Formal Languages and Compilers
Algorithms

IFJ25 Compiler – Documentation

Team xmervaj00, variant vv-BVS
with EXTFUN extension

Jaroslav Mervart	(xmervaj00)	25 %
Jozef Matus	(xmatusj00)	25 %
Jan Hájek	(xhajekj00)	25 %
Veronika Kubová	(xkubovv00)	25 %

2. December 2025

Contents

1 Introduction

This project deals with the design and implementation of an IFJ25 language compiler based on the Wren programming language. The input is the source code, and the output is a set of IFJcode25 instructions. The project is implemented in C without auxiliary generation libraries. Our variant uses a height-balanced tree data structure for the symbol table.

The compiler consists of four phases. Lexical analysis creates tokens and passes them to syntactic analysis. Here, recursive descent is used to check the syntax and create an abstract syntactic tree. This tree is passed on to semantic analysis, where function definitions, variable declarations, literal type compatibility, and range operations are checked. The checked AST is used to generate output instructions in the code generation phase.

An extension EXTFUN was implemented, which allows calling stand-alone functions. The program is run using Makefile and reads stdin after startup, but it can also read a file by adding its path and name to the input arguments. It uses stderr for error messages and writes the final instructions to stdout.

2 Disclaimer

During the development of this project, we utilized ChatGPT, along with other literature, for consulting, debugging, and refining the formulation of our ideas in this documentation.

3 Design

The design is very human. The compiler is composed of four main components: lexical analysis, Syntactic analysis, Semantic analysis, and code generation. Each of these components forms a well-defined stage in the compilation pipeline.

- **Lexical analysis** (files `lex.c/h`)
  The lexical analyzer implements a finite deterministic automaton to convert the character stream into tokens.
- **Syntactic analysis** (files `parser.c/h, psaParser.c/h`)
  The syntactic analyzer, based on recursive descent and precedence parsing, organizes these tokens into an abstract syntax tree (AST) by applying syntax-directed construction rules.
- **Semantic analysis** (files `semantic.c/h, symtable.c/h, functionTable.c/h`)
  The semantic analyzer annotates this tree by performing symbol table management, type checking, and control flow validation.
- **Code generation** (file `codegen.c/h`)
  The final stage code generation translates the semantically enriched AST into IFJcode25, closely following the operational semantics of the language.

Additional shared definitions (token types, AST structure, error codes, etc.) are placed in corresponding header files inside the `include/` directory.

4 Lexical analysis

Lexical analysis is the first part of code translation. It logically separates words and symbols from input source code into tokens and passes them to parser. Based on their content, tokens are labeled as identifiers, keywords, operators, special symbols, strings and numbers. The lexical analyser ignores all comments and does not create tokens for them.

4.1 Finite State Machine

A deterministic Finite State Machine was used when implementing token recognition. With a single character taken from input, it decides its next state and stores said character in a temporary buffer. When a token is identified, this buffer is passed on as data or id for the token. A more detailed description is provided in section A.

4.2 Tokens

A token is the fundamental structure used for communication between the lexer and the parser. Each token contains three fields: a `type`, an `id`, and `data`. The `type` determines the syntactic category of the lexeme (such as identifier, keyword, literal, operator, or special symbol). The `id` field stores the textual value of identifiers and certain operators. The `data` field is used for literal values, for example numeric and string literals.

During lexical analysis, the lexer normalizes the input into these structured tokens. Identifiers are checked against a keyword table and may be promoted to keyword types. Numeric literals, string literals, operators, punctuation, and control tokens (such as `NEWLINE`) are processed by dedicated finite-state machines. This representation allows the parser to work with well-typed symbolic units rather than raw characters, simplifying both the LL parser and the precedence parser.

4.3 Errors

Lexical analysis ends with an error only when an unexpected sequence of letters, numbers and/or symbols is found. The program terminates with a return value of 1.

5 Syntax

This section will focus on syntactic analysis. The syntactic analysis is composed from two parts. Top-down parser and Bottom-up parser.

