VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

IFJ PROJECT DOCUMENTATION

2022

TEAM XVECER29, BST VARIANT

Petr Večeřa	xvecer29	25%	leader
Matěj Šmida	xsmida06	25%	
René Češka	xceska06	25%	
Josef Unčovský	xuncov00	25%	

Contents

1 Introduction

This document was created as documentation for the IFJ project, describing implementation methods of all compiler components as well as problems during said implementation.

2 Compiler components

The task was to create a compiler of the **IFJ22** language, which is based on PHP 8, into the **IFJcode22** language. Since our team has chosen the BST variant of the task, table of symbols had to be implemented as a binary search tree. The compiler consists of the following components:

- Lexical analyzer
- Syntactic analyzer
- Semantic analyzer
- Code generator

The input code is processed by each component in the same order as written in the list above.

2.1 Lexical analyzer

The task of the lexical analyzer is to load individual lexical units (lexemes) and transform them into tokens, which represent the lexemes further on in other components of the compiler. The lexical analyzer component is implemented in the **scanner.c** module using a **finite state machine**. For situations in which an unrecognizable input token is provided, we use the **NOT_TOKEN** label. The parser component then contains a check for **NOT_TOKEN**, terminating the program upon encounter.

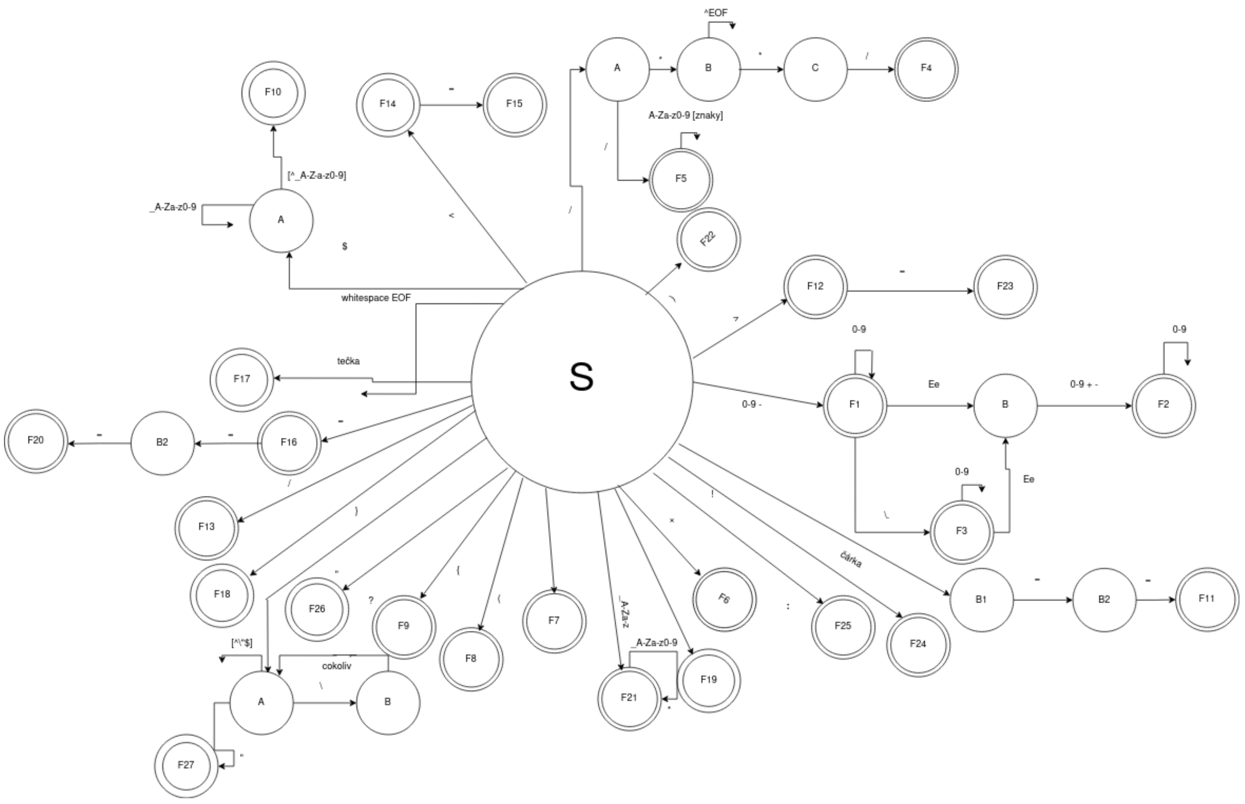

Figure 1: Finite state machine

2.2 Syntactic analyzer

The task of the syntactic analyzer is to fill the table of symbols (symtable). The component is implemented in the **parser.c** module and its input is a token from lexical analyzer. We used the top-down approach for implementation, precisely using **recursive descent**, which is based on LL grammar and LL table.

