МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ОРЛОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ И.С. ТУРГЕНЕВА»

Кафедра информационных систем и цифровых технологий

Работа допущена к защите

_____Руководитель

«____»_____20___г.

КУРСОВОЙ ПРОЕКТ

по дисциплине «Теория языков программирования и методы трансляции»

на тему: «Компилятор для подмножества языка Python»

Студент _____ Музалевский Н.С.

Шифр 190139

Институт приборостроения, автоматизации и информационных технологий

Направление подготовки 09.03.04 «Программная инженерия»

Группа 92ПГ

Руководитель _____ Гордиенко А.П.

Оценка: «_____» Дата _____

Орел 2022

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ОРЛОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ И.С. ТУРГЕНЕВА»

Кафедра информационных систем и цифровых технологий

УТВЕРЖДАЮ:

_____Зав. кафедрой

«___»_____2022г.

ЗАДАНИЕ
на курсовой проект

по дисциплине «Теория языков программирования и методы трансляции»

Студент Музалевский Н.С., Шифр 190139

Институт приборостроения, автоматизации и информационных технологий

Направление подготовки 09.03.04 «Программная инженерия»

Группа 92ПГ

1 Тема курсовой работы

«Компилятор для подмножества языка Python»

2 Срок сдачи студентом законченной работы «26» мая 2022

3 Исходные данные

Для подмножества языка Python сделать:

Определение переменных целого, вещественного, булевого типа; определение массивов и классов; определение функций и процедур; команды присваивания, условий, цикла, ввода-вывода, блока команд, вызова процедур и функций; использование таких операций как: обращение к элементу массива, обращение к полю класса, арифметические операции, операции сравнения, логические операции.

Для реализации использовать лексический анализатор на основе конечных автоматов, синтаксический анализатор методом рекурсивного спуска.

4 Содержание курсовой работы

Лексический анализ

Синтаксический анализ

Абстрактное синтаксическое дерево

5 Отчетный материал курсовой работы

Пояснительная записка курсового проекта, программа на съемном носителе

Руководитель _____ Гордиенко А.П.

Задание принял к исполнению: «11» марта 2022

Подпись студента_____

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В настоящие время сферу информационных технологий невозможно представить без компиляторов. Именно они позволили достигнуть текущего уровня развития сферы информационных технологий в целом и программирования в частности. Ведь писать код на каком-то C# или Python в миллион раз быстрее и легче чем на машинном коде. Компилятор — программа, переводящая написанный на языке программирования текст в набор машинных кодов [1].

Важность данной темы для программиста заключается в том, что глубокое понимание того, как работает компилятор, и его реализация, даёт программисту более широкое понимание синтаксиса языка и почему некоторые вещи сделаны именно так, например во многих языках программирования нельзя начинать имя переменой с цифры, так как если добавить возможность начинать с цифры сложность конечного автомата вырастет в несколько порядков и в этом нет какого-то особого смысла.

Объектом нашей работы является разработка компилятора для подмножества императивного языка Python. Предметом нашей работы является процесс создания лексического и синтаксического анализаторов, как составляющих частей компилятора для подмножества императивного языка Python.

Цель нашей работы: приобрести навыки разработки компиляторов на примере реализации компилятора для подмножества императивного языка Python.

Для достижения цели нам необходимо решить соответствующие задачи:

1) описание и построение лексического анализатора;

2) описание и построение синтаксического анализатора;

3) описание и построение абстрактного синтаксического дерева.

ЛЕКСИЧЕСКИЙ АНАЛИЗ

Первым этапом построения компилятора является лексический анализ. Лексический анализ — процесс аналитического разбора входной последовательности символов на распознанные группы — лексемы — с целью получения на выходе идентифицированных последовательностей, называемых «токенами» [1].

Для этого разработаем лексический анализатор на для подмножества Python на основе конечных автоматов. Для хранения переменной типа токен разработаем структуру:

```
struct Token
{
   enum TokenType type;
   char* value;
   int pos;
   int line;
};
```

Она будет иметь поле с типом токена, поле со значением токена, позицию начала токена в файле и номер строки, на котором расположен этот токен.

При составлении регулярных выражений надо учесть, что в python вложенность определяется символом “\t”, а окончание команды определяется концом строки. Также нужно помнить, что концом строки на Windows является “\r\n”, на Linux “\n”, на Mac “\r”.

Для начала построим недетерминированный конечный автомат для лексемы «число» (рисунок 1). Числом будет считаться последовательность цифр, с точкой по не с краю этой последовательности или же без точки. Регулярное выражение будет выглядеть следующим образом: [0-9]+(.[0-9]+)? «другой символ»

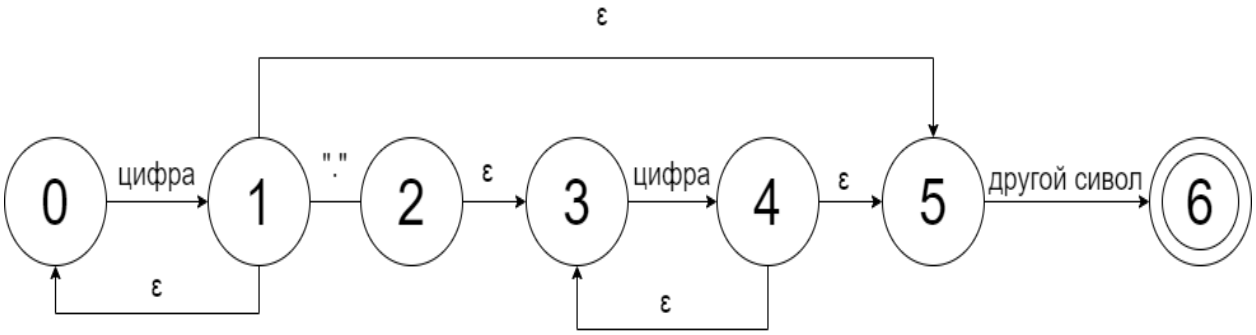

Рисунок 1 – Недетерминированный конечный автомат для числа

Преобразуем недетерминированный конечный автомат к детерминированному конечному автомату алгоритмом Томсона. Для наглядности построим таблицу переходов рисунок 2.

	цифра	"."	другой символ
{0}	{0,1,5}	-	-
{0,1,5}	{0,1,5}	{2,3}	{6}
{2,3}	{3,4,5}	-	-
{3,4,5}	{3,4,5}	-	{6}
{6}	-	-	-

Рисунок 2 – Таблица переходов для числа

По таблице переходов построим детерминированный конечный автомат рисунок 3.

Далее попробуем минимизировать детерминированный конечный автомат. Разделим в начале на 2 группы конечные состояния и неконечные. Получиться первая группа - S0, S1, S2, S3 и 2 группа S4. Во 2 группе всего 1 элемент его рассматривать не будем, так как ему не с кем быть эквивалентным. Теперь разберёмся с первой группой. Все состояния являются не эквивалентными. Получается у нас уже минимизированный конечный автомат.

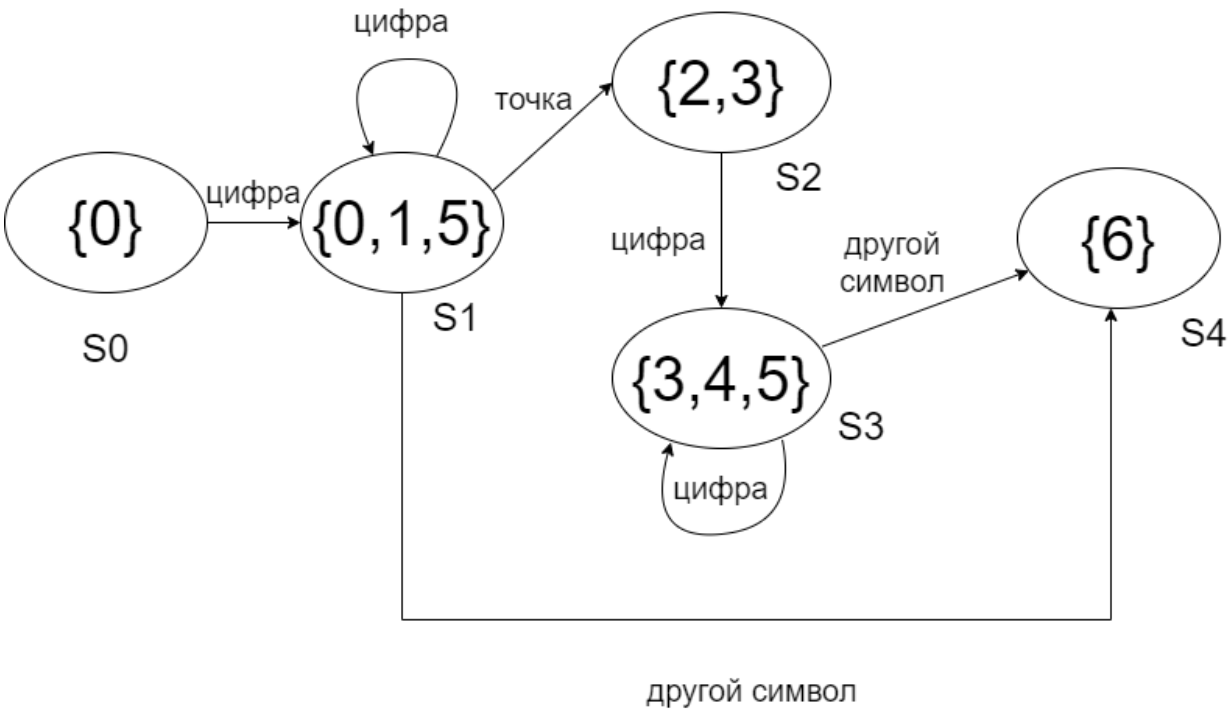

Рисунок 3 – Детерминированный конечный автомат для числа

Теперь рассмотрим лексему «переменная». Она будет состоять из последовательности букв, цифр и нижних подчёркиваний, и нельзя начинать с цифры. Регулярное выражение выглядит следующим образом ([A-Za-z] | _) ([A-Za-z0-9]|_)*

Для начала построим недетерминированный конечный автомат (рисунок 4)

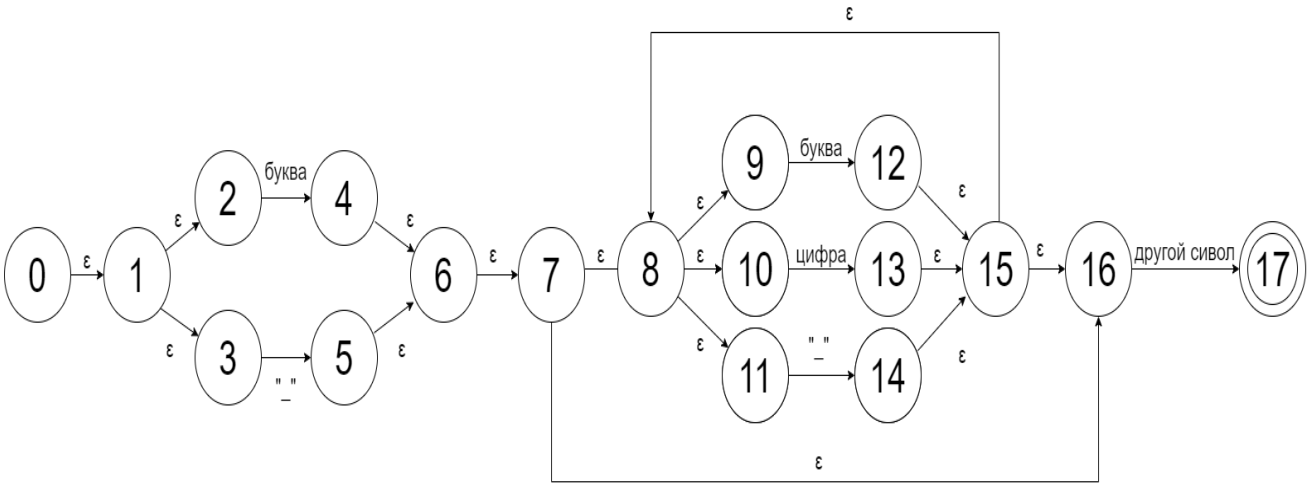

Рисунок 4 – Недетерминированный конечный автомат для переменной

Так как он слишком объёмный разделим его не 2 части R1 и R2. (рисунок 5)