5.1 Implementation of top-down parser

The top-down parser is implemented as a recursive descent parser. Each recursive function consumes tokens from the lexer, checks whether the input adheres to the LL grammar, and constructs the appropriate abstract syntax tree (AST) nodes.

The parser is supported by several helper functions. The 'peek' function compares the current token with the expected one and reports whether they match. The 'advance' function builds on peek: if the token is correct, it retrieves the next token from the lexer. Another helper, 'for_function', repeatedly calls advance inside a loop to verify whether an incoming sequence of tokens matches a predefined sequence of expected symbols. Additionally, 'nameHelperFunction' is used to determine whether the current token corresponds to any token within a given target set.

These helper functions integrate with the recursive descent structure to ensure correct LL parsing and to construct the AST alongside the syntactic analysis.

Implemented LL-grammar is describe in section B.

5.2 Implementation of bottom-up parser

The bottom-up parser is implemented using a stack adapted to store both tokens and AST nodes. The parsing procedure runs inside an iterative loop that terminates only when the stack contains the final symbol $E and the next token corresponds to the expected ending token.

Before applying any precedence operation, the parser determines the coordinates in the precedence table using a dedicated converter. Based on these coordinates, the appropriate action is selected.

In the case of a reduce action, the parser does not inspect only the top of the stack. Instead, it scans the stack backward to locate the nearest precedence marker $<$, which indicates the beginning of the handle. Once

this marker is found, the parser determines the size of the handle and decides which reduction rule to apply. The implementation supports two reduction patterns: a single-term reduction and a three-term reduction.

After applying the selected rule, the matched sequence is replaced with a new AST node representing the reduced expression. This process is repeated until the final reduction produces the complete AST for the expression.

The precedence table is described in section D.

6 Semantic analysis

Semantic analysis checks function definitions, variable scope validity, correctness of built-in function calls, and errors over literals. The input is an abstract syntax tree (AST) generated by the parser. It stores the necessary data and metadata about the entire program. The result of the analysis is a checked syntax tree supplemented with the information needed for the code generator.

6.1 Function registration

IFJ25 supports both normal user functions, which we know from standard languages such as C, as well as special "placeholder variables", known as static getters and setters.

First, it is necessary to save all functions, including getters and setters. The data structure **FuncTable** is used for this purpose; see section 8 for more details. The structure is filled with built-in functions Ifj.read_str, Ifj.read_num, Ifj.write, ... including their arity and expected argument types.

Then, user functions are saved from the AST, and a flag determining their type (user function, setter, getter) is attached to them. When adding each function, it is checked whether a function with the given name and arity has already been added; if so, it is a semantic error. At the end of writing the functions, a check is made to see if there is a **main** function that has no arguments and is not a setter or getter.

6.2 Checking program logic

6.2.1 Declaration of functions and local variables

The analysis then recursively traverses the AST and, depending on the type of nodes, delves into the given function that checks the IFJ25 language rules. When declaring a function, its parameters are verified, including whether they have a valid name (they must not contain the prefix for global variables "`__`") and whether they are already defined. If everything goes well, a new scope is created and placed on top of the stack with the other scopes. When declaring a local variable, we proceed similarly: first, we check the name to see if it contains a global prefix and then check for redefinition.

6.2.2 Assigment

First, the right side of the expression is checked recursively, followed by the left side. If it is a global variable, nothing needs to be checked, as all global variables are accessible at any time. If the first condition is not met, a local variable is searched for in all scopes; if it is not found, a setter is searched for last. If the search is unsuccessful, the program terminates with error 3.

6.2.3 Function call

When calling a function, the function is first searched for by name, but without the number of arguments, to determine whether a function with the same name exists. If not, it is error 3. If a function with the same name is written in the **FuncTable** structure, it is searched for again, this time with the number of arguments. If there is no function with the same arity, the program terminates with error 5. If it is a getter or setter, the number of arguments is checked. If the called function is a built-in function, the types of arguments must be checked if the given argument is a literal.

