Implementation of the IFJ24 imperative language compiler

Team xpator00, variant TRP

December 4, 2024

Contents

1 Team Work Division

1.1 Point Distribution Justification

Martin Jursa had to give up his active participation in the third week of the project for personal reasons. His contribution was limited to the implementation of the token.c code, for which we originally intended to award one percent of the points. However, after mutual agreement with Martin and the whole team, his contribution was set to zero percent. We appreciate his initial involvement despite circumstances that prevented him from further participation. His initial twenty-five percent share was assigned to Daria Kinash, who completed his tasks and replaced his position on the team for the remainder of the project implementation.

1.2 Project Organization

The project was divided into four main components:

- Lexical Analysis
- Syntactic Analysis
- Semantic Analysis
- Code Generation

Team coordination was maintained through:

- Weekly online meetings every Sunday
- Task distribution and progress review
- Collaborative problem-solving sessions

We allocated three weeks for each project component, maintaining this schedule until the trial submission phase required additional debugging and optimization work. Despite Martin's withdrawal, the team successfully adapted, with Daria Kinash taking on additional responsibilities to ensure project continuity.

1.3 Individual Contributions

- Lexikal analysis, syntactic analysis, semantic analysis, code generator
- Fixing errors
- Lexikal analysis, syntactic analysis, semantic analysis, code generator
- Additional codes for each part
- Fixing errors

- Additional codes for each part
- Project organisation and structure
- Testing for each part and fixing errors
- Documentation and presentation

2 Implementation

The implementation of this project is divided into several key components, each focusing on a specific stage of the compilation process. From lexical analysis to code generation, the design ensures efficient and logical handling of source code. Below is a detailed explanation of each phase.

2.1 Lexical Analysis

Lexical analysis in this project is implemented in the scanner.c and scanner.h, which use a finite state machine (FSM) designed by the team. The FSM is implemented using a switch-case structure, where each case corresponds to a specific state of the automaton. The parser interacts with the scanner by calling get_next_token() to retrieve tokens for syntactic analysis. The get_next_token() function reads characters from a temporary file, where the source code from stdin is stored. The FSM processes these characters to identify lexemes and convert them into tokens. Tokens are implemented using the token_t structure, which is defined in token.c, token.h. This structure contains a type representing the token type and an attribute for additional information, such as a literal value. The FSM also includes logic for processing single-line and multi-line strings with escape sequences, converting them to ASCII values, and ignoring single-line comments. The lexical analyzer ensures efficient tokenization of the source code to facilitate further processing by the parser.

2.2 Syntactic Analysis

Syntactic analysis in the project is implemented in the parser.c and parser.h, following an LL grammar designed by the team for the IFJ24 language. The parser performs a **double pass** over the source code. During the **first pass**, it processes only the headers of function definitions, ensuring they are added to the symbol table with their names, parameters, and return types. This allows the parser to recognize all functions before analyzing their bodies or handling function calls. The **second pass**performs a detailed analysis of the entire codebase, validating all constructs, including variable declarations, control flow statements, and etc. Parsing in parser.c follows a top-down approach, beginning with high-level constructs and progressively descending into specific details.

Expression parsing is implemented in pars_expr.c and follows a bottom-up approach. It employs a parsing stack to manage expressions efficiently. The process relies on precedence parsing, utilizing a precedence table to define operator hierarchy and associativity. Grammar rules are applied during the reduction phase to validate and simplify expressions.

2.3 Semantic Analysis

Semantic analysis ensures the program follows logical rules and type requirements. The symbol table (symtable.c) keeps track of variables, constants, and functions in their scopes.

The semantic analyzer checks that types match correctly across the program. It ensures that operands in arithmetic, logical, and relational operations are compatible and applies conversions when needed, like turning integers into floats. The analyzer also flags unused variables and constants to help maintain clean and efficient code.

2.4 Code Generation

Code generation in this project translates the parsed program into **IFJcode24**, writing it directly to stdout. The process occurs immediately during parsing, producing code as the parser processes constructs.

The generator starts with a program header that defines global variables and initializes built-in functions. It handles function definitions, conditional statements, loops, and expressions by emitting the corresponding instructions.