2.2.1 LL grammar

1. <PROLOG> → <?php declare(strict_types=1); <PROGRAM>
2. <PROGRAM> → <PROG_BODY> ?> EOF
3. <FUNC> → function ID (<PARAMS>) : <RETURN_EXPRESSION> {<BODY>}
4. <FUNC_CALL> → ID(<CALL_PARAMS>);
5. <CALL_PARAMS> → ϵ
6. <CALL_PARAMS> → $ID <CALL_PARAMS_N>
7. <CALL_PARAMS_N> → ϵ
8. <CALL_PARAMS_N> → ,$ID <CALL_PARAMS_N>
9. <PARAMS> → ϵ
10. <PARAMS> → $ID <PARAMS_N>
11. <PARAMS_N> → ϵ
12. <PARAMS_N> → , $ID <PARAMS_N>
13. <PROG_BODY> → <FUNC> <PROG_BODY>
14. <PROG_BODY> → ϵ
15. <PROG_BODY> → <EXPRESSION>; <PROG_BODY>
16. <PROG_BODY> → ; <PROG_BODY>
17. <PROG_BODY> → <CONSTRUCT> <PROG_BODY>
18. <ASSIGNMENT> → $ID = <EXPRESSION>;
19. <ASSIGNMENT> → $ID = <FUNC_CALL>;
20. <EXPRESSION> → <RETURN>
21. <CONSTRUCT> → <ASSIGNMENT>
22. <CONSTRUCT> → <WHILE_LOOP> <CONSTRUCT>

23. <CONSTRUCT> → if (<EXPRESSION>) {<BODY>} else {<BODY>}

24. <CONSTRUCT> → while (<CONSTRUCT>) {<BODY>}

25. <RETURN> → return <RETURN_EXPRESSION>;

26. <RETURN_EXPRESSION> → ϵ

27. <RETURN_EXPRESSION> → <EXPRESSION>

28. <BODY> → ϵ

29. <BODY> → <EXPRESSION>; <BODY>

30. <BODY> → <CONSTRUCT> <BODY>

31. <BODY> → ; <BODY>

2.2.2 LL table

	;	ID	FUNCTION	:	$ID	EOF	,	IF	ELSE	WHILE	RETURN	(	)	=
<PROLOG>	16		15											
<PROGRAM>	16		15			2								
<FUNC>		4	4	4										
<FUNC_CALL>		5												
<CALL_PARAMS>					7		7							
<CALL_PARAMS_N>					9		9							
<PARAMS>					11									
<PARAMS_N>					13		13							
<BODY>								23	27	24				
<ASSIGNMENT>					17									17
<EXPRESSION>			19					20						
<CONSTRUCT>								23						
<PROG_BODY>								24		26				
<RETURN>											26	4	4	
<RETURN_EXPRESSION>					28									

Table 1: LL Table

2.3 Semantic analyzer

2.3.1 Expression parsing

Our team didn't use precedence table, opting for using a combination of postfix conversions, binary and ternary trees instead to parse expressions. An **arithmetic search tree** (AST) was created for communication between parser and code generator, to help with work parallelization. AST contains all information needed to generate code. The AST is used for parsing expressions as a binary tree. At other instances it is also used as a ternary tree. Logic for parsing expressions is located in the files **atree.c** and **stree.c**.

2.4 Code generator

Code generator is implemented in the **generator.c** module. Its task is to create the final code, in our case **IFJcode2022** code. Code generator takes the tree structure passed from parser as input, nodes of which contain information about individual commands. The main component is node type **n_stList**, which acts as a list of individual commands in individual functions (command in the left node, another **n_stList** in the right node). Most of the nodes are generated recursively. The final generated code uses stack to pass on information. Generator also generates helping functions on the first rows of the source file written in IFJCODE, which are used in all individual generated commands. Since generating variables in loops proved to be complicated, all variable definitions are moved to a label which is then called at the beginning of a function. Functions (and what they return) are processed according to the **varT table**.

2.4.1 varT table

node type	left node	middle node	right node	cmpT	varT	name	valueInt	valueFloat	valueString	isNullable
n_stList	any		n_stList/null							
n_mul	any		any							
n_add	any		any							
n_sub	any		any							
n_div	any		any							
n_concat	any		any							
n_comp	any		any	set						
n_constant					set		set	set	set	
n_var						set				
n_assignVar			any			set				
n_defvar			any			set				
n_write	any									
n_read	n_argLcall									
n_if	n_comp/n_var/n_cons	any	any							
n_while	n_comp/n_var/n_cons	any								
n_funDef	n_argLfun				set	set				
n_argLfun			n_argLfun		set	set				set
n_funCall	n_argLcall					set				
n_argLcall	any		n_argLcall							
n_return			any		set					set

Table 2: varT Table

3 Division of labour & Teamwork

From the release of the project description, weekly meetings were held consistently until the end of the deadline. **Doodle** has been used for scheduling the meetings. We used **Git** as a version control system hosted on **GitLab**. A testing pipeline was created for every file entering the master branch, ensuring less errors overall. Because of the amount of various components, that had to be implemented, **Trello** was used to help us keep track of what needs to be done. Lastly, we used **Discord** for general communication.

3.1 Division of labour

- Petr Večeřa - Additional abstract data types, parser implementation
- Matěj Šmida - Scanner implementation, parser implementation
- René Češka - Code generator implementation, GitLab pipeline implementation, testing
- Josef Unčovský - LL grammar, LL table, documentation

4 Conclusion

The aim of the project was to put the concepts and techniques from IFJ and IAL lectures to practice and create a compiler. Studying theory, coming up with possible solutions and implementing them proved very beneficial for our skills as programmers and made it possible for us to fully grasp the concept of code compilation.

5 Sources, tools used for implementation, software used during implementation

- Alexandr Meduna, Roman Lukáš - Formal Languages And Compilers, Lecture slides
- Jan M. Honzík, Ivana Burgetová, Bohuslav Křena, Algorithms, Lecture slides
- GitLab, https://gitlab.com/
- Trello, https://trello.com/
- Doodle, https://doodle.com/
- Discord, https://discord.com/