Описание задания:

Программа: класс Main языка Java. Допускается описание функций с параметрами.

Типы данных: short, long, boolean.

Операции: простейшие арифметические, сравнения и логические.

Операторы: присваивания и do-while

Операнды: простые переменные, константы.

Константы: целые в 10 с/с, целые в 16 с/с, логические.

Типы данных (Sun JVM)

Тип	*Диапазон значений*	*Длинна памяти*
short	от –32768 до 32767	2 байта
long	от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807	8 байт
boolean	true или false	4 байта

Таблица приведения типов (Sun JVM)

Операнд	*Операнд*	*Результат*
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

Описание контекстно-зависимой грамматики:

<программа> → class <класс Main>

< класс Main > → Main {<содержание класса>};

<содержание класса> → < содержание класса > < описание > | <описание>

< описание > → < данные > |
< функция>

< данные > → < тип данных > < список переменных>;

< тип данных > → < тип int > | boolean

< тип int > → short | long

< список переменных > → < список переменных >, < переменная > | < переменная >

< переменная > → < идентификатор > | < идентификатор > = < выражение>

< константа> → < целая константа > |

< логическая константа> |

< 16-ричная константа > |

< целая константа > → < целая константа > < цифра > | < цифра >

< логическая константа > → true | false

< 16-ричная константа > → 0X< 16-ричное число> | 0x< 16-ричное число >

< 16-ричное число > → < 16-ричное число> <цифра > |

< 16-ричное число > < 16-ричные буквы > |

< цифра > | < 16-ричные буквы >

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

< функция > → <тип данных>

<идентификатор> (<переменные для описания функции>)

< составной оператор >

<переменные для описания функции> → <переменные для описания функции>,

<тип данных> <идентификатор> |

<тип данных> <идентификатор>

< составной оператор > → {<операторы и описания>}

<операторы и описания> → <операторы и описания> <данные> |

< операторы и описания > оператор |

ε

< оператор > → < присваивание >; |
< составной оператор >; |
< вызов функции >; |
< цикл do-while >; |
return < выражение >;

< присваивание > → < имя > = < выражение > ;

< имя > → < идентификатор >

< вызов функции > → <идентификатор> (<список выражений>);

<список выражений> → <список выражений>, <выражение> |
<выражение>

< цикл do-while > → do < составной оператор > while (< выражение >)

<выражение> → <выражение> || <Лог И> | <Лог И>

<Лог И> → <Лог И> && <Иск ИЛИ> | <Иск ИЛИ>

<Иск ИЛИ> → <Иск ИЛИ> ^ <Равенство> | <Равенство>

<Равенство> → <Равенство> == <Сравнение> |

<Равенство> != <Сравнение> |

<Сравнение>

<Сравнение> → <Сравнение> < <Слагаемое> |

<Сравнение> > <Слагаемое> |

<Сравнение> <= <Слагаемое> |

<Сравнение> >= <Слагаемое> |

<Слагаемое>

< слагаемое > → < слагаемое > + < множитель > |

< слагаемое > - < множитель > |

< множитель >

< множитель > → < множитель > * < префикс > |

< множитель > / < префикс > |

<множитель > % <префикс> |

<префикс>

< префикс > → ++< постфикс > |

--< постфикс > |

!< постфикс> |

< постфикс >

< постфикс > → < элементарное выражение >++ |

< элементарное выражение >-- |

< элементарное выражение >

< элементарное выражение > → < имя > |

< константа > |

(< выражение >) | <вызов функции>

< идентификатор > → < буква >< окончание >

< окончание > → < окончание >< буква > |

< окончание >< цифра > |

ε

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

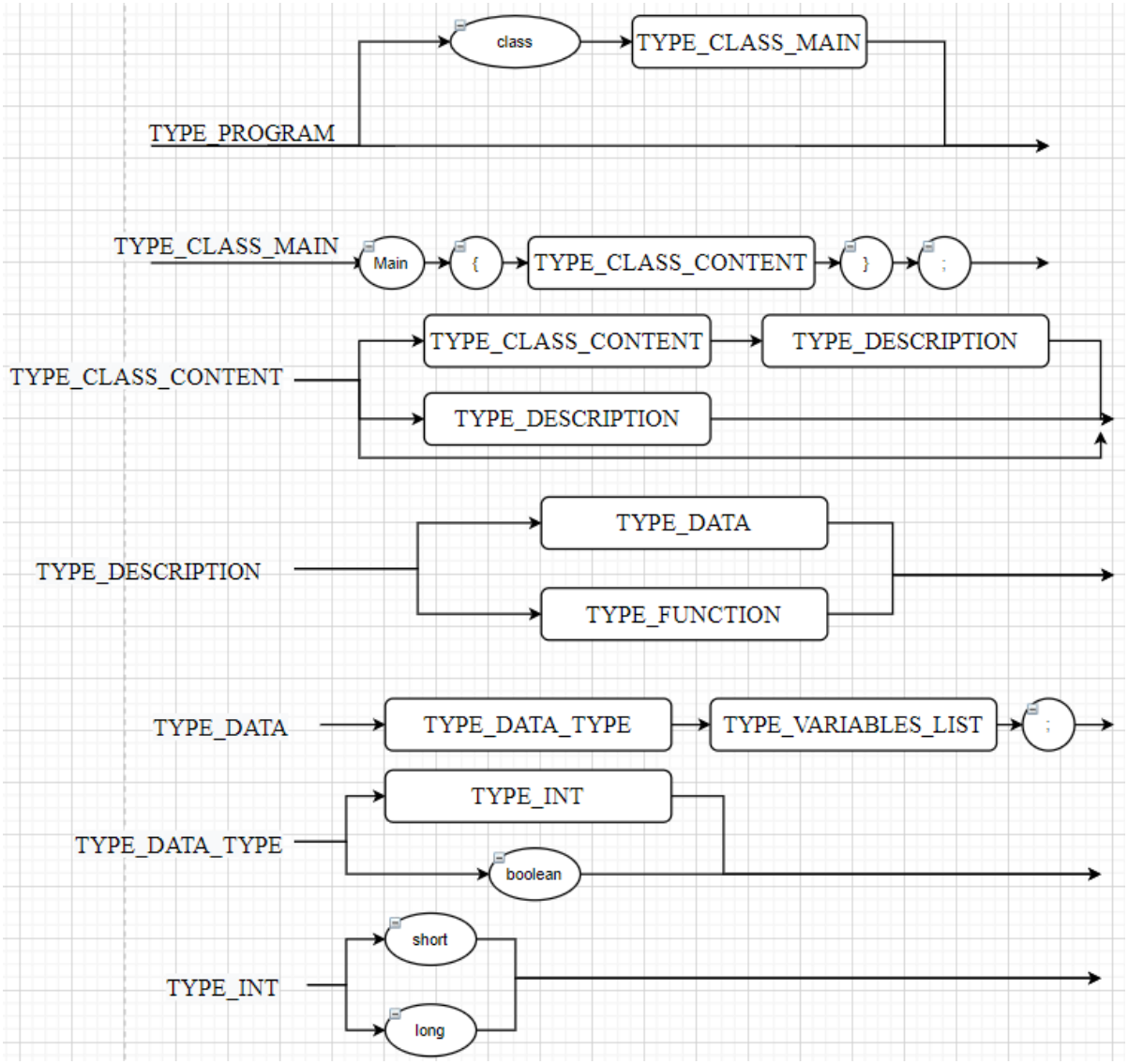
< цифра > → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