6.2.4 Identifiers

The identifier is searched for as a local variable in scopes. If unsuccessful, it continues as a global variable. If unsuccessful here as well, the existence of a getter is checked as a last resort. When calling a function, the number of arguments is also checked. If none of the above is found, the program terminates with error 3 (identifier not found).

6.2.5 Binary operations

The most complex part of semantic analysis is the handling of binary operations. First, the analyzer checks whether the operator is `is`, if so only left operand requires analysis and no further checking is applied.

For other operators, it is verified whether the operation belongs to group `is`, `==`, or `!=`. Only these operations allow keyword **Null** as an operand. If it appears elsewhere, the analysis ends with an error.

After both operands are analyzed, the semantic rules for the specific operation are checked. These include restrictions such as valid operand type combinations, requirements on operand order, or ensuring that a numeric literal used in repetition is an integer.

7 Code generation

In this final phase, IFJcode25 instructions are generated from the abstract syntax tree. By the time the program reaches this phase, the AST is already filled and checked. Before generating the instructions themselves, we create a three-address code. Output instructions are printed to stdout.

7.1 TAC

TAC is a representation of a three-address code that compilers use to facilitate the process of generating the resulting code. Complex expressions are broken down into simple steps containing a maximum of three addresses. This is handled by a doubly linked list `TAClist`, which links `TACnode` nodes storing instructions, arguments, and pointers. We chose this implementation because it is easier to optimize and extend.

7.2 Generating

This is handled by the main function `generate`, which reads the input AST, creates `TAClist`, and inserts a jump to the label `$$main`. This is followed by the generation of all functions and the printing of the resulting list of instructions in IFJcode25 format. The function definition is generated by first creating frames, then processing parameters, and allocating local variables. The body of the function is then generated recursively. For another example, when generating conditional statements, the expression is evaluated first. Based on the result, labels and the corresponding jump instructions are inserted.

8 Data structures

8.1 Function table

During semantic analysis, information about all functions is stored in the function table. This structure keeps track of the maximum and current number of functions and references to functions, which is another structure called **FuncInfo**, where the name, number of parameters, data type of parameters, type of function (whether it is a regular, built-in function, getter, or setter) and, finally, a reference to the function in the AST are stored.

8.2 Symbol table

The symbol table is implemented as a height-balanced binary tree. A single node of the tree is defined as a structure containing height, name, and then two references to its children, left and right. Local variables are stored in the symbol table, and since there may be multiple scopes for variables in a program, there must also

be multiple tables. Therefore, symbol tables are placed on a stack called **scopeStack** and are processed there during semantic analysis.

8.3 Stack

The bottom-up parser relies on a stack to operate correctly. In this project, the stack implementation was adapted from the materials used in the IAL course. The original structure was modified so that it can store not only tokens but also pointers to abstract syntax tree (AST) nodes, which allows the parser to integrate syntax analysis with AST construction.

8.4 Abstract syntax tree

AST is the main intermediate representation of the program. Each node contains an enum `ASTnodeType` (Program, function definition, variable declaration, assignment, literal, etc.). Depending on this type, a specific part of the union is used which contains structures tailored to the given node type. For example, the type `AST_PROGRAM`, which is the root of the entire tree, contains the current number of functions and also references to the functions themselves in the tree. A node of type `AST_BINOP` contains the type of operation (addition, subtraction, multiplication, etc.) and references to the left and right operands. The AST is created during syntactic analysis, where it is then traversed by semantic analysis to check the correctness of the program. Finally, the tree is traversed recursively and the IFJcode25 instructions are generated according to the node type.

9 Work in team

Communication took place primarily through the Discord platform, where we shared important information and made calls. There were also irregular meetings at the faculty, where we discussed progress since the last meeting and how we would proceed. In the code, we used the label `"#NEXT_MEETING"` for pieces of code that we wanted to discuss with the team. We used GitHub for version control, resulting in more than 130 commits by the end of the project.

