2.3. Разработка синтаксического анализатора

На данном этапе работ необходимо разработать грамматику в соответствии с формой Бекуса – Наура, разработать алгоритм построения дерева разбора, выполнить программную реализацию и провести тестировании.

2.3.1. Разработка грамматики

Исходный код программы представляет собой функции и директивы препроцессора. Последние определяется за счет ключевого слова «определить». Грамматика для определения функций и директив имеет следующий вид:

<code> := определить <define> | <proc>

Но так как в программе может быть функция и процедура вместе, не одна директива и/или не одна функция, то следует доработать грамматику и тогда она примет следующий вид:

<code> := определить <define> <code> | <proc> <code> | e

Такое правило означает, что символ <code> может заменяться на одну из последовательностей в зависимости от типа и значения текущего токена. То есть при значении токена равному «определить» происходит выбор последовательности, связанной с разбором директивы препроцессора. После разбора <define> происходит переход к следующему символу последовательности, так как он равен <code> , то снова происходит выбор последовательности. В случае, если токенов больше нет, то нетерминальный символ заменяется на e, обозначающий пустую последовательность и символизирующую окончание раскрытия нетерминального символа <code>.

В свою очередь нетерминальный символ <define>, отвечает за разбор директивы препроцессора. Как известно, препроцессор состоит из 2х частей: макроса и части, которая будет подставлена во время компиляции. Для простоты назовем последнюю просто препроцессором. Тогда грамматика для разбора директивы препроцессора будет следующей:

<define> := <macros> н <preproc> к

Макросом, как правило, является переменная или функция c параметрами. Но в любом из этих случаев сначала всегда идет идентификатор, а потом могут идти параметры, которые отделяются скобками, если это функция. Тогда грамматика для разбора будет выглядеть так:

<macros>:= <name> | <name> ( <parametr>

Грамматика для идентификатора имеет вид:

<name> := идентификатор

Что касается параметров, то их может быть несколько. Дополнительно стоит отметить, что параметры есть всегда, где определяются скобки, хотя бы один, но он обязательно присутствует. Это в значительной степени упрощает разработку грамматики для определения параметров и выглядит она так:

<parametr> := <value> | <value><parametr> | )

<value> := идентификатор | число

Как можно заметить, <value> и <name> отличаются лишь тем, что в первом дополнительно есть определение числа. Это было сделано с целью исключить лишнюю проверку на правильность введенных данных. То есть <name> применяется там, где должно быть только имя: имя переменной, которой присваивают значение; имя функции и т.д. А <value> используется там, где допустимо использование чисел. Например, при описании параметров.

В разрабатываем языке препроцессором может являться число, причем число может быть отрицательным, поэтому грамматика такая:

<preproc> := -<value> | <value>

Как было описано выше, программа может состоять из функций и для ее определения использовался символ <proc>. Так же, было отмечено, что функции могут иметь параметры. Кроме того, стоит помнить, что любая функция должна иметь реализацию, то есть тело. Учитывая это все, получается следующая грамматика:

<proc>:=<name> н <proc\_code> | <name> (<parametr> н <proc\_code>

Что касается тела функции, то оно состоит из различных операций:

<proc\_code> := <operation> к | <operation><proc\_code>

Так как кол-во операций, выполняющихся внутри функции может быть различным, то наиболее логичным вариантом для определения всех внутренних операций будет «отлов» конца тела функции, задаваемого скобкой «к». В случае, когда конец был обнаружен происходит завершение обработки кода процедуры и управление передается в последовательность, которая содержала <proc\_code> для дальнейшего разбора.

Операции, поддерживаемые языком следующие: арифметические (сложение, вычитание, сдвиги, деление, умножение), условие, переход, метка и выход из подпрограммы (конец), присваивание. Следовательно, грамматика разбора для операций следующая:

<operation>:=<action> | <metka> | вернуть | переход <name>| конец | если <if\_statment> | иначе <else\_statment>

<action>:= <name> | <name> (<operation> | <name> () | <name> = <value>| <name>=<name>() | <name>=<name>(<parametr>| <name> <op> <value>

<op>:=И | ИЛИ | НЕ | \* | - | + | / | >> | <<

Далее, будет разработана грамматика условия. В простом случае условный оператор формируется из самого условия, которое записывается в скобках, а потом следует код, который будет выполняться при выполнении заданного условия.