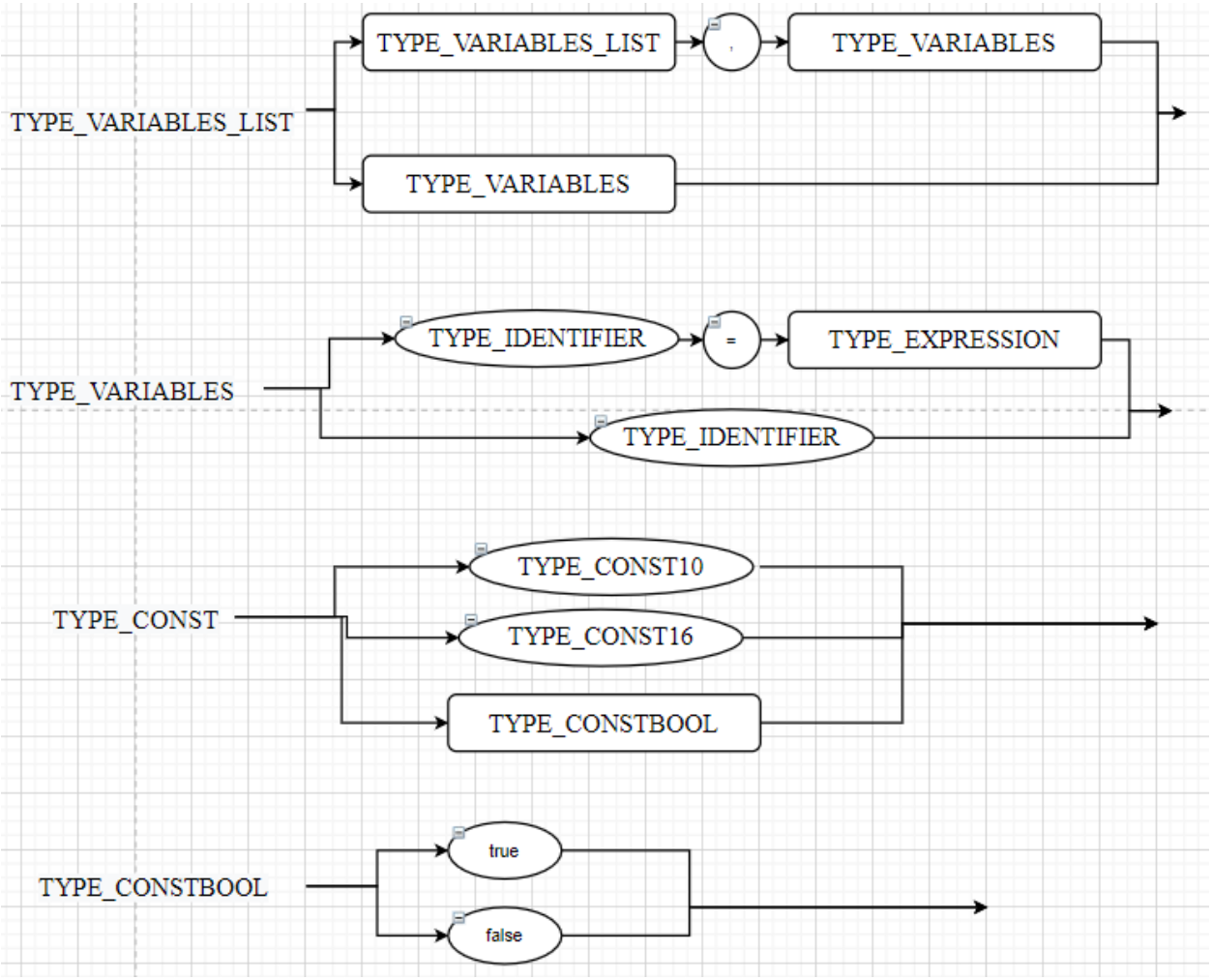
Обозначения нетерминальных символов

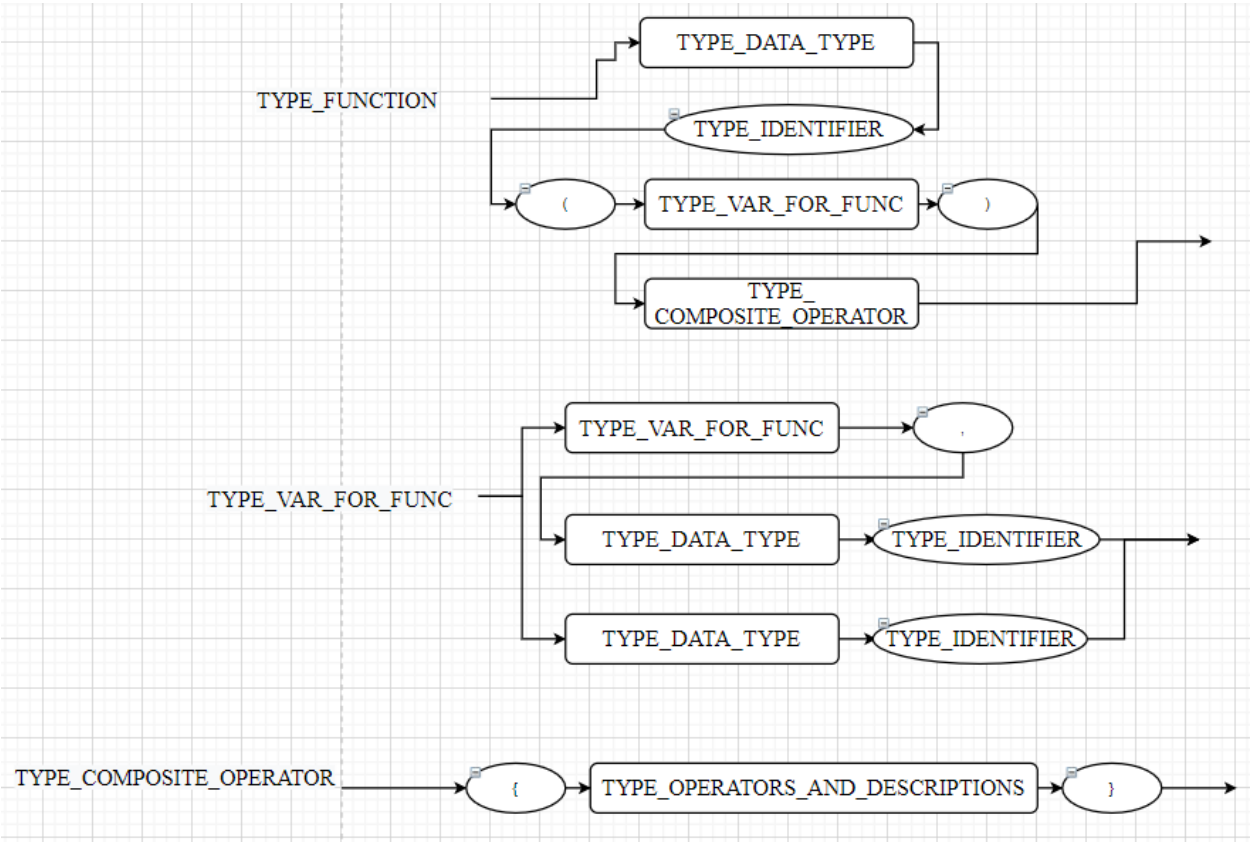
Нетерминальный символ	Обозначение
Программа	TYPE_PROGRAM
Класс Main	TYPE_CLASS_MAIN
Содержание класса	TYPE_CLASS_CONTENT
Описание	TYPE_DESCRIPTION
Данные	TYPE_DATA
Тип данных	TYPE_DATA_TYPE
Тип int	TYPE_INT
Список переменных	TYPE_VARIABLES_LIST
Переменная	TYPE_VARIABLES
Константа	TYPE_CONST
Логическая константа	TYPE_LOG_CONST
Функция	TYPE_FUNCTION
Переменные для описания функции	TYPE_VAR_FOR_FUNC
Составной оператор	TYPE_ COMPOSITE_OPERATOR
Операторы и описания	TYPE_OPERATORS_AND_DESCRIPTIONS
Оператор	TYPE_OPERATOR
Присваивание	TYPE_ ASSIGNMENT
Вызов функции	TYPE_FUNCTION_CALL
Список выражений	TYPE_LIST_OF_EXPRESSION
Цикл do-while	TYPE_LOOP
Выражение	TYPE_EXPRESSION
Лог И	TYPE_LOG_AND
Иск ИЛИ	TYPE_EXC_OR

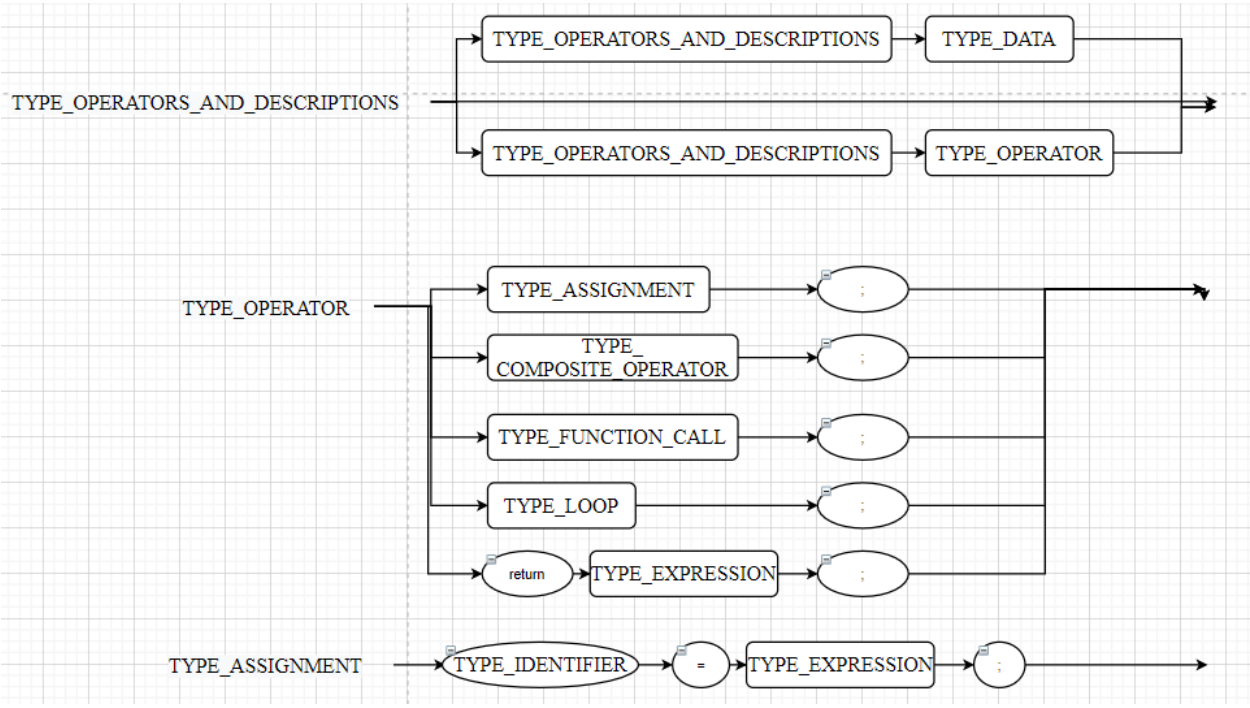
Равенство	TYPE_EQUALITY
Сравнение	TYPE_COMPARISON
Слагаемое	TYPE_ADDEND
Множитель	TYPE_MULTIPLIER
Префикс	TYPE_PREFIX
Элементарное выражение	TYPE_ELEMENTERY_EXPRESSION

