
# Project documentation

## Implementace překladače imperativního jazyka IFJ24

Tým xpopov10, varianta vv-BVS

December 4, 2024


| | | |
|---|---|---|
| **Popov Albert** | **(xpopov10)** | 25 % |
| Krasovskyi Oleh | (xkrasoo00) | 25 % |
| Savin Ivan | (xsavini00) | 25 % |
| Turar Nurdaulet | (xturarn00) | 25 % |

# 1 Introduction

The goal of the project was to create a C program that reads source code written in the IFJ24 source language, which is a simplified subset of the imperative programming language Zig 0.13, and translates it into the target language IFJcode24 (intermediate code).

The program is a console application that reads the source program from standard input and generates the resulting intermediate code to the standard output or, in the event of an error, returns the corresponding error code.

# 2 Work in the team

## 2.1 How the team works

At the beginning of the project, we divided the task into 4 main blocks and acted according to the interaction of the codes. If someone's work seemed less important, that person helped others or wrote tests.

### 2.1.1 Version control system

We used the Git versioning system to manage the project files. As a remote repository we used GitHub.

Git allowed us to work on multiple tasks on a project at the same time in so-called branches. Most of the tasks were first prepared into a branch, and only after testing and approving the modifications by other team members did we incorporate them into the main development branch.

### 2.1.2 Communication

Communication was mainly in a group chat and in private messages. During the first half of the semester, several offline meetings were conducted.

Towards the end of the project, we met to demonstrate how our program blocks worked, in order to find errors and increase the understanding of the project to the other participants.

## 2.2 Distribution of work among team members

We divided the work on the project evenly with regard to its complexity and time demands. So everyone got a percentage rating of 25 %. The table summarizes the distribution of the team's work among the individual members.

| Team Member | Assigned Work |
|---|---|
| **Popov Albert** | team management, testing, syntactic analysis |
| Krasovskyi Oleg | semantic analysis, testing |
| Savin Ivan | lexical analysis, documentation, testing |
| Turar Nurdaulet | target code generation, work organization, testing, project structure |

Table 1: Distribute the work of the team among the individual members

# 3 Design and implementation

The project is composed of several parts implemented by us, which are presented in this chapter. It also shows how the different subparts work together.

## 3.1 Compiler structure

The compiler consists of the following modules:
**Lexical analyzer (lexer):** Converts source code to tokens
**Parser:** It analyzes tokens using syntactic and semantic rules and then builds an abstract syntax tree (AST).
**Code Generator:** Generates target code from AST.

## 3.2 Lexical Analysis

The first stage of the compiler is lexical analysis. Initially, we assumed that a separator symbol would exist between all tokens, allowing the source code to be divided into discrete blocks. Each block could then be easily converted into corresponding tokens. However, this approach was abandoned due to the need to implement numerous exceptions in certain cases, which made the lexer harder to extend and reduced code readability.

The new version of the lexical analyzer employs a deterministic finite automaton (DFA). The main function, `run_lexer()`, reads the input code and starts the automaton in its initial state. It identifies the first non-empty character and transitions the automaton to the corresponding state. After processing that state, the control returns to `run_lexer()`, which continues reading until the end of the file.

**Processing Identifiers**

For identifiers, allowed characters are read sequentially. Once the sequence is fully read, the lexer determines whether it matches a keyword or should be classified as an identifier. This approach allows keywords to be correctly identified and associated tokens to be created.

**Processing Numbers**

For numerical tokens, digits, decimal points, and the characters `e/E` are read in sequence. If the input forms a valid number, the lexer generates an `f64` or `i32` token based on the data provided.

**Finite State Machine Design**

The entire lexical analyzer is implemented as a deterministic finite automaton based on a predefined state diagram (included in the appendix). This diagram defines the states and transitions, ensuring the lexer is easier to maintain and extend when required.

By utilizing a structured DFA approach, the lexer achieves a balance between functionality and simplicity, making it a reliable component of the compiler pipeline.

