BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Documentation of compiler for language IFJ24

Team xnovakf00 – variant vv-BVS

extensions: FUNEXP

Matúš Fignár xfignam00 – 25%
Tibor Malega xmalegt00 – 25%
Filip Novák xnovakf00 – team leader – 25%
Ján Skovajsa xskovaj00 – 25%

December 4, 2024

Contents

List of Figures

1 Teamwork

Implementing a compiler is not an easy task, that is why the work has begun as soon as the assignment was published. Firstly, tasks were distributed among all members based on preferences.

Test submission was set as deadline for implementation. This deadline was successfully met with time left for fixing errors and further testing. Pair programming as well as code reviews were practised.

1.1 Communication

Discord was chosen as primary mean of communication. In specialised voice and text channels meetings were held at least once a week. Git was used as versioning system with remote repository hosted by team leader at GitHub. Using webhook, Discord bot for notifications about changes on remote repository was added.

Email addresses and telephone numbers were shared among all members in case of an unexpected situation. Luckily, no problems were encountered and communication ran smoothly for the whole duration of the project.

1.2 Task distribution

- **Matúš Fignár xfignam00** – precedence table, "bottom-up" expression parsing
- **Tibor Malega xmalegt00** – finite automaton design, lexical analysis
- **Filip Novák xnovakf00** – "top-down" recursive parsing, table of symbols, testing, AST
- **Ján Skovajsa xskovaj00** – code generating, testing

The remaining parts of the project were developed by all members of the team simultaneously.

2 General information about implementation of compiler

Compiler uses double-traversal method of the source code based on "top-down" recursive parsing specified by LL(1) grammar[1]. Expressions are processed using precedence syntactic analysis. Lexical analyser (scanner) processes source code and tokenizes it based on deterministic finite automaton[2]. During parsing,
semantic analysis is performed and abstract syntax tree (AST) is constructed for internal representation. Compiler uses table of symbols, implemented as height-balanced binary search tree for keeping information about defined functions and variables. Three-address code is generated through recursive traversal of constructed AST. Compiler is written exclusively in `C` language. Test cases covering basic and advanced programs, both correct and incorrect were created. Exit codes and output were compared to reference. Programs expecting input from user were automated as well.

3 Lexical analysis

Lexical analysis is mediated by a **scanner** (`scanner.c, scanner.h`) which reads from standard input and processes it into **tokens**. It is entirely based on the **deterministic finite state automaton**. The lexical analysis starts in `getToken()` function and works based on the first character read from input and, according to it, continues to process the lexeme. More complicated lexemes such as number, ID or string lexemes have its own functions which process lexemes of those types. Additionally, the scanner indentifies **lexical errors** and calls `ERRORLEX` to handle all errors with the appropriate error message and the **line** and **column number**.

[1] See figures 2 and 3.
[2] See figure 1.

Each created token contains:

- a **type**, represented by an element from an enumeration `token_types`
- a **value** holding the content of the token, represented as a string of characters
- **line** and **column number** of the currently processed token

The value of token is dynamically resized when necessary to ensure sufficient memory for tokens of any length and saved into a linked list `TokenValues`, which is used to correctly free all allocated memory for token values. Every not empty value of a token is terminated with a null string at the end of processing the token. Since scanner and specifically `getToken()` function is called many times during the compiling process, line and column number are defined as global variables in order to keep track of them even when the process of a single lexeme ends.

3.1 Functionality

A **single character** lexemes are processed directly in `getToken()` function by assigning a type, line and column number before returning the token.

Number lexems are recognized in `getToken()` function by their first character and function `process_Number_Token()` is called. This function reads through input and accepts characters as shown in the finite state automaton. **Whole**, **decimal**, **exponential** and **zero** lexemes are all processed in this function and all lexical errors regarding those types are covered here.