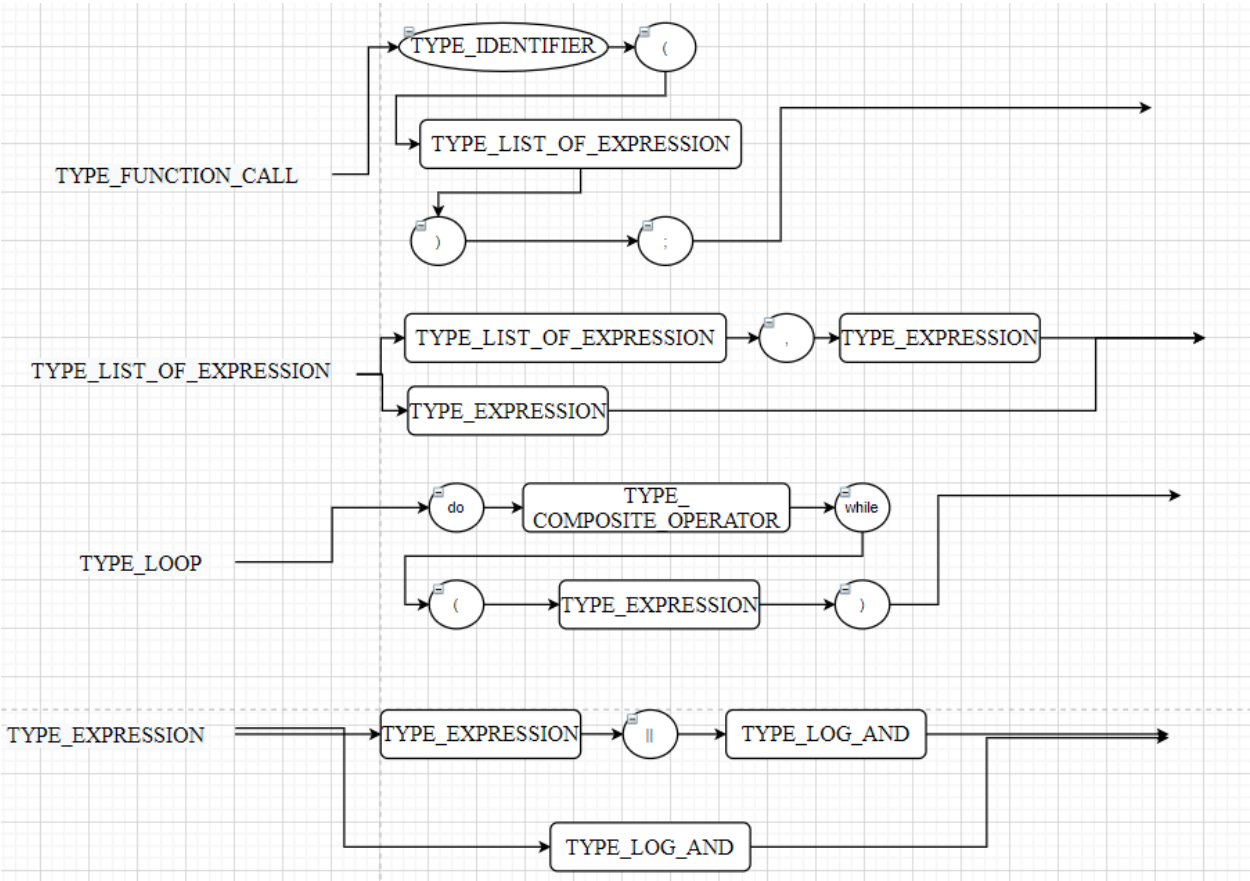
Синтаксические диаграммы контекстно-зависимой грамматики:

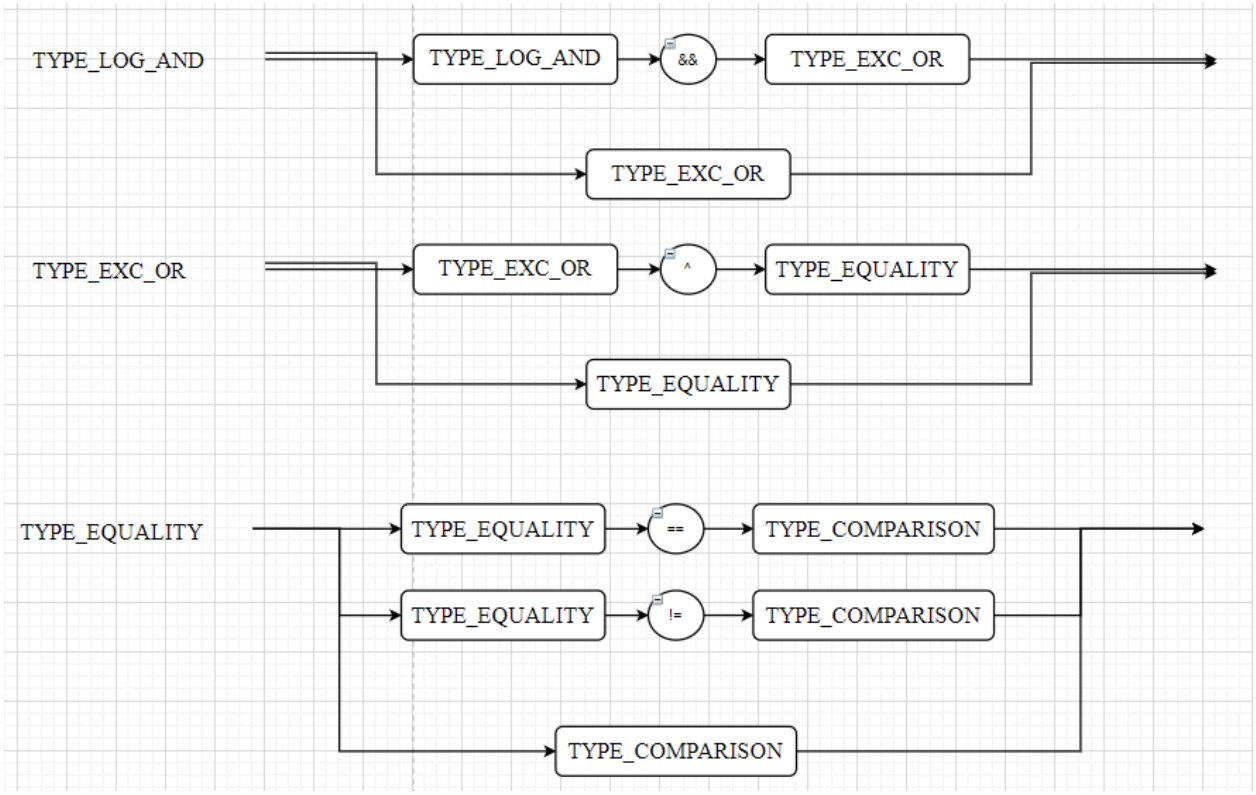


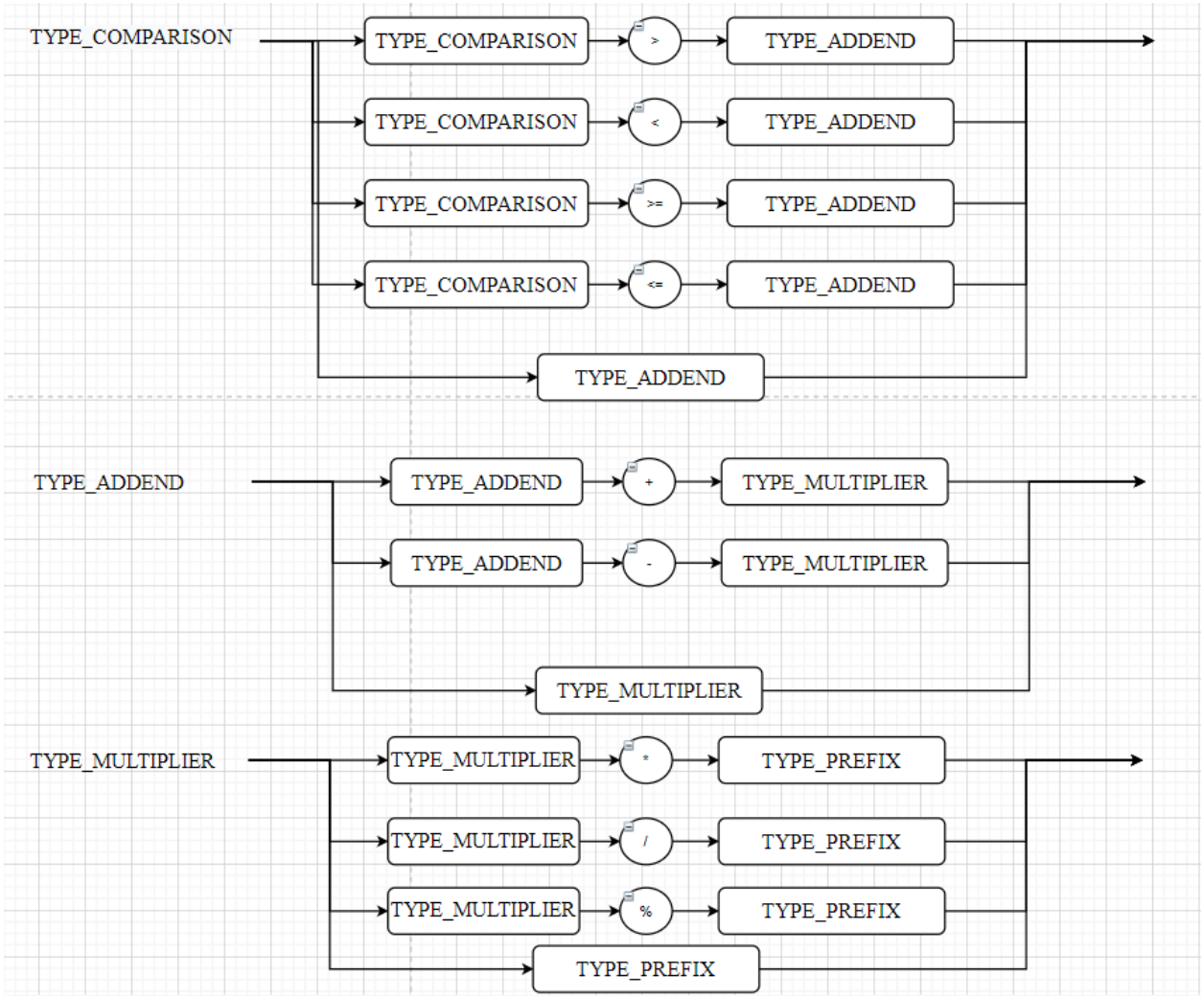

TYPE_VARIABLES_LIST
TYPE_VARIABLES_LIST
,
TYPE_VARIABLES
TYPE_VARIABLES
TYPE_VARIABLES
TYPE_IDENTIFIER
=
TYPE_EXPRESSION
TYPE_IDENTIFIER
TYPE_CONST
TYPE_CONST10
TYPE_CONST16
TYPE_CONSTBOOL
TYPE_CONSTBOOL
true
false