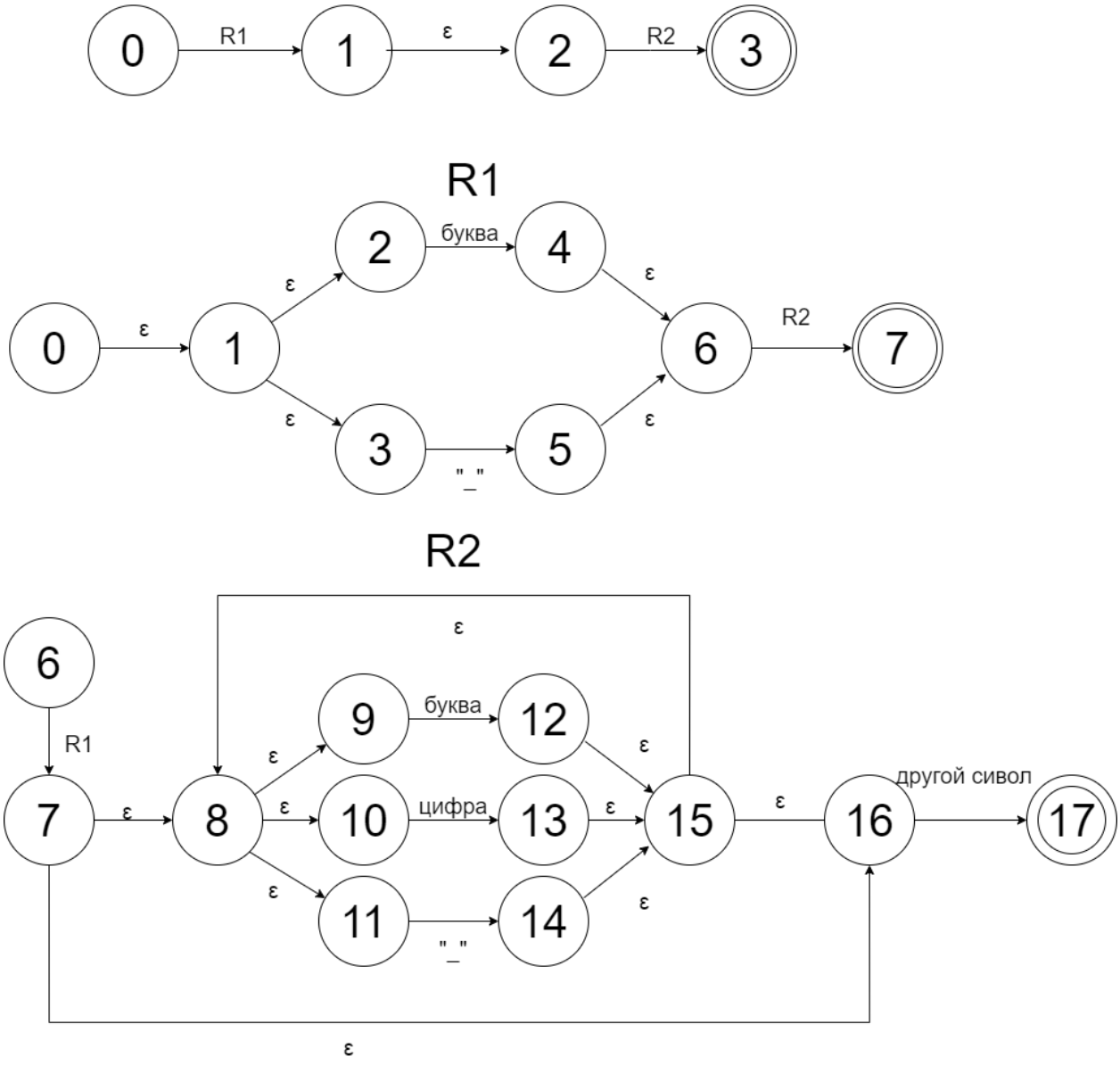

Рисунок 5 – Недетерминированный конечный автомат для переменной разделённый на 2 части

Преобразуем каждую часть недетерминированного конечный автомат к детерминированному конечному автомату алгоритмом Томсона. Для наглядности построим таблицы переходов рисунок 6 и рисунок 7.

Для R1

	буква	"_"	R2
{0}	{1,2,3,4,6,7	{1,2,3,5,6,7}	-
{1,2,3,4,6,7}	-	-	{7}
{1,2,3,5,6,7}	-	-	{7}

Рисунок 6 – Таблица переходов для R1

Для R2					
	R1	буква	цифра	"_"	другой символ
{6}	{7,8,9,10,11,16}				
{7,8,9,10,11,16}		{8,9,10,11,12,15,16}	{8,9,10,11,13,15,16}	{8,9,10,11,14,15,16}	{17}
{8,9,10,11,12,15,16}		{8,9,10,11,12,15,16}	{8,9,10,11,13,15,16}	{8,9,10,11,14,15,16}	{17}
{8,9,10,11,13,15,16}		{8,9,10,11,12,15,16}	{8,9,10,11,13,15,16}	{8,9,10,11,14,15,16}	{17}
{8,9,10,11,14,15,16}		{8,9,10,11,12,15,16}	{8,9,10,11,13,15,16}	{8,9,10,11,14,15,16}	{17}

Рисунок 7 – Таблица переходов для R2

По каждой таблице переходов построим детерминированный конечный автомат рисунок 8.

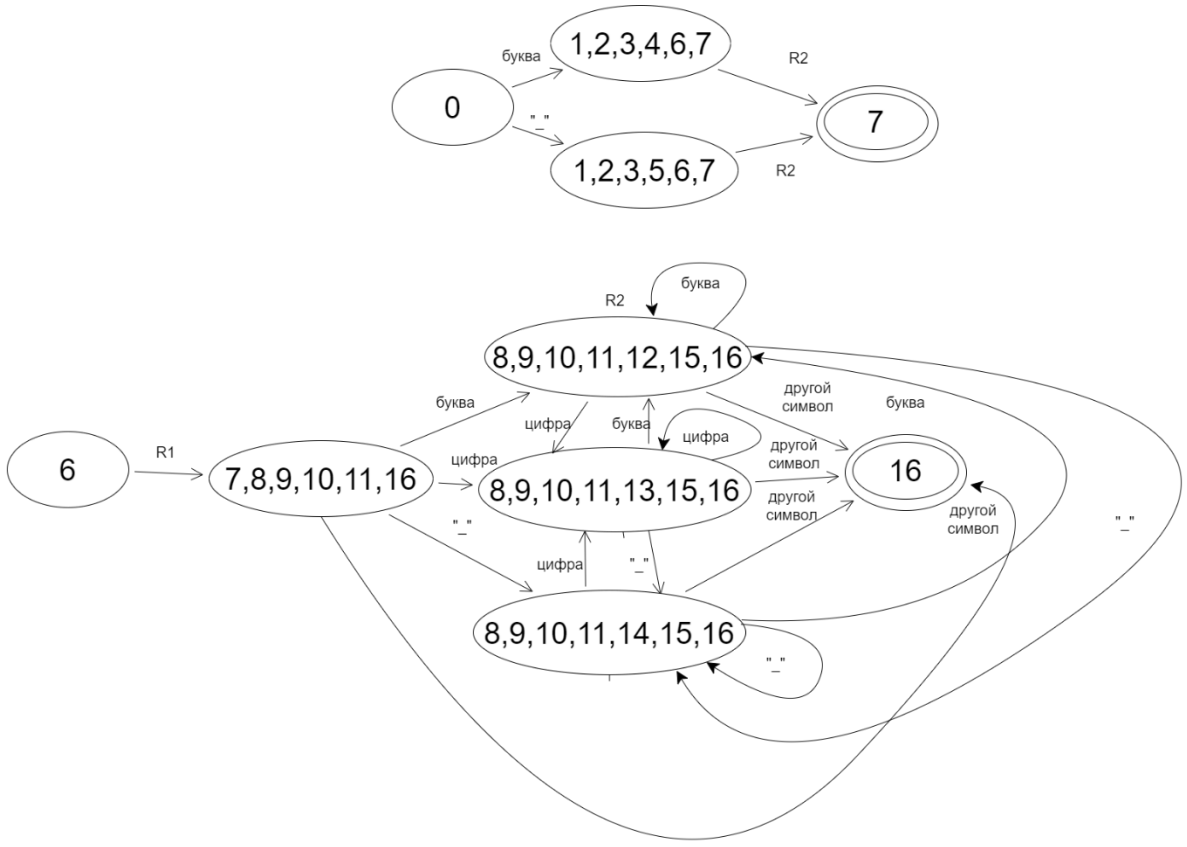

Рисунок 8 – Детерминированный конечный автомат R1 и R2

Сладом стоит объединить R1 и R2, так как «1,2,3,4,5,6,7» и «1,2,3,5,6,7» в R1 и «7,8,9,10,11,16» содержат цифру 7, они канкатенируют.

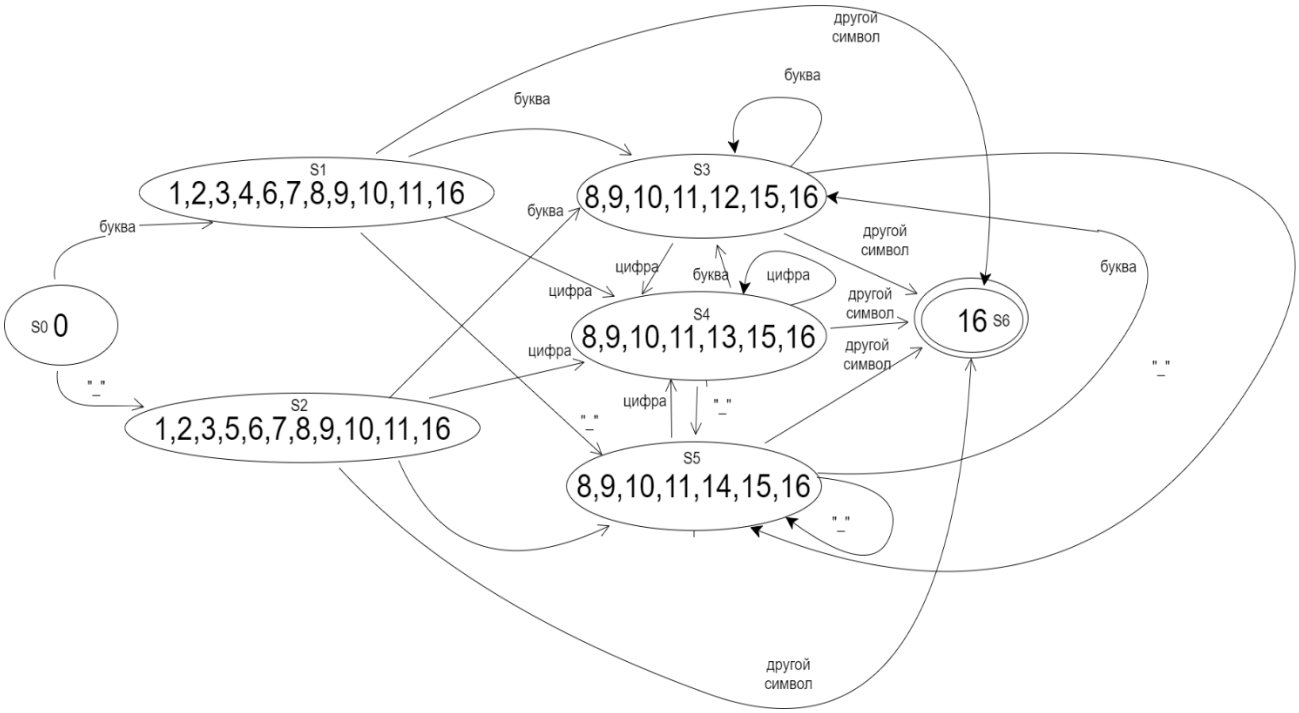

Рисунок 9 – Детерминированный конечный автомат переменной

Далее попробуем минимизировать детерминированный конечный автомат. Разделим в начале на 2 группы конечные состояния и неконечные. Получиться первая группа - S0, S1, S2, S3, S4, S5 и 2 группа S6. Во 2 группе всего 1 элемент его рассматривать не будем, так как ему не с кем быть эквивалентным. Теперь разберёмся с первой группой. Все элементы кроме S0 являются k0 эквивалентными, также они являются k1 эквивалентными, а это значит, что S1, S2, S3, S4, S5 являются эквивалентными и их можно упростить в 1 элемент. (рисунок 10)