String lexemes are recognized by their first character '"' in `getToken()` function. The lexeme is then processed in `process_String_Token()` function. This function reads and saves all characters until it reaches the second '"' or ends in lexical error. All **escape sequences** in the string are directly transformed to what they represent and inserted into the **token value**. **Hexadecimal numbers** in string are also transformed into their **decimal value** and then assigned to the token value. The other type of string, **multiline string**, is handled in a separate `process_Multiline_String_Token()` function. This function is called after encountering a **backslash** as the first character read. This function has a similar purpose to `process_String_Token()` but it **doesn't** transform escape sequences nor hexadecimal numbers and is used purely for Multiline strings.

Lexemes of type **ID** are processed in function `process_ID_Token()` which is called as a **default case** in switch located in `getToken()`. Lexeme is processed in this function if none of the above token types criteria were met.

Keywords are initially recognized as IDs and are processed in the **ID function**. After the ID is processed, a function for recognizing ID as a keyword is called and assigns it the correct keyword type as **element** from enumeration `token_types`. List of keywords is stored as an array of strings.

Import type lexemes are processed in `process_Import()` function after getting a '@' as the first character.

4 Syntactic analysis

Syntactic analysis is divided into two parts – expression parser and recursive parser processing the rest of the source code. These two are closely intertwined. Both call scanner to receive tokens to evaluate.

When recursive parser expects an expression in source code (denoted as `expression` in LL(1) grammar (2)), expression parser is called for processing.

4.1 "Top-down" syntactic analysis

Recursive parser is implemented as a set of functions, each representing certain non-terminal of LL(1) grammar, found in `parser.c, parser.h`. Functions validate sequence of tokens received by invoking scanner with function `getToken()` (or macro `GT` for compactness). When another non-terminal is expected, function

representing it is called. Each function returns `true` when processing of the non-terminal was successful. If an error occurs, `ERROR` is called[3].

In the first traversal, information about user-defined functions are collected. This includes:

- function IDs
- return data types
- number of parameters
- parameter names
- parameter data types

Other tokens, not representing function headers are skipped .This is implemented as loop with break condition being encountering token `pub`, signaling start of the next function definition.

Second traversal performs regular parsing of the source code with validation of token sequence, performing semantic analysis and constructing AST.

To support double traversal, function processing user-defined functions was split into two separate with adjusted logic: `def_func_first()` and `def_func_sec()`. For readability and reusability, `after_id()` function was split into two parts: `funCallHandle()`, processing standalone function calls in source code (or function calls in expressions) and `assignmentHandle()`, processing assignment to variables.

Validating compulsory prologue in source code is handled in LL(1) grammar as rule 2. For simplicity, `ifj` namespace and string literal `"ifj24.zig"` are represented as id terminal and expression. Further validation of their correctness is done in `prog()`. If either of them is incorrect, syntax error is raised and program exits with error code `2`.

4.2 "Bottom-up" syntactic analysis of expressions

The analysis involves processing expressions using a **precedence table**[4], which defines the relationships between operands, arithmetic and relational operators. A **stack** is employed to facilitate this analysis. The functions responsible for the analysis are implemented in `expression_parser.c`, `expression_parser.h`.

At the start of the analysis, the symbol `STOP` is added to the stack. Each subsequent element of the expression is retrieved using the `getToken()` function (the `GT` macro). The retrieved token is then assigned a symbol that represents its **role** in the expression, determined by the `evaluate_given_token()` function. If the token represents a **variable**, a **constant**, or a **function call**, control is handed over to the parser via the `wasDefined()` or `funCallHandle()` functions[5]. However, if a token does not conform to the rules defined in the precedence table, it is assigned the symbol `STOP`, signaling the termination of processing.

Each terminal symbol from the input is placed on the stack according to the rules of precedence analysis (via the `shift()` function) and subsequently reduced to **non-terminal symbols** based on reduction rules (via the `reduce()` function)[6]. If the precedence rules are violated, the program raises a **syntax error**.