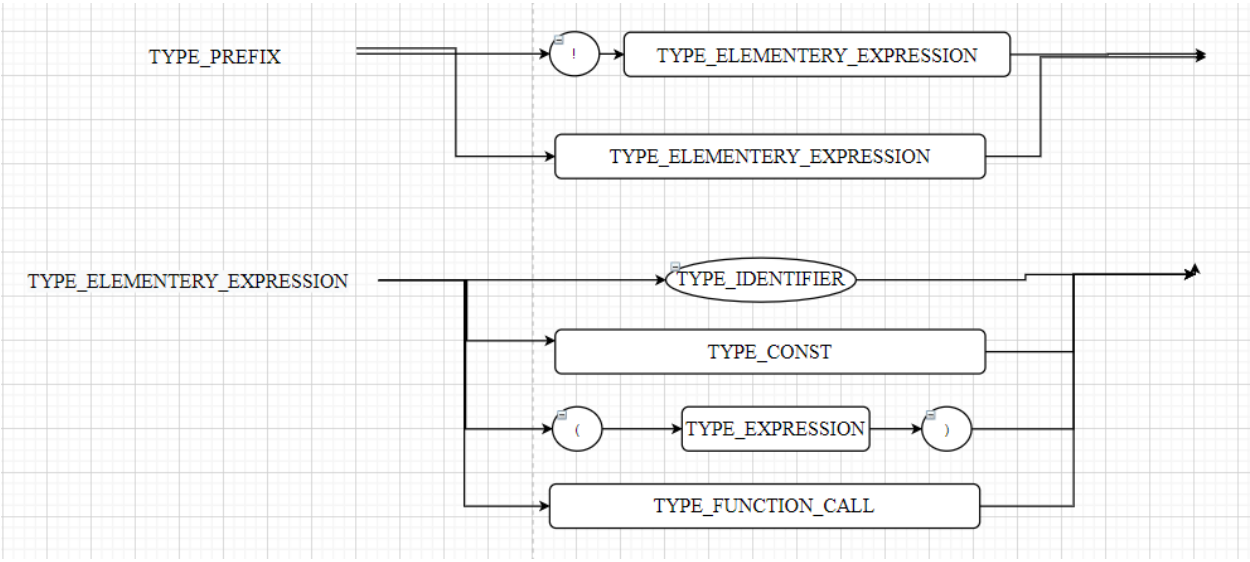

TYPE_FUNCTION
TYPE_DATA_TYPE
TYPE_IDENTIFIER
(
TYPE_VAR_FOR_FUNC
)
TYPE_
COMPOSITE_OPERATOR
TYPE_VAR_FOR_FUNC
TYPE_VAR_FOR_FUNC
,
TYPE_DATA_TYPE
TYPE_IDENTIFIER
TYPE_DATA_TYPE
TYPE_IDENTIFIER
TYPE_COMPOSITE_OPERATOR
{
TYPE_OPERATORS_AND_DESCRIPTIONS
}


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
;

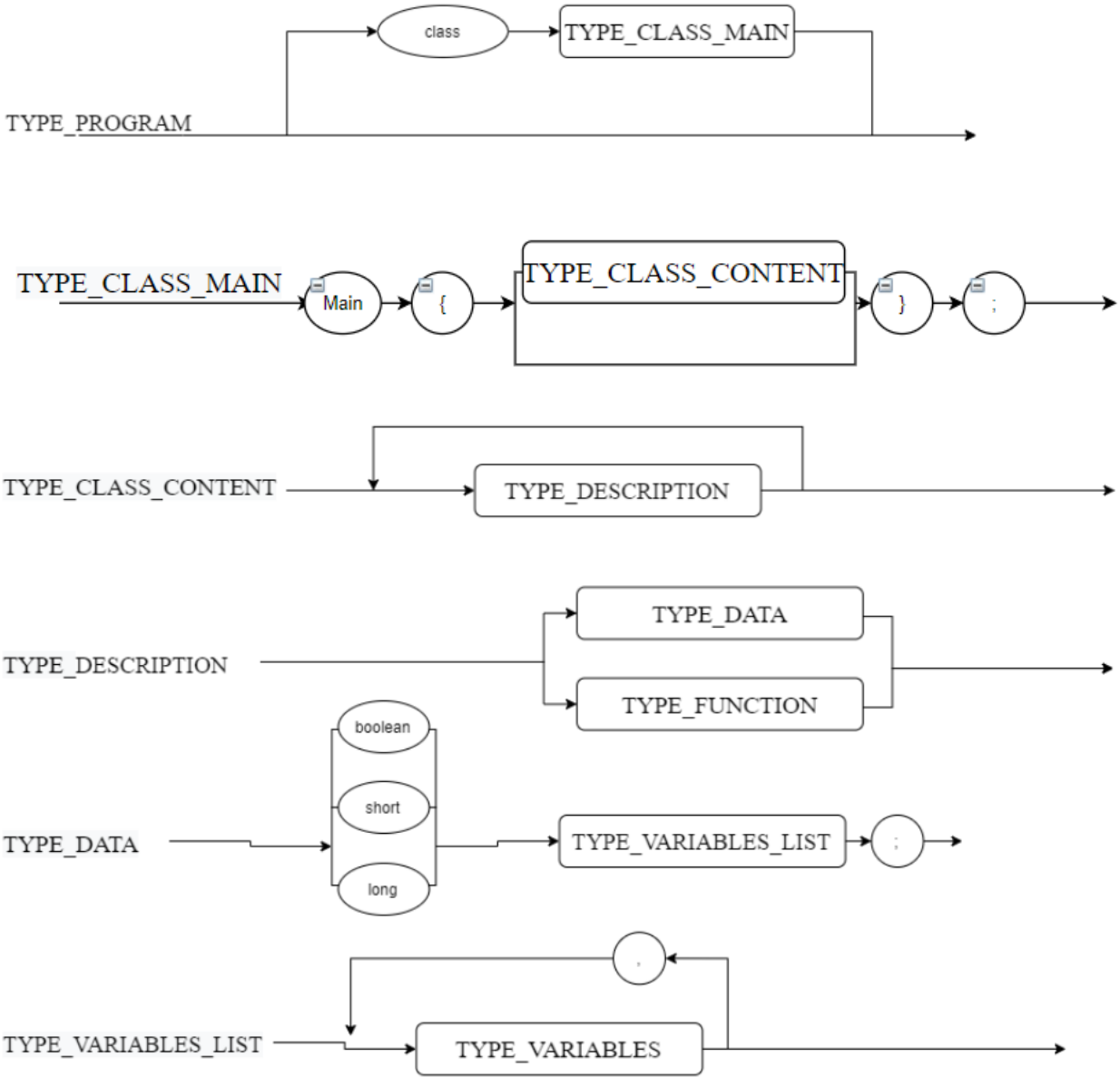

TYPE_IDENTIFIER
(
TYPE_FUNCTION_CALL
TYPE_LIST_OF_EXPRESSION
)
;
TYPE_LIST_OF_EXPRESSION
TYPE_LIST_OF_EXPRESSION
,
TYPE_EXPRESSION
TYPE_EXPRESSION
TYPE_LOOP
do
TYPE_
COMPOSITE_OPERATOR
while
(
TYPE_EXPRESSION
)
TYPE_EXPRESSION
TYPE_EXPRESSION
||
TYPE_LOG_AND
TYPE_LOG_AND

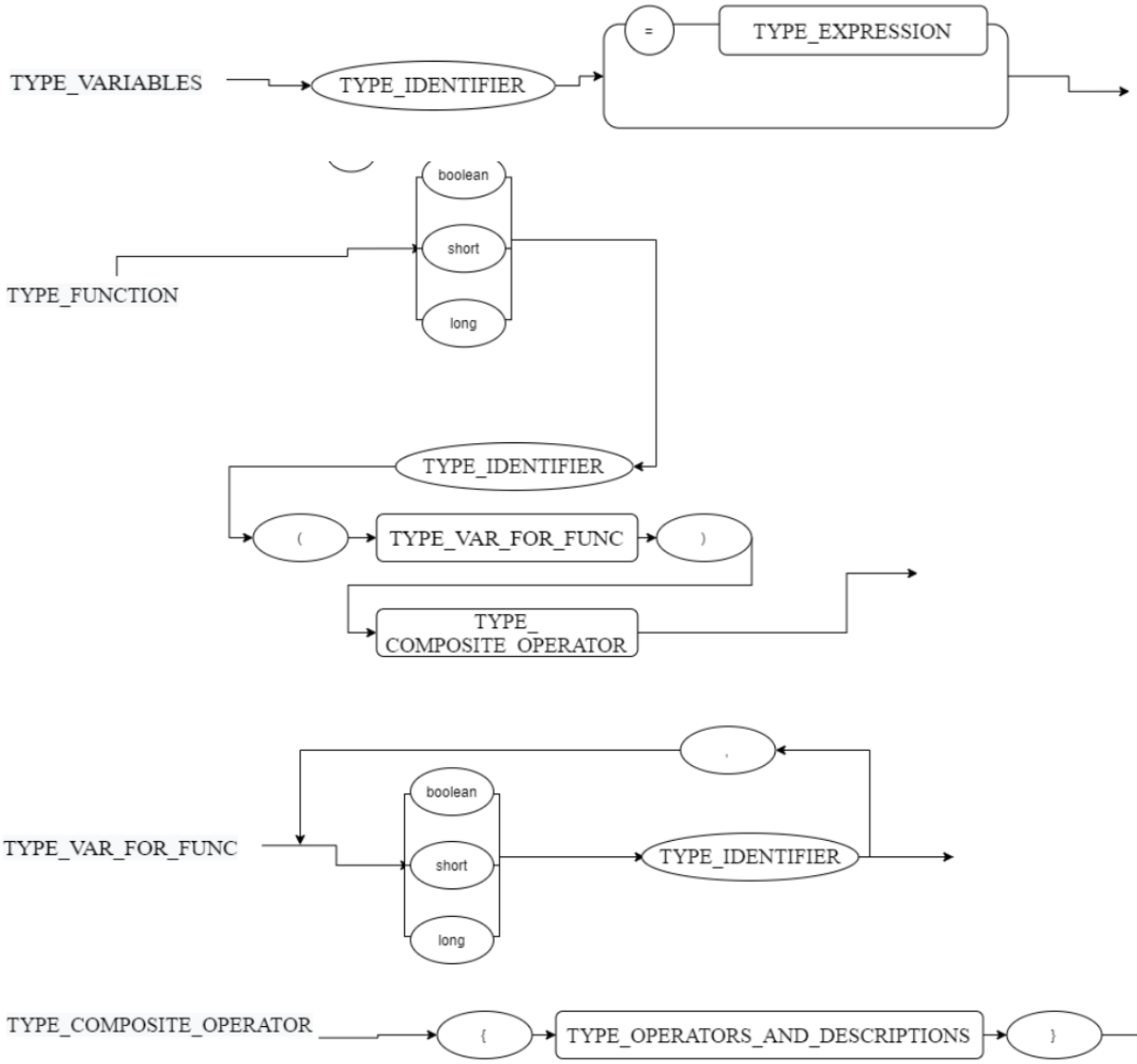

TYPE_LOG_AND
TYPE_LOG_AND
&&
TYPE_EXC_OR
TYPE_EXC_OR
TYPE_EXC_OR
TYPE_EXC_OR
^
TYPE_EQUALITY
TYPE_EQUALITY
TYPE_EQUALITY
TYPE_EQUALITY
==
TYPE_COMPARISON
TYPE_EQUALITY
!=
TYPE_COMPARISON
TYPE_COMPARISON