## 3.3 Syntactic Analysis

The syntactic analyzer, or parser, ensures that the input source code adheres to the defined grammar of the programming language. It processes the sequence of tokens produced by the lexical analyzer and organizes them into an Abstract Syntax Tree (AST), which represents the hierarchical structure of the program.

**How it works**

**Input:** The parser takes a sequence of tokens from the lexical analyzer. Each token consists of its type and optionally its attributes (e.g., literal values or identifier names).

**Parsing Strategy:** The parser uses a recursive descent parsing approach. Each grammatical rule is implemented as a specific function that matches tokens and recursively processes nested structures.

**Abstract Syntax Tree (AST):** The parser constructs an AST for syntactically valid input. The AST represents program constructs such as expressions, statements, and function definitions in a hierarchical form. For example, an expression like `a + b * c` is represented as a tree with `+` as the root, `a` as the left child, and a subtree for `b * c` as the right child.

**Error Handling:** The parser reports syntax errors if a token does not match the expected grammar rule. For example, if a semicolon is missing, the parser will report an error and gracefully terminate with a message indicating the expected token and its position.

**Workflow**

**Initialization:** Parsing begins with the `parseProgram` function, which processes the overall program structure. It first validates the prologue and then processes all function definitions.

**Recursive Descent Parsing:** Each grammar rule is implemented as a function (e.g., `parseExpression`, `parseStatement`). These functions match tokens and recursively call other functions to process nested constructs.

**Token Matching:** The `match` function ensures the current token matches the expected type. If not, a syntax error is raised.

**AST Construction:** For valid input, the parser creates AST nodes using helper functions like `createASTNode` and `createBinaryASTNode`. These nodes capture the syntactic structure of constructs such as expressions, statements, and function definitions.

**Completion:** After processing all tokens, the parser finalizes the AST, which is then passed to the semantic analyzer for further validation.

**Example Workflow**

```
const ifj = @import("ifj24.zig");

pub fn main() void {
    var x: i32 = 5;
    x = x + 1;
    return;
}
```

The parser verifies the validity of the prologue: `const ifj = @import("ifj24.zig");`. It analyzes the function definition `pub fn main() void`. Inside `main`, it analyzes: - Variable declaration: `var x: i32 = 5`. - Assignment: `x = x + 1`. - Return statement: `return`.

The resulting AST has the following structure:

```
Program
 Prolog
 FunctionDef (main)
     Parameters: None
     ReturnType: void
     StatementList
         VarDeclaration (x: i32 = 5)
         Assignment (x = x + 1)
         Return
```

The AST is then passed to the semantic analyzer.

## 3.4 Semantic Analysis

Semantic analysis is performed alongside syntax analysis, using AST nodes as storage units for various useful data, such as:

- Compile-time values for variables, constants, literals, and expressions.
- Value types for variables, constants, literals, expressions, and statements, for type-checking purposes.

- Names to store variables and constants' identifiers.

Semantic analysis constructs and utilizes the *symbol table*, which is implemented as a "spaghetti stack" structure. Each item of the stack represents a *scope*.

A *scope* is a structure that contains:

- A Red-Black binary search tree (BST) for efficient symbol storage and lookup.
- A counter for unused variables within the scope.

Additionally, the symbol table structure includes a regular stack to facilitate freeing all allocated scopes when exiting a block or function.

## 3.5 Target code generation

### Structure

Target code, (`IFJcode24`), is constructed using the AST and symtable. The final structure of a target program is the following:

```
.IFJcode24
CALL main
JUMP end_program
LABEL main
    ... # main function's body
LABEL f1
    DEFVAR LF@x
    DEFVAR LF@out
    ... # f1 function's body
    PUSHS LF@out
LABEL end_program
```

### Algorithm

The main idea of the algorithm for generating the target code is the following.