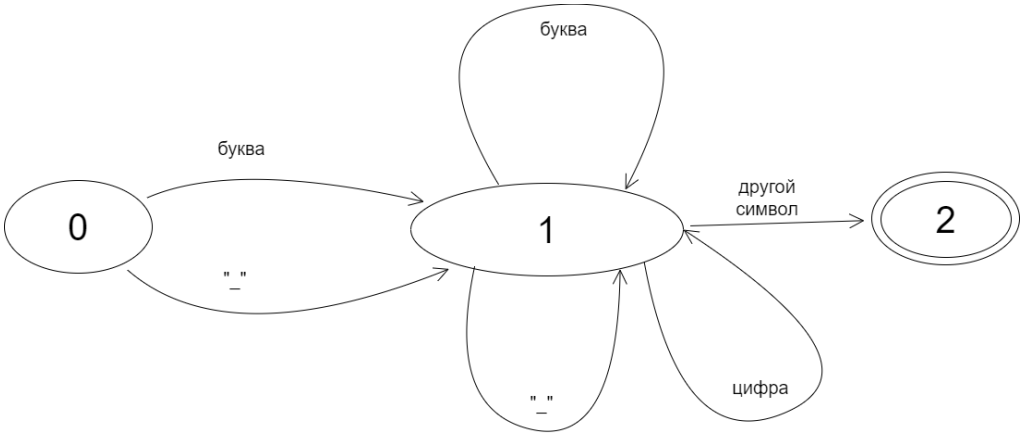

Рисунок 10 – Минимизированный конечный автомат переменной

Все ключевые слова имеют почти одинаковые регулярные выражения. Различия в них только в последовательности букв этих слов. Регулярное выражение будет выглядеть следующим образом: If [^([A-Za-z0-9] _)]. За место If можно поставить любое другое ключевое слово. В отличии от переменной и числа конец лексемы проверяется не любым другим символом, а именно теми символами, которые могут встретится в переменной. Это сделано для того, чтобы такие выражения как print1, не распознавались как ключевое слово «print» и цифра «1», а чтобы это была переменная print1.

Для обработки ключевых слов на примере if сделаем недетерминированный конечный автомат (рисунок 11).

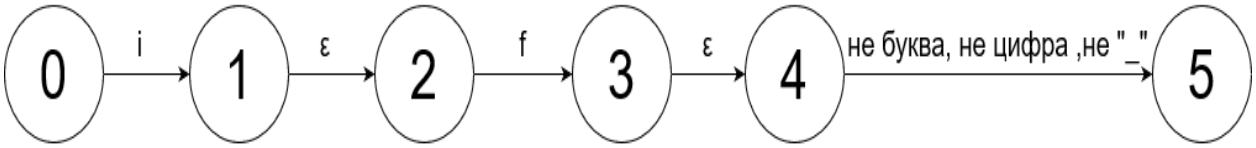

Рисунок 11 – Недетерминированный конечный автомат для ключевого слова if

Преобразуем недетерминированный конечный автомат к детерминированному конечному автомату алгоритмом Томсона. Для наглядности построим таблицу переходов рисунок 12.

	i	f	не буква, не цифра, не "_"
{0}	{1,2}	-	-
{1,2}	-	{2,3,4}	-
{2,3,4}	-	-	{4,5}
{4,5}	-	-	-

Рисунок 12 – Таблица переходов для ключевого слова if

По таблице переходов построим детерминированный конечный автомат рисунок 13.

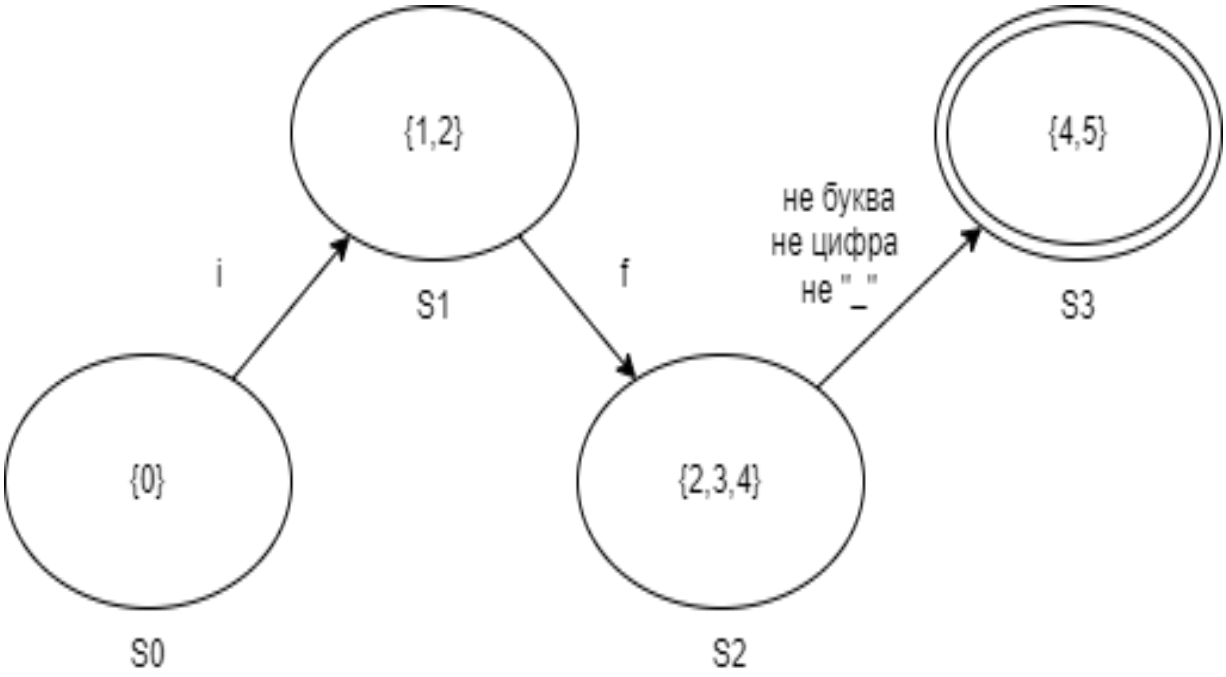

Рисунок 13 – Детерминированный конечный автомат для ключевого слова if

Далее попробуем минимизировать детерминированный конечный автомат. Разделим в начале на 2 группы конечные состояния и неконечные. Получиться первая группа - S0, S1, S2 и 2 группа S3. Во 2 группе всего 1 элемент его рассматривать не будем, так как ему не с кем быть эквивалентным. Теперь разберёмся с первой группой. Все состояния являются не эквивалентными. Получается у нас уже минимизированный конечный автомат.

Для остальных знаков проверка на символы, которые находятся после лексемы не нужна.

СИНТАКСИЧЕСКИЙ АНАЛИЗ

После лексического анализа, следует этап называемый синтаксическим анализом. Синтаксический анализ — процесс сопоставления линейной последовательности лексем (слов, токенов) естественного или формального языка с его формальной грамматикой [3]. Реализация синтаксического анализа буде происходить методом рекурсивного спуска. Метод рекурсивного спуска — алгоритм нисходящего синтаксического анализа, реализуемый путём взаимного вызова процедур, где каждая процедура соответствует одному из правил контекстно-свободной грамматики. Применения правил последовательно, слева-направо поглощают токены, полученные от лексического анализатора. Это один из самых простых алгоритмов синтаксического анализа, подходящий для полностью ручной реализации [5].

Перед началом реализации самого синтаксического анализатора нужно разработать ряд правил.

1) Так как в Python не функции main или ей подобных, то началом программы может быть любое выражение. Из-за этого Первая вызываемая функция будет обычная последовательность команд. Этот нетерминал был назван «блок». Он содержит следующий синтаксис: [таб] «обычное утверждение» «блок» | [таб] «утверждение класса» «блок» | [таб] «утверждение функции» «блок» | эпсилон;

2) Как мы увидели из предыдущего правила у нас есть 3 вида утверждений. Утверждение -это вызов какого-либо команды указанной в правиле. Так как команды которые вложены в класс, функцию или которые не имеют вложенности, могут быть использованы только в одном из перечисленных, было решение разделить на 3 вида утверждений:

1. «обычное утверждение» → «условный оператор if» | «оператор цикла while» | «оператор цикла for» | «операция присваивания» | «команда ввода» | «команда вывода» | «определение класса» | «определение функции» | «вызов функции» | “\n”;

2. «утверждение класса» → «операция присваивания» | «определение функции» | “\n”;

3. «утверждение функции» → «обычное утверждение» | «возврат».

Теперь разберём сами команды, которые находятся в утверждениях.