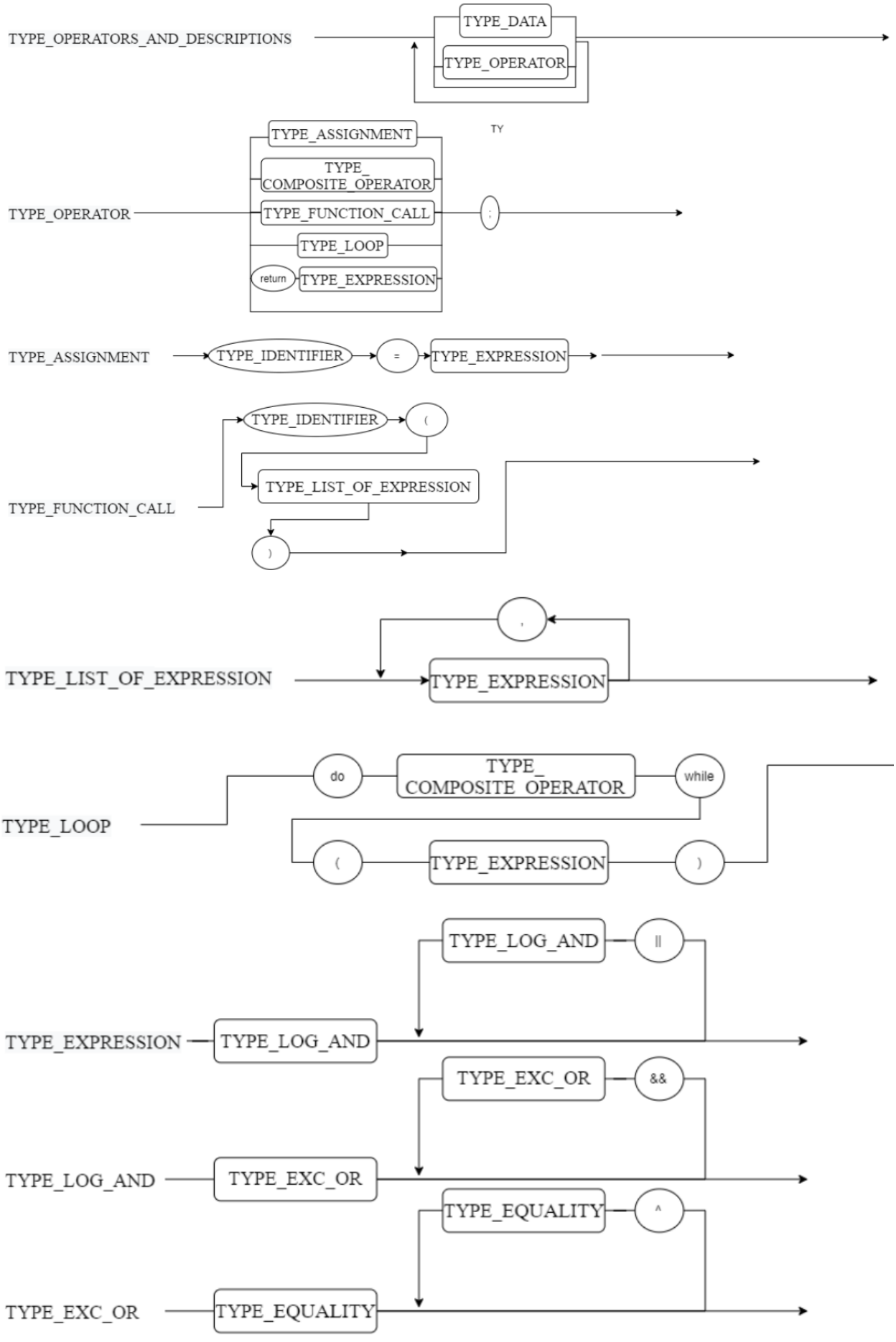

TYPE_COMPARISON
TYPE_COMPARISON
>
TYPE_ADDEND
TYPE_COMPARISON
<
TYPE_ADDEND
TYPE_COMPARISON
>=
TYPE_ADDEND
TYPE_COMPARISON
<=
TYPE_ADDEND
TYPE_ADDEND
TYPE_ADDEND
TYPE_ADDEND
+
TYPE_MULTIPLIER
TYPE_ADDEND
-
TYPE_MULTIPLIER
TYPE_MULTIPLIER
TYPE_MULTIPLIER
TYPE_MULTIPLIER
*
TYPE_PREFIX
TYPE_MULTIPLIER
/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

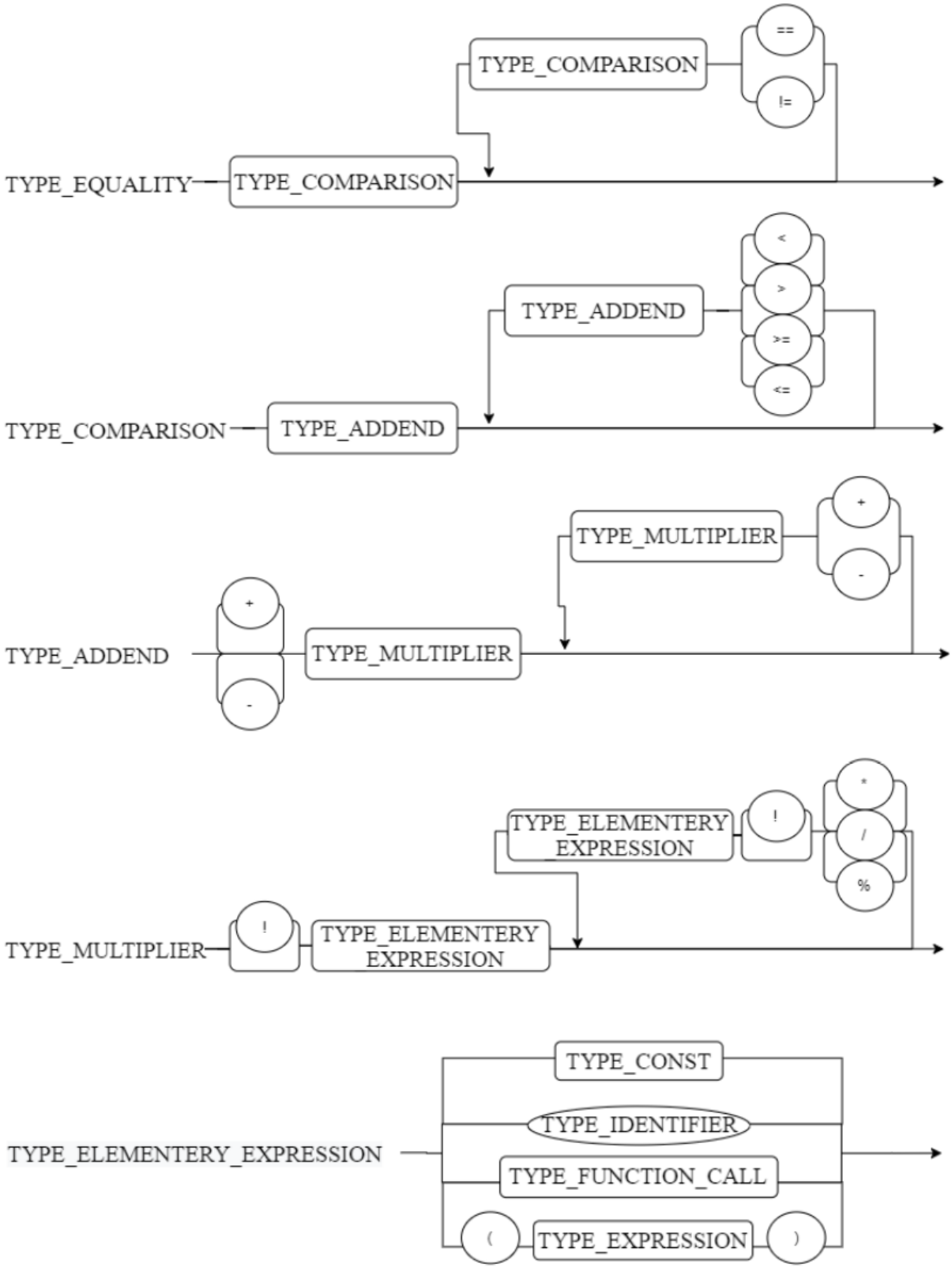


Оптимизированные синтаксические диаграммы




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


TYPE_EQUALITY
TYPE_COMPARISON
TYPE_COMPARISON
==
!=
TYPE_COMPARISON
TYPE_ADDEND
<
>
>=
<=
TYPE_ADDEND
TYPE_ADDEND
+
-
TYPE_MULTIPLIER
TYPE_MULTIPLIER
+
-
TYPE_MULTIPLIER
!
TYPE_ELEMENTERY EXPRESSION
TYPE_ELEMENTERY EXPRESSION
!
*
/
%
TYPE_ELEMENTERY_EXPRESSION
TYPE_CONST
TYPE_IDENTIFIER
TYPE_FUNCTION_CALL
(
TYPE_EXPRESSION
)

