Task description:

Program: Main class of Java language. Descriptions of functions with parameters are allowed.

Types data : short, long, boolean.

Operations: simple arithmetic, comparisons and logical.

Operators: assignments and do-while

Operands: simple variables, constants.

Constants: integer at 10 (decimal), integer at HEX, boolean.

Data Types (Sun JVM)

Type	*Range of values*	*Memory length*
short	from –32768 up to 32767	2 bytes
long	from –9 223 372 036 854 775 808 to 9 223 372 036 854 775 807	8 bytes
boolean	true or false	4 bytes

Type conversion table (Sun JVM)

Operand	*Operand*	*Result*
short	short	long
short	long	long
long	short	long
long	long	long
boolean	boolean	boolean
short	boolean	Error
boolean	short	Error
long	boolean	Error
boolean	long	Error

Description of context-sensitive grammar :

<program> → class <class Main >

<class Main > → Main {<class content>};

<class content> → <class content > <description> | <description>

<description> → <data> |
<function>

<data> → <data type> <list of variables> ;

<data type> → <int type> | boolean

<type int> → short | long

<variable list> → <variable list>, <variable> | <variable>

<variable> → <identifier> | <identifier> = <expression>

<constant> → <integer constant> |

<boolean constant> |

<HEX-constant> |

<integer constant> → <integer constant> <digit> | <digit>

<boolean constant> → true | false

<hexadecimal constant> → 0 X <hexadecimal number> | 0 x <hexadecimal number>

<hexadecimal number> → <hexadecimal number> <digit> |

<hexadecimal number> <hexadecimal letters> |

<digit> | <hexadecimal letters>

<16-case letters > → a | b | c | d | f | A | B | C | D | F

<function> → <data type>

<identifier> (<variables for describing the function>)

<compound operator>

<variables for describing a function> → <variables for describing a function>,

<data type> <identifier> |

<data type> <identifier>

<compound operator> → {<operators and descriptions>}

<statements and descriptions> → <statements and descriptions> <data> |

<operators and descriptions > operator |

ε

< operator > → <assignment>; |
<compound operator>; |
<function call>; |
<do-while loop >; |
return < expression >;

< assignment > → < name > = < expression>;

<name> → <id>

< function call> → <identifier> (<list of expressions>);

<list of expressions> → <list of expressions>, <expression> |
<expression>

<cycle do - while> → do <compound statement> while (<expression>)