1. Create labels in memory: `"main"` and `"end_program"`.
2. Output a call to label `"main"`.
3. Output a `JUMP` to `"end_program"` label.
4. For each node in the AST with a `Function` type:
   (a) Create and print a label representing the function's entry point.
   (b) For each statement within a function:
      i. Add appropriate instruction(s) to the function's instruction context.
      ii. If a new variable (including temporary) must be defined within the scope of the function, add it to the function's variables context.
   (c) Output every variable definition within the function's variables context.
   (d) Output every other instruction within the context.
5. Output `"end_program"` label.

### Development

When I began working on target generation, proper syntax and semantic analyses were not yet in place. As a result, the first sub-component I implemented was functionality for outputting instructions programmatically, which can be found in `instructions.h`. Following this, I started adding scaffolding to the `target_gen.h` component, creating functions with comments describing the intended behavior but without actual implementations.

After the AST was developed, the real work began. I implemented the outlined functions one by one, and during this process, I encountered two key challenges:

1. `DEFVAR` can be invoked only once per variable name within a scope. Redefining the same variable in a single scope causes an interpretation error. Since functions must operate within a single scope, this led to issues with variables defined inside loops in the AST. To resolve this, I grouped all `DEFVAR` instructions while traversing the AST and emitted them at the beginning of the function, before generating other instructions. This solution is implemented in `target_func_context.h`.

2. To prevent variable name collisions, I adopted a method to generate unique identifiers by appending a distinct suffix to each variable or label. A separate indexer is used for labels. New indexer is created for each function scope. This functionality is implemented in `id_indexer.h`.

3. When most of the functionality was ready, I've realized that we could've used polish notation to simplify the generation of arithmetic operations. Unfortunately, due to time constraints, I've decided to not implement it.

## 4 Conclusion

The project initially surprised us a bit with its scope and complexity. Over time, until we had enough knowledge about compiler development from the IFJ lectures, we started to tackle the project.

We had our team assembled very early on, and we had already agreed on the communication channels in advance, face-to-face meetings and the use of the versioning system, so we didn't have any and we worked very well together.

We started working on the project a bit late, so we didn't have time to spare, but in the end we got everything done. We worked on the different parts of the project mostly individually using knowledge from lectures or materials for IFJ and IAL courses.

During the development, we encountered minor problems related to ambiguities in the assignment, but these have been resolved through the project forum. The correctness of the solution was verified by automatic tests and test submissions, which allowed us to further fine-tune the project.

Overall, this project has given us a lot of knowledge about how compilers work, practically clarified the subject matter covered in the IFJ and IAL courses, and brought us experience with projects of this magnitude.