The analysis concludes when the terminal symbol in the input is `STOP` and the topmost terminal symbol on the stack is also `STOP`. The result of this analysis is an Abstract Syntax Tree (AST) representing the expression, which is then passed to the parser. **The parser** must verify whether the token represented by the `STOP` symbol during precedence analysis is valid within the program context. If it is not, a **syntax error** is reported.

[3] See section about errors 9.
[4] See figure 5.
[5] See section 10.
[6] See figure 4.

5 Semantic analysis

Semantic actions are executed together with syntactic analysis, both in "top-down" and "bottom-up" parsers. Various functions, encapsulating these actions were created to make the code readable and structured.

5.1 Semantic analysis in "top-down" parser

This subsection lists and briefly explains all semantic actions taken in "top-down" parser. By incorporating them directly into parser, there is no need for unnecessary and complicated traversals of AST.

5.1.1 Variables

When defining a variable, parser ensures no **redefinition** of already existing variable occurs within the same nested blocks. This is achieved by searching stack using `findInStack()`. If `NULL` is returned, redefinition is not detected. Similarly, `funSymtable` is also searched for an entry with matching ID.

Assignment to **undefined variables** are not allowed. Verification is similar to redefinition check, however non-`NULL` value is expected.

Matching data types of the variable and the assigned expression in assigning and defining must be confirmed. This is done by examining data type of the `astNode` representing expression. Expression cannot be logical.

As explicit data type of a variable during definition can be omitted, it is necessary to **inherit data type from expression**. If it is not possible (e.g. expression is `null` or `string` literal), an error is raised. If the data type can be deduced, it is set to symbol table entry of a variable along with nullability of the variable.

When parser is finished with processing block of code bounded with curly brackets, symbol table on the top of the stack is recursively traversed (preorder traversal, function `allUsed()`) and flags `changed` and `used` are observed, ensuring that **all modifiable variables were modified** and **all defined variables within block were used**. This includes `id_without_null` in *if-else* and *while* as well as function parameters.

5.1.2 Functions

Redefinition of functions is checked during the first traversal of the parser (same concept as variable redefinition check). With separate symbol table for built-in functions, matching IDs are allowed (e.g. `ifj.write()` and user-defined `write()`).

Validation that all user-defined functions are **used within program** is performed at the very end of second traversal of parser, by examining `used` flag in `funSymtable` nodes. Function `main()` has this flag set to `true` implicitly.

Non-void function not containing any return statements in all possible execution paths is evaluated as semantic error. This is analysed by `allReturns()` function, using depth-first recursive traversal of AST of a function body. Function returns `true`, if:

- `return` AST node is encountered
- all paths in `if` and `else` blocks return true
- all paths in `while` block return true

When a different AST node is encountered, `allReturns()` is called on the `next` of the AST node [7].

Type compatibility in returning expression is checked with each `return` statement. If it is not compatible with expected return type (from function definition), error is raised. Similarly, when a `void` function returns an expression, error is raised.

[7] TEMPLATETYPEDEF. *Stack overflow*. Online. Accessible through: stackoverflow.com/questions/21945891. [cited 2024-12-03].

When a function is called outside of an expression, it must be `void`. Return type of function is examined in symbol table entry.

Presence of main function in source code is validated by `mainDefined()` at the end of first traversal. At the same time, **parameters and return type of main** are checked, ensuring they match the expected specification.

5.1.3 Other semantic checks

When `while` or `if-else` statements are being processed by parser, condition expression is checked. In statements without the nullable variable part, logical expression is expected. In statements with `id_without_null`, condition has to be nullable.

5.2 Semantic analysis in "bottom-up" parser

The "bottom-up" semantic checks primarily focuse on the compatibility of types in expressions. Analysis verifies whether the right types of operands are used together in operations. When necessary, it performs retyping of these operands. The semantic functions for this parser are defined in the `expression_parser.c` and `parser.c` files.

5.2.1 Semantic analysis of ID Symbols

In the `evaluate_given_token()` function, when the parser evaluates a variable or a function call as valid, semantic type-casting verification is performed. Castable variables (**constants** with values **known at compile-time**) are stored in the AST as **literal nodes** to facilitate easier manipulation during subsequent semantic checks.