A genStack is used to manage labels for control structures like if statements and loops, ensuring proper handling of nested constructs. Some parts of code generation, such as variable definitions, assignments, and operations, are handled directly in the parser, while others are implemented in generator.c.

2.5 Dynamic String (dstring_t)

The dynamic string was implemented to handle strings efficiently throughout the project, particularly in lexical analysis. Since token attributes like identifiers, keywords, and string literals can vary in length, a resizable string structure was necessary to avoid fixed size limitations.

2.6 Precedence Parsing Stack (Stack)

The precedence parsing stack was implemented to support the bottom-up parsing of expressions using precedence parsing. It facilitates the management of operators, operands, and precedence symbols during expression evaluation. This stack ensures that expressions are parsed correctly according to operator precedence and associativity rules.

2.7 Code Generator Stack (genStack)

The code generator stack is used to manage nested control structures like loops and conditional statements. It helps keep track of labels and flow control, ensuring that the correct labels are generated and reused for jumps and conditions in nested or recursive constructs.

2.8 Symbol Table(symtable_t)

The symbol table is implemented as a hash table using **separate chaining** with linked lists. This design allows efficient management of variables, constants, and functions, with attributes such as their type, scope level, and associated data.

For this project, separate chaining was chosen over open addressing because of its distinct advantages in managing symbols and scopes:

1. **Efficient Name-Based Lookups**: In a compiler, symbols are primarily accessed by their names during parsing and semantic analysis. Separate chaining allows quick lookups within a bucket, even when multiple symbols hash to the same value.

2. **Handling Collisions Gracefully**: When two symbols have the same hash value, they are linked within the same bucket. This ensures that all symbols with similar names can still be accessed efficiently without the need for complex probing mechanisms.

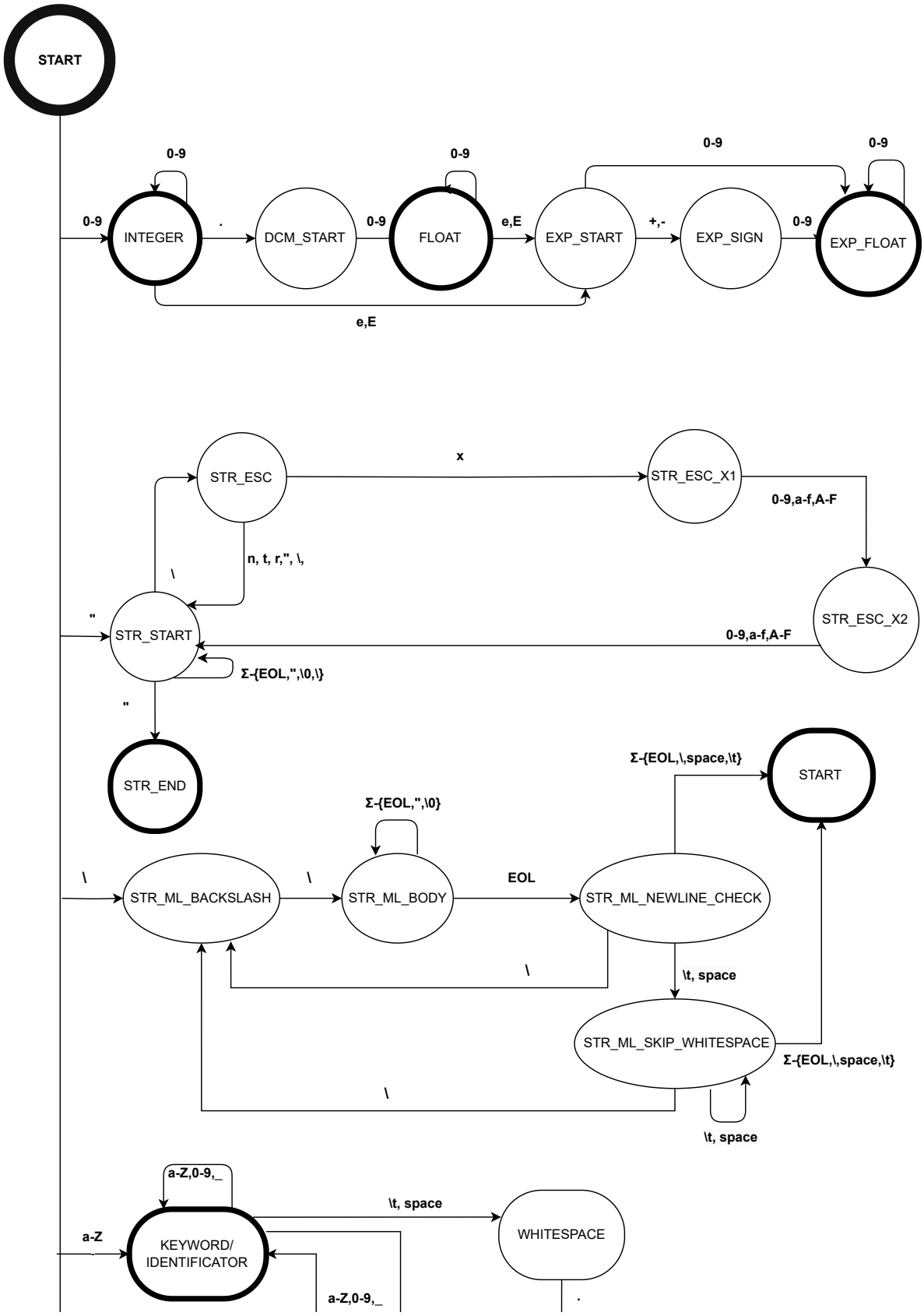
3. **Dynamic Addition and Deletion**: Separate chaining makes it straightforward to dynamically add or remove symbols. For example, adding a new function or variable involves appending it to the linked list in the relevant bucket

