



# Project documentation

## Imperative Language Compiler Implementation IFJ22

Team xmazur08, TRP variant

December 7, 2022

<b>Maryia Mazurava</b>	<b>(xmazur08)</b>	25 %
Ekaterina Krupenko	(xkrupe00)	25 %
Evgeniya Taipova	(xtaipo00)	25 %
Alina Vinogradova	(xvinog00)	25 %

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Implementation</b>	<b>2</b>
2.1	Lexical analysis . . . . .	2
2.2	Syntactic analysis . . . . .	2
2.3	Semantic analysis . . . . .	2
2.4	Expression processing using precedence syntactic analysis . . . . .	2
2.5	Code generation . . . . .	2
2.5.1	Variable generation . . . . .	2
2.5.2	Function generation . . . . .	3
2.5.3	Generation of built-in functions . . . . .	3
<b>3</b>	<b>Special algorithms and data structures</b>	<b>3</b>
3.1	Hash table for table of symbols . . . . .	3
3.2	Dynamic strings . . . . .	3
<b>4</b>	<b>Teamwork</b>	<b>3</b>
4.1	Communication and code sharing . . . . .	3
4.2	Division of work in the team between members . . . . .	3
<b>5</b>	<b>Conclusion</b>	<b>4</b>
<b>A</b>	<b>Finite state machine diagram for a lexical analyzer</b>	<b>5</b>
<b>B</b>	<b>LL – grammar</b>	<b>6</b>
<b>C</b>	<b>LL – table</b>	<b>7</b>
<b>D</b>	<b>Precedence table</b>	<b>7</b>

# 1 Introduction

The goal of this project was to create a C program that translates the IFJ22 source code based on the PHP programming language into the IFJcode22 target language.

## 2 Implementation

This chapter presents the implementation of several parts that make up the program. These parts work together to make a compiler.

### 2.1 Lexical analysis

The first part of the compiler is a scanner. It was implemented according to the created diagram of a finite state machine (diagram 1). The scanner consists of `scanner.c` and `scanner.h` files. The main function of this part is `get_next_token(&token)`, which reads symbol by symbol from the standard input with the function `getc()`. The state machine is implemented in this function as a single switch, where each case is one state of the state machine. All states are defined in the header with a `#define`. If an invalid sign is read or the function ends in a non-final state, the program will return lexical error 1. Also, to work with tokens, a structure `token_t` was created that records the type of token and its attributes. The attribute is used for strings, numbers, identifiers, keywords.

### 2.2 Syntactic analysis

Syntactic analysis is implemented in files `parser.c`, `parser.h`. The parser is the main part of our project, as it is responsible for the flow of the compiler. It parses the incoming file, accepts tokens from the scanner as input, and runs through it recursively. According to the grammar table, we check whether the token is correct, if not, syntactic error is returned.

### 2.3 Semantic analysis

### 2.4 Expression processing using precedence syntactic analysis

To parse expressions symbol stack is used. It is implemented in files `symstack.c` and `symstack.h`. With the help of a function `get_next_token` from the scanner, we get tokens until there is `{` or `;`, which for us will indicate the end of the expression, depending on the function call. Using the functions implemented in files `expressions.c` and `expressions.h`, we write the received tokens to the stack and applying the precedence rule, we reduce the expression.

### 2.5 Code generation

Code generation works at the same time as syntactic and semantic analysis. With the help of functions from `generator.c` and `generator.h` files, the beginning of the function with the declaration of the name is called in the parser, the function arguments are generated, which will be written as variables with a unique name for each, ordinary variables are also generated inside the function and the function closes, it also works with `if` and `while` statements as well as with built-in functions of the IFJ22 language such as `write()`, `readi()`.

#### 2.5.1 Variable generation

Variable functions - arguments are declared immediately after the declaration of the beginning of the file and push frame. Each argument has its own unique index in each function, the first argument starts at index 0. All arguments are written to the Temporary Frame for later use in the following functions.

### 2.5.2 Function generation

The beginning of the function is generated, the frame is pushed. Function arguments are declared, possible operations with function arguments or local variables inside the function are generated. When we hit `}` the function is loaded.