For elements of type **ID** (variables, function calls, or literals), the `control` structure is populated during token evaluation. This structure stores additional information necessary for further semantic checks of the given element.

If not-defined function or variable is encountered, semantic error of undefined variable or function is envoked.

5.2.2 Semantic check for binary operations

When the reduction of binary operations is required, the `semantic_check()` function is invoked to verify the compatibility of operand types with each other and with the given operation. This function ensures that:

- **Retyping:** Retyping is allowed only for operands whose values are known at **compile-time** and are of type `i32` or `f64` (with zero decimal part). Priority is given to retyping from `i32` to `f64`. The `retype()` function is called to traverse the entire operand **subtree** and retype all nodes to the required type. Based on the retypeability of both operands, it is determined whether the operands in the operation can be retyped for future use.
- **Prohibited Types:** Operands of type `string` or `u8` are not allowed.
- **Nullable Variables:** Binary operations (except **EQUAL** or **NOT EQUAL**) cannot involve operands with `null` or `nullable` types.
- **Null Comparisons:** A `null` value can only be compared with operands of type `null` or `nullable`.

Violating any of these rules results in a **semantic type error**.

6 Table of symbols

Data structure representing symbol tables in implementation is *height-balanced binary search tree*. Functions for working with symbol tables, structs representing nodes or node data and functions for printing .dot visualisation of symbol table can be found in `symtable.c` and `symtable.h`. Recursive approach was adopted because the binary tree itself is recursive in nature.

Compiler utilizes three distinct types of symbol tables:

- symbol table for user-defined functions
- symbol table for built-in functions
- symbol table for defined variables

For all types, ID of an element, represented as an array of characters, is used as key based on which the tree is arranged. Node representation (`struct symNode`) is the same for all, one thing that varies is data representation in union `data` (`struct funData` for functions and `struct varData` for variables).

6.1 User-defined functions

Symbol table for **user-defined functions** is a global variable `symtable funSymtable` declared in `symtable.h`. Parser calls `createSymtable()` to initialise it. Nodes include information collected during first traversal of parser and pointer to a symbol table for variables defined within the function.

6.2 Built-in functions

Symbol table for **built-in functions** is also a global variable `symtable builtinSymtable` declared in `symtable.h` sharing the same information as symbol table for user-defined functions. It is populated by parser using function `prepareBuiltinSymtable()`. All data are pre-initialised based on assignment. No new entry is added into this function after initialisation – it is only used for semantic checks and AST construction.

For built-in functions `ifj.write(term)` and `ifj.string(term)`, extra data types (`any, stringOru8`) in `enum dataType` were introduced, as parameters in these functions can take multiple data types.

Separation of symbol tables for built-in and user-defined functions was realised for support of user-defined functions with the same ID as an existing built-in one.

6.3 Variables

Symbol tables for **defined variables** are created during parsing of blocks of code bounded by curly brackets (functions, *while* and *if-else*). Scopes, in which defined variable is accessible are implemented using stack of symbol tables (global variable `stack symtableStack`). Stack is implemented as singly linked list. Function `findInStack()` is used to traverse symbol table stack and look for nodes with given key. If no entry is found in the whole stack, `NULL` is returned.

7 Abstract syntax tree

Abstract syntax tree (found in `ast.c, ast.h`) is implemented as acyclic graph with specialised node types to represent different components of the source code. The AST includes the following node types, each representing a specific syntactic construct:

- **Control Flow**
  - while
  - if-else
  - if
  - else
- **Statements**
  - assign
  - unused declaration
  - variable definition
  - function definition
  - return
- **Expressions**
  - expression
  - binary operator
  - function call
  - literal
  - variable

Each node has distinct set of information needed for semantic analysis or code generating, implemented with `struct`. Nodes may include pointer to a different nodes (e.g. condition in *while* or *if-else*, expression node in assign or return). All nodes in constructed AST are `struct astNode`, which serves a purpose of general representation. Information included in `astNode`:

- type of node
- pointer to the next statement in block (in case node itself represents a statement)
- union of node representations (exact representation of a node, as mentioned before)

Each node type has a designated function for its creation (`create*Node()`, where `*` is the type of the node) with all information needed to be stored as parameters.