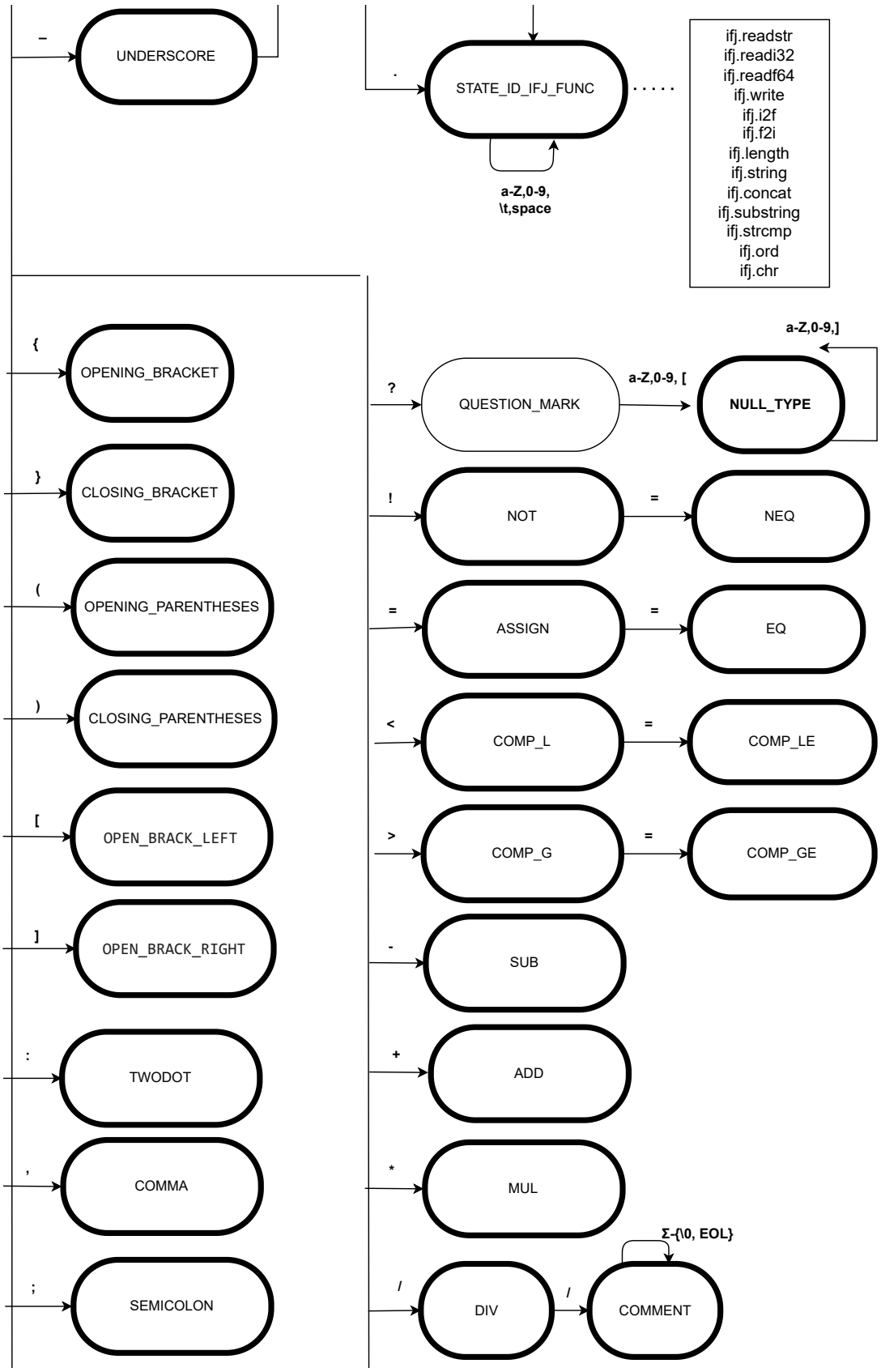
We chose to implement the symbol table using separate chaining with linked lists because it fits the needs of our project. This approach efficiently handles name-based lookups, manages collisions effectively, and allows for easy addition and removal of symbols. It ensures a simple and reliable way to manage variables, constants, and functions dynamically during compilation.