<if\_statment> := ( <condition> н <if\_code>

Условие может быть простым, то есть состоять из сравнения 2х операндов, а может быть составным, то есть состоять из сравнения нескольких операндов. В разрабатываем языке составные условия представляют собой логическое ?сравнение простых условий между собой.

Например, (a==b И a!=c).

Тогда грамматика будет иметь следующий вид:

<condition> := <first\_cond> <op2><condition> | <first\_cond>

<first\_cond> := <name> <op3><value>

<op2>:= И | ИЛИ

<op3>:= И | ИЛИ | == | !=

Что касается части для не выполнения условия, то ее грамматика:

<else\_statment> := н <else\_code>

<if\_code> и <else\_code> определяет операции, происходящие внутри условия, грамматика построена по принципу <proc\_code>.

Стоит отметить, что <else\_statment> и <if\_statment> были вынесены в разные последовательности одного правила для возможности осуществления вложенных условий.

Грамматика в общем виде представлена в приложении Б.

2.3.2. Разработка алгоритма построения дерева разбора

Результатом выполнения синтаксического разбора является построение дерева разбора, для его построения необходимо разработать соответствующий алгоритм. Схема алгоритма представлена на рисунках 6 – 14.

Дополнительно стоит отметить, что для наглядности схемы было принято решение об объединение действия «создать new\_node» для всех процедур и операций. В действительности же для каждой процедуры и операции будет создаваться свой new\_node, отличающийся типом создаваемого узла.