# A Finite state machine diagram specifying a lexical analyzer

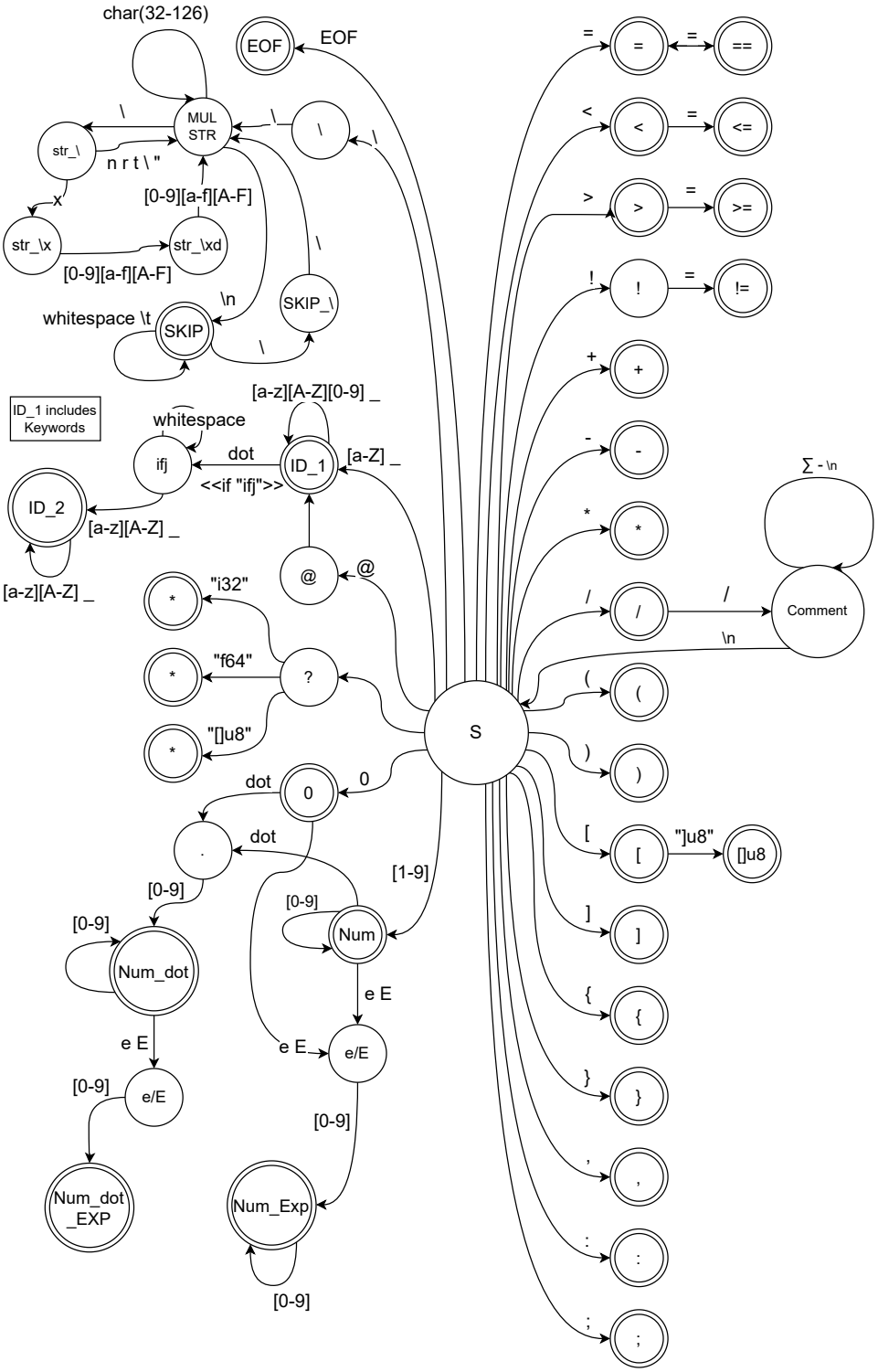


Figure 1: Finite state machine diagram specifying a lexical analyzer

# B grammar

```
1. [Program] ::= [Prolog] [FunctionDefList]

2. [Prolog] ::= 'const' 'ifj' '=' '@import' '(' '"ifj24.zig"' ')' ';'

3. [FunctionDefList] ::= [FunctionDef] [FunctionDefList]
4. [FunctionDefList] ::= ε

5. [FunctionDef] ::= 'pub' 'fn' [Identifier] '(' [ParamList] ')' [ReturnType] '{' [StatementList] '}'

6. [ParamList] ::= [Identifier] ':' [Type] [ParamListTail]
7. [ParamList] ::= ε

8. [ParamListTail] ::= ',' [Identifier] ':' [Type] [ParamListTail]
9. [ParamListTail] ::= ε

10. [ReturnType] ::= 'void'
11. [ReturnType] ::= [Type]

12. [StatementList] ::= [Statement] [StatementList]
13. [StatementList] ::= ε

14. [Statement] ::= 'const' [Identifier] [VarType] '=' [Expression] ';'
15. [Statement] ::= 'var' [Identifier] [VarType] '=' [Expression] ';'
16. [Statement] ::= [Identifier] '=' [Expression] ';'
17. [Statement] ::= 'if' '(' [Expression] ')' [NullableBinding] '{' [StatementList] '}' [OptionalElse]
18. [Statement] ::= 'while' '(' [Expression] ')' '{' [StatementList] '}'
19. [Statement] ::= 'return' [ReturnValue] ';'

20. [VarType] ::= ':' [Type]
21. [VarType] ::= ε

22. [ReturnValue] ::= [Expression]
23. [ReturnValue] ::= ε

24. [NullableBinding] ::= '|' [Identifier]
25. [NullableBinding] ::= ε