Special node `root` only includes a `next` pointer, which serves as a connecting piece for bodies of functions, *if-else* or *while*. Global variable `AST ASTree`, declared in `main.c` is an encapsulation of the `root` of the AST of the whole source code. It is the product of syntactic analysis.

Functions for printing .dot representation for visualisaton of AST are also implemented[8].

8 Generating of IFJCODE24

Code generating was performed by traversing the ABSTRACT SYNTAX TREE (AST). This traversal involved an iterative approach for the overall structure and recursive processing for function bodies. Implementation can be found in `code_generator.c, code_generator.h`.

8.1 Code Buffer

For generating of code, an additional data structure, `Buffer_ll`, was developed and implemented in the modules `code_buffer.h` and `code_buffer.c`. The `Buffer_ll` structure is designed as a linked list where each node contains a pointer to a string, typically representing one line of code. Exceptions include cases like generating headers or built-in functions.

The `Buffer_ll` also incorporates:

- A pointer to an **accumulator string** for temporarily storing a string before adding it to a node.
- A special pointer, `flag`, that marks a specific node. This allows inserting a new node after the flagged node instead of appending it to the end of the list. This feature was primarily used for managing variable declarations.

[8] See figures 6,7. Inspiration from BuDDy library.

8.2 Header and Built-In Functions

At the start of code generating, a header is generated. This includes:

- The declaration of the intermediate code header: `.IFJcode24`.
- Global variable declarations for storing function return values.
- Proper initialization for code execution.

Next, built-in functions without a direct representation in `IFJcode24` are implemented as standalone functions. These built-in functions are prefixed with `$$` to differentiate them from user-defined functions and keywords in `IFJcode24`. Other built-in functions, which correspond directly to `IFJcode24` instructions, are generated when called, primarily for optimization purposes.

8.3 User-Defined Functions

User-defined functions are prefixed with `$` to distinguish them from built-in functions and keywords in `IFJcode24`. After generating the corresponding label for a function, the code generator flags a node in `Buffer_ll`. This enables the declaration of all variables that are used in function at the beginning of the function, ensuring that in generated code will be no reinitialization. Note that reinitialization is not supported in `IFJcode24`.

For every user-defined variable, compiler adds the prefix _ to differentiate it from other variables used by the compiler for various purposes.

Function bodies are generated recursively by traversing the AST and producing the corresponding code for each node.

8.4 Generating *if* and *while* Statements

For generating *if* and *while* statements, unique labels are required throughout the generated code. To achieve this:

- A static variable, `count`, is used to store the number of generated labels.
- Before generating a new statement, `count` is incremented.
- A special function is called to generate a unique and relevant label for the statement.

8.5 Expressions

Expressions are generated using a stack, which supports `IFJcode24` instructions. This approach simplifies the processing of expressions derived from a binary tree. Expression generating involves:

- Traversing the top-level binary tree for expression recursively using *post-order* traversal.
- For binary operators, invoking the corresponding `IFJcode24` instruction ensures the correct order of values is maintained in the stack.

This design ensures efficient and reliable processing of expressions during code generating.

9 Error handling and memory management

When lexical, syntax or semantic error is detected during analysis, an error is raised using `ERROR` or `ERRORLEX` macros, defined in `error.h`. Allocated memory is freed[9]. Suitable error message along with line and column number is printed to `stderr` and program exits with one of the error codes.

This memory management, when encountering error, is the reason some elements are implemented as global variables. Returning back to `main()` function is not needed and process is simplified.

[9]This includes: main AST, tables of symbols and string values produced by scanner.