3) Начнём с определения класса оно состоит из ключевого сова (class), переменной, открывающийся и закрывающийся скобки, двоиточия и знака разделения строки. Далее повышается вложенность и вызывается новая последовательность «блок». Правило выглядит следующим образом: «определение класса» → “class” «переменная» “(“ “)” “:” “\n” «блок»;

4) Операторы циклов работают точно также как и класс, имеют последовательность терминалов и нетерминалов и следом за ним новая последовательность «блок»: «оператор цикла while» → “while” «логическая операция» ":" "\n"«блок», «оператор цикла for» → “for” «значение» “in” «значение» ":" "\n" «блок»;

5) После операторов циклов зададим плавило для оператора условия if. Сначала зададим последовательность терминалов и нетерминалов в соответствии с синтаксисом Python, после неё будет идти новый «блок», а за ним может быть будет else тоже с новым «блоком». Правило будет выглядить так: «условный оператор if» → “if” «логическая операция» ":" "\n"«блок» [“else” ":" "\n" «блок»];

6) Теперь рассмотрим определение функции оно очень похоже на определение класса за исключением некоторых терминалов и тем что в скобочках могут быть аргументы. Они являются ещё 1 правилом которое имеет синтаксис: «аргументы для определения функции» → «переменная» [“,” | «аргумент для определения функции»] | эпсилон, «аргумент для определения функции» → «переменная» [“,” | «аргумент для определения функции»]. Разделение на 2 правила нужно для того, чтобы не срабатывал случай, когда указана запятая, но при повторном вызове правила мог быть эпсилон, что является ошибкой;

7) Вызов функции имеет следующий синтаксис: «вызов функции» → «переменная» "(" «аргументы» ")" "\n". Правило «аргументы» отличается от правила «аргумент для определения функции» тем, что в «аргументы» используется правило значение (которое содержит и обращение к элементу класса и к элементу массива, обычная переменна, число, булевое значение), а не просто переменная;

8) Так как в Python нет как такого объявления переменных, а есть только их определение, или же обычное присвоение переменной какого-либо значения. Присвоить переменной мы можем арифметическое выражение, команду ввода, команду вызова функции и правило определения массива. Правило выглядит следующим образом: «операция присваивания» → «идентификатор» "=" «арифметическое выражение» "\n" | «идентификатор» "=" «команда чтения» "\n" | «идентификатор» "=" «вызов функции» "\n" | | «идентификатор» "=" «определение массива» "\n";

9) Рассмотрим теперь правило «определение массива», которое упоминалось чуть выше. Оно содержит в себе открывающуюся и закрывающуюся квадратную скобку, а посередине используется правило «аргументы», описанное выше;

10) Команды ввода вывода имеют простой синтаксис они просто проверяют последовательность терминалов и нетеминалов. Правило выглядит следующим образом: «команда ввода» → “input” “(“ “)” “\n”; «команда вывода» → “print” “(“ «арифметическое выражение»” “)” “\n;

11) Чтобы процедура стала функцией, надо добавить к ней return. Для этого в правиле «утверждение функции», добавлено правило возврат. Само правило имеет следующий синтаксис;

12) Осталось рассмотреть 3 правила: «логическая операция» «логическое выражение» и «арифметическое выражение». Они пости идентичны, каждая операция в начале может содержать скобку, потом на правило, в которое надо опуститься, далее идёт или не идёт соответствующий знак и такая же операция. Синтаксис 3 правил: «логическая операция» → [“(“]

«логическое выражение» [«логический знак» «логическая операция»] [“)“] «логическое выражение» → [“(“] «арифметическое выражение» [«знак сравнения» «логическое выражение»] [“)“] «арифметическое выражение» →[“(“]«значение» [«арифметический знак» «арифметическое выражение»] [“)“];

Со всеми правилами можно ознакомиться в приложении Б.

Сам синтаксический анализатор будет реализован на языке Си методом рекурсивного спуска вниз. Также во время выполнения будет строиться рекурсивного спуска будет строиться сразу абстрактное синтаксическое дерево. Рассмотрим пример выполнения правила «команда вывода»:

```
bool OutputCommand()
{
	tempCurrentToken = currentToken;
	if (Is(Output))
	{
		struct Node* outputNode = NewNode(tokens[currentToken]);
		if (Is(OpenBracket))
		{
			if (ArithmeticExpressionMain(true, outputNode))
			{
				if (Is(Delimiter))
				{
					AddChildNode(outputNode, current);
					currentToken = tempCurrentToken;
					return true;
				}
			}
		}
	}
	return false;
}
```

Сначала она проверяет первый токен является ли “print”, следом она проверяет токен открытая скобка “(“, потом проверяет правило арифметическое выражение, наконец она проверяет токен на наличие конца строки. Если все терминалы и нетерминалы совпали, то функция возвращает true.

АБСТРАКТНОЕ СИНТАКСИЧЕСКОЕ ДЕРЕВО

На прошлом шаге во время работы синтаксического анализатора было построено абстрактное синтаксическое дерево. Дерево абстрактного синтаксиса (ДАС) — в информатике конечное помеченное ориентированное дерево, в котором внутренние вершины сопоставлены (помечены) с операторами языка программирования, а листья — с соответствующими операндами. Таким образом, листья являются пустыми операторами и представляют только переменные и константы [4].

Для реализации дерева была создана структура данных «Node»:

```
struct Node
{
	enum NonTerminalType type;
	struct Token data;
	struct Node* parent;
	struct Node* next;
	struct Node* childs;
};
```

Она имеет 2 поля с данными для данных о узле, также каждый узел имеет указатель на родителя(parent), на брата(next) и на дочерний узел(childs). Так как правило «блок» может иметь несколько «утверждений» будет не удобно использовать обычное бинарное дерево, потому что будет слишком много лишних узлов, то это будет слишком затратно по памяти. Так как для левостороннего нисходящего обхода дерева нужно только последовательное прохождение по всем ветвям и нам нет необходимости обращаться произвольно к конкретной ветви, то вместо указателя на массив дочерних узлов присутствует только указатель на первый дочерний узел, а далее, чтобы пройтись по всем дочерним узлам дерева нужно прости пройти по одностороннему односвязному списку от указателя на дочерний узел текущего узла по указателям next.

Чтобы автоматизировать процесс добавления дочерних узлов и братьев, создадим несколько функций. Первая из которых добавление дочернего узла.(Рисунок 14)

```
struct Node * AddChild(struct Token data, struct Node* node)
{
    if (node == NULL)
    {
        node = malloc(sizeof(struct Node));
        node->data = data;
        node->type = Terminal;
        node->next = node->childs = NULL;
        return node;
    }
    else
    {
        if (node->childs == NULL)
        {
            node->childs = malloc(sizeof(struct Node));
            node->childs->data = data;
            node->childs->type = Terminal;
            node->childs->next = node->childs->childs = NULL;
            node->childs->parent = node;
            return node->childs;
        }
        struct Node** temp = &node->childs;
        while ((*temp)->next != NULL)
        {
            temp = (&((*temp)->next));
        }
        if ((*temp)->next == NULL)
        {
            (*temp)->next = malloc(sizeof(struct Node));
            (*temp)->next->parent = node;
            (*temp)->next->type = Terminal;
            (*temp)->next->data = data;
            (*temp)->next->next = (*temp)->next->childs = NULL;
            return (*temp)->next;
        }
    }
}
```

Рисунок 14 – Функция добавления дочернего узла

Для начала разберём аргументы функции. Первый аргумент - это данные которые мы добавляем в дочерний узел. Второй аргумент отвечает за указатель на узел к которому нужно добавить дочерний. Функция возвращает указатель на добавленный элемент.

В начале функции мы проверяем существует ли узел, к которому мы будем добавлять дочерний узел, в случае если не существует, то мы сначала выделяем память под узел, далее вносим данные в узел и обнуляем ссылку на следующий и на дочерний узел. В случае существования мы переходим к первому дочернему узу и проверяем существует ли он. В случае если не существует выделяем ему память, заносим данные, устанавливаем ссылку на родителя и обнуляем другие указатели. В противном случае проходим по связному списку в поисках пустого узла. При нахождении выделяем память устанавливаем ссылку на родителя, обнуляем указатели, заносим данные.

Также создадим подобную функцию, но только в 1 аргументе будет не данные, а дочерний узел. Так как в си нет как токовых перегрузок назовём её AddChildNode. Также создадим функции для добавления братьев и на добавление родителя.

В ветвях дерева может также содержаться несколько терминалов: BlockTerm, Params, LogicalOperationTerm. BlockTerm - отвечает за вложенность операций в нём, Params – отвечает за аргументы в объявлении функции, аргументы при вызове функции, параметры которыми заполнен массив (рисунок 15). LogicalOperationTerm – отвечает за условия в while, if.

Также стоит отмети как собирается узлы абстрактного синтаксического дерева. Каждой функции проверки правила грамматики передаётся ссылка на узел, к которому нужно добавить дочерний узел. При проверке правила начинает формируется узел, в случае удачной проверки, переданному в функцию узлу, присваивается сформированный узел.

Рисунок 15 – Пример вывода параметров функции

Как мы видим у узла функции (Def) если 3 дочерних узла. Первый из них отвечает за название функции. Второй отвечает за аргументы функции. Третий отвечает за последовательность утверждений входящих в функцию. Также строит отметить, что в конце последовательности утверждений был встречен return который возвращает значение 1.

Теперь рассмотрим сколько потомков имеют конкретные узлы. Начнём с узлов, которые имеют 3 дочерних узла это узлы с объявления функции, объявлением класса, if, while, for. Далее рассмотрим, узел Block который содержит произвольное количество дочерних узлов зависящие от количества утверждений в блоке. Остальные узлы содержат по 2 дочерних узла.

Реализация абстрактного синтаксического дерева приведена в приложении Е и Ё.

Для наглядности маленький фрагмент построения дерева, представлен на рисунке 16.

Рисунок 16 – Пример вывода параметров функции

ЗАКЛЮЧЕНИЕ

В результате выполнения курсового проекта были описаны принципы построения лексического анализатора, синтаксического анализатора и абстрактного синтаксического дерева для подмножества языка Python.

В ходе данной работы был реализован лексический анализатор на основе конечных автоматов, синтаксический анализатор методом рекурсивного спуска, а также абстрактное синтаксическое дерево.

В результате выполнения работы были получены знания о принципах работы лексического анализатора и синтаксического анализатора, а также закреплены на практике в результате их реализации на языке Си. Мы узнали, как лексический анализатор разбивает исходную последовательность на последовательность лексем(токенов), далее, как синтаксический анализатор проверяет последовательность токенов на соответствие с грамматикой языка Python и строит на основе рекурсивного спуска абстрактное синтаксическое дерево.

Исходя из результатов работы можно сделать вывод, что все поставленные задачи были выполнены, и, следовательно, цель данного курсового проекта была достигнута.

СПИСОК ЛИТЕРАТУРЫ

1. Компилятор [Электронный ресурс]. - Режим доступа: https://ru.wikipedia.org/wiki/Компилятор (дата обращения: 12.04.2022).

2. Лексический анализ [Электронный ресурс]. - Режим доступа: https://ru.wikipedia.org/wiki/Лексический_анализ (дата обращения: 19.04.2022).

3. Синтаксический анализ [Электронный ресурс]. - Режим доступа: https://ru.wikipedia.org/wiki/Синтаксический_анализ (дата обращения: 25.04.2022).

4. Абстрактное синтаксическое дерево [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/Абстрактное_синтаксическое_дерево (дата обращения: 09.05.2022).

5. Метод рекурсивного спуска [Электронный ресурс]. - Режим доступа: https://ru.wikipedia.org/wiki/Метод_рекурсивного_спуска (дата обращения: 10.05.2022).

6. Конечный автомат [Электронный ресурс]. - Режим доступа: https://ru.wikipedia.org/wiki/Конечный_автомат (дата обращения: 11.05.2022).