The work was divided as follows:

- **Jaroslav Mervart** (xmervaj00) – implemented the lexical analyzer and code generator.
- **Jozef Matus** (xmatusj00) – implemented the syntax analyzer and the AST construction. Wrote documentation.
- **Jan Hájek** (xhajekj00) – implemented the semantic analysis, AST definition, and the function/scope management. Wrote documentation.
- **Veronika Kubová** (xkubovv00) – helped with lexical analyzer, precedence table and code generator. Wrote documentation.

10 Development description

10.1 Workflow

Work on the project began on September 16, when a repository was also created on GitHub. First, lexical analysis was started, focusing primarily on the design and structure of tokens. This was followed by the design of the LL grammar and work on the parser. After creating the basic framework of the parser, the structure of the abstract syntax tree (AST) was created, and the symbol table was implemented. At this point, the lexer was already stable, and only debugging was taking place, so it was possible to start working on the code generator. The semantic analyzer already had a basic form and the structure for storing functions was gradually created. As functionality was added, individual parts were tested.

The project was tested using a trial submission. This gave us feedback on the compatibility of individual parts and helped us verify that the entire compiler could be compiled and run. Based on this feedback, critical errors were removed.

In the final phase, the parser and semantic analysis took their final form. All that remained was to finish the code generator and complete the documentation.

10.2 Tools

All team members used Visual Studio Code to write code, which, after adding an extension, allowed multiple users to edit files in real time, a feature we utilized during our meetings. We used the aforementioned git repository for versioning.

10.3 Important decisions

When creating the code generator, we decided to implement three-address code. We took this step because we wanted to optimize the code in the final stage of development, which was ultimately not implemented due to time constraints. However, the compiler is ready for optimization.

10.4 Testing

Testing was carried out continuously throughout the development process. Lexical and semantic tests were written by hand and used during integration to verify the code's functionality. Student tests were also used to assess the overall functionality. Finally, the available programs from the IFJ e-learning course were tested.

11 Conclusion

In this project, we implemented a complete compiler for IFJ25 following the classic multi-stage architecture used in modern language design. It integrates a deterministic lexical analyser, a combined top-down and bottom-up syntactic analysis, a comprehensive semantic analyser, and a generator producing valid IFJcode25 instructions. The internal structure - particularly the AST representation, function table, and height-balanced binary tree - proved crucial for ensuring correctness and consistency across compilation phases. We successfully implemented the EXTFUN extension and maintained rigorous testing throughout development. Although planned optimalizations were not completed, the system is architecturally prepared for such improvements. This project enhanced our understanding of compiler theory, practical implementation challenges, and effective collaborative development.

References

KUMAR SINGH, Ankit. Three address code in Compiler. Online. In: . 27 Dec, 2024. Avalible from: https://www.geeksforgeeks.org/compiler-design/three-address-code-compiler/. [cit. 2025-12-03].

HONZÍK, Prof. Ing.Jan M, CSc. Algoritmy - Studijní opora. Online. 6. 1. 2014. Brno. [cit. 2025-12-03].