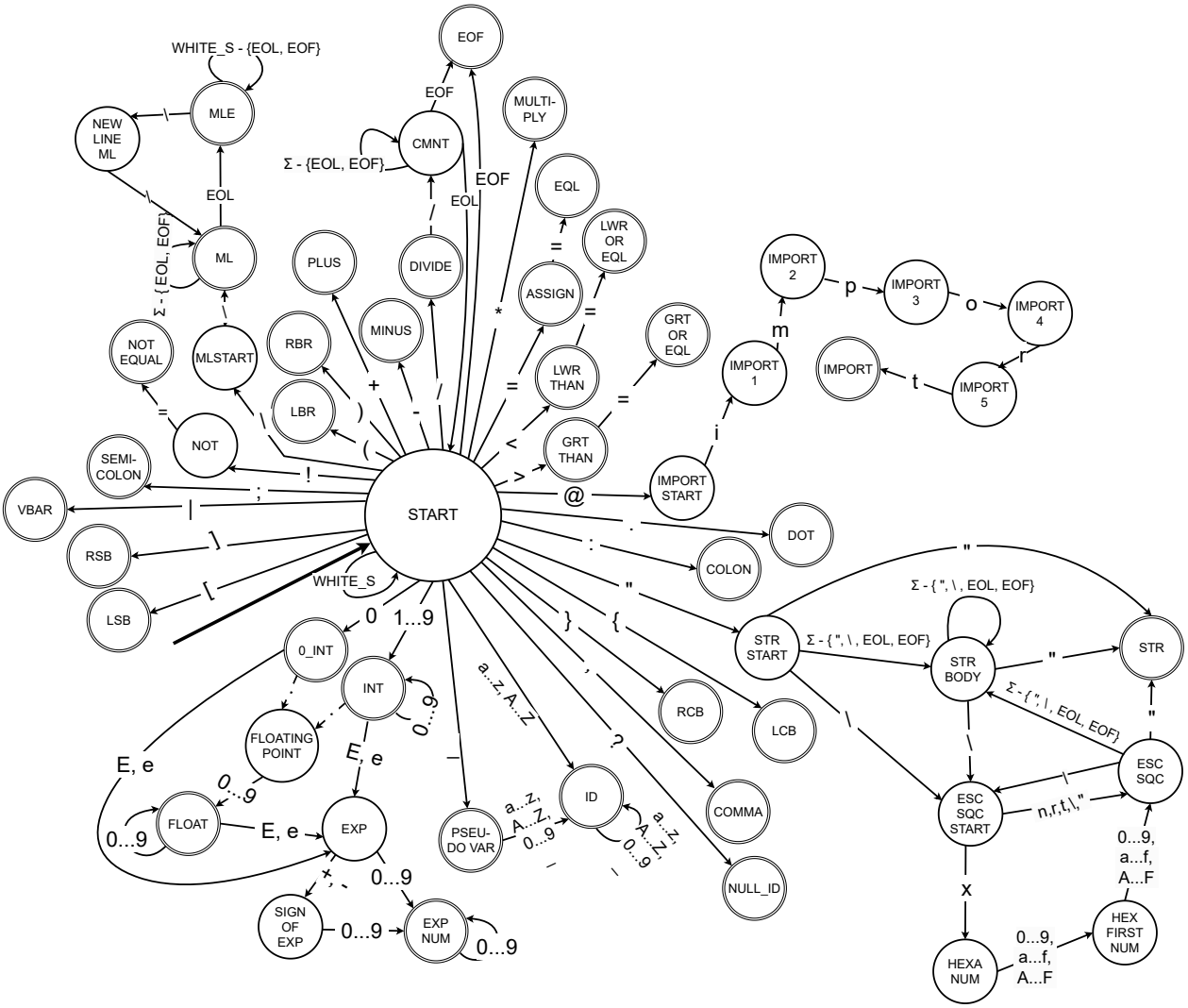
10 FUNEXP

If a token of type `id` is encountered as input during precedence syntactic analysis, it is first checked whether it represents a variable or a constant. If not, it is verified whether it might be a function call. For a token to be considered a function call, the next token must be of type `lbracket` (left bracket) or `dot` (period). In such case, the function `funCallHandle()` is invoked.