<expression> → <expression> || <Log AND> | <Log AND>

<Log AND> → <Log AND> && <XOR> | <XOR>

<XOR> → <XOR> ^ <Equality> | <Equality>

<Equality> → <Equality> == <Comparison> |

<Equality> ! = <Comparison> |

<Comparison>

<Comparison> → <Comparison> < <Command> |

<Comparison> > <Term> |

<Comparison> < = <Component> |

< Comparison > > = <Command> |

<Component>

< term > → <term> + <factor> |

<term> - <multiplier> |

<multiplier>

<multiplier> → <multiplier> * <prefix> |

<multiplier> / <prefix> |

<multiplier> % <prefix> |

<prefix>

<prefix> → ++<postfix> |

-- <postfix> |

!<postfix> |

<postfix>

<postfix> → <elementary expression>++ |

<elementary expression>-- |

<elementary expression>

<elementary expression> → <name> |

<constant> |

(<expression>) | <function call>

<identifier> → <letter> <ending>

< ending > → <ending><letter> |

< ending >< number > |

ε

< letter > → | a | b | c | d | e | ... | z | A | B | C | D | E |...| Z |

<digit> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

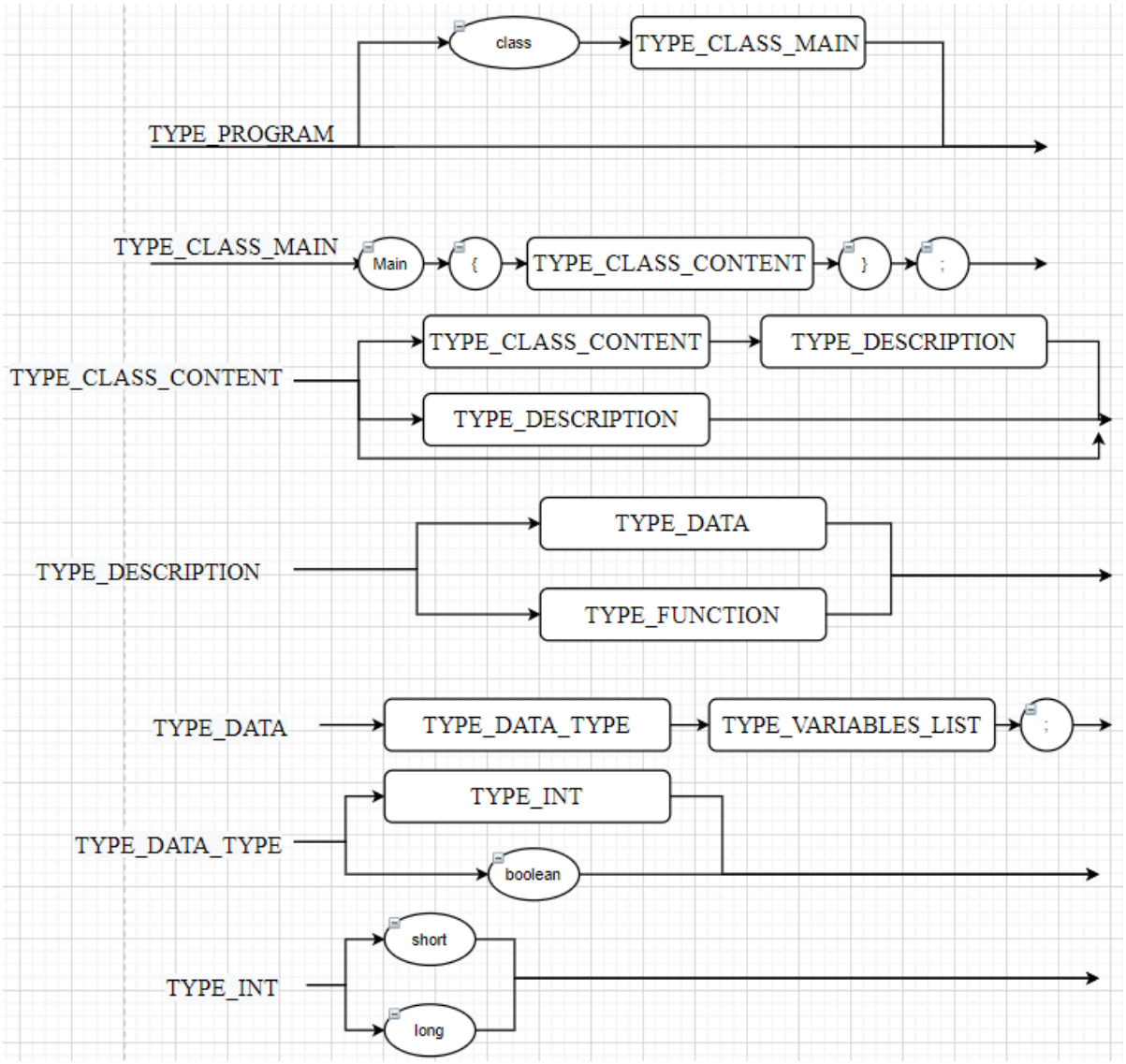
Notation of non-terminal symbols

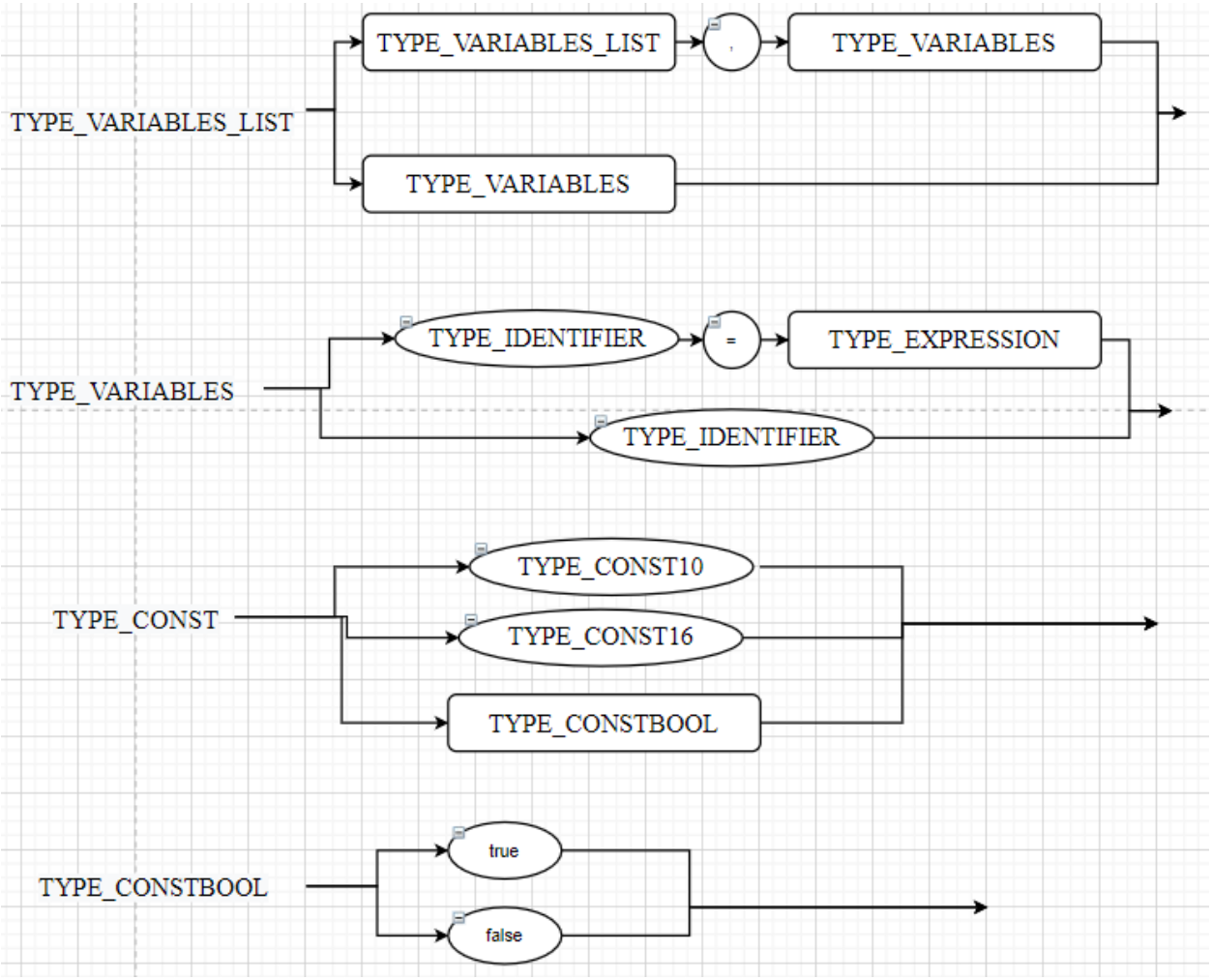
Non-terminal symbol	Designation

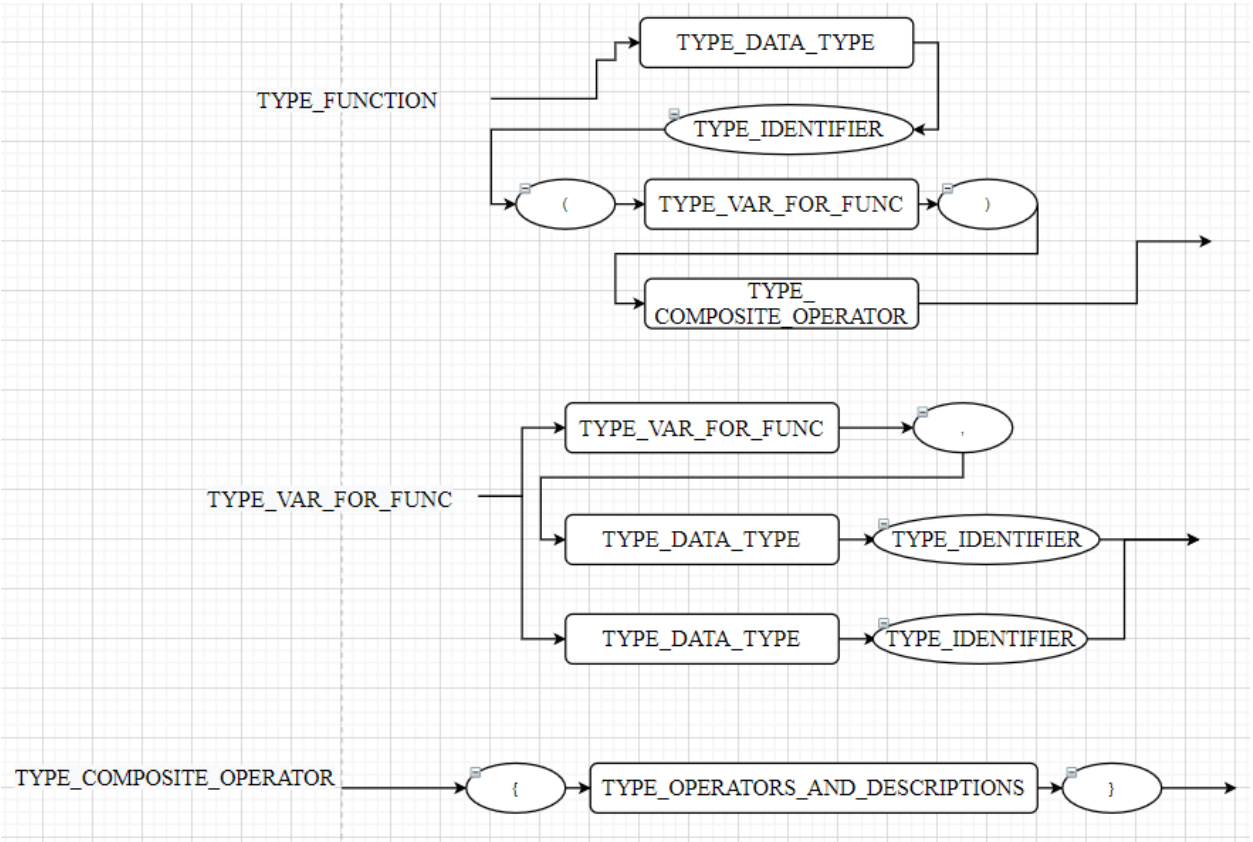
Program	TYPE_PROGRAM
Main class	TYPE_CLASS_MAIN
Class content	TYPE_CLASS_CONTENT
Description	TYPE_DESCRIPTION
Data	TYPE_DATA
Data type	TYPE_DATA_TYPE
Type int	TYPE_INT
List of variables	TYPE_VARIABLES_LIST
Variable	TYPE_VARIABLES
Constant	TYPE_CONST
Boolean constant	TYPE_LOG_CONST
Function	TYPE_FUNCTION
Variables to describe a function	TYPE_VAR_FOR_FUNC
Compound operator	TYPE_COMPOSITE_OPERATOR
Operators and descriptions	TYPE_OPERATORS_AND_DESCRIPTIONS
Operator	TYPE_OPERATOR
Assignment	TYPE_ASSIGNMENT
Calling a function	TYPE_FUNCTION_CALL
List of Expressions	TYPE_LIST_OF_EXPRESSION
do-while loop	TYPE_LOOP
Expression	TYPE_EXPRESSION
Log I	TYPE_LOG_AND
XOR	TYPE_EXC_OR
Equality	TYPE_EQUALITY

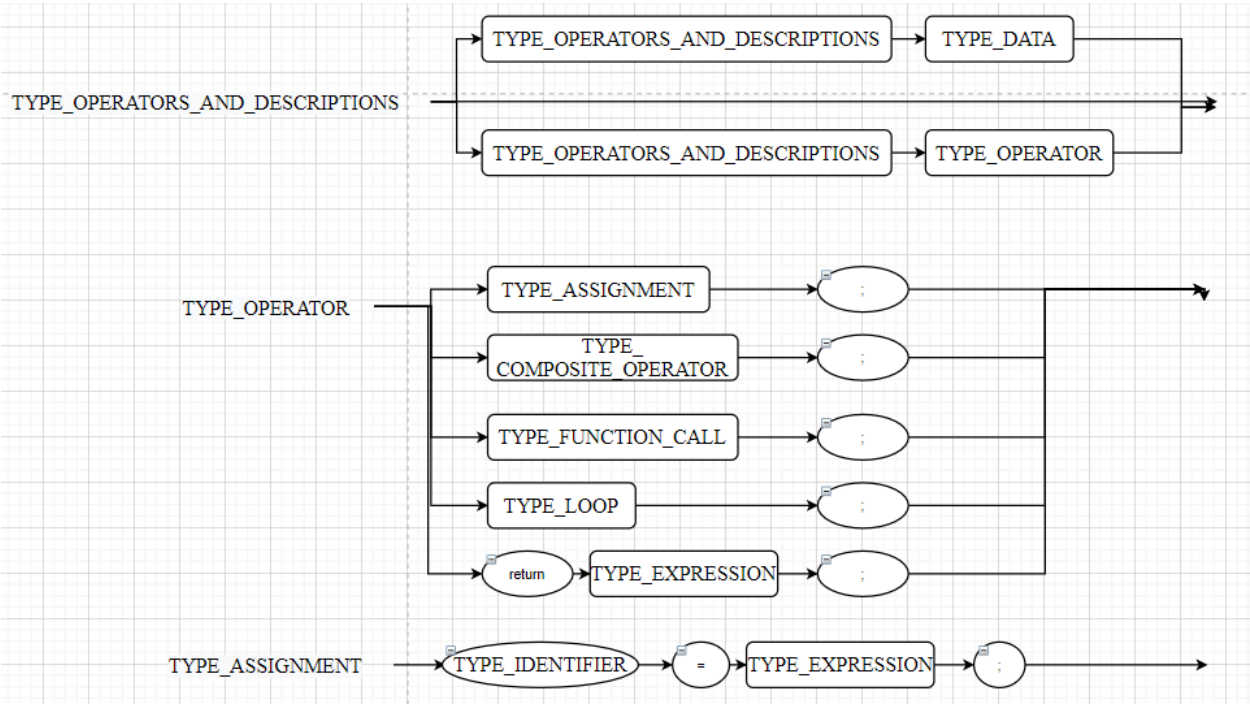
Comparison	TYPE_COMPARISON
Term	TYPE_ADDEND
Factor	TYPE_MULTIPLIER
Prefix	TYPE_PREFIX
Elementary expression	TYPE_ELEMENTERY_EXPRESSION

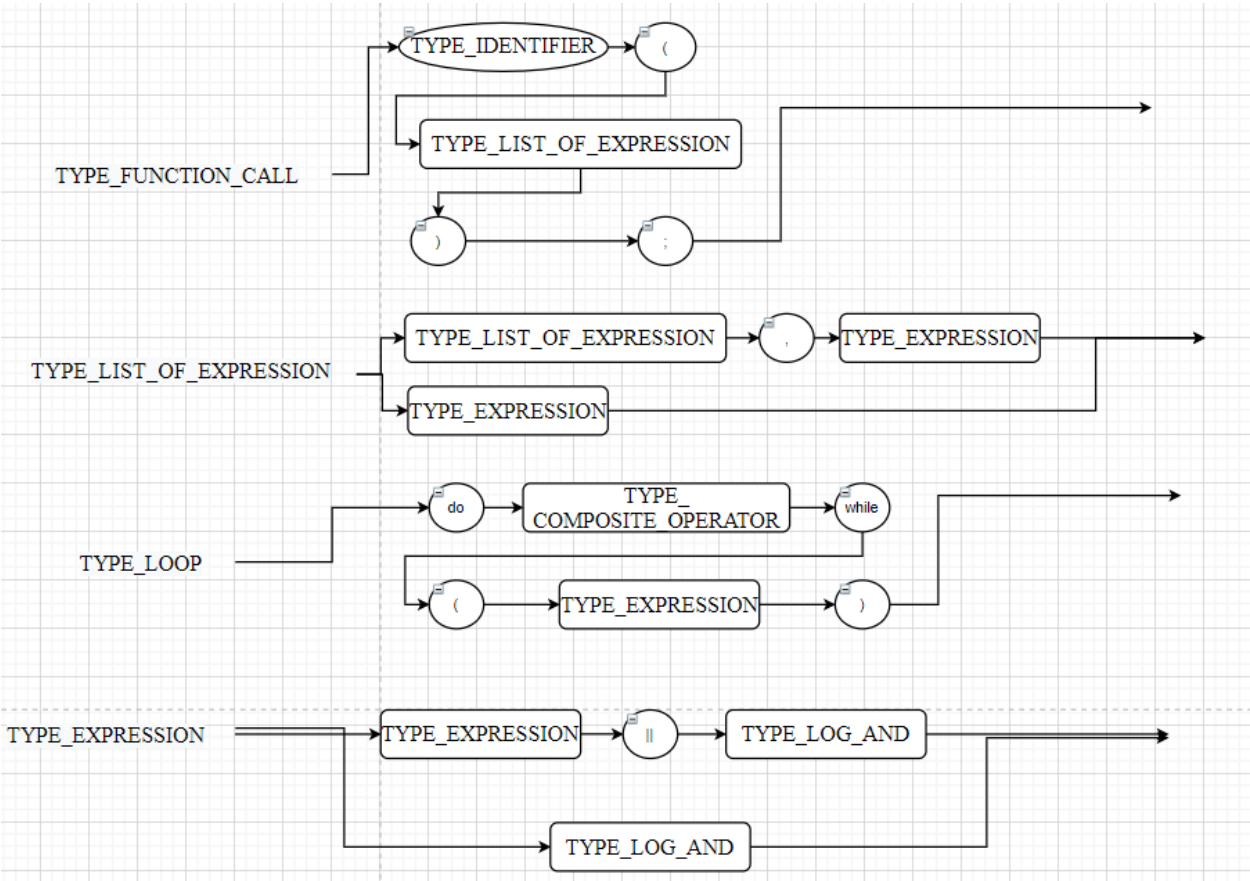
Syntactic diagrams of context-sensitive grammar:

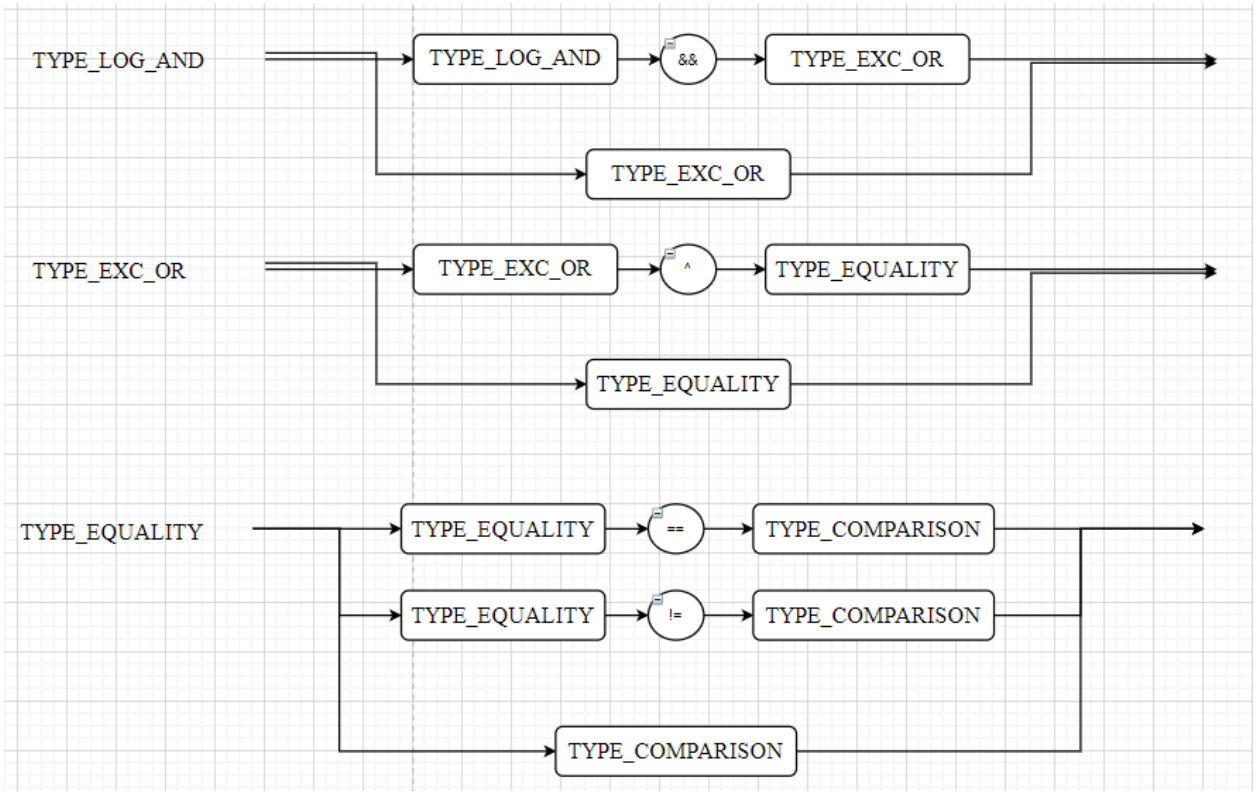


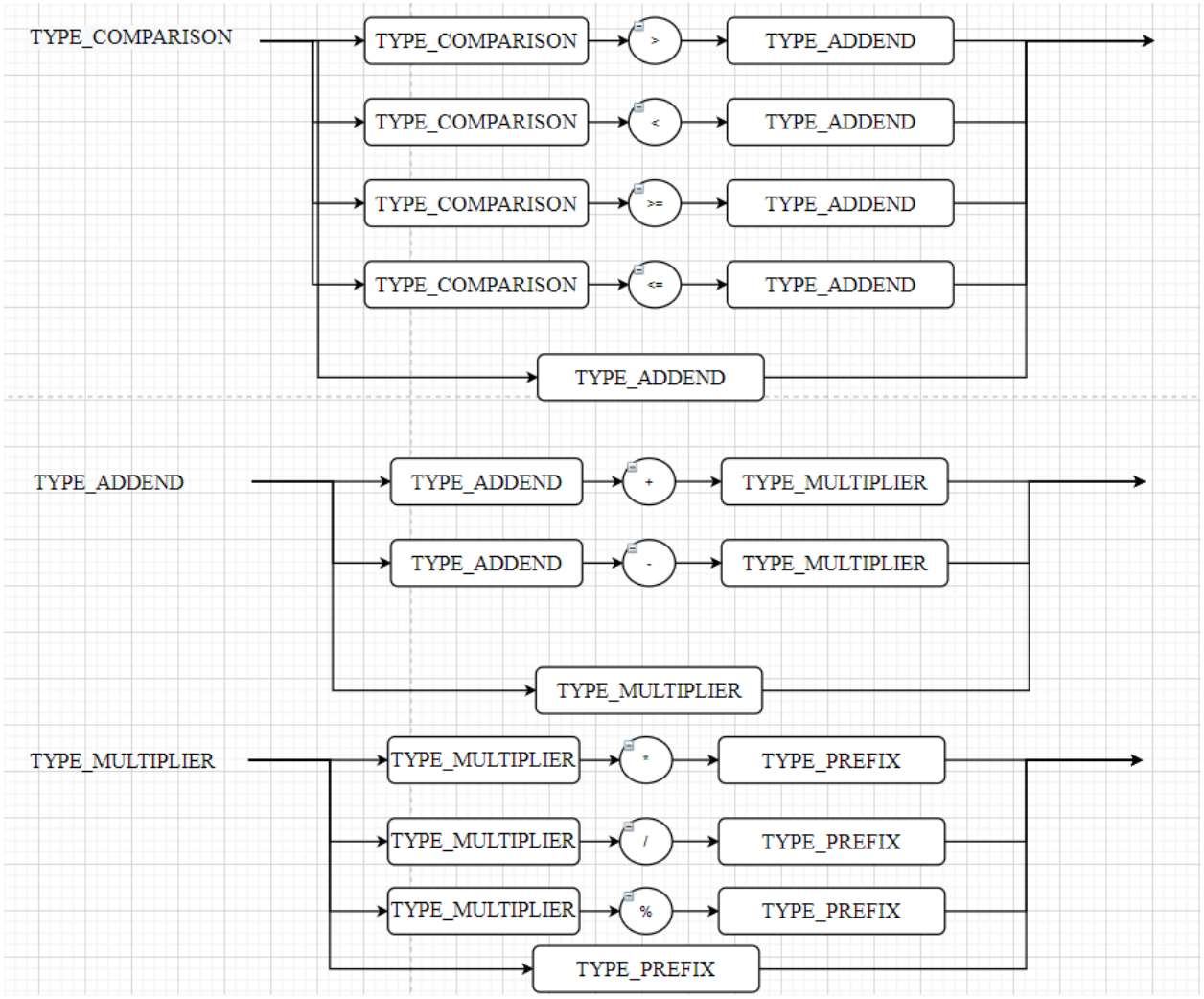

TYPE_VARIABLES_LIST
TYPE_VARIABLES_LIST
,
TYPE_VARIABLES
TYPE_VARIABLES
TYPE_VARIABLES
TYPE_IDENTIFIER
=
TYPE_EXPRESSION
TYPE_IDENTIFIER
TYPE_CONST
TYPE_CONST10
TYPE_CONST16
TYPE_CONSTBOOL
TYPE_CONSTBOOL
true
false

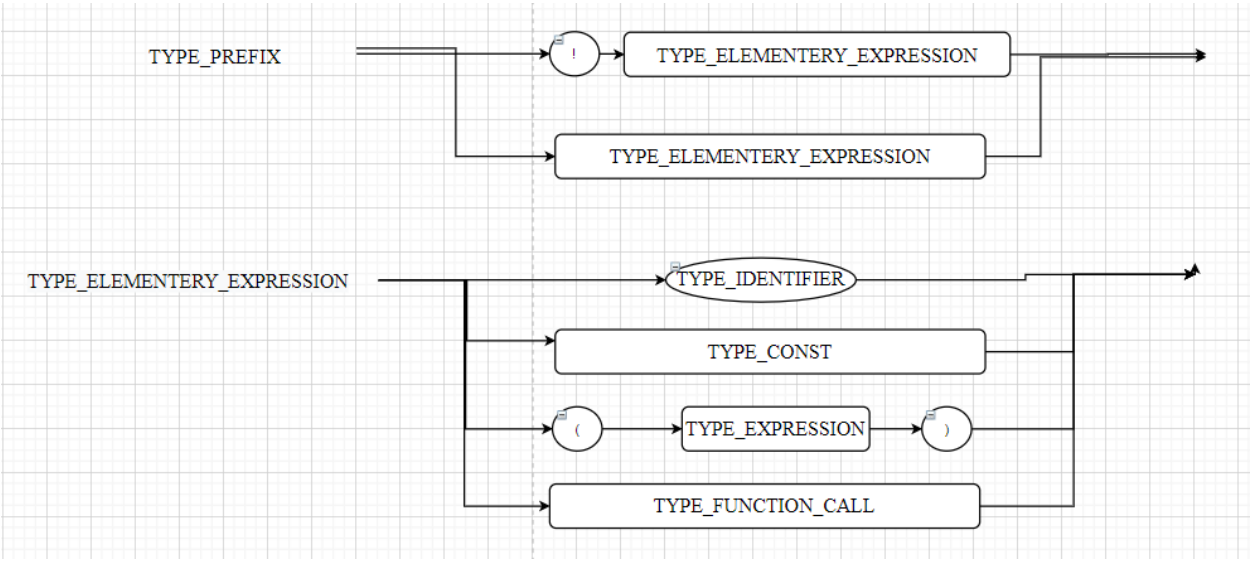

TYPE_FUNCTION
TYPE_DATA_TYPE
TYPE_IDENTIFIER
(
TYPE_VAR_FOR_FUNC
)
TYPE_
COMPOSITE_OPERATOR
TYPE_VAR_FOR_FUNC
TYPE_VAR_FOR_FUNC
,
TYPE_DATA_TYPE
TYPE_IDENTIFIER
TYPE_DATA_TYPE
TYPE_IDENTIFIER
TYPE_COMPOSITE_OPERATOR
{
TYPE_OPERATORS_AND_DESCRIPTIONS
}


