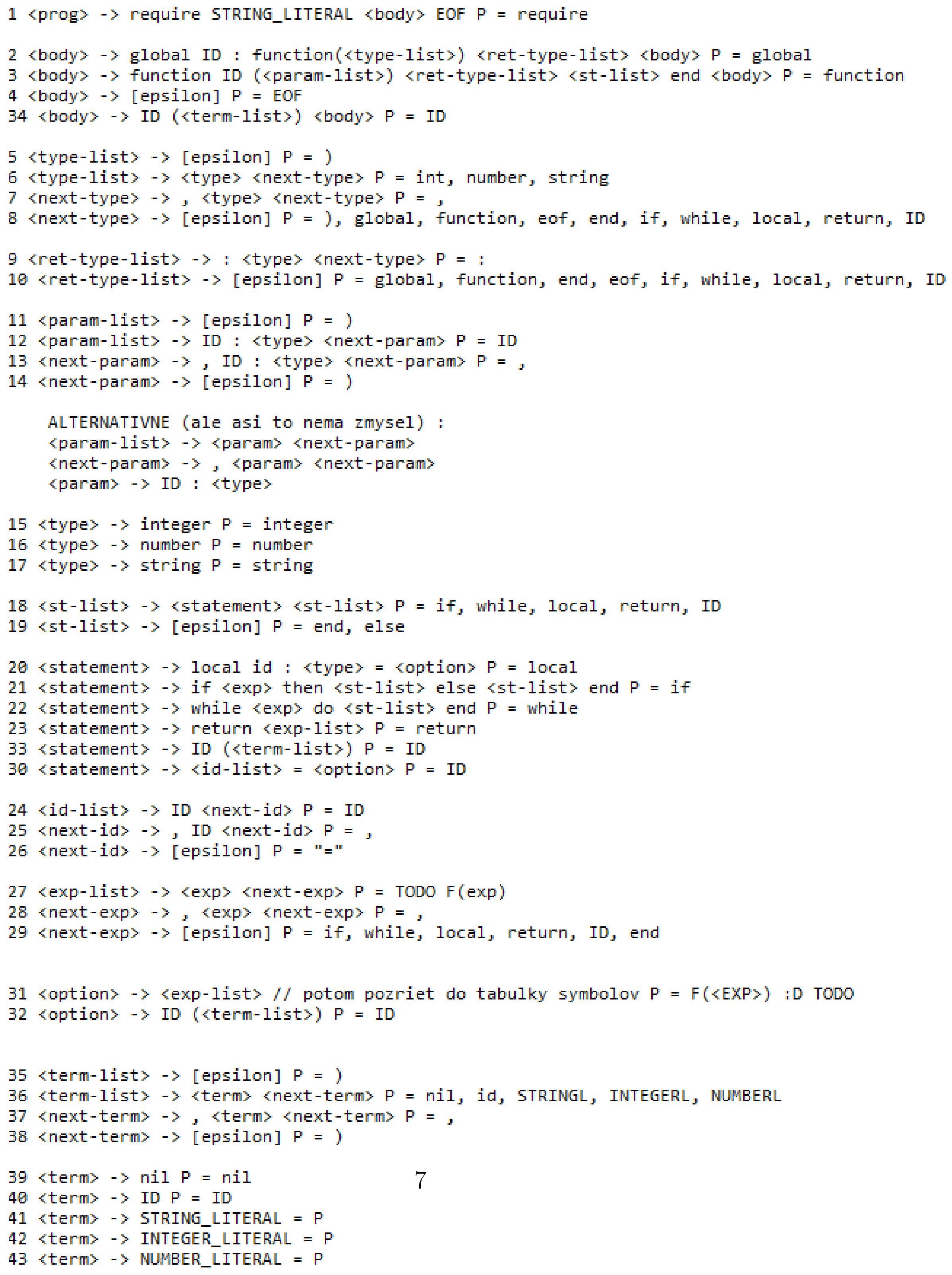


Figure 1: Final state machine diagram for lexical analysis

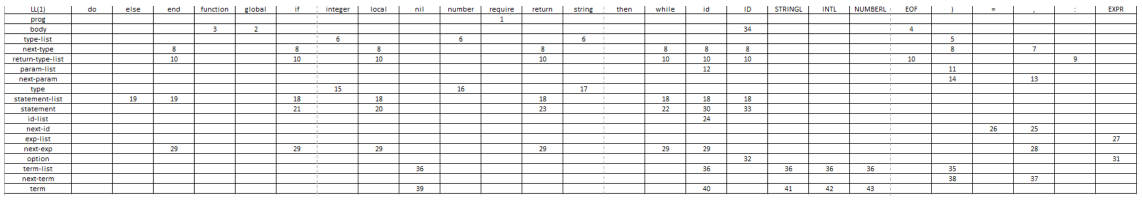
5.2 LL GRAMMAR



```
1 <prog> -> require STRING_LITERAL <body> EOF P = require

2 <body> -> global ID : function(<type-list>) <ret-type-list> <body> P = global
3 <body> -> function ID (<param-list>) <ret-type-list> <st-list> end <body> P = function
4 <body> -> [epsilon] P = EOF
34 <body> -> ID (<term-list>) <body> P = ID

5 <type-list> -> [epsilon] P = )
6 <type-list> -> <type> <next-type> P = int, number, string
7 <next-type> -> , <type> <next-type> P = ,
8 <next-type> -> [epsilon] P = ), global, function, eof, end, if, while, local, return, ID

9 <ret-type-list> -> : <type> <next-type> P = :
10 <ret-type-list> -> [epsilon] P = global, function, end, eof, if, while, local, return, ID

11 <param-list> -> [epsilon] P = )
12 <param-list> -> ID : <type> <next-param> P = ID
13 <next-param> -> , ID : <type> <next-param> P = ,
14 <next-param> -> [epsilon] P = )

    ALTERNATIVNE (ale asi to nema zmysel) :
    <param-list> -> <param> <next-param>
    <next-param> -> , <param> <next-param>
    <param> -> ID : <type>

15 <type> -> integer P = integer
16 <type> -> number P = number
17 <type> -> string P = string

18 <st-list> -> <statement> <st-list> P = if, while, local, return, ID
19 <st-list> -> [epsilon] P = end, else

20 <statement> -> local id : <type> = <option> P = local
21 <statement> -> if <exp> then <st-list> else <st-list> end P = if
22 <statement> -> while <exp> do <st-list> end P = while
23 <statement> -> return <exp-list> P = return
33 <statement> -> ID (<term-list>) P = ID
30 <statement> -> <id-list> = <option> P = ID

24 <id-list> -> ID <next-id> P = ID
25 <next-id> -> , ID <next-id> P = ,
26 <next-id> -> [epsilon] P = "="

27 <exp-list> -> <exp> <next-exp> P = TODO F(exp)