26. [OptionalElse] ::= 'else' '{' [StatementList] '}'
27. [OptionalElse] ::= ε

28. [Expression] ::= [SimpleExpression] [RelationalTail]

29. [RelationalTail] ::= [RelationalOperator] [SimpleExpression]
30. [RelationalTail] ::= ε

31. [SimpleExpression] ::= [Term] [SimpleExpressionTail]

32. [SimpleExpressionTail] ::= '+' [Term] [SimpleExpressionTail]
33. [SimpleExpressionTail] ::= '-' [Term] [SimpleExpressionTail]
34. [SimpleExpressionTail] ::= ε
```

Table 2: grammar 1

```
35. [Term] ::= [Factor] [TermTail]

36. [TermTail] ::= '*' [Factor] [TermTail]
37. [TermTail] ::= '/' [Factor] [TermTail]
38. [TermTail] ::= ε

39. [Factor] ::= '(' [Expression] ')'
40. [Factor] ::= [Identifier]
41. [Factor] ::= [Literal]

42. [Type] ::= '?' [BaseType]
43. [Type] ::= [BaseType]

44. [BaseType] ::= 'i32'
45. [BaseType] ::= 'f64'
46. [BaseType] ::= '[]u8'

47. [Identifier] ::= id [CallParam]
48. [Identifier] ::= '_'

49. [CallParam] ::= '(' [Expression] [CallParamTail] ')'
50. [CallParam] ::= ε

51. [CallParamTail] ::= ',' [Expression] [CallParamTail]
52. [CallParamTail] ::= ε


53. [Literal] ::= integer
54. [Literal] ::= floating-point
55. [Literal] ::= string literal
56. [Literal] ::= 'null'

57. [RelationalOperator] ::= '=='
58. [RelationalOperator] ::= '!='
59. [RelationalOperator] ::= '['
60. [RelationalOperator] ::= ']'
61. [RelationalOperator] ::= '[='
62. [RelationalOperator] ::= ']='
```