TYPE_OPERATORS_AND_DESCRIPTIONS
TYPE_OPERATORS_AND_DESCRIPTIONS
TYPE_DATA
TYPE_OPERATORS_AND_DESCRIPTIONS
TYPE_OPERATOR
TYPE_OPERATOR
TYPE_ASSIGNMENT
;
TYPE_
COMPOSITE_OPERATOR
;
TYPE_FUNCTION_CALL
;
TYPE_LOOP
;
return
TYPE_EXPRESSION
;
TYPE_ASSIGNMENT
TYPE_IDENTIFIER
=
TYPE_EXPRESSION
;

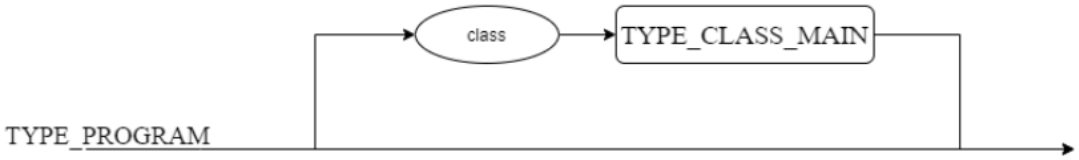

TYPE_IDENTIFIER
(
TYPE_FUNCTION_CALL
TYPE_LIST_OF_EXPRESSION
)
;
TYPE_LIST_OF_EXPRESSION
TYPE_LIST_OF_EXPRESSION
,
TYPE_EXPRESSION
TYPE_EXPRESSION
TYPE_LOOP
do
TYPE_
COMPOSITE_OPERATOR
while
(
TYPE_EXPRESSION
)
TYPE_EXPRESSION
TYPE_EXPRESSION
||
TYPE_LOG_AND
TYPE_LOG_AND

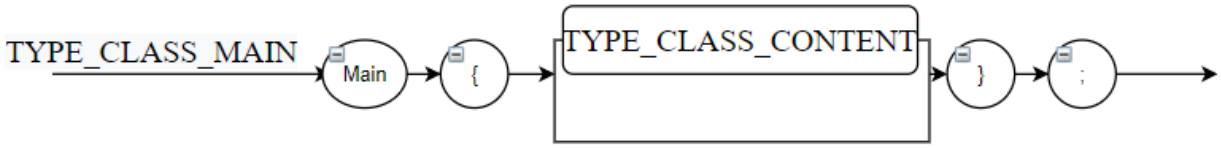

TYPE_LOG_AND
TYPE_LOG_AND
&&
TYPE_EXC_OR
TYPE_EXC_OR
TYPE_EXC_OR
TYPE_EXC_OR
^
TYPE_EQUALITY
TYPE_EQUALITY
TYPE_EQUALITY
TYPE_EQUALITY
==
TYPE_COMPARISON
TYPE_EQUALITY
!=
TYPE_COMPARISON
TYPE_COMPARISON

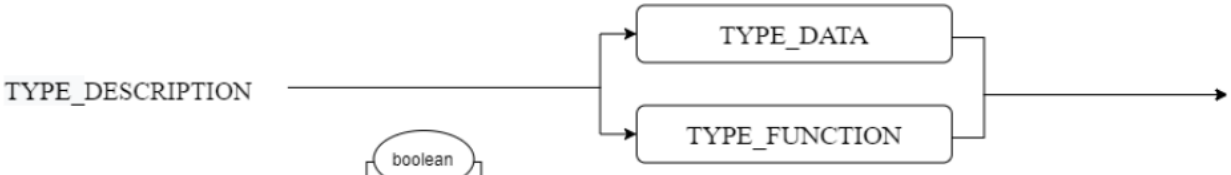

TYPE_COMPARISON
TYPE_COMPARISON
>
TYPE_ADDEND
TYPE_COMPARISON
<
TYPE_ADDEND
TYPE_COMPARISON
>=
TYPE_ADDEND
TYPE_COMPARISON
<=
TYPE_ADDEND
TYPE_ADDEND
TYPE_ADDEND
TYPE_ADDEND
+
TYPE_MULTIPLIER
TYPE_ADDEND
-
TYPE_MULTIPLIER
TYPE_MULTIPLIER
TYPE_MULTIPLIER
TYPE_MULTIPLIER
*
TYPE_PREFIX
TYPE_MULTIPLIER
/
TYPE_PREFIX
TYPE_MULTIPLIER
%
TYPE_PREFIX
TYPE_PREFIX