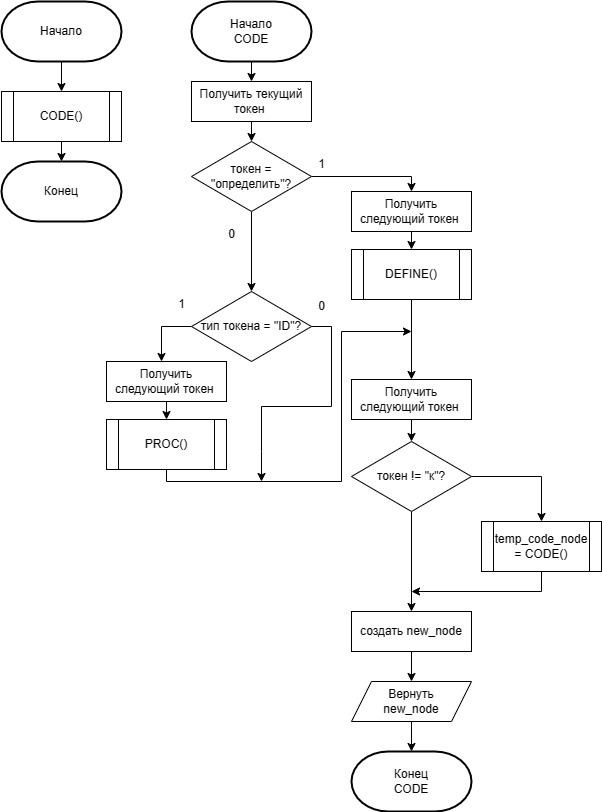


Рисунок 6 – Схема алгоритма, часть 1

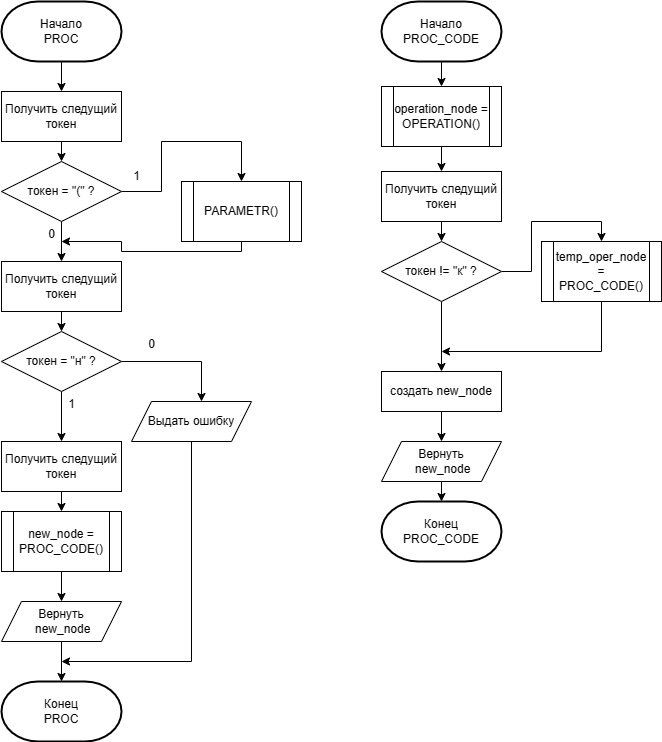


Рисунок 7 – Схема алгоритма, часть 2

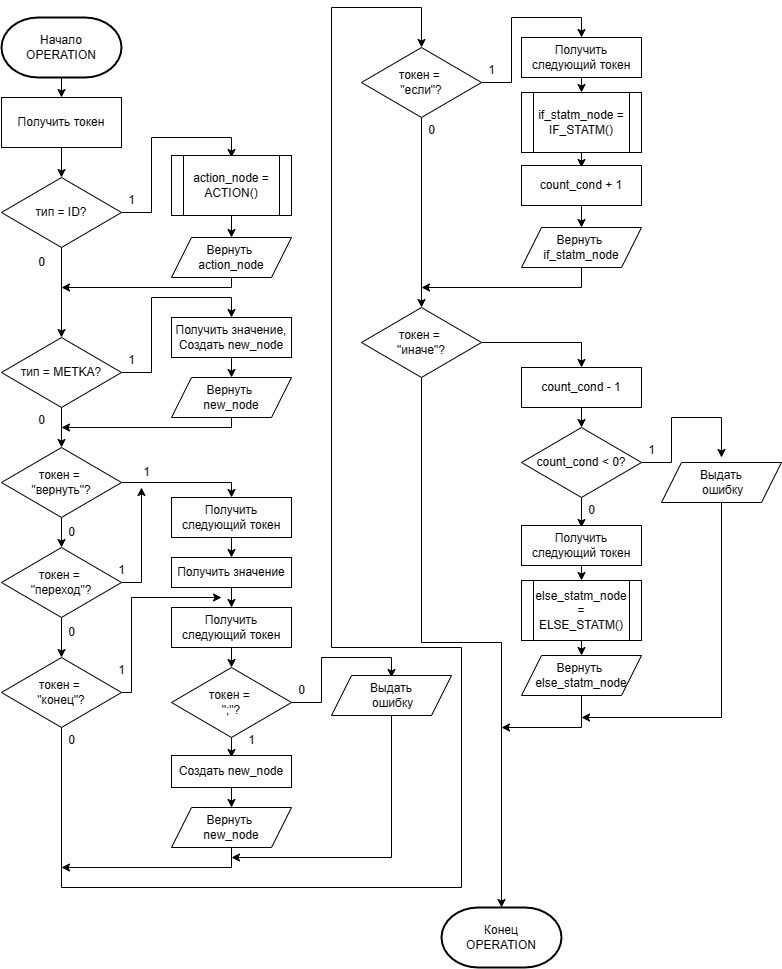


Рисунок 8 – Схема алгоритма, часть 3

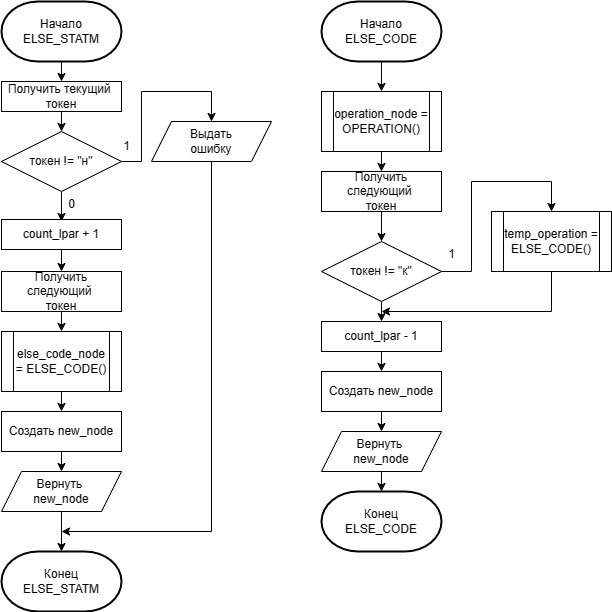


Рисунок 9 – Схема алгоритма, часть 4

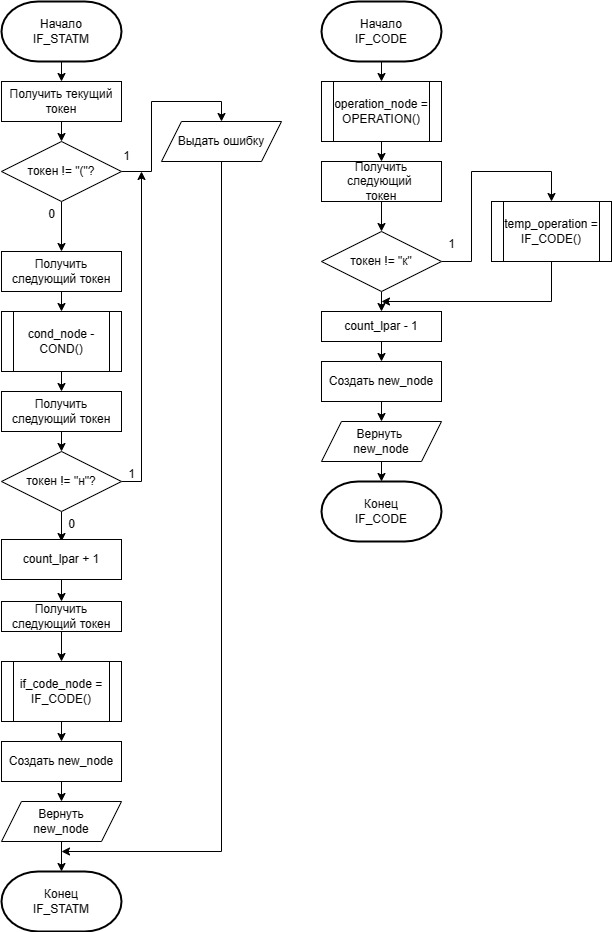


Рисунок 10 – Схема алгоритма, часть 5

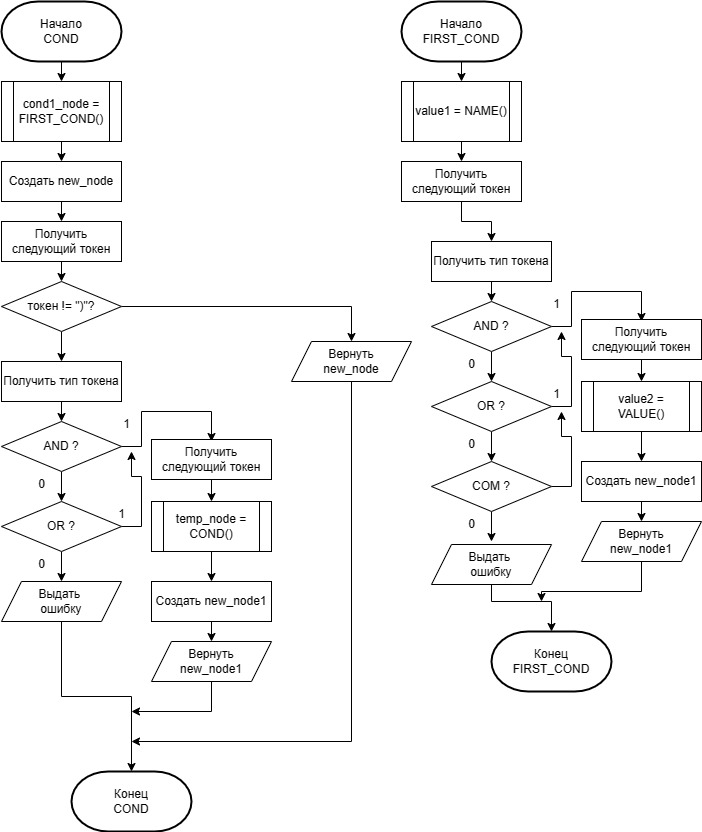


Рисунок 11 – Схема алгоритма, часть 6