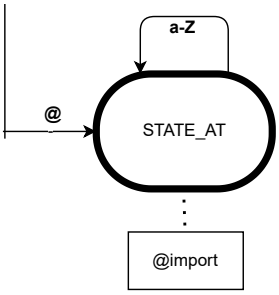
3 Structure of the Implementation Solution

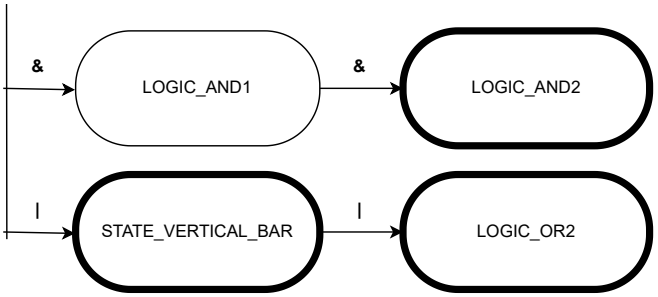
root
- **dstring.c** – Dynamic string operations for lexical analysis
- dstring.h – Dynamic string interface
- error_codes.c – Error handling system implementation
- error_codes.h – Error handling and reporting interface
- file.c – Input stream management system
- file.h – Input stream processing interface
- **generator.c** – Target code generation for IFJcode24
- generator.h – Code generation module interface
- main.c – Compiler entry point and initialization
- parse_expr.c – Expression parsing with precedence analysis
- parse_expr.h – Expression parsing module interface
- **parser.c** – Recursive descent parser implementation
- parser.h – Parser module interface
- prec_stack.c – Stack for precedence expression analysis
- prec_stack.h – Parser implementation
- prec_sym_types.h – Precedence analysis symbol types
- **scanner.c** – Lexical analyzer implementation
- scanner.h – Lexical analyzer interface
- stack.c – Stack operations for code generator
- stack.h – Generator stack interface
- **symtable.c** – Symbol table with TRP implementation
- symtable.h – Symbol table module interface
- token.c – Token operations implementation
- token.h – Token handling interface

4 Diagram of a Finite Automat


START
0-9
0-9
0-9
0-9
0-9
INTEGER
.
DCM_START
0-9
FLOAT
e,E
EXP_START
+,-
EXP_SIGN
0-9
EXP_FLOAT
e,E
STR_ESC
x
STR_ESC_X1
0-9,a-f,A-F
n, t, r,", \,
\
"
STR_START
0-9,a-f,A-F
STR_ESC_X2
Σ-{EOL,",\0,\}
"
STR_END
Σ-{EOL,\,space,\t}
START
Σ-{EOL,",\0}
\
STR_ML_BACKSLASH
\
STR_ML_BODY
EOL
STR_ML_NEWLINE_CHECK
\
\t, space
STR_ML_SKIP_WHITESPACE
Σ-{EOL,\,space,\t}
\
\t, space
a-Z,0-9,_
\t, space
a-Z
_
KEYWORD/
IDENTIFICATOR
WHITESPACE
a-Z,0-9,_
.