A Lexer diagrams

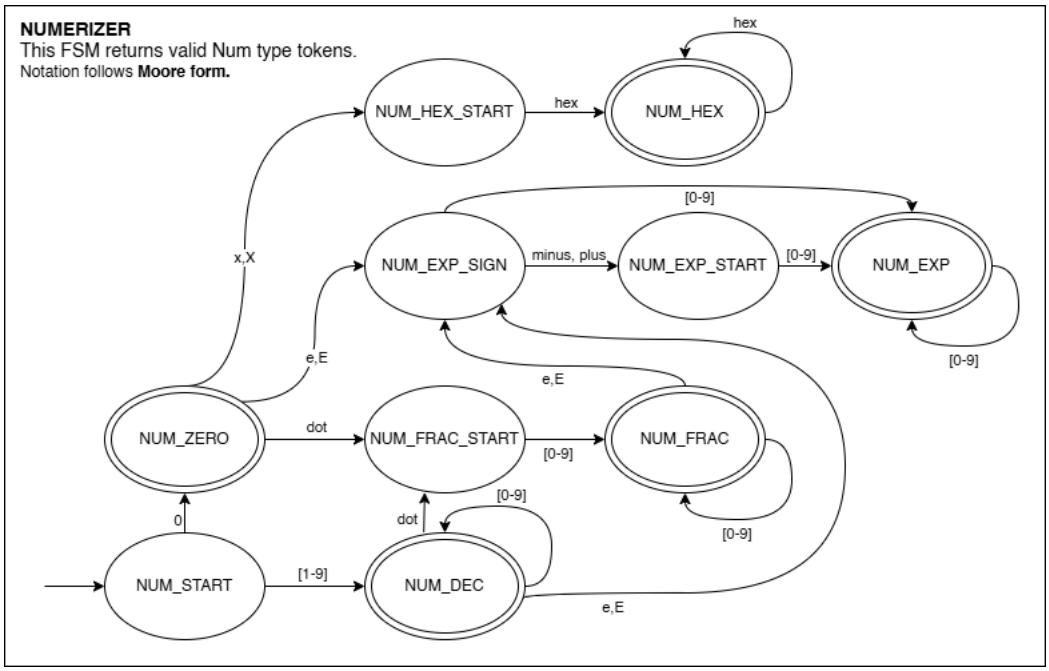

Figure 1: Numerizer FSM

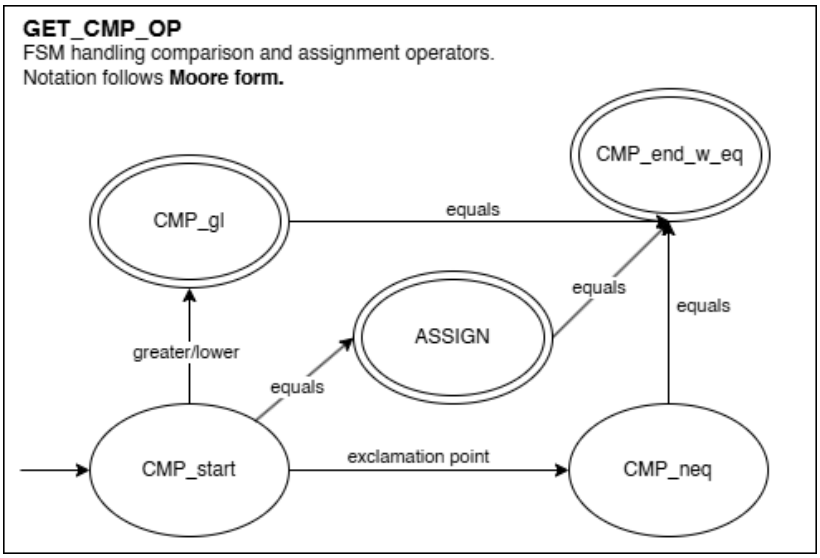

Figure 2: Get compare operator FSM

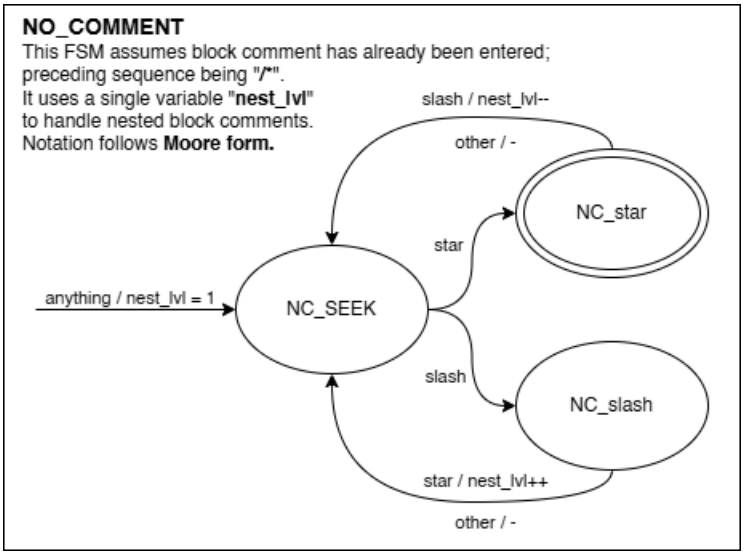

Figure 3: Remove comments FSM

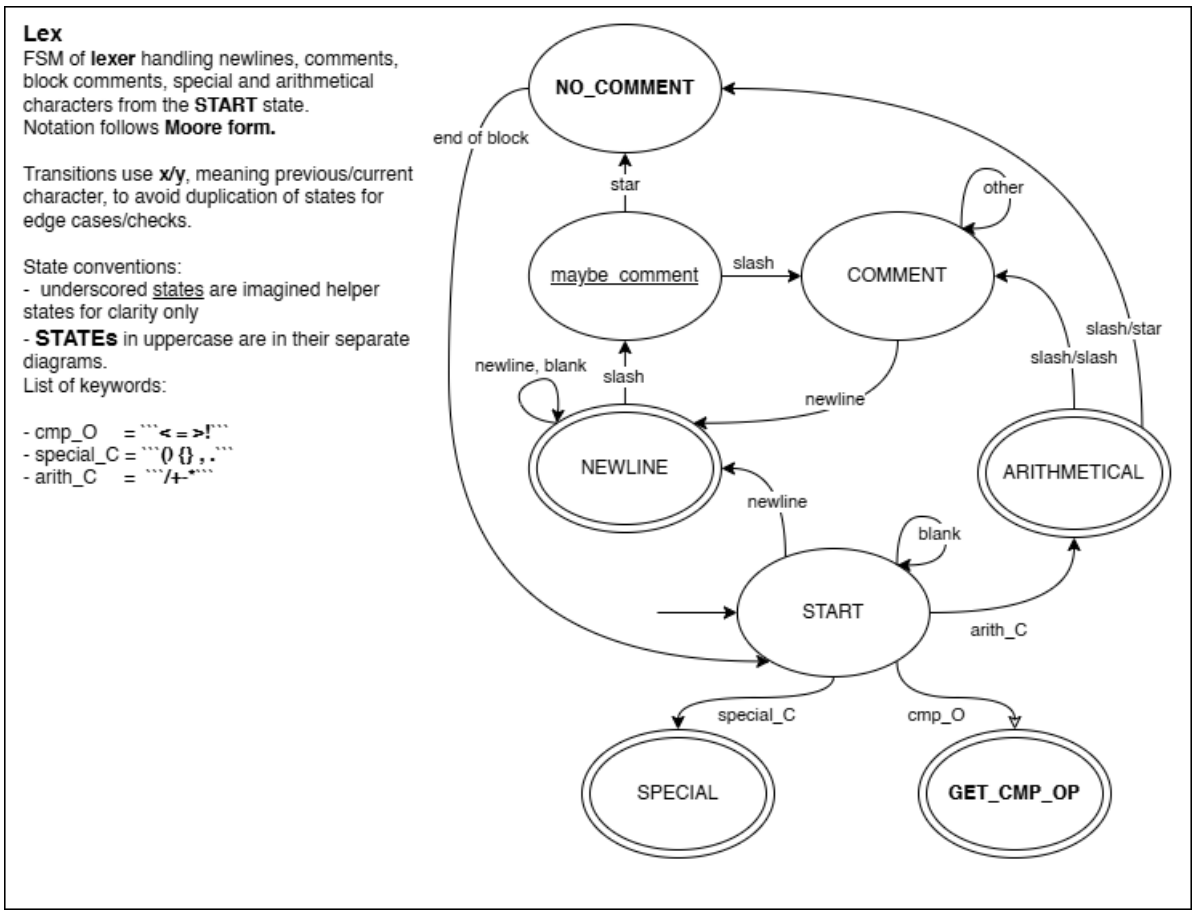

Figure 4: Main FSM

B LL grammar

The symbol "/" in the grammar represents the end-of-line (EOL) token.

```
PROGRAM ::= PROLOG CLASS
PROLOG ::= import "ifj25" for Ifj /
CLASS ::= class Program { / FUNCTIONS }

FUNCTIONS ::= static id FUNC_GET_SET_DEF FUNCTIONS
FUNCTIONS ::= ε

FUNC_GET_SET_DEF ::= ( PAR ) { / FUNC_BODY } /
FUNC_GET_SET_DEF ::= { / FUNC_BODY } /
FUNC_GET_SET_DEF ::= = ( id ) { / FUNC_BODY } /

PAR ::= ε
PAR ::= id NEXT_PAR
NEXT_PAR ::= , id NEXT_PAR
NEXT_PAR ::= ε

ARG ::= ε
ARG ::= ARG_NAME NEXT_ARG
NEXT_ARG ::= ε
NEXT_ARG ::= , ARG_NAME NEXT_ARG

ARG_NAME ::= int
ARG_NAME ::= string
ARG_NAME ::= float