This approach was chosen due to the existence of the `funCallHandle()` function, which the parser already uses to process function calls in the program. A **valid** function call is further processed in the expression as a type `ID`. This reuse efficiently and effectively handles function calls as well as expressions as function call parameters [10]. Maximum number of parameters for user defined functions is set to 300 (`MAX_PARAM_NUM` in `symtable.h`). This number should be more than enough and reallocation is therefore not needed.

[10] See rule 29. in LL(1) grammar (2).

Attachments



Legend:

- Σ = all printable ASCII characters $\cup$ `WHITE_S`
- `WHITE_S` = {`EOL`, `EOF`, `TAB`, `SPACE`}

Figure 1: Deterministic finite automaton for lexical analysis

1. <prog> → <prolog> <code>
2. <prolog> → const id = @import (expression) ;
3. <code> → <def_func> <code>
4. <code> → ε
5. <def_func> → pub fn id (<params>) <type_func_ret> { <body>}
6. <params> → id : <type> <params_n>
7. <params> → ε
8. <params_n> → , <params>
9. <params_n> → ε
10. <def_variable> → <varorconst> id <type_var_def> = expression ;
11. <varorconst> → const
12. <varorconst> → var
13. <unused_decl> → _ = expression ;
14. <type_normal> → i32
15. <type_normal> → f64
16. <type_normal> → [ ] u8
17. <type_null> → ? <type_normal>
18. <type> → <type_normal>
19. <type> → <type_null>
20. <type_func_ret> → <type>
21. <type_func_ret> → void
22. <type_var_def> → : <type>
23. <type_var_def> → ε
24. <expr_params> → expression <expr_params_n>
25. <expr_params> → ε
26. <expr_params_n> → , <expr_params>
27. <expr_params_n> → ε
28. <after_id> → = expression ;
29. <after_id> → <builtin> (<expr_params>) ;
30. <assign_or_f_call> → id <after_id>
31. <builtin> → . id
32. <builtin> → ε
33. <st> → <def_variable>
34. <st> → <assign_or_f_call>
35. <st> → <unused_decl>
36. <st> → <while_statement>
37. <st> → <if_statement>
38. <st> → <return>
39. <body> → ε
40. <body> → <st> <body>
41. <return> → return <exp_func_ret>;
42. <exp_func_ret> → ε
43. <exp_func_ret> → expression
44. <id_without_null> → | id |
45. <id_without_null> → ε
46. <while_statement> → while (expression) <id_without_null> { <body>}
47. <if_statement> → if (expression) <id_without_null> { <body>} else { <body>}

Figure 2: LL(1) grammar

LL1 TABLE	i32	f64	const	var	pub	void	return	while	if	id	expression	=	(	)	{	}	[	|	?	;	,	:	.	_	$
<prog>			1																						
<prolog>			2																						
<code>					3																				4
<def_func>					5																				
<params>										6				7											
<params_n>														9							8				
<def_variable>			10	10																					
<varorconst>			11	12																					
<unused_decl>																								13	
<type_normal>	14	15															16								
<type_null>																			17						
<type>	18	18															18		19						
<type_func_ret>	20	20				21											20		20						
<type_var_def>												23										22			
<st>			33	33			38	36	37	34														35	
<body>			40	40			40	40	40	40						39								40	
<return>							41																		
<exp_func_ret>											43									42					
<id_without_null>															45			44							
<while_statement>								46																	
<if_statement>									47																
<expr_params>											24			25											
<expr_params_n>														27							26				
<after_id>												28	29										29		
<assign_or_f_call>										30															
<builtin>													32										31		