7. Python [Электронный ресурс]. - Режим доступа: https://ru.wikipedia.org/wiki/Python (дата обращения: 11.05.2022).

ПРИЛОЖЕНИЕ А

(обязательное)

РЕАЛИЗАЦИЯ ЛЕКСИЧЕСКОГО АНАЛИЗАТОРА ДЛЯ ПОДМНОЖЕСТВА ЯЗЫКА PYTHON(lexer.h)

```
#ifndef LEXER_H
#define LEXER_H
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <ctype.h>
#include "token.h"

enum number {
   numbers,
   one_dot_numbers,
   finish_number,
   no_number
};

bool IsNumber(char* str);
void FinishToken(enum TokenType type, int* tokenQuantity);
struct Token* Lexer(char* content,int * tokenQuantity);
bool CompareStrings(char* str);
bool CompareStringCheck(char* str);

#endif
```

ПРИЛОЖЕНИЕ Б

(обязательное)

РЕАЛИЗАЦИЯ ЛЕКСИЧЕСКОГО АНАЛИЗАТОРА ДЛЯ ПОДМНОЖЕСТВА ЯЗЫКА PYTHON(lexer.c)

```
#include "lexer.h"

char* fileContent;

int currentTokenLength = 0;
int idx = 0;

int lines = 0;

struct Token * tokenS;

void FinishToken(enum TokenType type, int * tokenQuantity)
{
   struct Token* temp = malloc((*tokenQuantity + 1)*sizeof(struct Token));
   for (int i = 0; i < *tokenQuantity; i++)
   {
      temp[i] = tokenS[i];
   }
   temp[*tokenQuantity].type = type;
   char buffer[currentTokenLength + 1];
   memcpy(buffer, &fileContent[idx - currentTokenLength], currentTokenLength);
   buffer[currentTokenLength] = '\0';
   temp[*tokenQuantity].value = malloc(strlen(buffer) + 1);
   strcpy(temp[*tokenQuantity].value, buffer);
   temp[*tokenQuantity].pos = idx - currentTokenLength;
   temp[*tokenQuantity].line = lines;
   if (tokenS)
   {
      free(tokenS);
   }
   tokenS = temp;
   (*tokenQuantity)++;
   currentTokenLength = 0;
}

struct Token * Lexer(char* content, int * tokenQuantity)
{
   enum TokenType state = Delimiter;
   tokenS = malloc(0);
   fileContent = content;
   bool io_Rooted = false;
   bool identificatorFirst = true;

   while (idx < strlen(fileContent))
   {
```

```
switch (state)
{
case Delimiter:
   if ((idx + 1 < strlen(fileContent) && fileContent[idx] == '\r'&& fileContent[idx+1] =='\n'))
   {
      lines++;
      idx+=2;
      currentTokenLength+=1;
      FinishToken(Delimiter, tokenQuantity);
      state = Delimiter;
   }
   else if ( fileContent[idx] == '\n' || fileContent[idx] == '\0')
   {
      lines++;
      idx++;
      currentTokenLength++;
      FinishToken(Delimiter, tokenQuantity);
      state = Delimiter;
   }
   else
   {
      state = Tab;
   }
   break;
case Tab:
   if(CompareStrings("\t"))
   {
      FinishToken(Tab, tokenQuantity);
      state = Delimiter;
   }
   else
   {
      state = Comma;
   }
   break;
case Comma:
   if (CompareStrings(","))
   {
      FinishToken(Comma, tokenQuantity);
      state = Delimiter;
   }
   else
   {
      state = Dot;
   }
   break;
case Dot:
   if (CompareStrings("."))
   {
      FinishToken(Dot, tokenQuantity);
      state = Delimiter;
   }
   else
```

```
    {
        state = DoubleDot;
    }
    break;
case DoubleDot:
    if (CompareStrings(":"))
    {
        FinishToken(DoubleDot, tokenQuantity);
        state = Delimiter;
    }
    else
    {
        state = Bool;
    }
    break;
case Bool:
    if (CompareStringCheck("true")||CompareStringCheck("false"))
    {
        FinishToken(Bool, tokenQuantity);
        state = Delimiter;
    }
    else
    {
        state = Logical;
    }
    break;
case Logical:
    if (CompareStringCheck("and")||CompareStringCheck("or")||CompareStringCheck("not"))
    {
        FinishToken(Logical, tokenQuantity);
        state = Delimiter;
    }
    else
    {
        state = OpenBraces;
    }
    break;
case OpenBraces:
    if (CompareStrings("["))
    {
        FinishToken(OpenBraces, tokenQuantity);
        state = Delimiter;
    }
    else
    {
        state = CloseBraces;
    }
    break;
case CloseBraces:
    if (CompareStrings("]"))
    {
        FinishToken(CloseBraces, tokenQuantity);
        state = Delimiter;
```

```
        }
        else
        {
          state = OpenBracket;
        }
        break;
      case OpenBracket:
        if (CompareStrings("("))
        {
          FinishToken(OpenBracket, tokenQuantity);
          state = Delimiter;
        }
        else
        {
          state = CloseBracket;
        }
        break;
      case CloseBracket:
        if (CompareStrings(")"))
        {
          FinishToken(CloseBracket, tokenQuantity);
          state = Delimiter;
        }
        else
        {
          state = Comparison;
        }
        break;
      case Comparison:
        if
(CompareStrings("!=")||CompareStrings("==")||CompareStrings("<=")||CompareStrings(">=")||Compa
reStrings(">")||CompareStrings("<"))
        {
          FinishToken(Comparison, tokenQuantity);
          state = Delimiter;
        }
        else
        {
          state = Assignment;
        }
        break;
      case Assignment:
        if (CompareStrings("=")||CompareStrings("+=")||CompareStrings("-
=")||CompareStrings("*=")||CompareStrings("/=")||CompareStrings("\%="))
        {
          FinishToken(Assignment, tokenQuantity);
          state = Delimiter;
        }
        else
        {
          state = MathSign;
        }
        break;
```

```
      case MathSign:
        if (CompareStrings("+") || CompareStrings("-") || CompareStrings("*") || CompareStrings("/") ||
CompareStrings("\%"))
        {
          FinishToken(MathSign, tokenQuantity);
          state = Delimiter;
        }
        else
        {
          state = If;
        }
        break;
      case If:
        if (CompareStringCheck("if"))
        {
          FinishToken(If, tokenQuantity);
          state = Delimiter;
        }
        else
        {
          state = Else;
        }
        break;
      case Else:
        if (CompareStringCheck("else"))
        {
          FinishToken(Else, tokenQuantity);
          state = Delimiter;
        }
        else
        {
          state = For;
        }
        break;
      case For:
        if (CompareStringCheck("for"))
        {
          FinishToken(For, tokenQuantity);
          state = Delimiter;
        }
        else
        {
          state = While;
        }
        break;
      case While:
        if (CompareStringCheck("while"))
        {
          FinishToken(While, tokenQuantity);
          state = Delimiter;
        }
        else
        {
```

```
        state = Input;
    }
    break;
case Input:
    if (CompareStringCheck("input"))
    {
        FinishToken(Input, tokenQuantity);
        state = Delimiter;
    }
    else
    {
        state = Output;
    }
    break;
case Output:
    if (CompareStringCheck("print"))
    {
        FinishToken(Output, tokenQuantity);
        state = Delimiter;
    }
    else
    {
        state = Class;
    }
    break;
case Class:
    if (CompareStringCheck("class"))
    {
        FinishToken(Class, tokenQuantity);
        state = Delimiter;
    }
    else
    {
        state = Def;
    }
    break;
case Def:
    if (CompareStringCheck("def"))
    {
        FinishToken(Def, tokenQuantity);
        state = Delimiter;
    }
    else
    {
        state = In;
    }
    break;
case In:
    if (CompareStringCheck("in"))
    {
        FinishToken(In, tokenQuantity);
        state = Delimiter;
    }
```

```
            else
            {
               state = Return;
            }
            break;
         case Return:
            if (CompareStringCheck("return"))
            {
               FinishToken(Return, tokenQuantity);
               state = Delimiter;
            }
            else
            {
               state = Variable;
            }
            break;
         case Variable:
            if (isalpha(fileContent[idx]) || CompareStrings("_") || (!identificatorFirst &&
isdigit(fileContent[idx])))
            {
               identificatorFirst = false;
               currentTokenLength++;
               idx++;
            }
            else if (!identificatorFirst)
            {
               identificatorFirst = true;
               FinishToken(Variable, tokenQuantity);
               state = Delimiter;
            }
            else
            {
               state = Number;
            }
            break;
         case Number:
            if (IsNumber(fileContent))
            {
               FinishToken(Number, tokenQuantity);
               state = Delimiter;
            }
            else
            {
               state = Error;
            }
            break;
         case Error:
            if(fileContent[idx]!=' ')
            {
               printf("\nLexical error on the line %d - unreserved character: %c  ",lines+1 , fileContent[idx]);
            }
            idx++;
            state = Delimiter;
```

```
      default:
         break;
      }
   }
   FinishToken(Delimiter, tokenQuantity);
   FinishToken(END, tokenQuantity);
   return tokenS;
}

bool CompareStrings(char* str)
{
   if (idx + strlen(str)-1 < strlen(fileContent))
   {
      for (int i = 0; i < strlen(str); i++)
      {
         if (fileContent[idx + i] != str[i])
         {
            return false;
         }
      }
      idx += strlen(str);
      currentTokenLength += strlen(str);
      return true;
   }
   return false;
}

bool CompareStringCheck(char* str)
{
   if (idx + strlen(str)-1 < strlen(fileContent))
   {
      for (int i = 0; i < strlen(str); i++)
      {
         if (fileContent[idx + i] != str[i])
         {
            return false;
         }
      }
      if(idx + strlen(str) < strlen(fileContent))
      {
         if(isalpha(fileContent[idx + strlen(str)]) || isdigit(fileContent[idx + strlen(str)]))
         {
            return false;
         }
      }
      idx += strlen(str);
      currentTokenLength += strlen(str);
      return true;
   }
   return false;
}

bool IsNumber(char* str)
```

```
{
   enum number stage = numbers;
   bool RootNumber = false;
   while (true)
   {
      switch (stage)
      {
      case numbers:
      {
         if (idx < strlen(str))
         {
            if (isdigit(str[idx]))
            {
               RootNumber = true;
               currentTokenLength++;
               idx++;
            }
            else if (str[idx] == '.' && RootNumber)
            {

               currentTokenLength++;
               idx++;
               stage = one_dot_numbers;
            }
            else if ((str[idx] != ' ' || str[idx] != '\n' || str[idx] != '\r') && RootNumber)
            {
               stage = finish_number;
            }
            else
            {
               stage = no_number;
            }
         }
         else
         {
            stage = finish_number;
         }
         break;
      }
      case one_dot_numbers:
      {
         if (idx < strlen(str) && str[idx] != ' ' && str[idx] != '\n' && str[idx] != '\r')
         {
            if (isdigit(str[idx]))
            {
               currentTokenLength++;
               idx++;
            }
            else
            {
               stage = no_number;
            }
         }
```

```
            else
            {
               stage = finish_number;
            }
            break;
         }
         case no_number:
         {
            return false;
         }
         case finish_number:
         {
            return true;
         }
         }
      }
   }
```

ПРИЛОЖЕНИЕ В

(обязательное)

ФОРМАЛЬНАЯ ГРАММАТИКА ДЛЯ ПОДМНОЖЕСТВА ЯЗЫКА PYTHON

«значение» → «переменная"."«переменная» | «переменная»“[“«арифметическое выражение»”]“ | «переменная» | «число» | «булевые значения»

«идентификатор» → «переменная"."«переменная» | «переменная» “[“ «арифметическое выражение»”]“ | «переменная»

«знак сравнения» → “>” | “<” | “==“ | “!=” | “>=”| “<=”

«арифметический знак» → “+” | “/” | “-“ | “*” | “%” | “**” | “//”

«логический знак» → “or” | “and“

«булевые значения» → “true“ | “fasle“

«возврат» → "return" («значение» "\n" | "\n")

«операция присваивания» → «идентификатор» "=" «арифметическое выражение» "\n" | «идентификатор» "=" «команда чтения» "\n" | «идентификатор» "=" «вызов функции» "\n" | | «идентификатор» "=" «определение массива» "\n"

«определение массива» → "[" [«аргументы»] "]"

«определение класса» → “class” «переменная» “(“ “)” “:” “\n” «блок»

«определение функций» → "def" «переменная» "(" «аргументы для определения функции» ")" ":" "\n" «блок»

«аргументы для определения функции» → «переменная» [“,” | «аргумент для определения функции»] | эпсилон

«аргумент для определения функции» → «переменная» [“,” | «аргумент для определения функции»]

