Project Documentation

IFJ24 Compiler

Authors:

Denys Malytskyi (xmalytd00) 33%

Zhdanovich Iaroslav (xzhdan00) 33%

Zhukava Aryna (xzhuka01) 33%

Panchenko Artem (xpanch01) 1%

December 4, 2024

Contents

1 Creation and Implementation

1.1 Lexical Analysis

1.1.1 Structure and Workflow

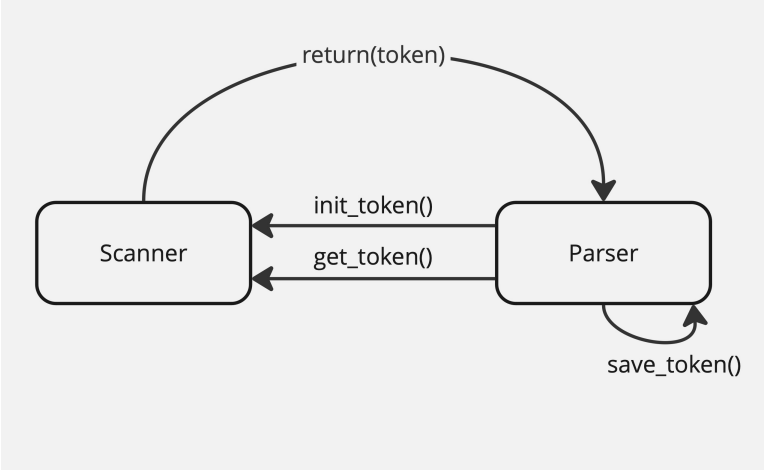
The Lexical Analyzer is implemented in the `scanner.c(h)` files. Its work begins immediately after the scanner is initialized by the parser and the `get_token` function is called.

The lexical analyzer's task is to read characters from `STDIN` using a `WHILE` loop. Reading continues until the lexical analyzer encounters the end-of-file (`EOF`). Depending on the character encountered, a function for reading a certain data type is called (e.g., reading numbers, identifiers, operators, etc.).

When the reading function encounters the `token_end` flag, indicating the end of a token, the scanner returns a ready-made token with its text and type to the parser. All information about the token is stored in the structure `token_t`, located in the `token.c(h)` files, which consist of:

- The type of the token
- The text of the token

To optimize the parser work the scanner provides tokens with very clear for understanding types `(TOKEN_ELSE, TOKEN_FLOAT, ect.)`.



1.1.2 FSM of Lexical Analyser

The following figure illustrates the finite state machine (FSM) of the lexical analyzer:

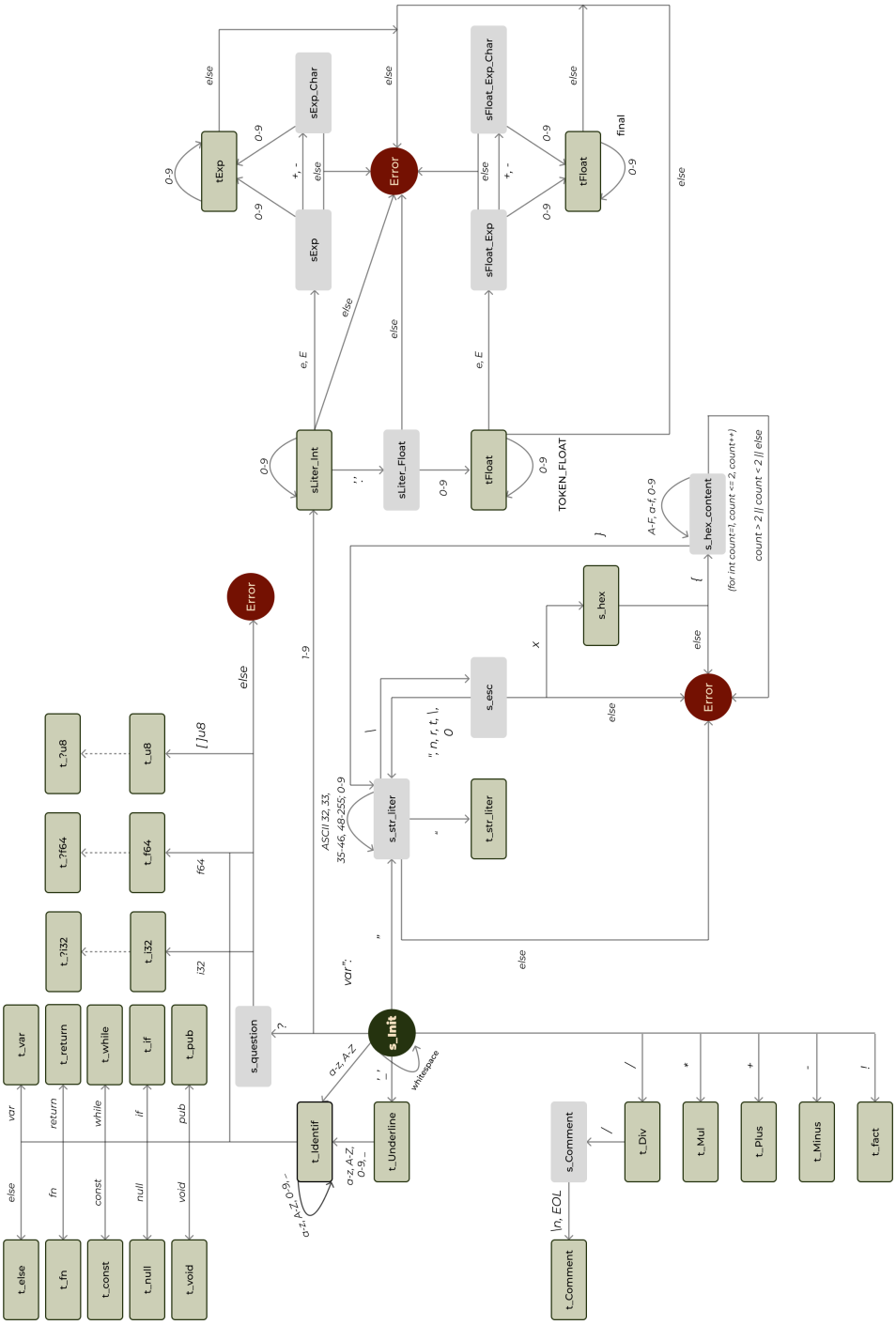


Figure 1: FSM of Lexical Analyser

1.2 Syntax Analysis

1.2.1 Structure and Workflow of Parser

The parser serves as the core module of the IFJ24 compiler, orchestrating interactions with all other modules and managing the compiler's overall functionality. It performs syntax analysis based on LL(1) grammar rules.

Using a recursive descent approach, the parser processes input from top to bottom. It receives tokens from the lexer, saves them in the `all_tokens` array, and validates them against the defined syntactic structure. If a current token's type deviates from the expected rule, a syntax error is triggered. The parser's implementation resides in the `topDown.c` and `topDown.h` files. Once initialized and prepared, it begins parsing the program by invoking the `program_rule()` function. This function handles the necessary "prologue" of each input program.

While the parser covers all aspects of the LL(1) grammar, it delegates the handling of expressions to a dedicated expression parser. When encountering an expression, it builds an expression (a singly linked list of tokens) and calls the `parse_expression()` function, which determines if the expression is valid or raises an error otherwise.