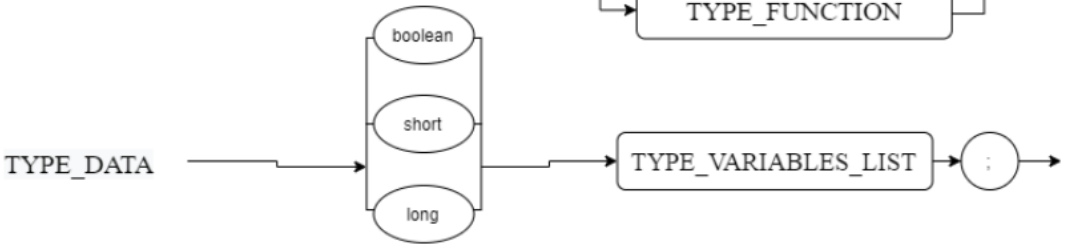


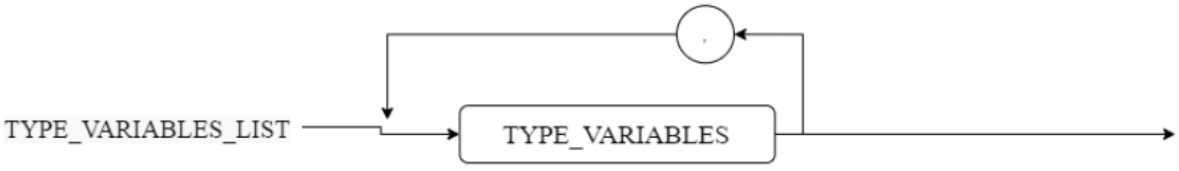
Optimized syntax diagrams

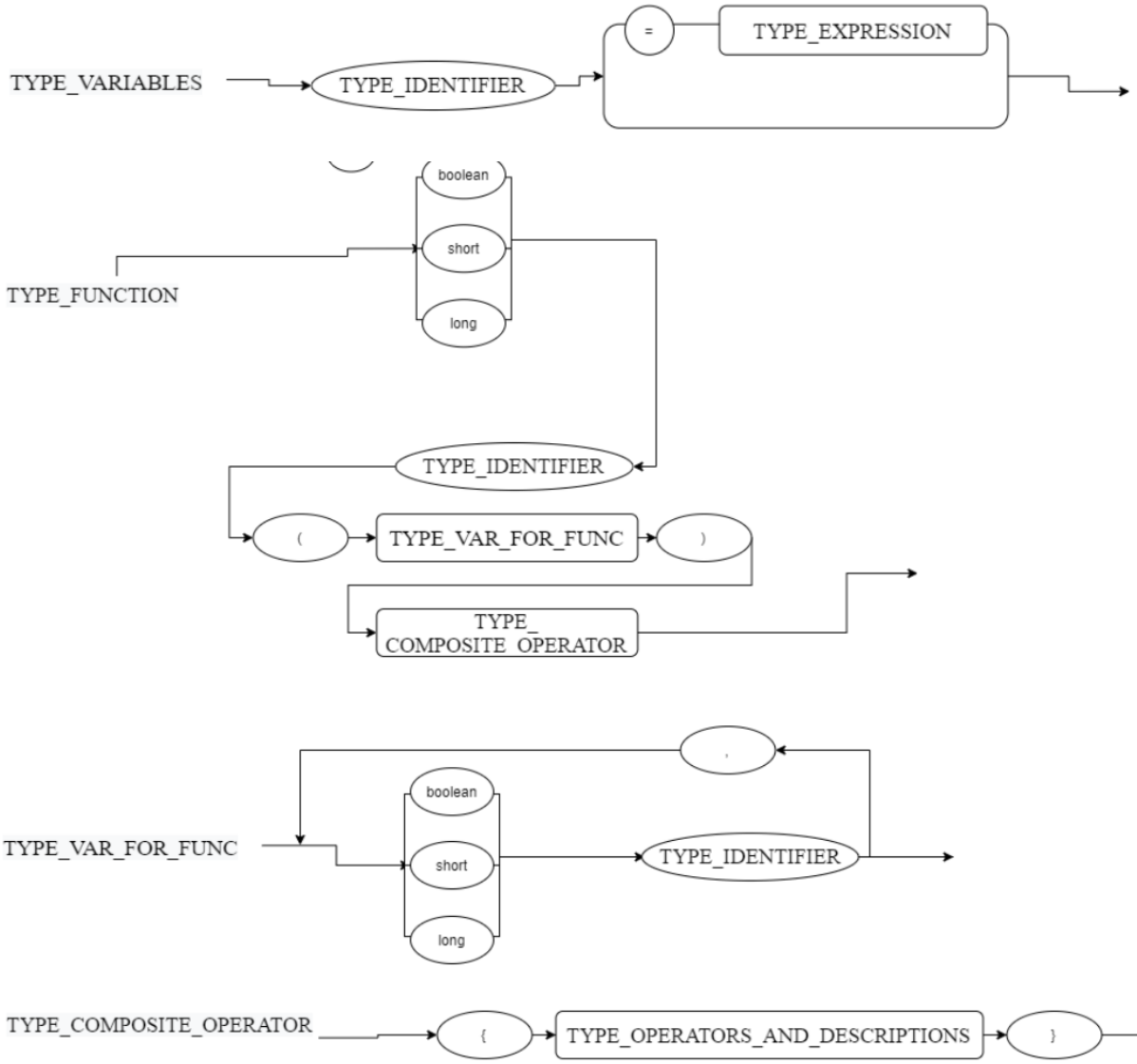


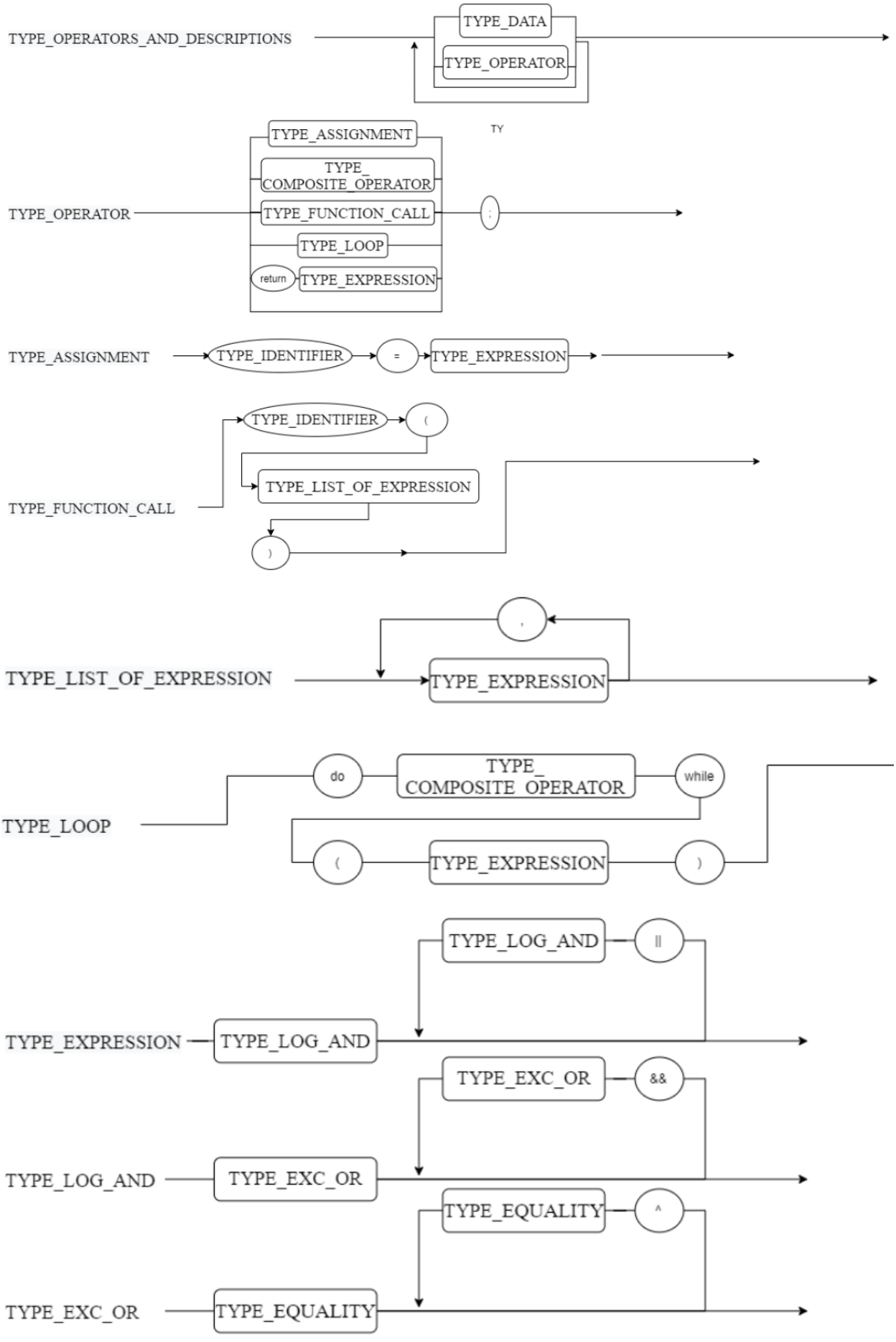


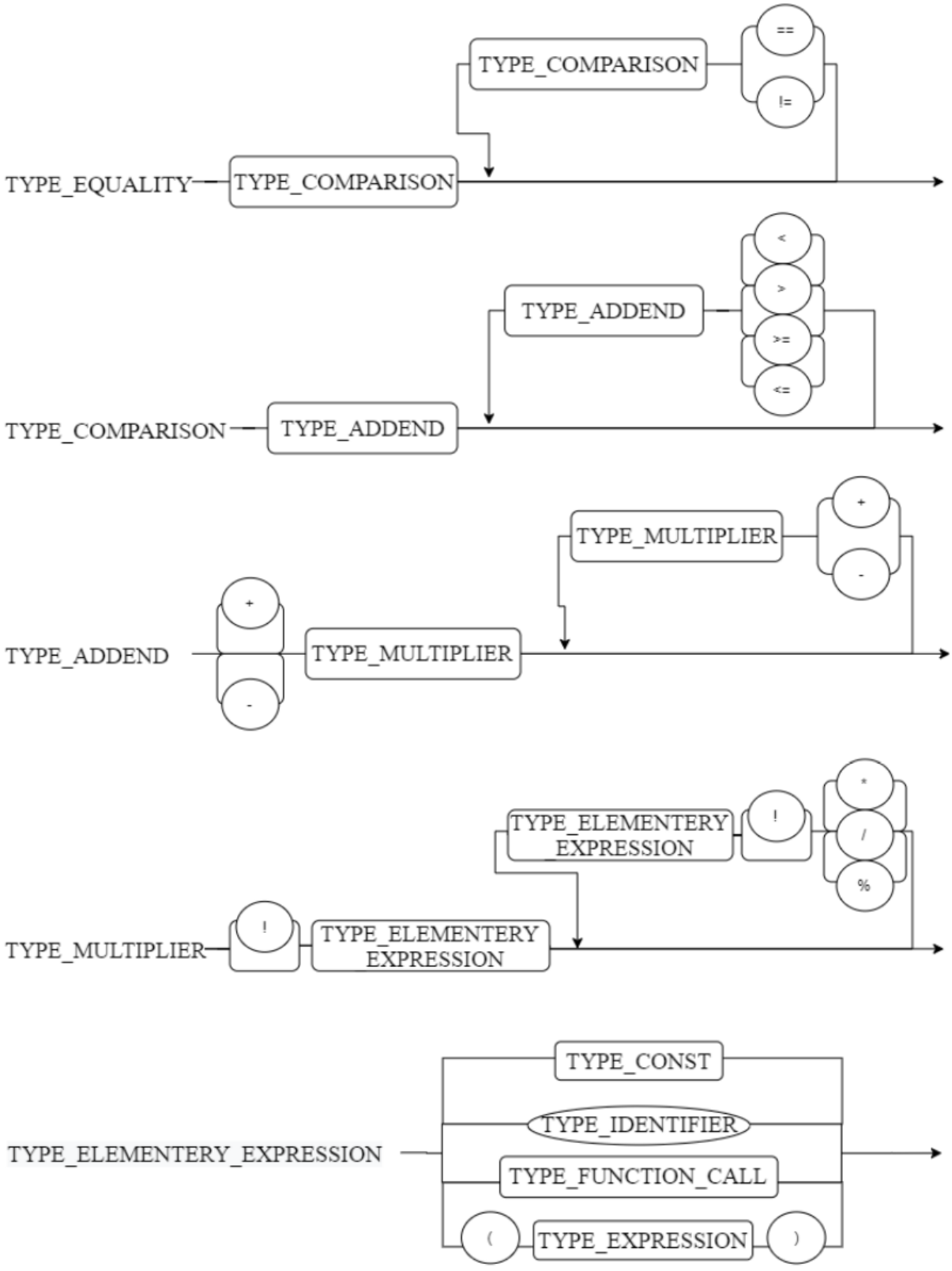







TYPE_VARIABLES
TYPE_IDENTIFIER
=
TYPE_EXPRESSION
TYPE_FUNCTION
boolean
short
long
TYPE_IDENTIFIER
(
TYPE_VAR_FOR_FUNC
)
TYPE_
COMPOSITE_OPERATOR
TYPE_VAR_FOR_FUNC
,
boolean
short
long
TYPE_IDENTIFIER
TYPE_COMPOSITE_OPERATOR
{
TYPE_OPERATORS_AND_DESCRIPTIONS
}


TYPE_OPERATORS_AND_DESCRIPTIONS
TYPE_DATA
TYPE_OPERATOR
TY
TYPE_OPERATOR
TYPE_ASSIGNMENT
TYPE_
COMPOSITE_OPERATOR
TYPE_FUNCTION_CALL
TYPE_LOOP
return
TYPE_EXPRESSION
;
TYPE_ASSIGNMENT
TYPE_IDENTIFIER
=
TYPE_EXPRESSION
TYPE_FUNCTION_CALL
TYPE_IDENTIFIER
(
TYPE_LIST_OF_EXPRESSION
)
TYPE_LIST_OF_EXPRESSION
,
TYPE_EXPRESSION
TYPE_LOOP
do
TYPE_
COMPOSITE_OPERATOR
while
(
TYPE_EXPRESSION
)
TYPE_EXPRESSION
TYPE_LOG_AND
||
TYPE_LOG_AND
TYPE_EXC_OR
&&
TYPE_EXC_OR
TYPE_EQUALITY
^
TYPE_EQUALITY