Figure 3: LL(1) table

1. E → i
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

Figure 4: Precedence analysis grammar

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

Figure 5: Precedence table

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

pub fn main() void {

    const number : i32 = 5;

    if(number == 9){
        ifj.write(number);
    }
    else{
        return;
    }

}
```

Figure 6: Simple program in IFJ24 language

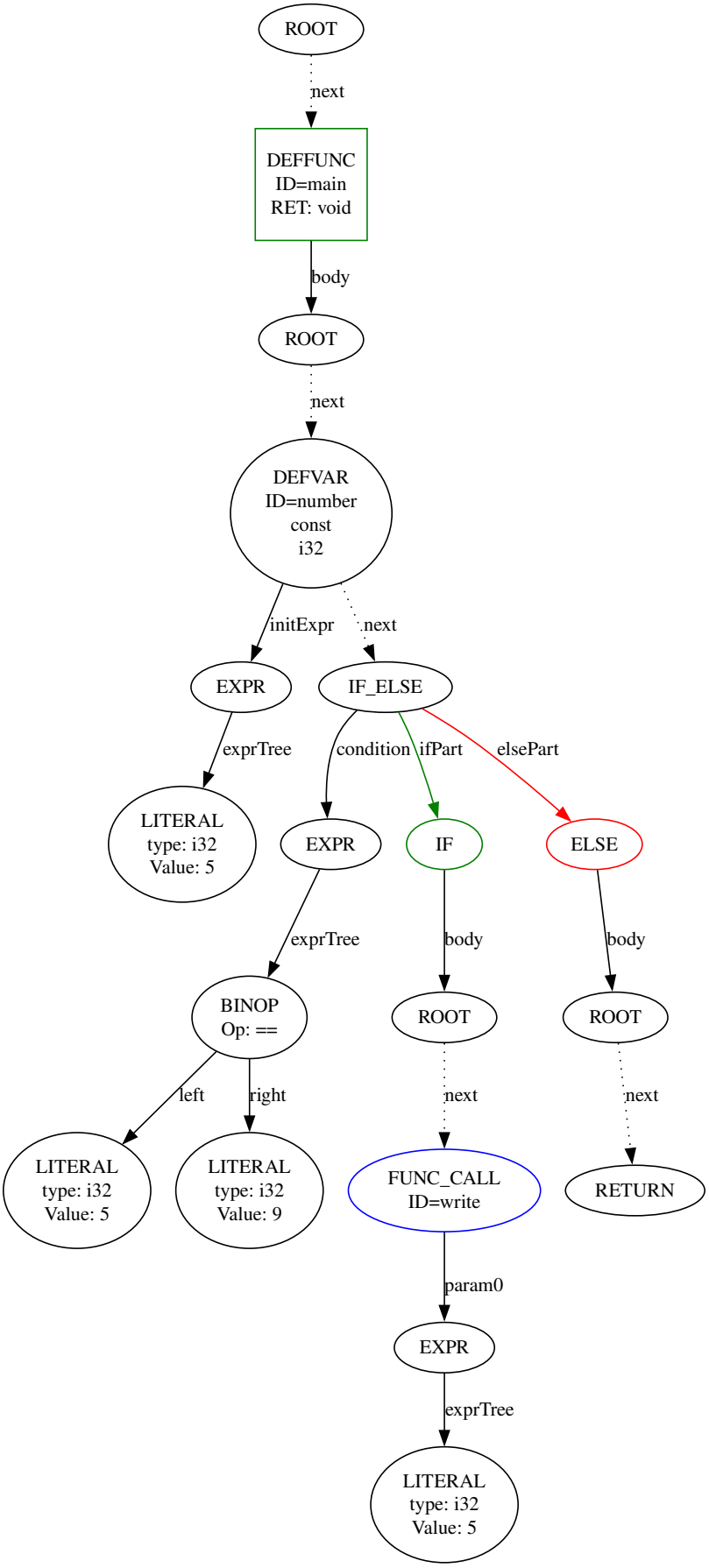


Figure 7: AST visualisation for program in figure 6