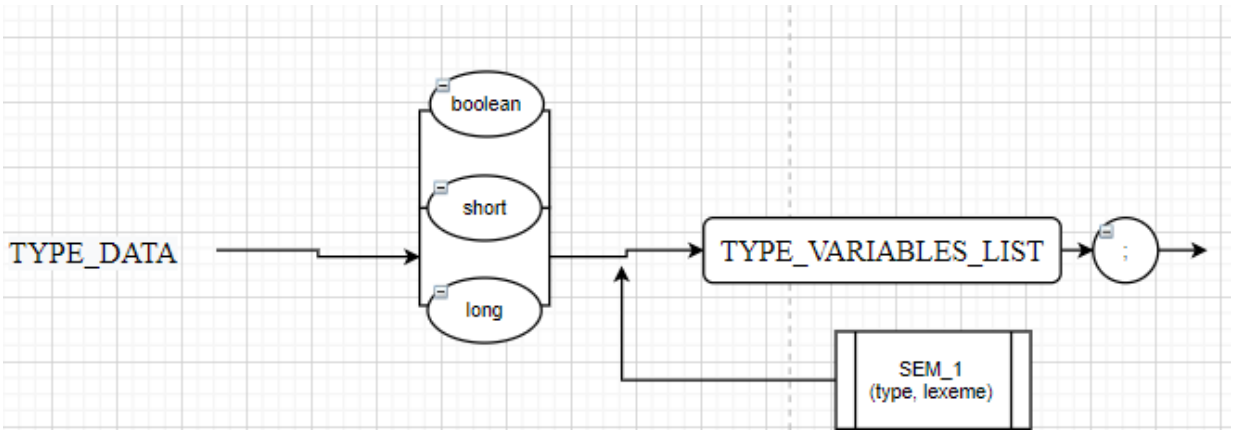
Типы объектов программы

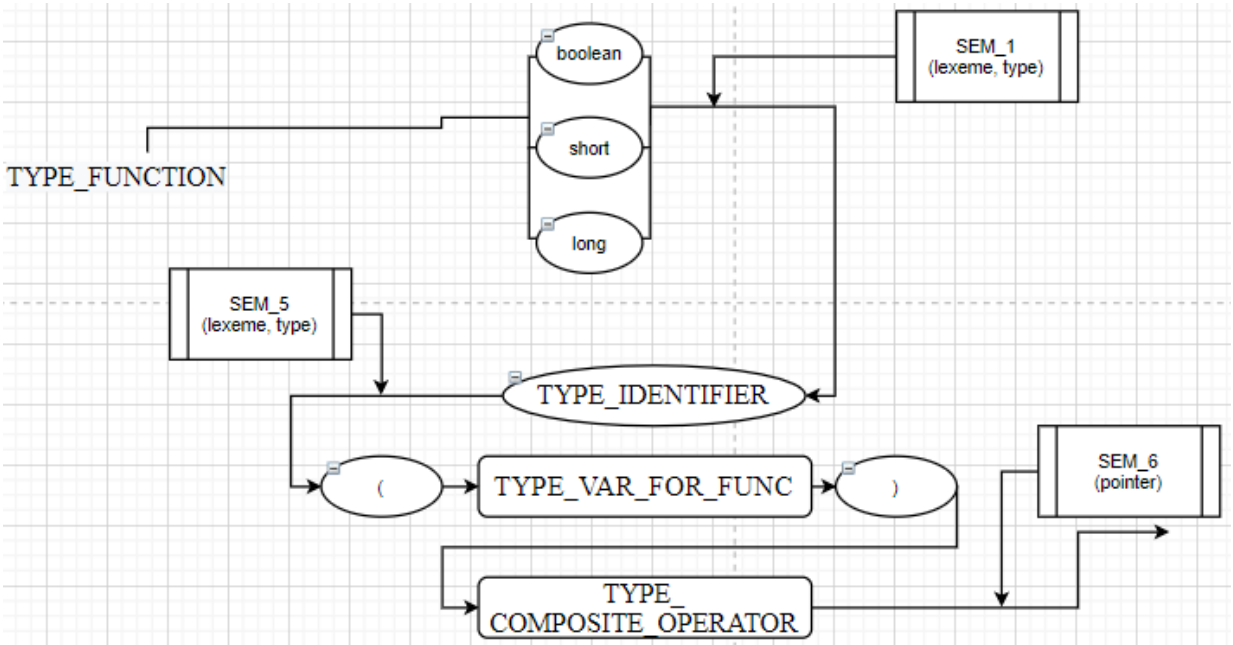
- Простые переменные
- Функции с параметрами

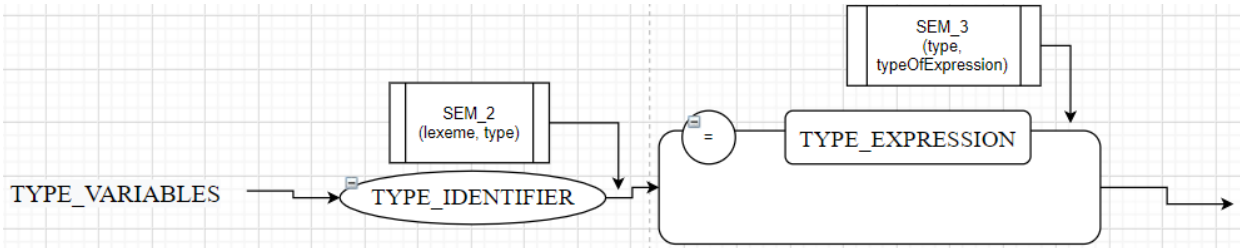
Перечень контекстных условий

1. Каждый объект должен быть описан.
2. В программе разрешаются неявные приведения типов.
3. Область использования объекта должна быть согласована с областью его действия.

- Описания переменных и функций

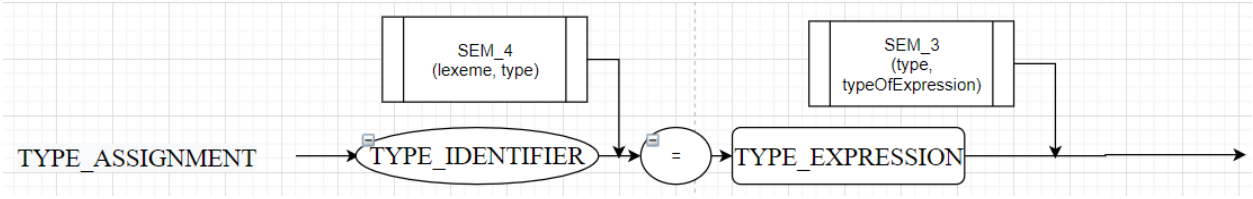






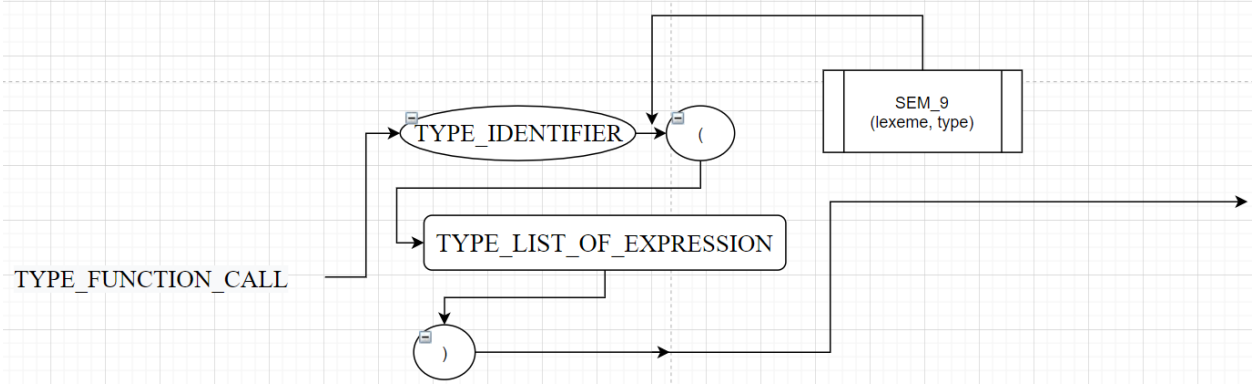
SEM_1 (lexeme, type) – возвращает тип
SEM_2 (lexeme, type)– заносит идентификатор в таблицу с типом определенным Sem1
SEM_3 (type, typeOfExpression) – осуществляет контроль приведения типов при присваивании
SEM_5 (lexeme, type) – заносит идентификатор в таблицу с типом определенным SEM_1, создает пустого правого потомка, возвращает указатель на текущую вершину, устанавливает указатель на созданного потомка.
SEM_6 (pointer) – восстанавливает указатель

- Присваивание



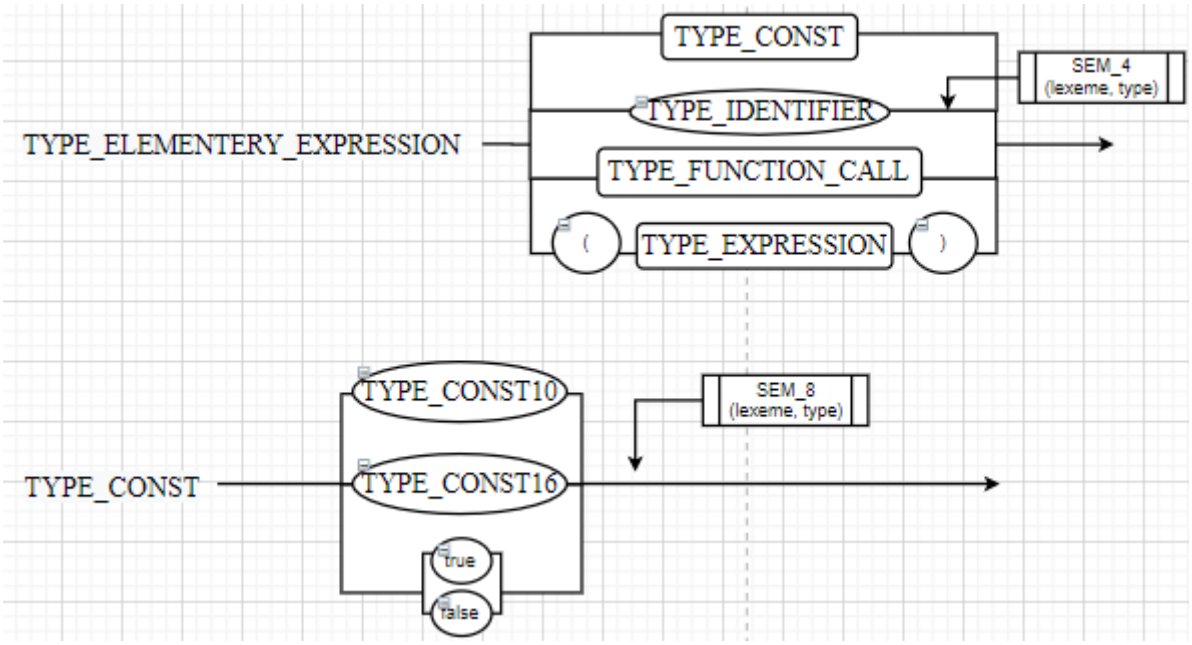
SEM_4 (lexeme, type) – проверяет идентификатор в таблице, возвращает его тип.
SEM_3 (type, typeOfExpression) – осуществляет контроль приведения типов при присваивании