### 2.5.3 Generation of built-in functions

Built-in functions include `readi`, `readf`, `reads`, `write`, `strlen`, `substr`, `ord`, `chr`. The beginning of the function is generated - exception `write`, `strlen` which are instructions for the IFJ-code22 language and do not require function declaration, function arguments are declared - exception `readi`, `readf`, `reads` for which it is not required to declare arguments. The function code already written in the `generator.c` file is called, and at the closing of the file, the symbol `}` declares the closing of the function, `pop` frame and return.

## 3 Special algorithms and data structures

### 3.1 Hash table for table of symbols

Hash table is implemented in `syntable.c`, `syntable.h` files. It is used to save information about global and local variables that are written in the code, as well as to save identifiers, functions in the symbol table. Depending on the context, we write to a local or global table. So, for example, if we are inside function definition or inside `if` or `while` statement, we write to the local symbol table, otherwise to the global one.

### 3.2 Dynamic strings

To work with dynamic strings, functions from files `str.c` and `str.h` were used. These files were taken from `jednoduchý interpret` template. Thanks to these functions, we can initialize a new string, add a new character there, compare strings, and also free them.

## 4 Teamwork

### 4.1 Communication and code sharing

We divided the parts of the project between team members. Each team member did their part alone or in pairs, and when needed, others helped if there was any problem.

The communication of the team members took place with the help of discord, as well as through personal meetings, where further plans for the project were discussed and problems and questions were resolved.

To work with the code, a repository was created in Github and also used live share in Visual Studio Code.

### 4.2 Division of work in the team between members

We divided the work in the project equally, so everyone gets 25%. The table 1 summarizes the division of work in the team between individual members.

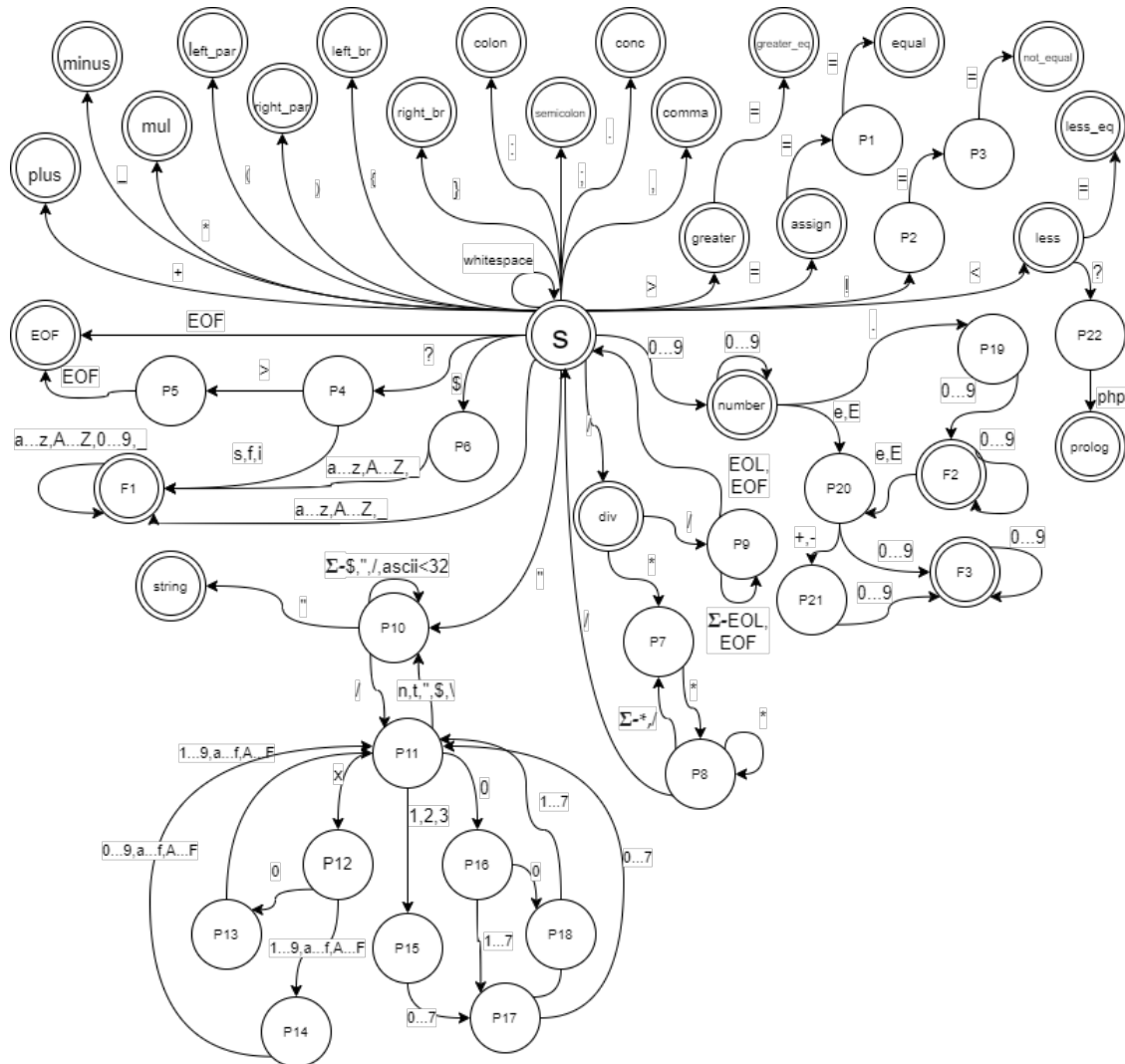
Team member	Assigned work
<b>Maryia Mazurava</b>	team management, syntactic analysis, semantic analysis, LL-grammar, LL-table, precedence analysis, error handler
Ekaterina Krupenko	code generator
Evgeniya Taipova	scanner, documentation, presentation, finite state machine diagram
Alina Vinogradova	syntactic analysis, semantic analysis, LL-grammar, LL-table, precedence analysis, project structure, testing