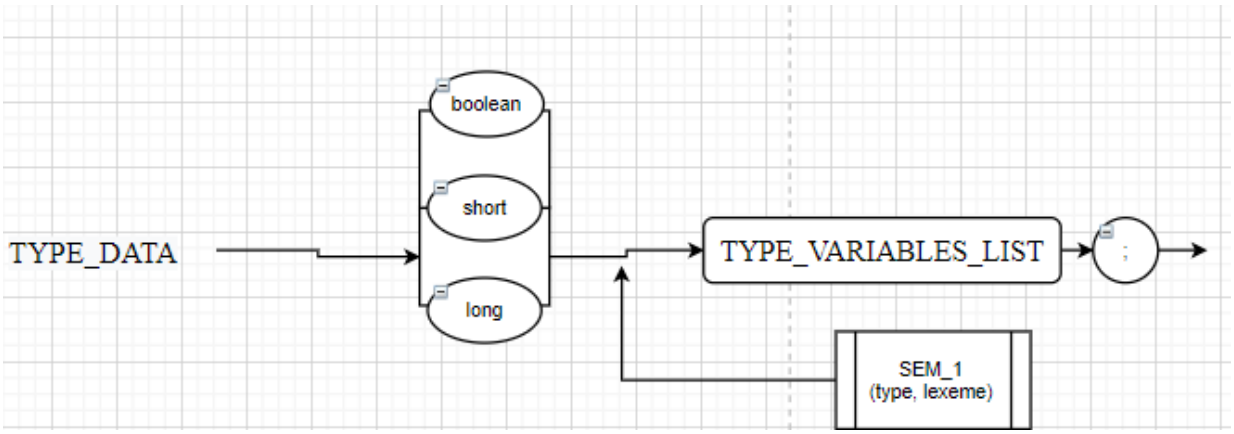
Types of program objects

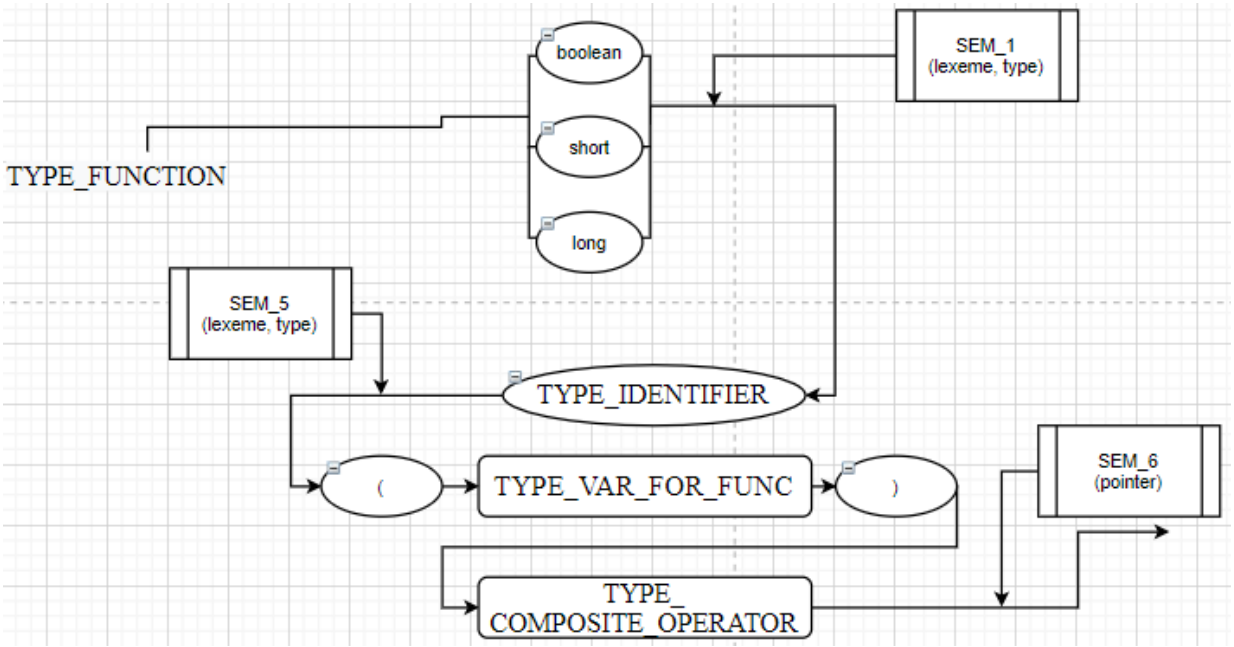
- Simple Variables
- Functions with parameters

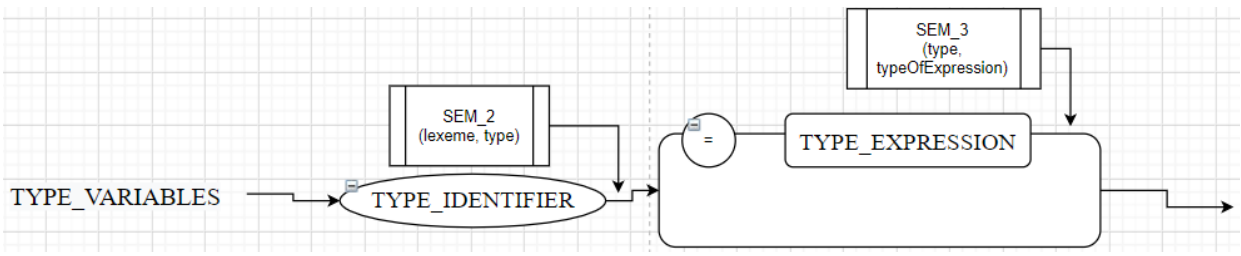
List of context conditions

1. Each object must be described.
2. The program allows implicit type casts.
3. The scope of use of an object must be consistent with the scope of its action.

- Descriptions of variables and functions



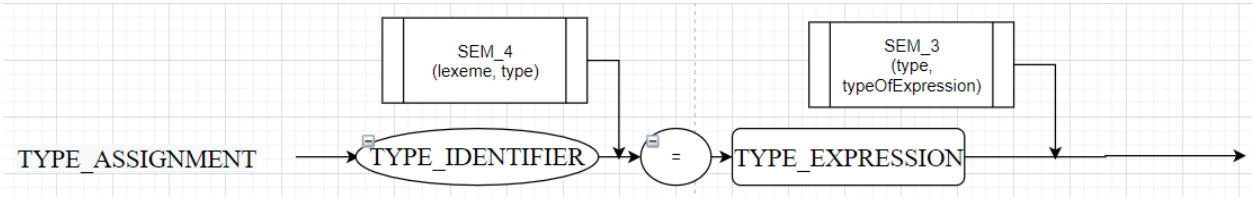




SEM _1 (lexeme , type) – returns the type

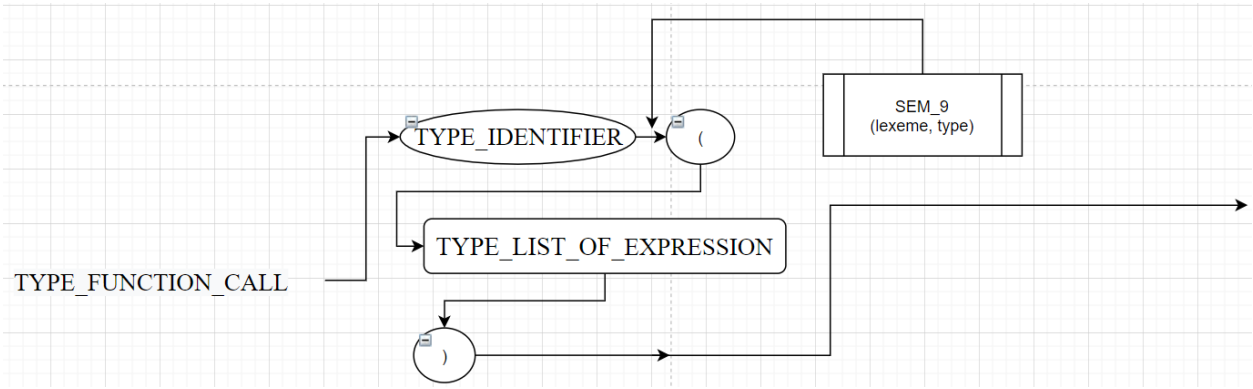
SEM _2 (lexeme , type) – enters the identifier into the table with the type defined SEM_1
SEM _3 (type , typeOfExpression) – controls type conversion during assignment
SEM _5 (lexeme , type) – enters an identifier into a table with a type defined by SEM _1, creates an empty right child, returns a pointer to the current vertex, sets a pointer to the created child.
SEM_6 (pointer) – restores the pointer

- Assignment



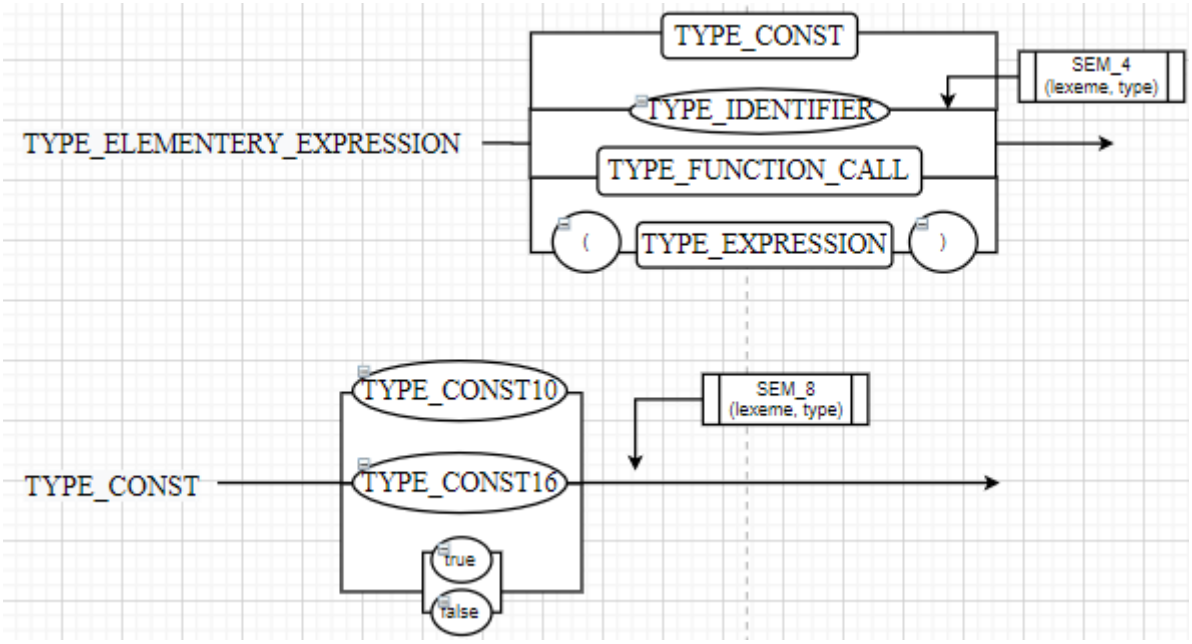
SEM _4 (lexeme , type) – checks the identifier in the table, returns its type.
SEM _3 (type , typeOfExpression) – controls type conversion during assignment