_
UNDERSCORE
.
STATE_ID_IFJ_FUNC
a-Z,0-9,
\t,space
ifj.readstr
ifj.readi32
ifj.readf64
ifj.write
ifj.i2f
ifj.f2i
ifj.length
ifj.string
ifj.concat
ifj.substring
ifj.strcmp
ifj.ord
ifj.chr
{
OPENING_BRACKET
}
CLOSING_BRACKET
(
OPENING_PARENTHESES
)
CLOSING_PARENTHESES
[
OPEN_BRACK_LEFT
]
OPEN_BRACK_RIGHT
:
TWODOT
,
COMMA
;
SEMICOLON
?
QUESTION_MARK
a-Z,0-9, [
NULL_TYPE
a-Z,0-9,]
!
NOT
=
NEQ
=
ASSIGN
=
EQ
<
COMP_L
=
COMP_LE
>
COMP_G
=
COMP_GE
-
SUB
+
ADD
*
MUL
/
DIV
/
COMMENT
Σ-{\0, EOL}


a-Z
@
STATE_AT
@import


&
LOGIC_AND1
&
LOGIC_AND2
|
STATE_VERTICAL_BAR
|
LOGIC_OR2

5 LL-grammar, LL-table and Precedence Table

5.1 LL - grammar

1. <program> ::= <prologue> <module_body>
2. <prologue> ::= const ifj = @ import ("ifj24.zig") ;
3. <module_body> ::= <module_section> <module_body>
4. <module_body> ::= eps
5. <module_section> ::= pub fn <function_name> (<parameters>) <return_type> { <statements> }
6. <function_name> ::= main
7. <function_name> ::= <id>
8. <parameters> ::= <parameter> <parameters_n>
9. <parameters> ::= eps
10. <parameters_n> ::= , <parameter> <parameters_n>
11. <parameters_n> ::= eps
12. <parameter> ::= <id> : <type>
13. <return_type> ::= <type>
14. <return_type> ::= void
15. <statements> ::= <statement> <statements_n>
16. <statements_n> ::= <statement> <statements_n>
17. <statements_n> ::= eps
18. <statement> ::= <var_decl>
19. <statement> ::= <const_decl>
20. <statement> ::= <id_statement>
21. <statement> ::= <conditional>
22. <statement> ::= <cycle>
23. <statement> ::= <return_statement>
24. <statement> ::= _ = <expression> ;
25. <id_statement> ::= <id> <id_statement_tail>
26. <id_statement_tail> ::= = <expression> ;
27. <id_statement_tail> ::= (<arguments>)
28. <var_decl> ::= var <id> <type_annot> = <expression> ;

29. <const_decl> ::= const <id> <type_annot> = <expression> ;
30. <type_annot> ::= : <type>
31. <type_annot> ::= eps
32. <conditional> ::= if (<cond_content>) <cond_content_tail>
33. <cond_content> ::= <expression>
34. <cond_content_tail> ::= <reg_block>
35. <cond_content_tail> ::= <binding_part> <reg_block>
36. <reg_block> ::= { <statements> } <else_block>
37. <binding_part> ::= | <id> |
38. <else_block> ::= else { <statements> }
39. <cycle> ::= while (<cycle_content>) <cycle_content_tail>
40. <cycle_content> ::= <expression>
41. <cycle_content_tail> ::= <while_reg_block>
42. <cycle_content_tail> ::= <binding_part> <while_reg_block>
43. <while_reg_block> ::= { <statements> }
44. <return_statement> ::= return <return_value>
45. <return_value> ::= <expression> ;
46. <return_value> ::= ;
47. <type> ::= i32
48. <type> ::= f64
49. <type> ::= []u8
50. <type> ::= ?i32
51. <type> ::= ?f64
52. <type> ::= ?[]u8
53. <expression> ::= <term> <expression_tail>
54. <expression_tail> ::= <operator> <term> <expression_tail>
55. <expression_tail> ::= eps
56. <term> ::= <literal>
57. <term> ::= <id_term>
58. <term> ::= (<expression>)