- Вызов функции



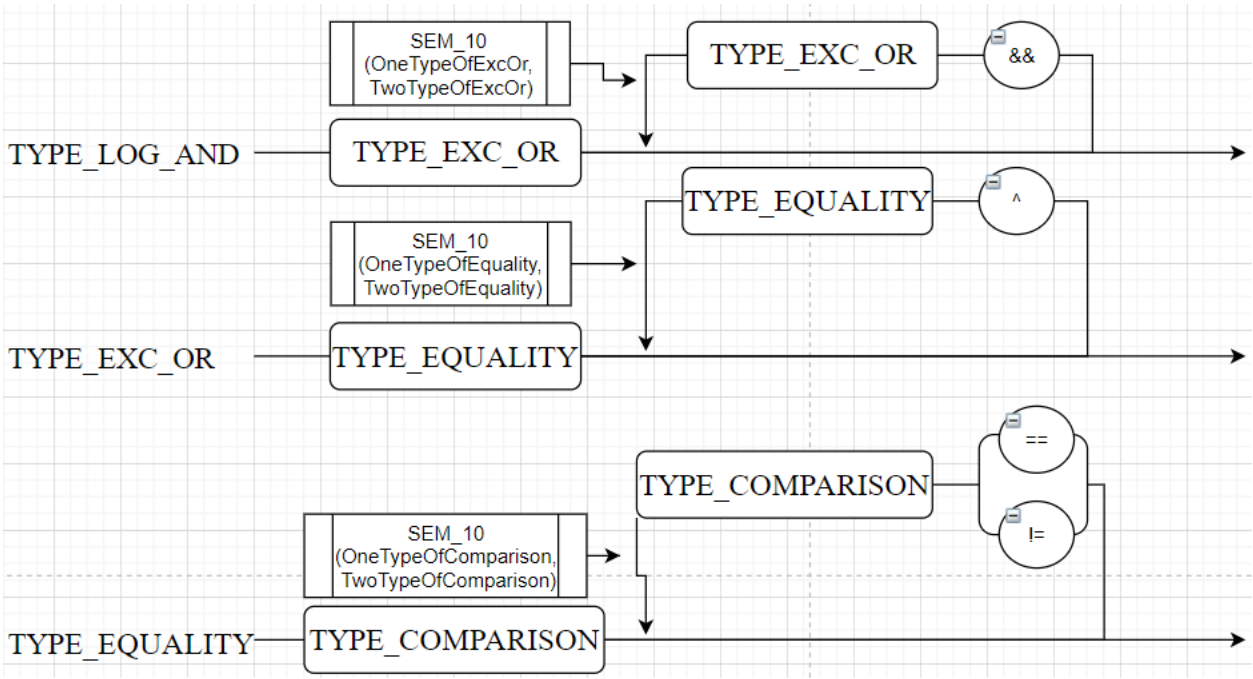
SEM_9 (lexeme, type) – проверяет идентификатор функции в таблице, возвращает его тип.

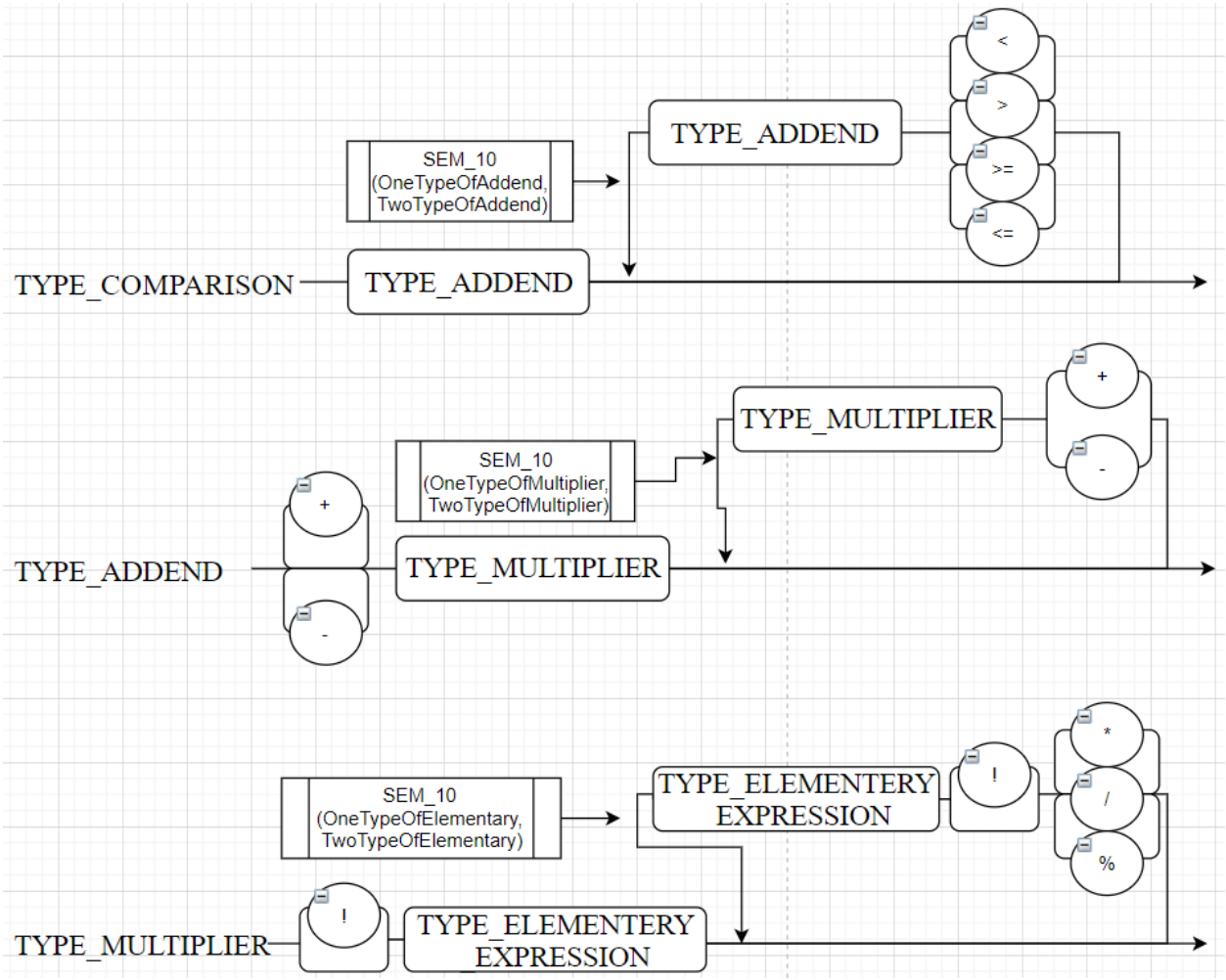
- Выражение



SEM_4 (lexeme, type) – проверяет идентификатор в таблице, возвращает его тип.

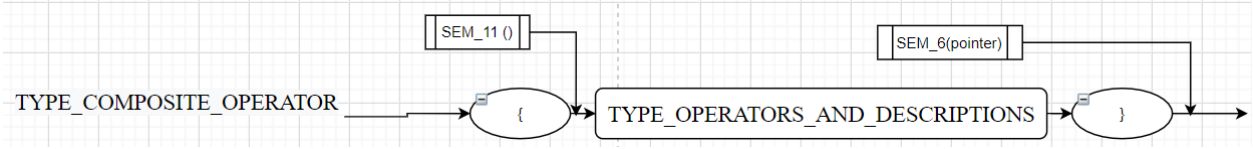
SEM_8 (lexeme, type) – возвращает тип константы





SEM_10 – вычисляет тип результата выполнения операции в соответствии таблицей приведений.

- Составной оператор



SEM_11 () – создает пустого левого потомка и правого от него, возвращает указатель на созданного левого потомка, устанавливает указатель на созданного правого.
SEM_6 (pointer) – восстанавливает указатель

Перечень данных, хранящихся в семантическом дереве

- Простые переменные
  - Идентификатор
  - Тип данных
  - Значение

    - Тип объекта
- Функции с параметрами
    - Идентификатор
    - Тип возвращаемого значения
    - Тип объекта
    - Число параметров
    - Тип каждого параметра

Хранение значений в семантической таблице

Для хранения используется конструкция union, в которой данные хранятся по следующему принципу:

1) Простые переменные (short, long):

- short dataForShort (хранение значения типа short)

- long dataForLong (хранение значения типа long)

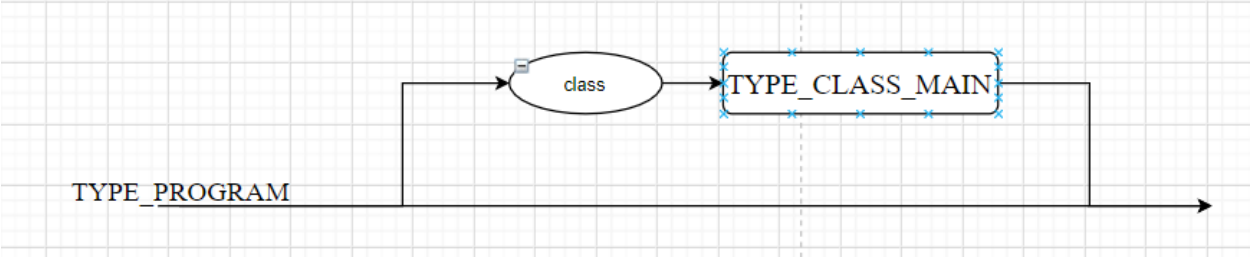
2) Функции с параметрами (short, long):

- int dataForShort (хранение возвращаемого значения типа short)