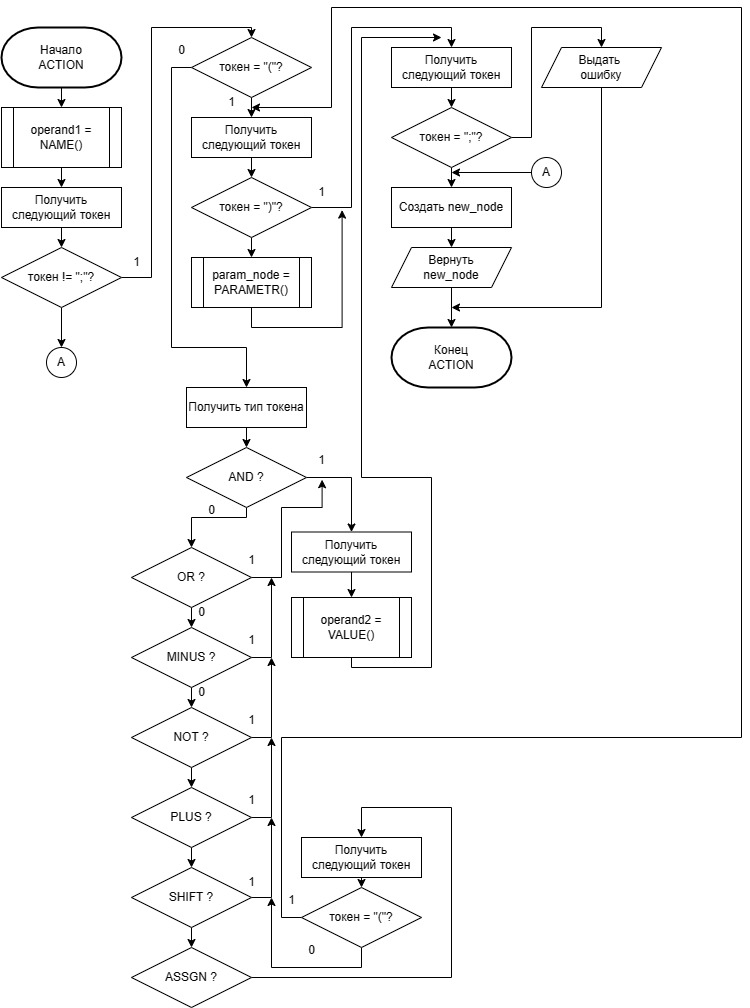


Рисунок 12 – Схема алгоритма, часть 7

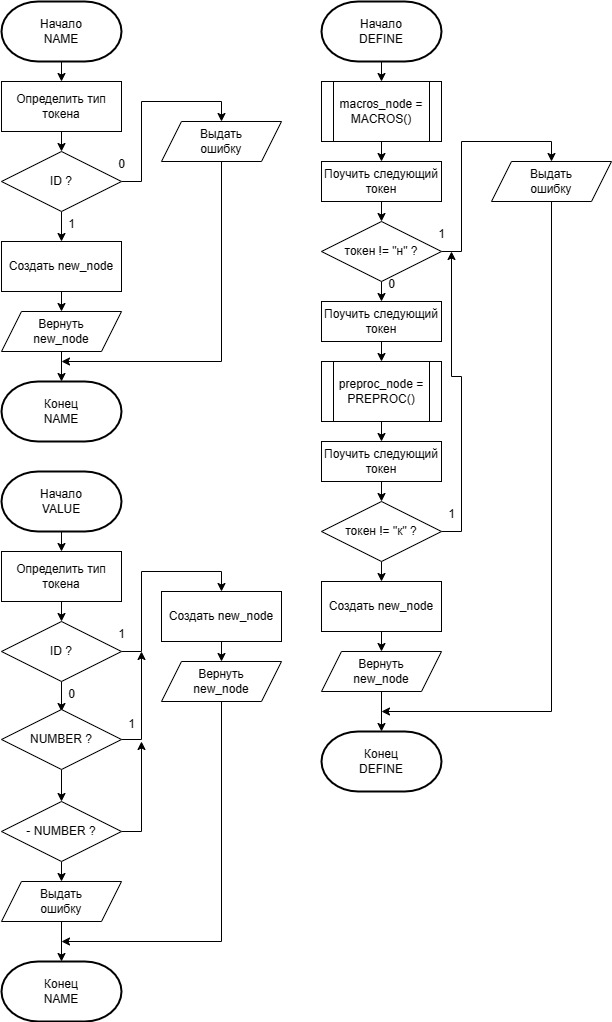


Рисунок 13 – Схема алгоритма, часть 8

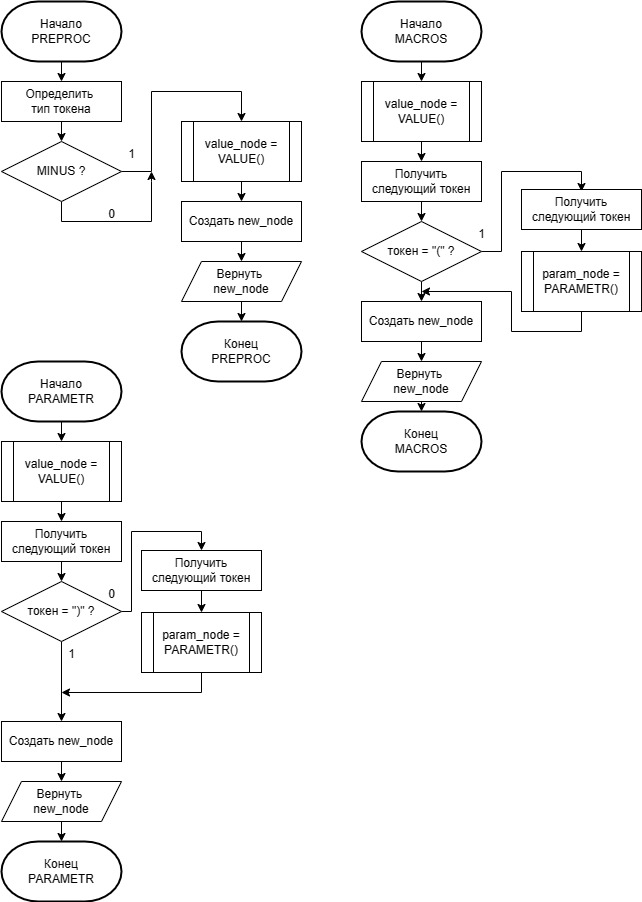


Рисунок 14 – Схема алгоритма, часть 9

2.3.3. Программная реализация

Для построения дерева разбора использовался стандартный алгоритм рекурсивного спуска. Как было отмечено выше для каждого узла выполняется действие «создать new\_node». В свою очередь new\_node является экземпляром класса node. Реализация класса node представлена ниже:

class node

{

private:

string value;

node\_type type;

node\* operand1;

node\* operand2;

public:

node(node\_type type, const string& value = "",

node\* operand1 = nullptr, node\* operand2 = nullptr)

{

this->type = type;

this->value = value;

this->operand1 = operand1;

this->operand2 = operand2;

}

};

Для построения дерева был разработан класс AST, экземпляр которого создавался в классе лексического анализатора, Parser, и вызывал функцию code. Диаграмма классов представлена на рисунке 15.

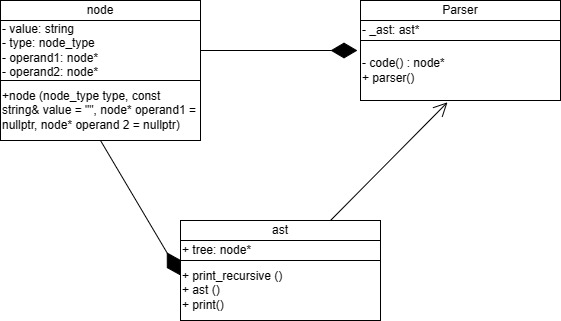


Рисунок 15 – Диаграмма классов

Создать родителя node, потом 2 экземпляра node1 и node2, эти оба

экземпляра ассоциативность с ast и с parser. Между ними сделать ассоциативность с черной стрелкой.

Пример работы синтаксического анализатора представлен на рисунках 16 -17.

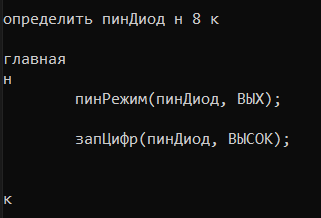


Рисунок 16 – Исходный код программы

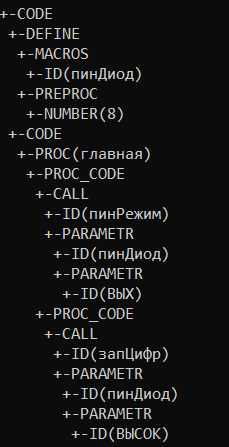


Рисунок 17 – Дерево разбора