«вызов функции» → «переменная» "(" «аргументы» ")" "\n"

«аргументы» → «идентификатор» [“,” | «аргумент»] | | эпсилон

«аргумент» → «идентификатор» [“,” | «аргумент»]

«условный оператор if» → “if” «логическая операция» ":" "\n" «последовательность» [“else” ":" "\n" «блок»]

«оператор цикла while» → “while” «логическая операция» ":" "\n" «блок»

«оператор цикла for» → “for” «значение» “in” «значение» ":" "\n"
«блок»

«логическая операция» → [“(“] «логическая операция» [«логический знак» «логическая операция»] [“)“]

«логическое выражение» → [“(“] «арифметическое выражение» [«знак сравнения» «логическое выражение»] [“)“]

«арифметическое выражение» →
[“(“]«значение» [«арифметический знак» «арифметическое выражение»] [“)“]

«команда ввода» → “input” “(“ “)” “\n”

«команда вывода» → “print” “(“ «арифметическое выражение»” “)” “\n”

«обычное утверждение» → «условный оператор if» | «оператор цикла while» | «оператор цикла for» | «операция присваивания» | «команда ввода» | «команда вывода» | «определение класса» | «определение функции» | «вызов функции» | “\n”

«утверждение класса» → «операция присваивания» | «определение функции» | “\n”

«утверждение функции» → «обычное утверждение» | «возврат»

«блок» → [«таб»] «обычное утверждение» «блок» | [«таб»] «утверждение класса» «блок» | [«таб»] «утверждение функции» «блок» | эпсилон

ПРИЛОЖЕНИЕ Г

(обязательное)

ФРАГМЕНТЫ РЕАЛИЗАЦИИ СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА ДЛЯ ПОДМНОЖЕСТВА ЯЗЫКА PYTHON(parser.h)

```
#ifndef PARSER_H
#define PARSER_H

#include <stdbool.h>
#include "token.h"
#include "tree.h"

enum StateBlock
{
	CommonBlock,
	ClassBlock,
	DefBlock,
};

bool Block();
bool Statement();
bool StatementDef();
bool StatementClass();

bool Condition();
bool WhileLoop();
bool ForLoop();

bool OutputCommand();
bool InputCommand(struct Node* inputNode);

bool AssignmentOperation();

bool ArrayDefinition(struct Node* arrayFunctionDefinition);

bool FunctionDefinition();
bool FunctionCall(struct Node* functionCallNode);
bool Return1();

bool LogicalOperationMain(struct Node* arifNode);
bool LogicalOperation(int* bracketCountDifference, struct Node* arifNode);
bool LogicalExpression(int* bracketCountDifference, struct Node* arifNode);
bool ArithmeticExpressionLO(int* bracketCountDifference, struct Node* arifNode);

bool ArithmeticExpressionMain(bool open, struct Node* arifNode);
bool ArithmeticExpression(int* bracketCount, struct Node* arifNode);

bool ArgumentsFunctionDefinition(struct Node* argumentsFunctionNode);
```

```
bool ArgumentFunctionDefinition(struct Node* argumentFunctionNode);
bool Arguments(struct Node* argumentsNode);
bool Argument(struct Node* argumentNode);

bool ClassDefinition();

bool Value(struct Node* valueNode);
bool Identificator(struct Node* identificatorNode);

bool Is(enum TokenType type);
void Parser(struct Token* token,int tokenQuantity);

#endif
```

ПРИЛОЖЕНИЕ Д

(обязательное)