28 <next-exp> -> , <exp> <next-exp> P = ,
29 <next-exp> -> [epsilon] P = if, while, local, return, ID, end


31 <option> -> <exp-list> // potom pozriet do tabulky symbolov P = F(<EXP>) :D TODO
32 <option> -> ID (<term-list>) P = ID


35 <term-list> -> [epsilon] P = )
36 <term-list> -> <term> <next-term> P = nil, id, STRINGL, INTEGERL, NUMBERL
37 <next-term> -> , <term> <next-term> P = ,
38 <next-term> -> [epsilon] P = )

39 <term> -> nil P = nil
40 <term> -> ID P = ID
41 <term> -> STRING_LITERAL = P
42 <term> -> INTEGER_LITERAL = P
43 <term> -> NUMBER_LITERAL = P
```

Figure 2: LL rules

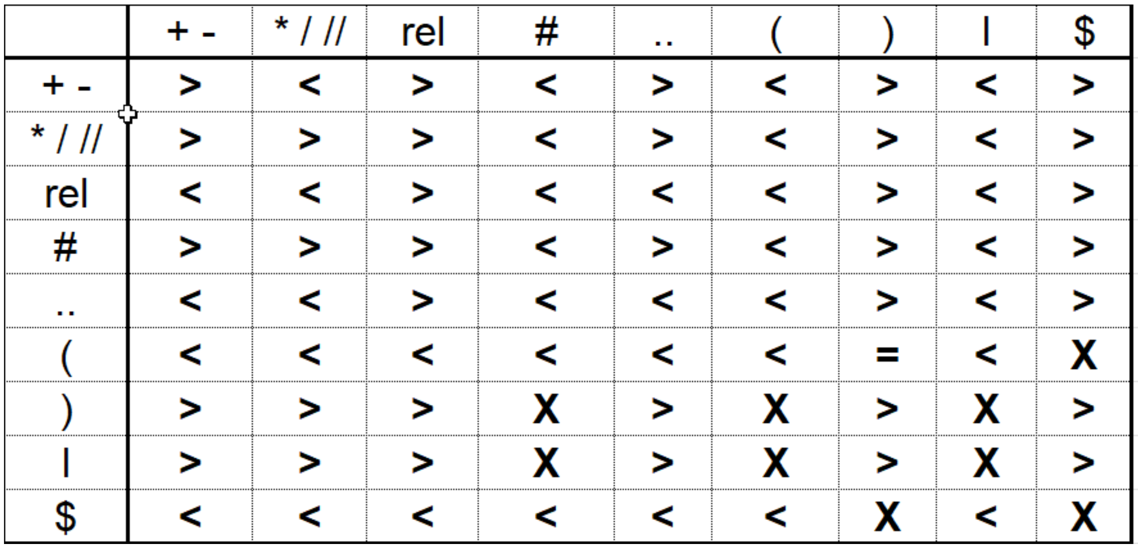
5.3 LL TABLE



LL(1)	do	else	end	function	global	if	integer	local	nil	number	require	return	string	then	while	id	ID	STRINGL	INTL	NUMBERL	EOF	)	=	,	:	EXPR
prog											1															
body				3	2												34				4					
type-list							6			6			6									5				
next-type			8			8		8				8			8	8	8					8		7		
return-type-list			10			10		10				10			10	10	10				10				9	
param-list																12						11				
next-param																						14		13		
type							15			16			17													
statement-list		19	19			18		18				18			18	18	18									
statement						21		20				23			22	30	33									
id-list																24										
next-id																							26	25		
exp-list																										27
next-exp			29			29		29				29			29	29								28		
option																	32									31
term-list									36							36		36	36	36		35				
next-term																						38		37		
term									39							40		41	42	43						

Figure 3: LL table

5.4 PRECEDENCE TABLE



	+ -	* / //	rel	#	..	(	)	I	$
+ -	>	<	>	<	>	<	>	<	>
* / //	>	>	>	<	>	<	>	<	>
rel	<	<	>	<	<	<	>	<	>
#	>	>	>	<	>	<	>	<	>
..	<	<	>	<	<	<	>	<	>
(	<	<	<	<	<	<	=	<	X
)	>	>	>	X	>	X	>	X	>
I	>	>	>	X	>	X	>	X	>
$	<	<	<	<	<	<	X	<	X

Figure 4: Precedence table