ARG_NAME ::= id
ARG_NAME ::= global_id
ARG_NAME ::= null

VAR_NAME ::= id
VAR_NAME ::= global_id

FUNC_BODY ::= ε
FUNC_BODY ::= VAR_DECL FUNC_BODY
FUNC_BODY ::= VAR_ASS_CALL_GET FUNC_BODY
FUNC_BODY ::= IF_STAT FUNC_BODY
FUNC_BODY ::= WHILE FUNC_BODY
FUNC_BODY ::= RETURN FUNC_BODY
FUNC_BODY ::= { / FUNC_BODY } / FUNC_BODY
FUNC_BODY ::= Ifj . id ( ARG ) / FUNC_BODY

VAR_DECL ::= var VAR_NAME /
VAR_ASS_CALL_GET ::= id VAR_OR_CALL
VAR_ASS_CALL_GET ::= global_id = RSA

VAR_OR_CALL ::= = RSA
VAR_OR_CALL ::= ( ARG ) /

RSA ::= EXPRESSION /
RSA ::= id FUNC_TYPE /
RSA ::= Ifj . id ( ARG ) /

FUNC_TYPE ::= ε
FUNC_TYPE ::= ( ARG )
```

```
IF_STAT ::= if ( EXPRESSION ) { / FUNC_BODY } else { / FUNC_BODY } /
WHILE ::= while ( EXPRESSION ) { / FUNC_BODY } /
RETURN ::= return EXPRESSION /
```

C Transition Table

	$	import	"ifj25"
S		S::=PROGRAM $	
PROGRAM		PROGRAM::=PROLOG CLASS	