- Calling a function

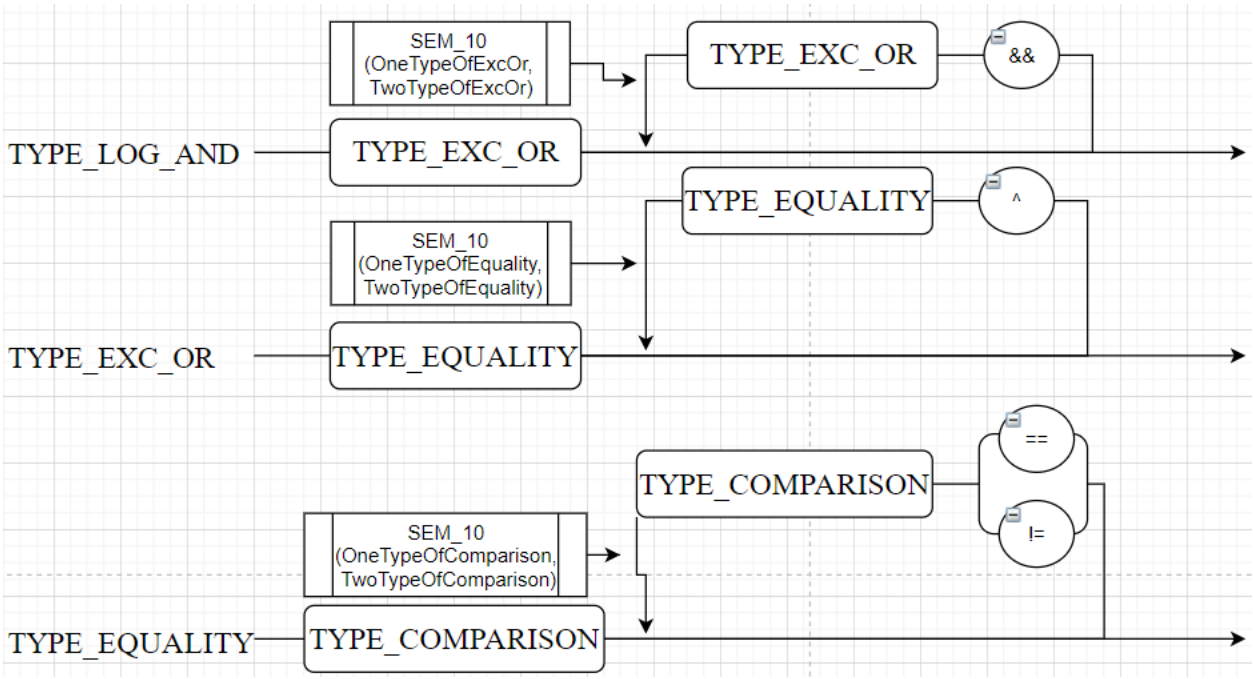


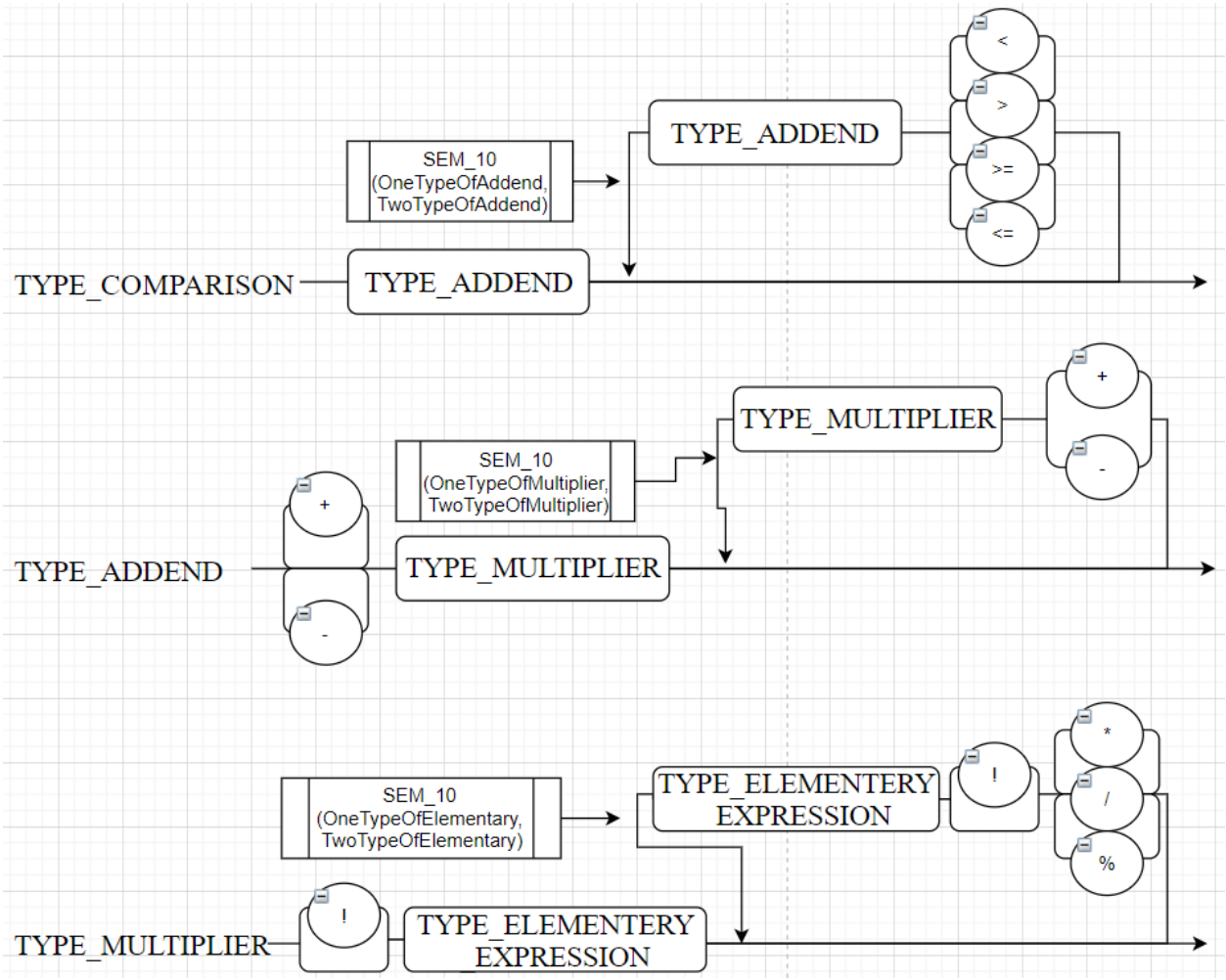
SEM _9 (lexeme , type) – checks the function identifier in the table, returns its type.

- Expression



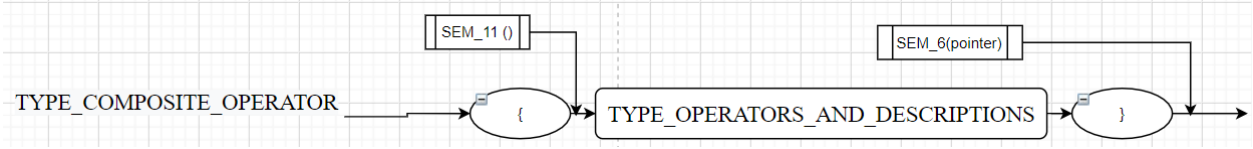
SEM_4 (lexeme , type) – checks the identifier in the table, returns its type.
SEM_8 (lexeme , type) – returns the type of the constant





SEM _10 – calculates the type of the result of the operation in accordance with the cast table.

- Compound operator



SEM _11 () – creates an empty left child and its right child, returns a pointer to the created left child, sets a pointer to the created right child.
SEM _6 (pointer) – restores the pointer

List of data stored in the semantic tree

- Simple Variables
  - Identifier
  - Data type
  - Value
  - Object type

- Functions with parameters
    - Identifier
    - Return type
    - Object type
    - Number of parameters
    - Type each parameter

Storing values in a semantic table

For storage, the union construction is used , in which data is stored according to the following principle:

1) Simple variables (short , long):

-short dataForShort (storing a short value)

-long dataForLong (storing a long value)

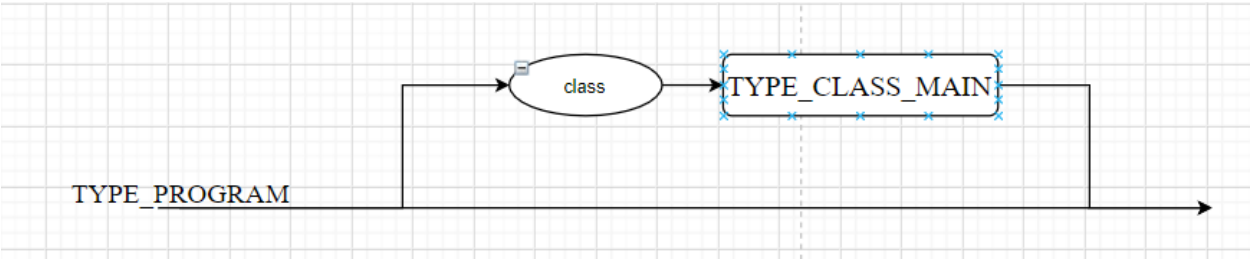
2) Functions with parameters (short , long):

- int dataForShort (storing the return value of type short)

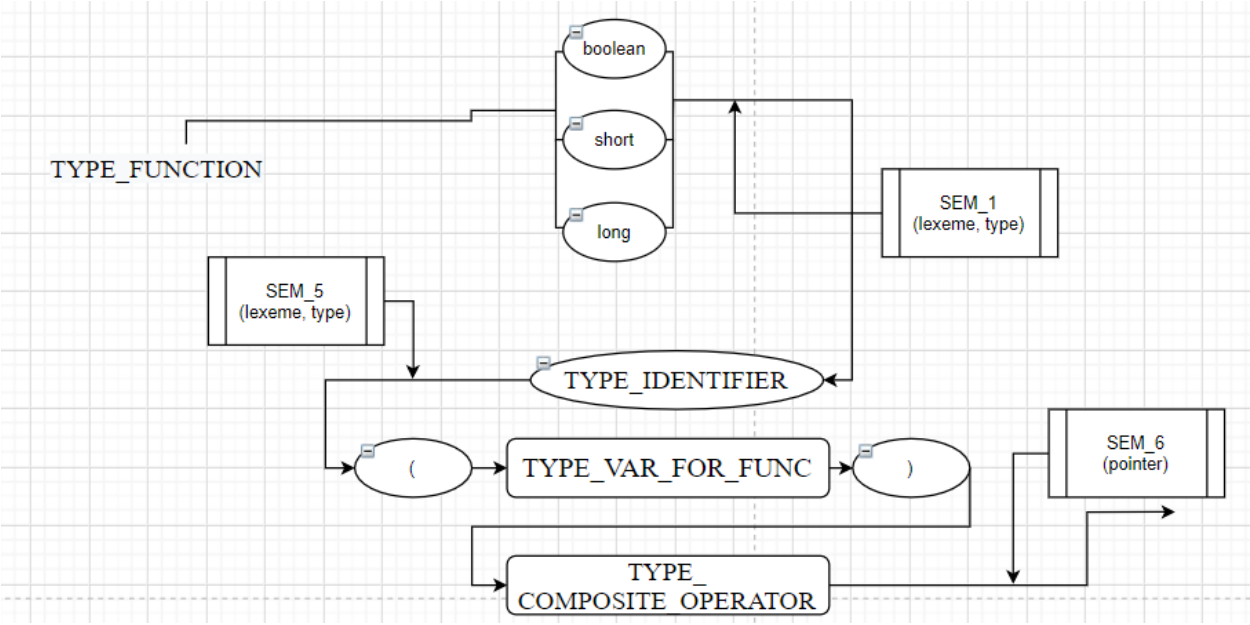
-long dataForLong (storing the return value of type long)