Table 1: Division of work in the team between members

## 5 Conclusion

Thanks to the lectures on subjects IFJ and IAL, we gained enough knowledge about how the compiler works and used them to implement the basic work of interpreters and translators. While writing the code, we encountered many difficulties, such as implementing dynamic strings in the scanner, parsing from top to bottom using the recursive descent method in the parser, or indexing arguments and declaring variables in the code generator. By closely linking the parts of the project for which different team members were responsible, we learned to work by listening to each other's problems, trying to help and look at the problem from different angles.

## A Finite state machine diagram for a lexical analyzer



P1 STATE\_EQUAL  
 P2 STATE\_NOT\_EQUAL\_START  
 P3 STATE\_NOT\_EQUAL  
 P4 STATE\_QUESTION  
 P5 STATE\_EOF  
 P6 STATE\_DOLLAR  
 P7 STATE\_BLOCK\_COMMENTARY  
 P8 STATE\_BLOCK\_COMMENTARY\_LEAVE  
 P9 STATE\_COMMENTARY  
 P10 STATE\_STRING\_START  
 P11 STATE\_STRING\_SEQUENCE  
 P12 STATE\_STRING\_SEQUENCE\_X  
 P13 STATE\_STRING\_SEQUENCE\_X\_ZERO  
 P14 STATE\_STRING\_SEQUENCE\_X\_NOT\_ZERO  
 P15 STATE\_STRING\_SEQUENCE\_ONE\_THREE  
 P16 STATE\_STRING\_SEQUENCE\_ZERO  
 P17 STATE\_STRING\_SEQUENCE\_REST  
 P18 STATE\_STRING\_SEQUENCE\_ZERO\_ZERO  
 P19 STATE\_NUMBER\_POINT  
 P20 STATE\_NUMBER\_EXPONENT  
 P21 STATE\_NUMBER\_EXPONENT\_SIGN  
 P22 STATE\_PROLOG\_START