PROLOG		PROLOG::=import "ifj25" for Ifj /	

Table 1: LL(1) table – part 1

	for	Ifj	/
FUNC_BODY		FUNC_BODY::=Ifj.id(ARG) / FUNC_BODY	
RSA		RSA::=ifj.id(ARG)/	

Table 2: LL(1) table – part 2

	class	Program	{
CLASS	CLASS ::= class Program { / FUNCTIONS }		
FUNC_GET_SET_DEF			FUNC_GET_SET_DEF ::= { / FUNC_BODY } /
FUNC_BODY			FUNC_BODY ::= { / FUNC_BODY } / FUNC_BODY

Table 3: LL(1) table – part 3

	}	static
FUNCTIONS		FUNCTIONS ::= static id FUNC_GET_SET_DEF FUNCTIONS

Table 4: LL(1) table – part 4

	id	EPS
FUNCTIONS		FUNCTIONS ::= EPS
PAR	PAR ::= id NEXT_PAR	PAR ::= EPS
NEXT_PAR		NEXT_PAR ::= EPS
ARG	ARG ::= ARG_NAME NEXT_ARG	ARG ::= EPS
NEXT_ARG		NEXT_ARG ::= EPS
ARG_NAME	ARG_NAME ::= id	
VAR_NAME	VAR_NAME ::= id	
FUNC_BODY	FUNC_BODY ::= VAR_ASS_CALL_GET FUNC_BODY	FUNC_BODY ::= EPS
FUNC_TYPE		FUNC_TYPE ::= EPS
VAR_ASS_CALL_GET	VAR_ASS_CALL_GET ::= id VAR_OR_CALL	
RSA	RSA ::= id FUNC_TYPE /	