Table 3: grammar 2

# C LL – table

| | $ | 'const' | 'ifj' | '=' | '@import' | '(' | '"ifj24.zig"' | ')' | ';' | ε | 'pub' | 'fn' | '{' | '}' | ':' | ',' | 'void' | 'var' | 'if' | 'while' | 'return' | '|' | 'else' | '+' | '-' | '*' | '/' | '?' | 'i32' | 'f64' | '[]u8' | id | '_' | integer | floating-point | string | 'null' | '==' | '!=' | '[' | ']' | '[=' | ']=' |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| S | | 0 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| [Program] | | 1 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| [Prolog] | | 2 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| [FunctionDefList] | | | | | | | | | | 4 | 3 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| [FunctionDef] | | | | | | | | | | | 5 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| [ParamList] | | | | | | | | | | 7 | | | | | | | | | | | | | | | | | | | | | 6 | 6 | | | | | | | | | | |
| [ParamListTail] | | | | | | | | | | 9 | | | | | | 8 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| [ReturnType] | | | | | | | | | | | | | | | | | 10 | | | | | | | | | | 11 | 11 | 11 | 11 | | | | | | | | | | | | | |
| [StatementList] | | 12 | | | | | | | | 13 | | | | | | | | 12 | 12 | 12 | 12 | | | | | | | | | | 12 | 12 | | | | | | | | | | | |
| [Statement] | | 14 | | | | | | | | | | | | | | | | 15 | 17 | 18 | 19 | | | | | | | | | | 16 | 16 | | | | | | | | | | | |
| [ReturnValue] | | | | | | # | | | | 23 | | | | | | | | | | | | | | | | | | | | | 22 | 22 | 22 | 22 | 22 | 22 | | | | | | |
| [NullableBinding] | | | | | | | | | | 25 | | | | | | | | | | | | 24 | | | | | | | | | | | | | | | | | | | | | |
| [OptionalElse] | | | | | | | | | | 26 | | | | | | | | | | | | | 26 | | | | | | | | | | | | | | | | | | | |
| [Expression] | | | | | | # | | | | | | | | | | | | | | | | | | | | | | | | | 28 | 28 | 28 | 28 | 28 | 28 | | | | | | |
| [RelationalTail] | | | | | | | | | | 30 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 29 | 29 | 29 | 29 | 29 | 29 |
| [SimpleExpression] | | | | | | # | | | | | | | | | | | | | | | | | | | | | | | | | 31 | 31 | 31 | 31 | 31 | 31 | | | | | | |
| [SimpleExpressionTail] | | | | | | | | | | 34 | | | | | | | | | | | | | | 32 | 33 | | | | | | | | | | | | | | | | | | |
| [Term] | | | | | | # | | | | | | | | | | | | | | | | | | | | | | | | | 35 | 35 | 35 | 35 | 35 | 35 | | | | | | |
| [TermTail] | | | | | | | | | | 38 | | | | | | | | | | | | | | | | 36 | 37 | | | | | | | | | | | | | | | | |
| [Factor] | | | | | | # | | | | | | | | | | | | | | | | | | | | | | | | | 40 | 40 | 40 | 40 | 40 | 40 | | | | | | |
| [Type] | | | | | | | | | | | | | | | | | | | | | | | | | | | 42 | 43 | 43 | 43 | | | | | | | | | | | | |
| [BaseType] | | | | | | | | | | | | | | | | | | | | | | | | | | | | 44 | 45 | 46 | | | | | | | | | | | | |
| [Identifier] | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 47 | 48 | | | | | | | | | | |
| [CallParam] | | | | | | # | | | | 50 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| [CallParamTail] | | | | | | | | | | 52 | | | | | | 51 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| [Literal] | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 53 | 54 | 55 | 56 | | | | | | |
| [RelationalOperator] | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 57 | 58 | 59 | 60 | 61 | 62 |

Table 4: LL – table used in syntactic analysis

# D Precedence table

| $/_x$ | * | / | + | - | “==” | “!=” | “<” | “>” | “<=” | “>=” | ( | id | ) | $ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| * | > | > | > | > | > | > | > | > | > | > | < | < | > | > |
| / | > | > | > | > | > | > | > | > | > | > | < | < | > | > |
| + | < | < | > | > | > | > | > | > | > | > | < | < | > | > |
| - | < | < | > | > | > | > | > | > | > | > | < | < | > | > |
| “==” | < | < | < | < | > | > | > | > | > | > | < | < | > | > |
| “!=” | < | < | < | < | > | > | > | > | > | > | < | < | > | > |
| “<” | < | < | < | < | > | > | > | > | > | > | < | < | > | > |
| “>” | < | < | < | < | > | > | > | > | > | > | < | < | > | > |
| “<=” | < | < | < | < | > | > | > | > | > | > | < | < | > | > |
| “>=” | < | < | < | < | > | > | > | > | > | > | < | < | > | > |
| ( | < | < | < | < | < | < | < | < | < | < | < | < | = | |
| id | > | > | > | > | > | > | > | > | > | > | | | > | > |
| ) | > | > | > | > | > | > | > | > | > | > | | | > | > |
| $ | < | < | < | < | < | < | < | < | < | < | < | < | | |

Table 5: Precedence table used in precedent syntactic analysis of expressions