59. <id_term> ::= <id> <id_term_tail>
60. <id_term_tail> ::= (<arguments>)
61. <id_term_tail> ::= eps
62. <literal> ::= <int_literal>
63. <literal> ::= <float_literal>
64. <literal> ::= <string_literal>
65. <literal> ::= null
66. <operator> ::= +
67. <operator> ::= -
68. <operator> ::= *
69. <operator> ::= /
70. <operator> ::= ==
71. <operator> ::= !=
72. <operator> ::= <
73. <operator> ::= >
74. <operator> ::= <=
75. <operator> ::= >=
76. <operator> ::= !
77. <operator> ::= &&
78. <operator> ::= ||
79. <arguments> ::= <expression> <arguments_n>
80. <arguments> ::= eps
81. <arguments_n> ::= , <expression> <arguments_n>
82. <arguments_n> ::= eps

5.2 Precedence Table

	+	−	*	/	==	! =	!	<	>	<=	>=	&&	||	(	)	i	$
+	>	>	<	<	>	>	<	>	>	>	>	>	>	<	>	<	>
−	>	>	<	<	>	>	<	>	>	>	>	>	>	<	>	<	>
*	>	>	>	>	>	>	<	>	>	>	>	>	>	<	>	<	>
/	>	>	>	>	>	>	<	>	>	>	>	>	>	<	>	<	>
==	<	<	<	<			<					>	>	<	>	<	>
! =	<	<	<	<			<					>	>	<	>	<	>
!	>	>	>	>	>	>	<	>	>	>	>	>	>	<	>	<	>
<	<	<	<	<			<					>	>	<	>	<	>
>	<	<	<	<			<					>	>	<	>	<	>
<=	<	<	<	<			<					>	>	<	>	<	>
>=	<	<	<	<			<					>	>	<	>	<	>
&&	<	<	<	<	<	<	<	<	<	<	<	>	>	<	>	<	>
||	<	<	<	<	<	<	<	<	<	<	<	<	>	<	>	<	>
(	<	<	<	<	<	<	<	<	<	<	<	<	<	<	=	<	
)	>	>	>	>	>	>		>	>	>	>	>	>		>		>
i	>	>	>	>	>	>		>	>	>	>	>	>		>		>
$	<	<	<	<	<	<	<	<	<	<	<	<	<	<		<	

5.3 LL - table

	<arguments>	<arguments_n>	<binding_part>	<cond_content>	<cond_content_tail>	<conditional>	<const_decl>	<cycle>	<cycle_content>	<cycle_content_tail>	<else_block>	<expression>	<expression_tail>	<function_name>	<id_statement>	<id_statement_tail>	<id_term>	<id_term_tail>	<literal>	<module_body>	<module_section>	<operator>	<parameter>	<parameters>	<parameters_n>	<program>	<prologue>	<reg_block>	<return_statement>	<return_type>	<return_value>	<statement>	<statements>	<statements_n>	<term>	<type>	<type_annot>	<var_decl>	<while_reg_block>
!													54					61				76																	
! =													54					61				71																	
&&													54					61				77																	
(	79			33					40			53				27		60												45				58					
)	80	82											55					61						9	11														
*													54					61				68																	
+													54					61				66																	
,		81																							10														
−													54					61				67																	
/													54					61				69																	
:																																				30		
;													55					61												46									
<													54					61				72																	
<=													54					61				74																	
<float_lit>	79			33					40			53							63											45				56					
<id>	79			33					40			53		7	25		59						12	8						45	20	15	16	57					
<int_lit>	79			33					40			53							62											45				56					
<string_lit>	79			33					40			53							64											45				56					
=																26																				31		
==													54					61				70																	
>													54					61				73																	
>=													54					61				75																	
?[]u8																														13					52				
?f64																														13					51				
?i32																														13					50				
[]u8																														13					49				
-																																24	15	16					
const							29																			1	2					19	15	16					
else											38																													
f64																														13					48				
i32																														13					47				
if						32																										21	15	16					
main														6																									
null	79			33					40			53							65											45				56					
pub																				3	5																	
return																													44			23	15	16					
var																																18	15	16				28	
void																														14									
while								39																								22	15	16					
{					34					41																	36											43
|			37		35					42																													
||													54					61				78																	
}																																		17					
$																				4																			