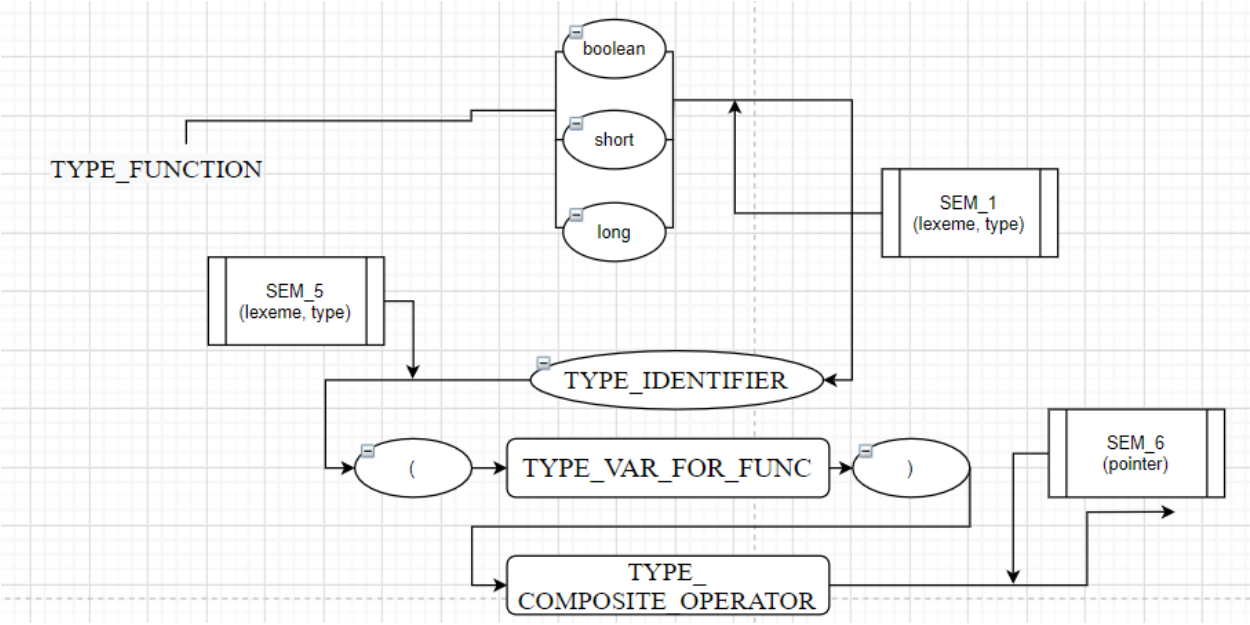
- long dataForLong (хранение возвращаемого значения типа long)

Синтаксические диаграммы, в которых выполняется выделение или освобождение памяти

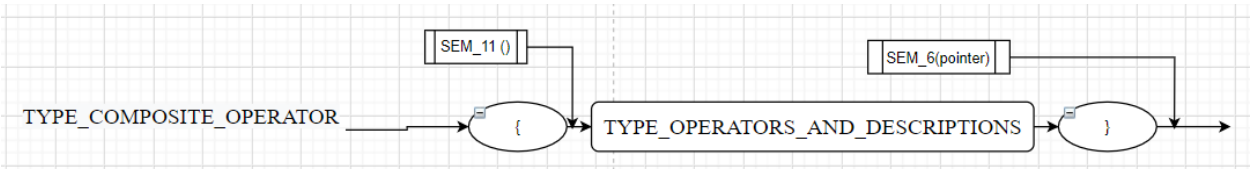
Выделение памяти под дерево происходит до начала анализа программы. Освобождении памяти происходит после завершения анализа программы.



При описании функций семантическая подпрограмма SEM_5 выделяет память для нового блока. После завершения тела функции необходимо освободить выделенную память.



При обработке составного оператора аналогично выделяется память для нового блока, которую, разумеется, необходимо освободить после выхода из составного оператора.



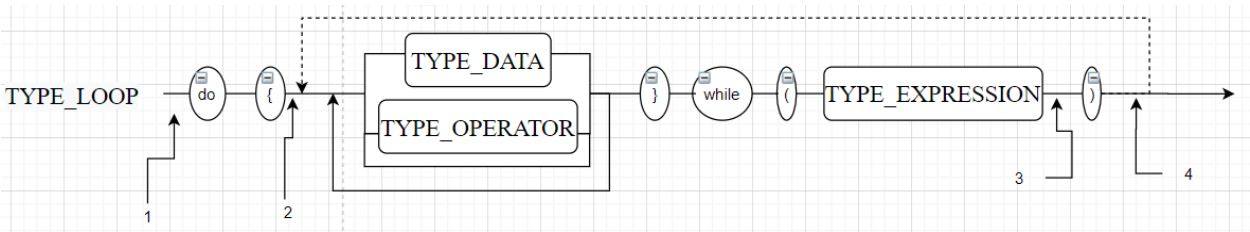
Флаг интерпретации

Встроить в программу флаг интерпретации flagInterpret.

Для всех синтаксических диаграмм, ответственных за вычисление flagInterpret, указать правила его вычисления

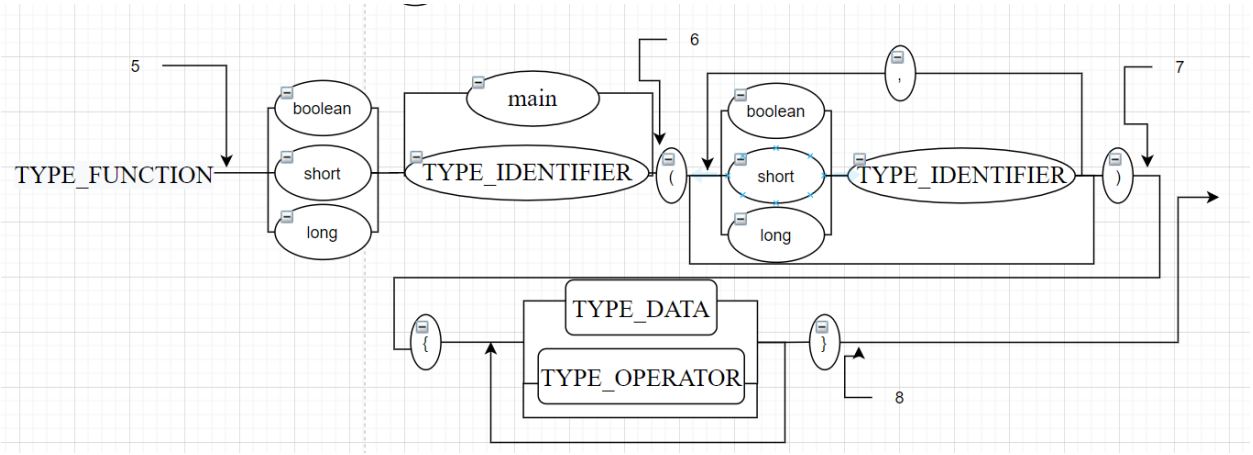
Ни одна семантическая подпрограмма не выполняется при flagInterpret равном fasle.

Цикл do-while:



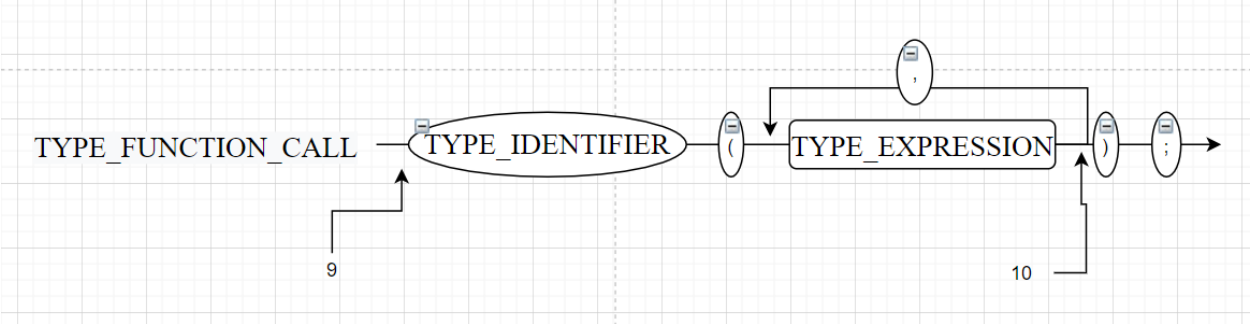
1. Сохранение флага интерпретации.
2. Сохранение позиции указателя в тексте.
3. Вычисление нового значения флага интерпретации.
4. В зависимости от значения флага интерпретации выход из цикла или повторное выполнение.

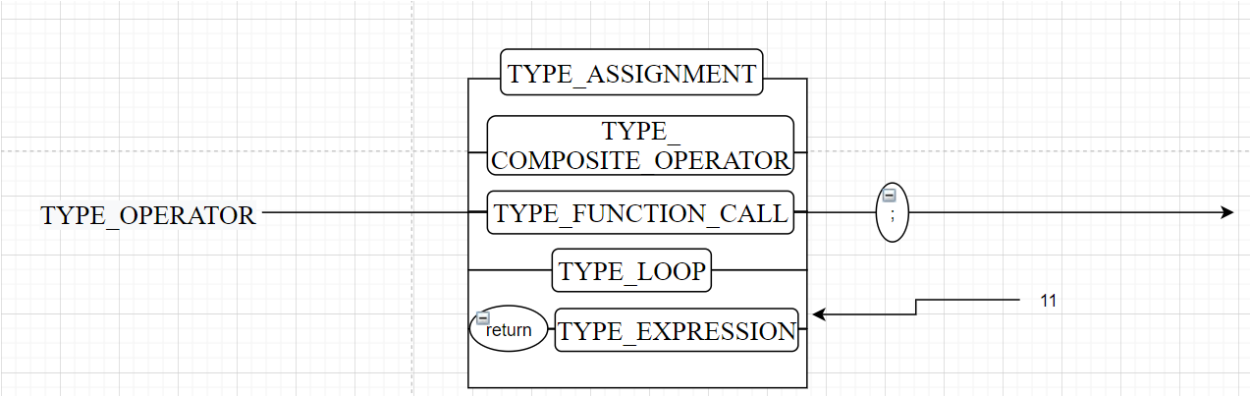
Описание функции:



5. Сохранение флага интерпретации в локальную переменную. Заголовок функции и список параметров должны интерпретироваться в любом случае.
6. Сохраняем тип терминала: main или другой идентификатор
7. Сохраняем начало тела функции и в зависимости от типа терминала:
    a. Если main, то интерпретируем тело функции;
    b. Если другой идентификатор, то не интерпретируем тело функции.
8. Восстановление флага интерпретации.

Вызов функции:





9. Запоминание позиции в тексте и указателя currentNode в дереве.
10. Вычислить и записать фактические параметры и перейти к телу функции.
11. Вычислить выражение и назначить его как значение функции. Восстановить контекст точки вызова.

ТЕСТ

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.