BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Project documentation

Imperative Language Compiler Implementation IFJ22

Team xmazur08, TRP variant

December 8, 2022

Maryia Mazurava	**(xmazur08)**	25 %
Ekaterina Krupenko	(xkrupe00)	25 %
Evgeniya Taipova	(xtaipo00)	25 %
Alina Vinogradova	(xvinog00)	25 %

Contents

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()`. Finite 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

Semantic analysis is implemented in `parser.c` and `expressions.c` files. Semantic errors are found using data structures and functions for semantic controls. The symbol table, which is in `symtable.c`, `symtable.h` files, helps us to find the information about the identifiers and functions to provide semantic controls.

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. Using the functions implemented in files `expressions.c` and `expressions.h`, we write the received tokens to the stack and reduce the expression according to the precedence table, which defines what operator has higher priority. The implementation of an algorithm to collect and reduce expressions according to the rules is done. and successfully tested. During the processing of the expression we were expected to test our expression on types compatibility between operands, but unfortunately we weren't able to finish it.

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 Local 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 `symtable.c`, `symtable.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.

3.3 Error handler

The implementation of errors processing is in the `error.c` and `error.h` files. This handler helps us easily return errors during the compilation. It is mostly used in `parcer.c` and `expression.c`.

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.

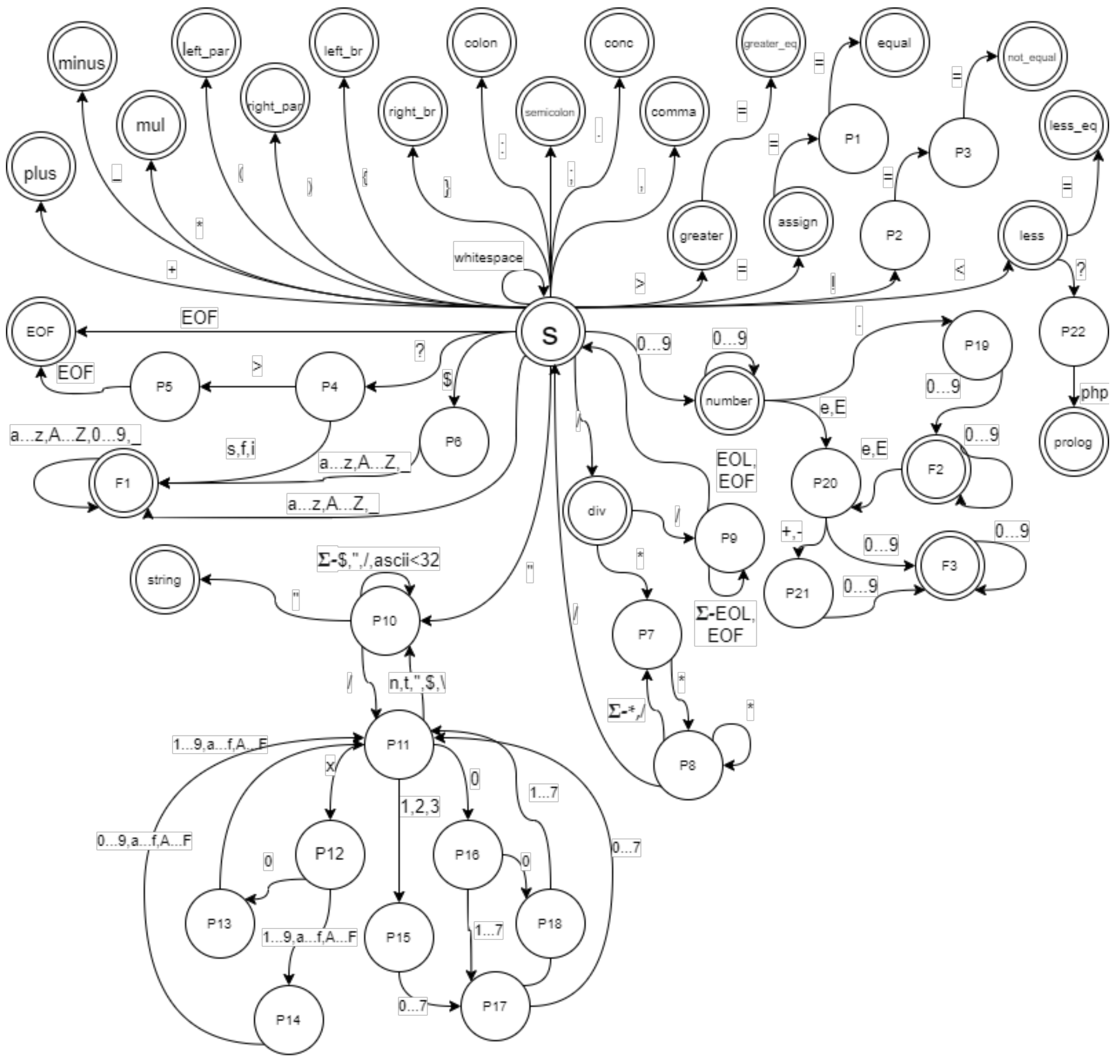
Team member	Assigned work
Maryia Mazurava	team management, syntactic analysis, semantic analysis, LL-grammar, LL-table, precedence analysis, error handler
Ekaterina Krupenko	code generator
Evgeniya Taipova	scanner, documentation, presentation, finite state machine diagram
Alina Vinogradova	syntactic 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_QUEST
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_SUQUENCE_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

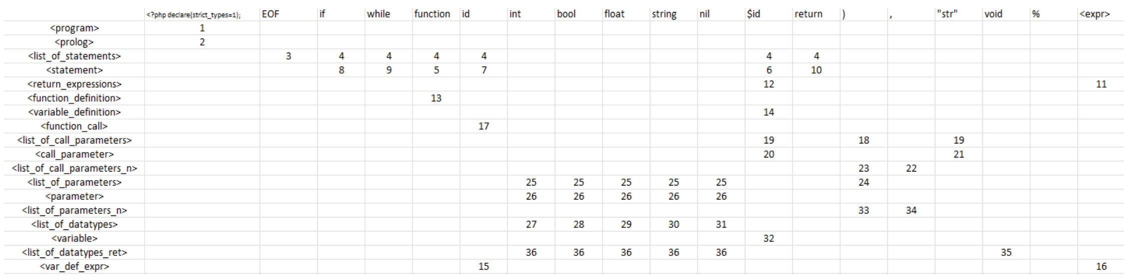
B LL – grammar

```
 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> -> ε
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> -> ε
25. <list_of_parameters> -> ε
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> -> ε
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
```

(some of the rules are not implemented in a final version)

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_call_parameters_n>														23	22				
<list_of_parameters>							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

1. `E->id`
2. `E->(E)`
3. `E->E*E`
4. `E->E/E`
5. `E->E+E`
6. `E->E-E`
7. `E->E.E`
8. `E->E>E`
9. `E->E>=E`
10. `E->E<E`
11. `E->E<=E`
12. `E->E===E`
13. `E->E!==E`