ФРАГМЕНТЫ РЕАЛИЗАЦИИ СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА ДЛЯ ПОДМНОЖЕСТВА ЯЗЫКА PYTHON(parser.c)

```
#include "parser.h"
int tempCurrentToken;

int currentToken;

int tokenLength;

int tabCount = 0;

int needTabCount = 0;

struct Token * tokens;

enum StateBlock sequence = CommonBlock;

enum StateBlock prevBlock = CommonBlock;

struct Node root;

struct Node * current;

void InitFirstTree()
{
	enum NonTerminalType RootTerm = BlockTerm;

	root = *NewNodeTerminal(RootTerm);
	current = &root;
}

bool ErrorRecovery()
{
	if(tokens[currentToken].type != END)
	{
		printf("\nSyntax error on the line: %d\n",tokens[currentToken].line+1);
		while (currentToken < tokenLength)
		{
			if (tokens[currentToken].type == END)
			{
				return false;
			}
			else if (tokens[currentToken].type != Delimiter)
			{
				currentToken++;
```

```
                        }
                        else
                        {
                                tempCurrentToken=currentToken;
                                return true;
                        }
                }
        }
        return false;
}

void Parser(struct Token * token, int tokenQuantity)
{
        InitFirstTree();

        tokens = token;
        tokenLength = tokenQuantity;
        currentToken = 0;
        printf("\nSTART_PARSER\n");
        if (Block())
        {

                printf("\nSuccessfully (A good program without errors well done)\n");
        }
        else
        {
                printf("\nVery bad needs to be redone\n");
        }
        Print2D(&root);
        printf("\nEND_PARSER\n");
}

bool Block()
{
        currentToken = tempCurrentToken;
        if(currentToken < tokenLength)
        {
                //printf("\nnew Block token - %s #%d\n", NameType(tokens[currentToken].type),
currentToken);
        }
        if(currentToken >= tokenLength)
        {
                tabCount -= 1;
                return true;
        }
        else if (tokens[currentToken].type == END)
        {
                tabCount -= 1;
                return true;
        }
        else
        {
                tempCurrentToken = currentToken;
```

```
		if (needTabCount != 0)
		{
			int currentTab = 0;
			while (Is(Tab))
			{
				currentTab += 1;
			}
			tempCurrentToken = currentToken;
			if (currentTab != needTabCount)
			{
				if (currentTab > needTabCount)
				{
					printf("\nError - incorrect number of tabs(a lot of tabs)\n");
				}
				else
				{
					printf("\nError - incorrect number of tabs(small number of
tabs)\n");
				}
				ErrorRecovery();
				Block();
				return false;
			}
			needTabCount = 0;
		}
		for (int i = 0; i < tabCount; i++)
		{
			if (!Is(Tab))
			{
				current = current->parent;
				tempCurrentToken = currentToken;
				tabCount -= 1;
				return true;
			}
		}

		currentToken = tempCurrentToken;
		if (sequence == CommonBlock)
		{
			if (Statement() && Block())
			{
				sequence = CommonBlock;
				needTabCount = 0;
				return true;
			}
		}
		else if (sequence == ClassBlock)
		{
			if (StatementClass() && Block())
			{
				prevBlock = CommonBlock;
				sequence = CommonBlock;
				needTabCount = 0;
```

```
                        return true;
                    }
                }
                else if (sequence == DefBlock)
                {
                    if (StatementDef() && Block())
                    {
                        sequence = prevBlock;
                        needTabCount = 0;
                        return true;
                    }
                }
                ErrorRecovery();
                Block();
                return false;

        }
        if(ErrorRecovery())
        {
            Block();
        }
        return false;
}

bool StatementDef()
{
        if (Statement())
        {
            return true;
        }
        else
        {
            if (Return1())
            {
                currentToken = tempCurrentToken;
                return true;
            }
            return false;
        }
}

bool Return1()
{
        if (Is(Return))
        {
            struct Node* returnNode = AddChild(tokens[tempCurrentToken - 1], current);
            if (Is(Delimiter) || (Value(returnNode) && Is(Delimiter)))
            {
                return true;
            }
        }
        return false;
}
```

```
bool StatementClass()
{
	tempCurrentToken = currentToken;
	if (Is(Delimiter))
	{
		return true;
	}
	if (AssignmentOperation())
	{
		currentToken = tempCurrentToken;
		return true;
	}
	else
	{
		if (FunctionDefinition())
		{
			currentToken = tempCurrentToken;
			return true;
		}
	}
}

bool Statement()
{
	tempCurrentToken = currentToken;
	if(Is(Delimiter))
	{
		return true;
	}
	if (Condition())
	{
		currentToken = tempCurrentToken;
		return true;
	}
	else
	{
		tempCurrentToken = currentToken;
		if (AssignmentOperation())
		{
			currentToken = tempCurrentToken;
			return true;
		}
		else
		{
			tempCurrentToken = currentToken;
			if (InputCommand(current))
			{
				currentToken = tempCurrentToken;
				return true;
			}
			else
			{
```

```
                                tempCurrentToken = currentToken;
                                if (OutputCommand())
                                {
                                        currentToken = tempCurrentToken;
                                        return true;
                                }
                                else
                                {
                                        if(WhileLoop())
                                        {
                                                currentToken = tempCurrentToken;
                                                return true;
                                        }
                                        else
                                        {
                                                if(ForLoop())
                                                {
                                                        currentToken = tempCurrentToken;
                                                        return true;
                                                }
                                                else
                                                {
                                                        if(FunctionDefinition())
                                                        {
                                                                currentToken = tempCurrentToken;
                                                                return true;
                                                        }
                                                        else
                                                        {
                                                                if(FunctionCall(current))
                                                                {
                                                                        currentToken =
tempCurrentToken;

                                                                        return true;
                                                                }
                                                                else
                                                                {
                                                                        if (ClassDefinition())
                                                                        {
                                                                                currentToken =
tempCurrentToken;

                                                                                return true;
                                                                        }
                                                                }
                                                        }
                                                }
                                        }
                                }
                        }
                }
        }
        return false;
}
```

```
bool ClassDefinition()
{
	if (Is(Class))
	{
		struct Node* funcNode = NewNode(tokens[tempCurrentToken - 1]);
		if (Is(Variable))
		{
			struct Node* funcNodetemp = AddChild(tokens[tempCurrentToken - 1],
funcNode);
			if (Is(OpenBracket))
			{
				if (Is(CloseBracket))
				{
					if (Is(DoubleDot))
					{
						if (Is(Delimiter))
						{
							tabCount += 1;
							needTabCount = tabCount;
							sequence = ClassBlock;

							enum NonTerminalType body = BlockTerm;

							struct Node * returnCurrent = current;
							AddChildNode(funcNode, current);

							current = AddNextTerminal(body,
funcNodetemp);
							if (Block())
							{
								current = returnCurrent;

								currentToken = tempCurrentToken;
								return true;
							}
						}
					}
				}
			}
		}
	}
}

bool FunctionDefinition()
{
	tempCurrentToken = currentToken;
	if(Is(Def))
	{
		struct Node* funcNode = NewNode(tokens[tempCurrentToken - 1]);
		if(Is(Variable))
		{
```

```
                struct Node* funcNodetemp = AddChild(tokens[tempCurrentToken - 1],
funcNode);
                if (Is(OpenBracket))
                {
                        enum NonTerminalType param = Params;
                        struct Node * params = AddNextTerminal(param, funcNodetemp);
                        if (ArgumentsFunctionDefinition(params))
                        {
                                if (Is(CloseBracket))
                                {
                                        if (Is(DoubleDot))
                                        {
                                                if (Is(Delimiter))
                                                {
                                                        tabCount += 1;
                                                        needTabCount = tabCount;
                                                        prevBlock = sequence;
                                                        sequence = DefBlock;

                                                        enum NonTerminalType body =
BlockTerm;

                                                        struct Node* returnCurrent = current;
                                                        AddChildNode(funcNode, current);

                                                        current = AddNextTerminal(body,
funcNodetemp);

                                                        if (Block())
                                                        {
                                                                current = returnCurrent;
                                                                currentToken =
tempCurrentToken;

                                                                return true;
                                                        }
                                                }
                                        }
                                }
                        }
                }
        }
        return false;
}

bool ArrayDefinition(struct Node* ArrayDefinition)
{
        if (Is(OpenBraces))
        {
                struct Node* arrayDefinition = NewNode(tokens[tempCurrentToken - 1]);

                enum NonTerminalType ParamsTerm = Params;
```

```
                struct Node* paramsNode = AddChildTerminal(ParamsTerm, arrayDefinition);
                if (Arguments(paramsNode))
                {
                        if (Is(CloseBraces))
                        {
                                AddChildNode(arrayDefinition, ArrayDefinition);
                                AddChild(tokens[tempCurrentToken - 1], arrayDefinition);
                                return true;
                        }
                }
        }
}

bool FunctionCall(struct Node * functionCallNode)
{
        if(Is(Variable))
        {
                enum NonTerminalType funcTerm = Function;
                struct Node* functionCallTemp = NewNodeTerminal(funcTerm);

                struct Node* functionCallChild = AddChild(tokens[tempCurrentToken - 1],
functionCallTemp);
                enum NonTerminalType params = Params;

                functionCallChild = AddNextTerminal(params, functionCallChild);

                if(Is(OpenBracket))
                {
                        if (Arguments(functionCallChild))
                        {
                                if (Is(CloseBracket))
                                {
                                        if (Is(Delimiter))
                                        {
                                                AddChildNode(functionCallTemp, functionCallNode);
                                                currentToken = tempCurrentToken;
                                                return true;
                                        }
                                }
                        }
                }
        }
        return false;
}

bool ArgumentsFunctionDefinition(struct Node * argumentsFunctionNode)
{
        if (Is(Variable))
        {
                struct Node* argFuncTemp = NewNode(tokens[tempCurrentToken-1]);
                if (Is(Comma))
                {
```

```
                        struct Node* commaNode = AddNext(tokens[tempCurrentToken - 1],
argFuncTemp);
                        if (ArgumentFunctionDefinition(commaNode))
                        {
                                AddChildNode(argFuncTemp, argumentsFunctionNode);
                                return true;
                        }
                        else
                        {
                                return false;
                        }
                }
                else
                {
                        AddChildNode(argFuncTemp, argumentsFunctionNode);
                        return true;
                }
        }
        else
        {
                return true;
        }
}

bool ArgumentFunctionDefinition(struct Node* argumentFunctionNode)
{
        if (Is(Variable))
        {
                struct Node* argFuncTemp = NewNode(tokens[tempCurrentToken - 1]);
                if (Is(Comma))
                {
                        struct Node* commaNode = AddNext(tokens[tempCurrentToken - 1],
argFuncTemp);
                        if (ArgumentFunctionDefinition(commaNode))
                        {
                                AddChildNode(argFuncTemp, argumentFunctionNode);
                                return true;
                        }
                        else
                        {
                                return false;
                        }
                }
                else
                {
                        AddChildNode(argFuncTemp, argumentFunctionNode);
                        return true;
                }
        }
        else
        {
                return false;
        }
```

```
}

bool Arguments(struct Node* argumentsNode)
{
	if (Value(argumentsNode))
	{
		if (Is(Comma))
		{
			struct Node * commaNode = AddChild(tokens[tempCurrentToken - 1],
argumentsNode);
			if (Argument(commaNode))
			{
				return true;
			}
			else
			{
				return false;
			}
		}
		else
		{
			return true;
		}
	}
	else
	{
		return true;
	}
}

bool Argument(struct Node* argumentNode)
{
	if (Value(argumentNode))
	{
		if (Is(Comma))
		{
			struct Node * commaNode = AddChild(tokens[tempCurrentToken - 1],
argumentNode);
			if (Argument(commaNode))
			{
				return true;
			}
			else
			{
				return false;
			}
		}
		else
		{
			return true;
		}
	}
	else
```

```
        {
                return false;
        }
}

bool AssignmentOperation()
{
        tempCurrentToken = currentToken;
        struct Token assignment;
        assignment.value = "=";
        assignment.type = Assignment;
        struct Node* assignmentNode = NewNode(assignment);
        if (Identificator(assignmentNode))
        {
                if (Is(Assignment))
                {
                        AddChildNode(assignmentNode, current);
                        currentToken = tempCurrentToken;
                        if (ArithmeticExpressionMain(false, assignmentNode) && Is(Delimiter))
                        {
                                currentToken = tempCurrentToken;
                                return true;
                        }
                        tempCurrentToken = currentToken;
                        if (InputCommand(assignmentNode))
                        {
                                currentToken = tempCurrentToken;
                                return true;
                        }
                        tempCurrentToken = currentToken;
                        if (FunctionCall(assignmentNode))
                        {
                                currentToken = tempCurrentToken;
                                return true;
                        }
                        tempCurrentToken = currentToken;
                        if (ArrayDefinition(assignmentNode) && Is(Delimiter))
                        {
                                currentToken = tempCurrentToken;
                                return true;
                        }
                }
        }
        return false;
}

bool OutputCommand()
{
        tempCurrentToken = currentToken;
        if (Is(Output))
        {
                struct Node* outputNode = NewNode(tokens[currentToken]);
                if (Is(OpenBracket))
```

```
		{
			if (ArithmeticExpressionMain(true, outputNode))
			{
				if (Is(Delimiter))
				{
					AddChildNode(outputNode, current);
					currentToken = tempCurrentToken;
				}
				return true;
			}
		}
	}
	return false;
}

bool InputCommand(struct Node * inputNode)
{
	if (Is(Input))
	{
		if (Is(OpenBracket))
		{
			if (Is(CloseBracket))
			{
				if (Is(Delimiter))
				{
					AddChild(tokens[tempCurrentToken - 4], inputNode);
					return true;
				}
			}
		}
	}
	return false;
}

bool LogicalOperationMain(struct Node* arifNode)
{
	int bracketCountDifference = 0;
	if (Is(OpenBracket))
	{
		bracketCountDifference++;
	}
	if (LogicalOperation(&bracketCountDifference, arifNode))
	{
		if (Is(CloseBracket))
		{
			bracketCountDifference--;
		}
		if (bracketCountDifference == 0)
		{
			return true;
		}
		else
		{
```

```
                printf("\ncountbracket:! %d !\n", bracketCountDifference);
                printf("\nError - the difference in the number of opening brackets to the closing
ones: %d\n", tokens[tempCurrentToken].pos);
            }
        }
        return false;
}

bool LogicalOperation(int * bracketCountDifference, struct Node* arifNode)
{
        if (Is(OpenBracket))
        {
            (*bracketCountDifference)++;
        }
        struct Node* valueNode = NewNode(tokens[currentToken]);
        if (LogicalExpression(bracketCountDifference, valueNode))
        {
            if (Is(Logical))
            {
                struct Node* childLog = AddChild(tokens[tempCurrentToken - 1], arifNode);
                AddChildNode(valueNode->childs, childLog);
                if (LogicalOperation(bracketCountDifference, childLog))
                {
                    currentToken = tempCurrentToken;
                }
                else
                {
                    return false;
                }
            }
            else
            {
                AddChildNode(valueNode->childs, arifNode);
            }
            if (Is(CloseBracket))
            {
                (* bracketCountDifference)--;
            }
            return true;
        }
        return false;
}

bool LogicalExpression(int * bracketCountDifference, struct Node* arifNode)
{
        if (Is(OpenBracket))
        {
            (*bracketCountDifference)++;
        }
        if (ArithmeticExpressionLO(bracketCountDifference, arifNode))
        {
            if(Is(Comparison))
            {
```

```
                        struct Node* childComp = AddChild(tokens[tempCurrentToken - 1], arifNode);
                        if(LogicalExpression(bracketCountDifference, childComp))
                        {
                                currentToken = tempCurrentToken;
                        }
                        else
                        {
                                return false;
                        }
                }
                if (Is(CloseBracket))
                {
                        (*bracketCountDifference)--;
                }
                return true;
        }
        return false;
}

bool ArithmeticExpressionLO(int * bracketCountDifference, struct Node* arifNode)
{
        int temp = tempCurrentToken;
        if (ArithmeticExpression(bracketCountDifference, arifNode))
        {
                return true;
        }
        tempCurrentToken = temp;
        return false;
}

bool ArithmeticExpressionMain(bool open, struct Node* arifNode)
{
        int temp = tempCurrentToken;
        int bracketCountDifference = open;
        if (ArithmeticExpression(&bracketCountDifference, arifNode))
        {
                if (bracketCountDifference == 0)
                {
                        return true;
                }
                else
                {
                        printf("\ncountbracket:%d\n", bracketCountDifference);
                        printf("\nError - the difference in the number of opening brackets to the closing
ones: %d\n", tokens[tempCurrentToken].pos);
                }
        }
        tempCurrentToken = temp;
        return false;
}

struct Node* bracketNode;
```

```
bool ArithmeticExpression(int * bracketCount, struct Node* arifNode)
{
	if (Is(OpenBracket))
	{
		(*bracketCount)++;
	}
	struct Node* valueNode=NewNode(tokens[currentToken]);
	if (Value(valueNode))
	{
		if (Is(MathSign))
		{
			struct Node* mathNode = AddChild(tokens[tempCurrentToken-1],arifNode);
			AddChildNode(valueNode->childs, mathNode);
			if(ArithmeticExpression(bracketCount, mathNode))
			{
				currentToken = tempCurrentToken;
			}
		}
		else
		{
			AddChildNode(valueNode->childs, arifNode);
		}
		if (Is(CloseBracket))
		{
			(*bracketCount)--;
			if ((*bracketCount) < 0)
			{
				printf("Error - missing opening parenthesis before: %d",
tokens[tempCurrentToken].pos);
				return false;
			}
		}
		return true;
	}
	return false;
}

bool Value(struct Node* valueNode)
{
	int temp = tempCurrentToken;
	if (Is(Variable) && Is(Dot) && Is(Variable))
	{
		struct Node* temp = AddChild(tokens[tempCurrentToken - 3], valueNode);
		struct Node* forNext = AddChild(tokens[tempCurrentToken - 2], temp);
		AddNext(tokens[tempCurrentToken - 1], forNext);
		return true;
	}
	tempCurrentToken = temp;
	if (Is(Variable) && Is(OpenBraces) )
	{
		if (ArithmeticExpressionMain(false, valueNode) && Is(CloseBraces))
		{
			return true;
```

```
                }
        }
        tempCurrentToken = temp;
        if (Is(Variable) || Is(Number) || Is(Bool))
        {
                AddChild(tokens[tempCurrentToken - 1], valueNode);
                return true;
        }
        return false;
}

bool Identificator(struct Node* identificatorNode)
{
        int temp = tempCurrentToken;
        if (Is(Variable) && Is(Dot) && Is(Variable))
        {
                struct Node* temp = AddChild(tokens[tempCurrentToken - 3], identificatorNode);
                struct Node* forNext = AddChild(tokens[tempCurrentToken - 2], temp);
                AddNext(tokens[tempCurrentToken - 1], forNext);
                return true;
        }
        tempCurrentToken = temp;
        if (Is(Variable) && Is(OpenBraces) && ArithmeticExpressionMain(false, identificatorNode) &&
Is(CloseBraces))
        {
                return true;
        }
        tempCurrentToken = temp;
        if (Is(Variable))
        {
                AddChild(tokens[tempCurrentToken - 1], identificatorNode);
                return true;
        }
        return false;
}

bool Condition()
{
        int tempCurrentToken1 = currentToken;
        tempCurrentToken = currentToken;

        if (Is(If))
        {
                struct Node* ifNode = NewNode(tokens[currentToken]);
                enum NonTerminalType lo = LogicalOperationTerm;
                struct Node* loNode = AddChildTerminal(lo, ifNode);
                if (LogicalOperationMain(loNode) && Is(DoubleDot) && Is(Delimiter))
                {
                        tabCount += 1;
                        needTabCount = tabCount;

                        enum NonTerminalType body = BlockTerm;
```

```
                struct Node* returnCurrent = current;
                AddChildNode(ifNode, current);

                current = AddNextTerminal(body, loNode);

                if (Block())
                {
                        current = returnCurrent;

                        currentToken = tempCurrentToken;
                        if (Is(Else))
                        {
                                struct Node* elseNode = NewNode(tokens[currentToken]);
                                if(Is(DoubleDot) && Is(Delimiter))
                                {
                                        tabCount += 1;
                                        needTabCount = tabCount;

                                        enum NonTerminalType body = BlockTerm;

                                        returnCurrent = current;
                                        AddChildNode(elseNode, current);

                                        current = AddChildTerminal(body, elseNode);

                                        if (Block())
                                        {
                                                current = returnCurrent;
                                                currentToken = tempCurrentToken;
                                        }
                                }
                        }

                        return true;
                }
        }
    }
    else
    {
        currentToken = tempCurrentToken1;
        return false;
    }
}

bool WhileLoop()
{
    if(Is(While))
    {
        struct Node* whileNode = NewNode(tokens[currentToken]);
        enum NonTerminalType lo = LogicalOperationTerm;
        struct Node* loNode = AddChildTerminal(lo, whileNode);
        if(LogicalOperationMain(loNode))
        {
```

```
                        if(Is(DoubleDot))
                        {
                                if(Is(Delimiter))
                                {
                                        tabCount += 1;
                                        needTabCount = tabCount;

                                        enum NonTerminalType body = BlockTerm;

                                        struct Node* returnCurrent = current;
                                        AddChildNode(whileNode, current);

                                        current = AddNextTerminal(body, loNode);

                                        if(Block())
                                        {
                                                current = returnCurrent;
                                                currentToken = tempCurrentToken;
                                                return true;
                                        }
                                }
                        }
                }
        }
}

bool ForLoop()
{
        tempCurrentToken = currentToken;
        if(Is(For))
        {
                struct Node* forNode = NewNode(tokens[currentToken]);
                if(Identificator(forNode))
                {
                        if(Is(In))
                        {
                                if(Identificator(forNode))
                                {
                                        if(Is(DoubleDot))
                                        {
                                                if(Is(Delimiter))
                                                {
                                                        tabCount += 1;
                                                        needTabCount = tabCount;

                                                        enum NonTerminalType body = BlockTerm;

                                                        struct Node* returnCurrent = current;
                                                        AddChildNode(forNode, current);

                                                        current = AddChildTerminal(body, forNode);

                                                        if(Block())
```

```
                                                {
                                                        current = returnCurrent;
                                                        currentToken = tempCurrentToken;
                                                        return true;
                                                }
                                        }
                                }
                        }
                }
        }
}

bool Is(enum TokenType type)
{
        if (tokens[tempCurrentToken].type == type)
        {
                tempCurrentToken++;
                return true;
        }
        return false;}
```