Syntax diagrams in which memory is allocated or deallocated

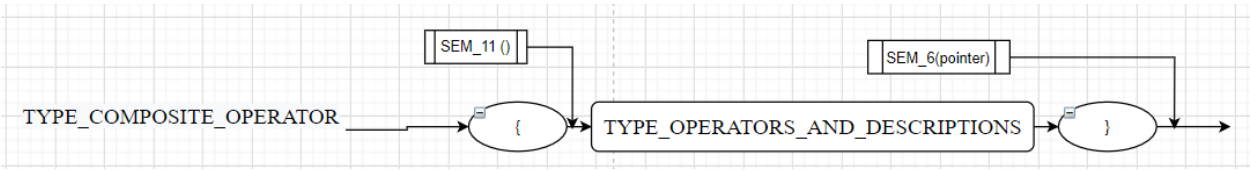
Memory allocation for the tree occurs before the program analysis begins. Memory is freed after the program is analyzed.



When describing functions, the semantic subroutine S EM_5 allocates memory for a new block. After the function body completes, the allocated memory must be freed.



When processing a compound statement, memory is similarly allocated for a new block, which, of course, must be freed after exiting the compound statement.



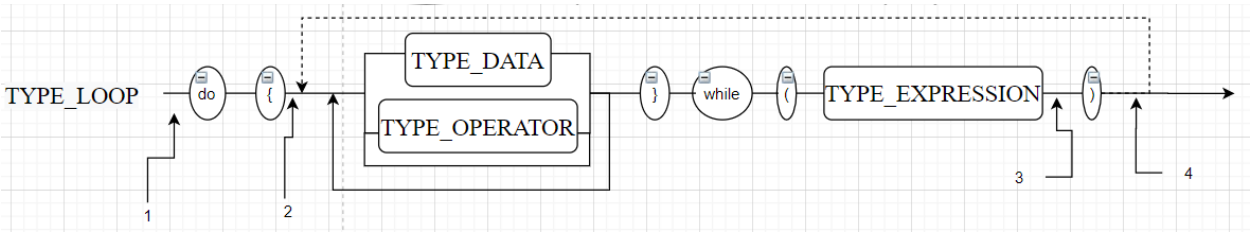
Interpretation flag

flagInterpret into the program.

For all syntax diagrams responsible for calculating flagInterpret , specify the rules for its calculation

No semantic routine is executed when flagInterpret is equal to fasle .

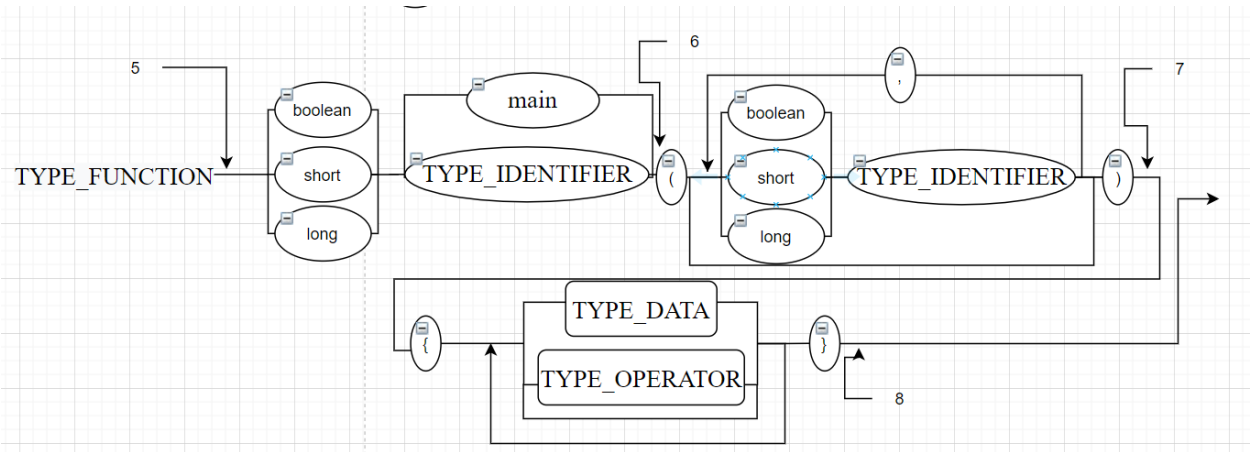
do - while loop :



1. Saving the interpretation flag.
2. Maintaining the pointer position in the text.

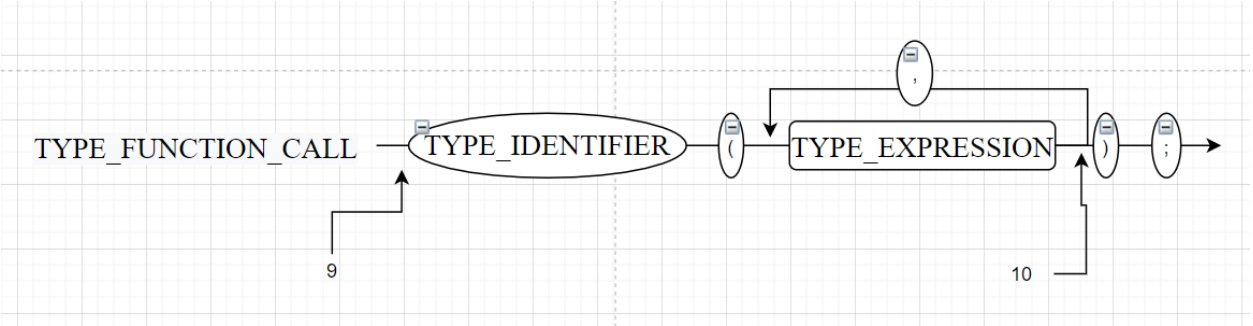
3. Calculate the new interpretation flag value.
4. Depending on the value of the interpretation flag, exit the loop or re-execute.

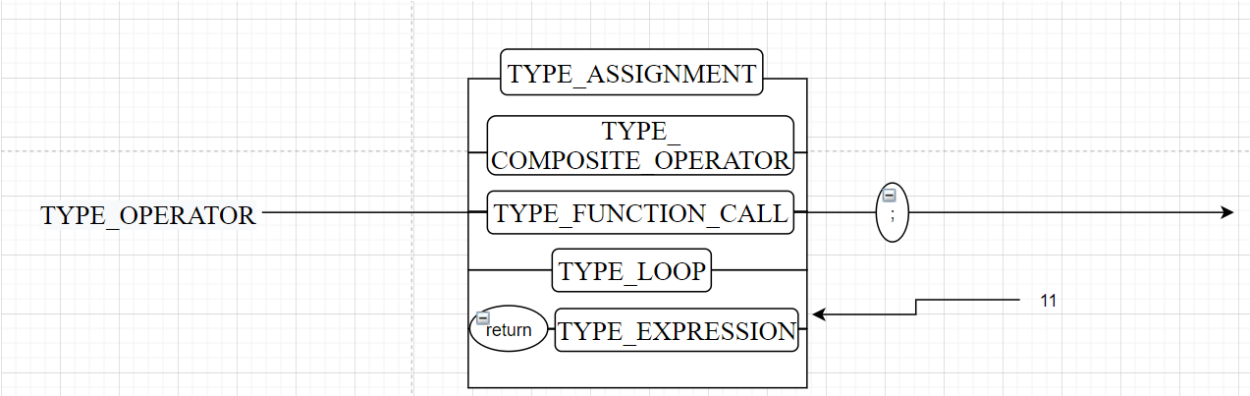
Function description:



5. Saving the interpretation flag to a local variable. The function header and parameter list must be interpreted in any case.
6. Save the terminal type: main or another identifier
7. We save the beginning of the function body and depending on the terminal type:
    a. If main , then we interpret the function body;
    b. If the identifier is different, then we do not interpret the function body.
8. Restoring the interpretation flag.

Function call:





9. Remembering the position in the text and the currentNode pointer in the tree.
10. Calculate and record the actual parameters and move on to the body of the function.
11. Evaluate the expression and assign it as the value of the function. Restore call peer context .

TEST

INPUT:

```
class Main

{

	long add(long a, long b)

	{

		return a + b;

	}

	short pow2(short d)

	{

		return d * d;

	}

	long func()

	{

		long var = 10;

		var = var - 10;

		var = var - var + var + 1;

		return add(var, 5);
```

```
	}
	long funcWithLoop()
	{
		long varForLoop = 0;
		do
		{
			varForLoop = varForLoop + 2;
		} while (varForLoop <= 10);
		return varForLoop;
	}
	long main()
	{
		long varInMain = 10;
		varInMain = add(101, 4);
		varInMain = pow2(10);
		varInMain = func();
		varInMain = funcWithLoop();
	}
};
```

OUTPUT:

Added function: (long) add = 0

Added function: (short) pow2 = 0

Added function: (long) func = 0

Added function: (long) funcWithLoop = 0

Added function: (long) main = 0

Initialization variable: (long) varInMain

Assignment variable: (long) varInMain = 10

Assignment variable: (long) varInMain = 105

Assignment variable: (long) varInMain = 100

Initialization variable: (long) var

Assignment variable: (long) var = 10

Assignment variable: (long) var = 0

Assignment variable: (long) var = 1

Assignment variable: (long) varInMain = 6

Initialization variable: (long) varForLoop

Assignment variable: (long) varForLoop = 0

Assignment variable: (long) varForLoop = 2

Assignment variable: (long) varForLoop = 4

Assignment variable: (long) varForLoop = 6

Assignment variable: (long) varForLoop = 8

Assignment variable: (long) varForLoop = 10

Assignment variable: (long) varForLoop = 12

Assignment variable: (long) varInMain = 12

No syntax errors were found.