Table 5: LL(1) table – part 5

	(	)
FUNC_GET_SET_DEF	FUNC_GET_SET_DEF ::= (PAR) { / FUNC_BODY } /	
NEXT_PAR		
NEXT_ARG		
VAR_OR_CALL	VAR_OR_CALL ::= (ARG) /	
FUNC_TYPE	FUNC_TYPE ::= (ARG)	

Table 6: LL(1) table – part 6

	=	,
FUNC_GET_SET_DEF	FUNC_GET_SET_DEF ::= = (id) { / FUNC_BODY } /	
NEXT_PAR		NEXT_PAR ::= , id NEXT_PAR
NEXT_ARG		NEXT_ARG ::= , ARG_NAME NEXT_ARG
VAR_OR_CALL	VAR_OR_CALL ::= = RSA	
FUNC_TYPE		

Table 7: LL(1) table – part 7

	int	string	float	null
ARG	ARG ::= ARG_NAME NEXT_ARG	ARG ::= ARG_NAME NEXT_ARG	ARG ::= ARG_NAME NEXT_ARG	ARG ::= ARG_NAME NEXT_ARG
ARG_NAME	ARG_NAME ::= int	ARG_NAME ::= string	ARG_NAME ::= float	ARG_NAME ::= null

Table 8: LL(1) table – part 8

	global_id	.	var	EXPRESSION
ARG	ARG ::= ARG_NAME NEXT_ARG			
ARG_NAME	ARG_NAME ::= global_id			
VAR_NAME	VAR_NAME ::= global_id			
FUNC_BODY	FUNC_BODY ::= VAR_ASS_CALL_GET FUNC_BODY		FUNC_BODY ::= VAR_DECL FUNC_BODY	
VAR_DECL			VAR_DECL ::= var VAR_NAME /	
VAR_ASS_CALL_GET	VAR_ASS_CALL_GET ::= global_id = RSA			
RSA				RSA ::= EXPRESSION /

Table 9: LL(1) table – part 9

	if	else
FUNC_BODY	FUNC_BODY ::= IF_STAT FUNC_BODY	
IF_STAT	IF_STAT ::= if (EXPRESSION) { / FUNC_BODY } else { / FUNC_BODY } /	

Table 10: LL(1) table – part 10

	while	return
FUNC_BODY	FUNC_BODY ::= WHILE FUNC_BODY	FUNC_BODY ::= RETURN FUNC_BODY
WHILE	WHILE ::= while (EXPRESSION) { / FUNC_BODY } /	
RETURN		RETURN ::= return EXPRESSION /

Table 11: LL(1) table – part 11

D Precedence table

	*	/	+	-	==	!=	<	>	<=	>=	(	)	id	is	$
*	>	>	>	>	>	>	>	>	>	>	<	>	<	>	>
/	>	>	>	>	>	>	>	>	>	>	<	>	<	>	>
+	<	<	>	>	>	>	>	>	>	>	<	>	<	>	>
-	<	<	>	>	>	>	>	>	>	>	<	>	<	>	>
==	<	<	<	<							<	>	<	<	>
!=	<	<	<	<							<	>	<	<	>
<	<	<	<	<							<	>	<		>
>	<	<	<	<							<	>	<		>
<=	<	<	<	<							<	>	<		>
>=	<	<	<	<							<	>	<		>
(	<	<	<	<	<	<	<	<	<	<	<	=	<	<	
)	>	>	>	>	>	>	>	>	>	>		>		>	>
id	>	>	>	>	>	>	>	>	>	>		>		>	>
is					>	>							=		>
$	<	<	<	<	<	<	<	<	<	<	<		<	<	$

Table 12: Precedence table for the expression parser