F1 STATE\_IDENTIFIER\_OR\_KEYWORD  
 F2 STATE\_NUMBER\_DOUBLE  
 F3 STATE\_NUMBER\_EXPONENT\_FINAL

Figure 1: Finite state machine diagram for a lexical analyzer

## B LL – grammar

1. `<program> -> <prolog> <list_of_statements>`
2. `<prolog> -> <?php declare(strict_types=1);`
3. `<list_of_statements> -> EOF`
4. `<list_of_statements> -> <statement> <list_of_statements>`
5. `<statement> -> <function_definition>`
6. `<statement> -> <variable_definition>`
7. `<statement> -> <function_call>`
8. `<statement> -> if ( <expression> ) { <list_of_statements> }  
    else { <list_of_statements> }`
9. `<statement> -> while ( <expression> ) { <list_of_statements> }`
10. `<statement> -> return <return_expressions>;`
11. `<statement> -> <expression>;`
12. `<return_expressions> -> <expression>`
13. `<return_expressions> -> <variable>`
14. `<function_definition> -> function ID( <list_of_parameters> ) :  
    <list_of_datatypes_ret> { <list_of_statements> }`
15. `<variable_definition> -> <variable> = <var_def_expr>`
16. `<var_def_expr> -> <function_call>`
17. `<var_def_expr> -> <expression>;`
18. `<function_call> -> ID( <list_of_call_parameters> );`
19. `<list_of_call_parameters> ->  $\epsilon$`
20. `<list_of_call_parameters> -> <call_parameter> <list_of_call_parameters_n>`
21. `<call_parameter> -> <variable>`
22. `<call_parameter> -> "string"`
23. `<list_of_call_parameters_n> -> , <call_parameter><list_of_call_parameters_n>`
24. `<list_of_call_parameters_n> ->  $\epsilon$`
25. `<list_of_parameters> ->  $\epsilon$`
26. `<list_of_parameters> -> <parameter> <list_of_parameters_n>`
27. `<parameter> -> <list_of_datatypes> <variable>`
28. `<list_of_datatypes> -> int`
29. `<list_of_datatypes> -> float`
30. `<list_of_datatypes> -> string`
31. `<list_of_datatypes> -> nil`
32. `<variable> -> $ID`
33. `<list_of_parameters_n> ->  $\epsilon$`
34. `<list_of_parameters_n> -> , <parameter> <list_of_parameters_n>`
35. `<list_of_datatypes_ret> -> void`
36. `<list_of_datatypes_ret> -> <list_of_datatypes>  
    *"string" is a pseudoterm`

Table 2: LL – grammar

## C LL – table

	<?php declare(strict_types=1);	EOF	if	while	function	id	int	bool	float	string	nil	\$id	return	)	,	"str"	void	%	<expr>
<program>	1																		
<prolog>	2																		
<list_of_statements>		3	4	4	4	4						4	4						
<statement>			8	9	5	7						6	10						
<return_expressions>												12							11
<function_definition>					13														
<variable_definition>												14							
<function_call>						17													
<list_of_call_parameters>												19		18			19		
<call_parameter>												20					21		
<list_of_parameters_n>															23	22			
<list_of_parameters_n>								25	25	25	25	25			24				
<parameter>								26	26	26	26	26							
<list_of_parameters_n>															33	34			
<list_of_datatypes>								27	28	29	30	31							
<variable>													32						
<list_of_datatypes_ret>								36	36	36	36	36					35		
<var_def_expr>						15													16

Table 3: LL – table

## D Precedence table

	id	(	)	*	/	+	-	.	>	>=	!	<=	===	!==	\$
id	e	e	>	>	>	>	>	>	>	>	!	<	>	>	>
(	<	<	=	<	<	<	<	<	<	<	<	<	<	<	e
)	<	e	>	>	>	>	>	>	>	>	>	>	>	>	>
*	<	<	>	>	>	>	>	>	>	>	>	>	>	>	>
/	<	<	>	>	>	>	>	>	>	>	>	>	>	>	>
+	<	<	>	<	<	>	>	>	>	>	>	>	>	>	>
-	<	<	>	<	<	>	>	>	>	>	>	>	>	>	>
.	<	<	>	<	<	>	>	>	>	>	>	>	>	>	>
>	<	<	>	<	<	<	<	<	>	>	>	>	>	>	>
>=	<	<	>	<	<	<	<	<	>	>	>	>	>	>	>
<	<	<	>	<	<	<	<	<	>	>	>	>	>	>	>
<=	<	<	>	<	<	<	<	<	>	>	>	>	>	>	>
===	<	<	>	<	<	<	<	<	<	<	<	<	<	<	>
!==	<	<	>	<	<	<	<	<	<	<	<	<	<	<	>
\$	<	<	e	<	<	<	<	<	<	<	<	<	<	<	e

Table 4: Precedence table