1.2.2 LL-Grammar

1. <Program> ::= <Prologue> <functions > <main_func> eof
2. <Prologue> ::= const ifj = @import("ifj24.zig") ;
3. <functions> ::= <function> <functions>
4. <functions> ::= eps
5. <function> ::= pub fn func_id <Param_list> <Return_type> <Block>
6. <main_func> ::= pub fn main () void <Block>
7. <Block> ::= { <Statements> }
8. <Param_list> ::= (<Params>)
9. <Params> ::= <Param> <Param*>
10. <Params > ::= eps
11. <Param*> ::= , <Param> <Param*>
12. <Param*> ::= eps
13. <Param> ::= param_id : <Base_type >
14. <Base_type> ::= i32
15. <Base_type> ::= f64
16. <Base_type> ::= []u8
17. <Base_type> ::= ?i32
18. <Base_type> ::= ?f64
19. <Base_type> ::= ?[]u8
20. <Return_type> ::= <Base_type>
21. <Return_type> ::= void
22. <Statements> ::= <Statement > <Statements>
23. <Statements> ::= <Conditionals > <Statements>
24. <Statements> ::= <while_statement> <Statements>
25. <Statements> ::= <Return> <Statements>
26. <Statements> ::= eps
27. <Statement> ::= <var_def> ;
28. <Statement > ::= <func_call> ;
29. <Statement > ::= <assignment> ;
30. <var_def> ::= <var_mode> id <var_type> = Expression ;
31. <var_mode> ::= const
32. <var_mode> ::= var
33. <var_type> ::= : :<Base_type >
34. <var_type> ::= eps
35. <Conditionals> ::= if (Expression) <Optional_null> <Block> <Optional_else>