ПРИЛОЖЕНИЕ Е

(обязательное)

ФРАГМЕНТЫ РЕАЛИЗАЦИИ ПОСТРОЕНИЯ АБСТРАКТНОГО СИНТАКСИЧЕСКОГО ДЕРЕВА(tree.h)

```
#include<stdio.h>
#include<malloc.h>
#include "token.h"

#define COUNT 5

struct Node
{
	enum NonTerminalType type;
	struct Token data;
	struct Node* parent;
	struct Node* next;
	struct Node* childs;
};

struct Node* NewNode(struct Token data);
struct Node* NewNodeTerminal(enum NonTerminalType data);


struct Node* AddChild(struct Token data, struct Node* node);
struct Node* AddChildNode(struct Node* child, struct Node* node);
struct Node* AddChildTerminal(enum NonTerminalType data, struct Node* node);

void AddParent(struct Token data, struct Node* node);

struct Node* AddNext(struct Token data, struct Node* node);
struct Node* AddNextTerminal(enum NonTerminalType data, struct Node* node);

void PrintTree(struct Node *root, int space);
void Print2D(struct Node *root);
```

ПРИЛОЖЕНИЕ Ё

(обязательное)

ФРАГМЕНТЫ РЕАЛИЗАЦИИ ПОСТРОЕНИЯ АБСТРАКТНОГО СИНТАКСИЧЕСКОГО ДЕРЕВА(tree.c)

```
#include "tree.h"

struct Node* NewNode(struct Token data)
{
	struct Node* node = malloc(sizeof(struct Node));
	node->data = data;
	node->type = Terminal;
	node->next = node->childs = NULL;
	return node;
}

struct Node* NewNodeTerminal(enum NonTerminalType data)
{
	struct Node* node = malloc(sizeof(struct Node));
	node->type = data;
	node->next = node->childs = NULL;
	return node;
}

struct Node * AddChild(struct Token data, struct Node* node)
{
	if (node == NULL)
	{
		node = malloc(sizeof(struct Node));
		node->data = data;
		node->type = Terminal;
		node->next = node->childs = NULL;
		return node;
	}
	else
	{
		if (node->childs == NULL)
		{
			node->childs = malloc(sizeof(struct Node));
			node->childs->data = data;
			node->childs->type = Terminal;
			node->childs->next = node->childs->childs = NULL;
			node->childs->parent = node;
			return node->childs;
		}
		struct Node** temp = &node->childs;
		while ((*temp)->next != NULL)
		{
			temp = (&((*temp)->next));
		}
```

```
		if ((*temp)->next == NULL)
		{
			(*temp)->next = malloc(sizeof(struct Node));
			(*temp)->next->parent = node;
			(*temp)->next->type = Terminal;
			(*temp)->next->data = data;
			(*temp)->next->next = (*temp)->next->childs = NULL;
			return (*temp)->next;
		}
	}
}

struct Node* AddChildTerminal(enum NonTerminalType data, struct Node* node)
{
	if (node == NULL)
	{
		node = malloc(sizeof(struct Node));
		node->type = data;
		node->next = node->childs = NULL;
		return node;
	}
	else
	{
		if (node->childs == NULL)
		{
			node->childs = malloc(sizeof(struct Node));
			node->childs->type = data;
			node->childs->next = node->childs->childs = NULL;
			node->childs->parent = node;
			return node->childs;
		}
		struct Node** temp = &node->childs;

		while ((*temp)->next != NULL)
		{
			temp = (&((*temp)->next));
		}
		if ((*temp)->next == NULL)
		{
			(*temp)->next = malloc(sizeof(struct Node));
			(*temp)->next->parent = node;
			(*temp)->next->type = data;
			(*temp)->next->next = (*temp)->next->childs = NULL;
			return (*temp)->next;
		}
	}
}

struct Node* AddChildNode(struct Node* child, struct Node* node)
{
	if (node == NULL)
	{
		node = malloc(sizeof(struct Node));
```

```
			node->childs = child;
			child->parent = node;
			return child;
		}
		else
		{
			if (node->childs == NULL)
			{
				struct Node** temp = &child;
				while ((*temp)->next != NULL)
				{
					(*temp)->next->parent = node;
					temp = (&((*temp)->next));
				}
				node->childs = malloc(sizeof(struct Node));
				node->childs = child;
				node->childs->parent = node;

				return child;
			}
			struct Node** temp2 = &node->childs;
			while ((*temp2)->next != NULL)
			{
				temp2 = (&((*temp2)->next));
			}
			if ((*temp2)->next == NULL)
			{
				struct Node** temp1 = &child;
				while ((*temp1)->next != NULL)
				{
					(*temp1)->next->parent = node;
					temp1 = (&((*temp1)->next));
				}

				(*temp2)->next = malloc(sizeof(struct Node));
				(*temp2)->next = child;
				(*temp2)->next->parent = node;

				return child;
			}
		}
}

void AddParent(struct Token data, struct Node* node)
{
		if (node == NULL)
		{
			node = malloc(sizeof(struct Node));
			node->data = data;
			node->next = node->childs = NULL;
		}
		else
		{
```

```
                if (node->parent == NULL)
                {
                        node->parent = malloc(sizeof(struct Node));
                        node->parent->data = data;
                        node->parent->next = NULL;
                        node->parent->childs = node;
                        return;
                }
                struct Node** temp = &node->childs;

                while ((*temp)->next != NULL)
                {
                        temp = (&((*temp)->next));
                }
                if ((*temp)->next == NULL)
                {
                        (*temp)->next->parent = node;
                        (*temp)->next = malloc(sizeof(struct Node));
                        (*temp)->next->data = data;
                        (*temp)->next->next = (*temp)->next->childs = NULL;
                }
        }
}

struct Node * AddNext(struct Token data, struct Node* node)
{
        if (node == NULL)
        {
                node = malloc(sizeof(struct Node));
                node->data = data;
                node->type = Terminal;
                node->next = node->childs = NULL;
                return node;
        }
        else
        {
                if (node->next == NULL)
                {
                        node->next = malloc(sizeof(struct Node));
                        node->next->data = data;
                        node->next->type = Terminal;
                        node->next->parent = node->parent;
                        node->next->next = node->next->childs = NULL;
                        return node->next;
                }
                struct Node** temp = &node->next;
                while ((*temp)->next != NULL)
                {
                        temp = (&((*temp)->next));
                }
                if ((*temp)->next == NULL)
                {
                        (*temp)->next = malloc(sizeof(struct Node));
```

```
                        (*temp)->next->parent = node->parent;
                        (*temp)->next->data = data;
                        (*temp)->next->type = Terminal;
                        (*temp)->next->next = (*temp)->next->childs = NULL;
                        return (*temp)->next;
                }
        }
}

struct Node* AddNextTerminal(enum NonTerminalType data, struct Node* node)
{
        if (node == NULL)
        {
                node = malloc(sizeof(struct Node));
                node->type = data;
                node->next = node->childs = NULL;
                return node;
        }
        else
        {
                if (node->next == NULL)
                {
                        node->next = malloc(sizeof(struct Node));
                        node->next->type = data;
                        node->next->parent = node->parent;
                        node->next->next = node->next->childs = NULL;
                        return node->next;
                }
                struct Node** temp = &node->next;
                while ((*temp)->next != NULL)
                {
                        temp = (&((*temp)->next));
                }
                if ((*temp)->next == NULL)
                {
                        (*temp)->next = malloc(sizeof(struct Node));
                        (*temp)->next->parent = node->parent;
                        (*temp)->next->type = data;
                        (*temp)->next->next = (*temp)->next->childs = NULL;
                        return (*temp)->next;
                }
        }
}

void PrintTree(struct Node *root, int space)
{
        if (root == NULL)
                return;
        for (int i = 0; i < space; i++)
                if(i==space-4)
                {
                        printf("|");
                }
```

```
                else if (i>space-4)
                {
                        printf("-");
                }
                else
                {
                        printf(" ");
                }
        if (root->type == Terminal)
        {
                if (root->childs == NULL || root->data.type == Variable || root->data.type == MathSign
|| root->data.type == Comparison || root->data.type == Logical)
                {
                        printf("%s - %s\n", NameType(root->data.type), ((*root).data.value));
                }
                else
                {
                        printf("%s\n", NameType(root->data.type));
                }
        }
        else
        {
                printf("%s\n", NameNonTerminalType(root->type));
        }
        PrintTree(root->childs, space + COUNT);
        PrintTree(root->next, space);
}

void Print2D(struct Node *root)
{
        printf("\nTree: \n\n");
   PrintTree(root, 0);
        printf("\nTree end; \n");
}
```