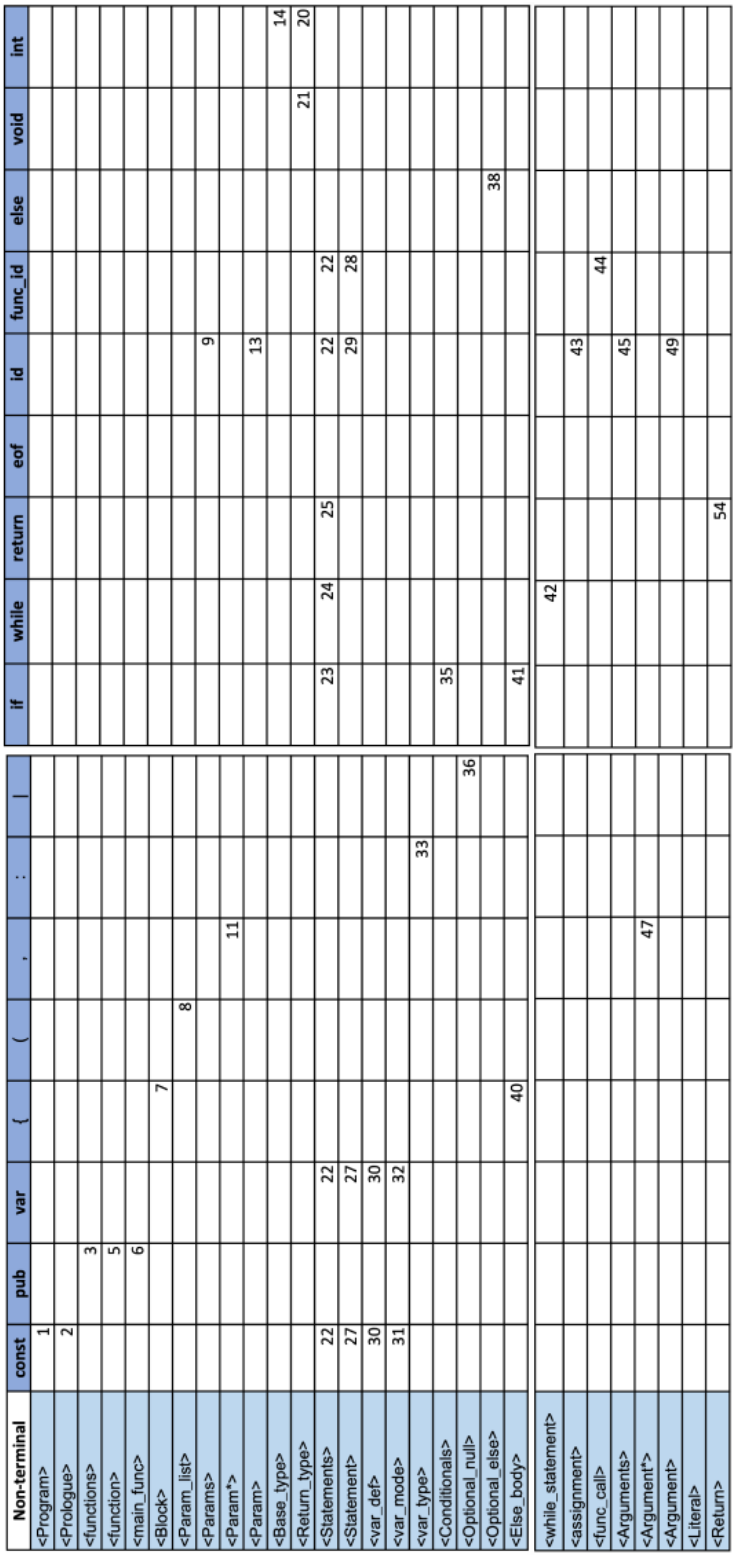
Figure 2: LL Grammar part 1

36. <Optional_null> ::= |id|
37. <Optional_null> ::= eps
38. <Optional_else > ::= else <Else_body>
39. <Optional_else > ::= eps
40. <Else_body> ::= <Block>
41. <Else_body> ::= if (Expression) <Optional_null> <Block> <Optional_else>
42. <while_statement> ::= while(Expression) <Optional_null> <Block>
43. <assignment> ::= id = Expression;
44. <func_call> ::= func_id(<Arguments>) ;
45. <Arguments> ::= <Argument> <Argument*>
46. <Arguments> ::= eps
47. <Argument*> ::= ,<Argument> <Argument*>
48. <Argument*> ::= eps
49. <Argument> ::= id
50. <Argument> ::= <Literal>

51. <Literal> ::= int_lit
52. <Literal> ::= float_lit
53. <Literal> ::= string_lit
54. <Return> ::= return Expression ;
55. <Return> ::= eps

Figure 3: LL Grammar part 2

1.2.3 LL-Table



Non-terminal	const	pub	var	{	(	,	:	|	if	while	return	eof	id	func_id	else	void	int
<Program>	1																	
<Prologue>	2																	
<functions>		3																
<function>		5																
<main_func>		6																
<Block>				7														
<Param_list>					8													
<Params>													9					
<Param*>						11												
<Param>													13					
<Base_type>																		14
<Return_type>																	21	20
<Statements>	22		22						23	24	25		22		22			
<Statement>	27		27										29		28			
<var_def>	30		30															
<var_mode>	31		32															
<var_type>							33											
<Conditionals>									35									
<Optional_null>								36										
<Optional_else>																38		
<Else_body>				40					41									
<while_statement>										42								
<assignment>													43					
<func_call>															44			
<Arguments>													45					
<Argument*>						47												
<Argument>													49					
<Literal>																		
<Return>											54							

Figure 4: LL Table part 1

Non-terminal	func_id	else	void	int	float	string	int?	float?	string?	int_lit	float_lit	string_lit	$
<Program>													
<Prologue>													
<functions>													4
<function>													
<main_func>													
<Block>													
<Param_list>													
<Params>													10
<Param*>													12
<Param>													
<Base_type>				14	15	16	17	18	19				
<Return_type>			21	20	20	20	20	20	20				
<Statements>	22												26
<Statement>	28												
<var_def>													
<var_mode>													
<var_type>													
<Conditionals>													34
<Optional_null>													37
<Optional_else>		38											39
<Else_body>													
<while_statement>													
<assignment>													
<func_call>	44												
<Arguments>										45	45	45	46
<Argument*>													48
<Argument>										50	50	50	
<Literal>										51	52	53	
<Return>													55

Figure 5: LL Table part 2

1.2.4 Structure and Workflow of Expression Parser

In bottom-up parsing, a precedence table guides the processing of expressions. The `parse_expression` function sequentially pushes tokens onto the stack, identifies their types, and uses the table to determine the relationship between the topmost unprocessed token in the stack and the current token in the input. If the table indicates that a rule is applicable, the reduce function is invoked to decide which rule to apply. If no rule matches the tokens at the top of the stack, the syntax analysis terminates with a syntax error.

1.2.5 Precedence Table

	(	)	+	-	*	/	==	!=
(	<	=	<	<	<	<	<	<
)	>	>	>	>	>	>	>	>
+	<	>	>	>	<	<	>	>
-	<	>	>	>	<	<	>	>
*	<	>	>	>	>	>	>	>
/	<	>	>	>	>	>	>	>
==	<	>	<	<	<	<	<	<
!=	<	>	<	<	<	<	<	<
<	<	>	<	<	>	>		
>	<	>	<	<	>	>		
<=	<	>	<	<	>	>		
>=	<	>	<	<	>	>		
i		>	>	>	>	>	>	>
$	<		<	<	<	<	<	<

<	>	<=	>=	i	$
<	<	<	<	<	
>	>	>	>		>
>	>	>	>	<	>
>	>	>	>	<	>
>	>	>	>	<	>
>	>	>	>	<	>
<	<	<	<	<	>
<	<	<	<	<	>
				<	>
				<	>
				<	>
				<	>
>	>	>	>		>
<	<	<	<	<	V

Figure 6: LL Grammar part 2

Precedence Rules.

Rule	Production
1.	E → i
2.	E → 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)

1.3 Semantic Analyses

Semantic analysis in the IFJ24 compiler is conducted alongside syntactic analysis, adhering to the language's semantic rules. During specific stages of parsing, checks are performed to verify that functions and variables are defined before use, that the data types and quantities of variables and expressions in assignments are consistent, and that function return values align with their declarations. If a semantic error is encountered, the program terminates with an appropriate return value.

1.3.1 Symbol Table

In the IFJ24 compiler, all scopes, variables, functions, and their arguments are managed using a symbol table. The symbol table is implemented in `symTable.c` and `symTable.h` and is based on a stack and an AVL tree. Each stack element includes attributes and methods for managing the scope.

For instance, when the LL parser begins parsing a function definition, it automatically captures the function's name, type, and other details, then pushes this information onto the stack. This allows the expression parser to recognize each scope and perform semantic actions, such as validating the function's return type.

Each stack element contains an AVL tree to store nodes representing variables, functions, and their parameters. Node searches within the AVL tree are performed based on the node's name. Nodes representing functions can include arguments organized in a linked list. Additionally, each node offers useful methods to facilitate semantic actions.

1.4 Code Generation

The final stage of our compiler is generating output code in `IFJcode24`. This process is managed by the code generator, which relies on pre-defined functions within the generator module. These functions are invoked at the appropriate points during parsing.

To prevent redeclaring variables inside loops, variable declarations are generated before the start of the `while` loop.

Special variables like `supa_giga_expr_res` are used to store the results of expressions, and `$tmp_<id>` is used for intermediate results, aiding in expression evaluation. Labels for `while` loops and conditionals are uniquely identified using specific identifiers (e.g., `while_case_<id>`, `while_end_case_<id>`).

2 Teamwork

2.1 Work Distribution

- **Common:**
  - Organization of the Git project

  Denys and Iaroslav did most of the work together:

- **Denys Malytskyi:**
  - Top-Down parser implementation
- **Iaroslav Zhdanovich:**
  - Bottom-Up parser implementation
- **Denys Malytskyi and Iaroslav Zhdanovich:**
  - Syntactic and Semantic analysis implementation:
    - Creating a symbol table
    - Creation of LL-table (+ rules)
    - Implementing an expression parser (+ rules)
    - Implementation of semantics in the expression parser
  - Code generation in the expression parser
  - Tests
- **Aryna Zhukava:**
  - Creation of tokens for the scanner
  - Dynamic string implementation
  - Scanner
  - Error output
  - Documentation
  - Presentation
- **Artem Panchenko:**
  - No impact

2.2 Communication Methods

Most often, we used "Telegram" for communication, and "Discord" for emergency calls. In addition, we kept all intermediate files and changes on our closed git hub. Thanks to teamwork, we all improved our skills with git. And we sometimes had a meetings at the university, where we analyzed what we had already done, went through the necessary theory and made plans for the next week.

Unfortunately, one person from our team did not participate in the writing project, so we are changing the point split to three people (instead of four).

3 References

C-Program to implement AVL tree - https://www.geeksforgeeks.org/c-program-to-implement-avl-tree/