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

«24» мая 2022 г.

КУРСОВОЙ ПРОЕКТ

по дисциплине «Теория языков программирования и методы трансляции»

на тему: «Компилятор для подмножества языка Pascal»

Студент _____ Бессонов М.П.

Шифр 191009

Институт приборостроения, автоматизации и информационных технологий

Направление подготовки 09.03.04 «Программная инженерия»

Группа 92ПГ

Руководитель _____ Гордиенко А.П.

Оценка: «отл.» Дата 26.05.22

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
«___»_____20__г.

ЗАДАНИЕ
на курсовой проект

по дисциплине «Теория языков программирования и методы трансляции»

Студент Бессонов М.П. Шифр 191009

Институт приборостроения, автоматизации и информационных технологий

Направление подготовки 09.03.04 «Программная инженерия»

Группа 92ПГ

1 Тема курсового проекта

«Компилятор для подмножества языка Pascal»

2 Срок сдачи студентом законченной работы «26» мая 2022

3 Исходные данные

Для подмножества языка Pascal сделать:

Объявление переменных целого, вещественного, булевого типа; объявление массивов и записей; определение функций, процедур и главной программы; команды присваивания, условий, цикла, ввода-вывода, блока команд и вызова процедуры; использование таких операций как: обращение к элементу массива, обращение к полю записи, арифметические операции, операции сравнения, логические операции.

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

4 Содержание курсового проекта

Лексический анализ

Синтаксический анализ

Абстрактное синтаксическое дерево

5 Отчетный материал курсового проекта

Пояснительная записка курсового проекта, программа на съемном носителе

Руководитель _____ Гордиенко А.П.

Задание принял к исполнению: «11» марта 2022

Подпись студента _____

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Современный мир предъявляет высокие требования к выпускникам вуза, который по завершении обучения должен обладать рядом компетенций. Одной из важнейших является компетенция техническая. Современный специалист в области IT должен уметь работать в разных программах и с разными ресурсами. Кроме того, он должен уметь работать с различными языками программирования, а также уметь разрабатывать свои собственные компиляторы для них.

Основной задачей компилятора является построение эквивалентного кода, то есть перевод исходных инструкций, заданных на одном языке, в исполняемый код на целевом языке.

Изучение принципов разработки компиляторов является актуальной, так как знания из этой области можно применить в других сферах разработки программного обеспечения. Например, методы, используемые для распознавания лексем, можно также использовать для чтения и обработки любого структурированного документа (XML, список адресов и другие).

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

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

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

Лексический анализ является неотъемлемой составляющей компиляции. На этом этапе последовательность символов исходного файла преобразуется в последовательность лексем [5]. Это необходимо для получения на выходе идентифицирующих последовательностей (токенов).

Нам необходимо разработать лексический анализатор для подмножества языка Pascal. Для распознавания блока команд, который может включать в себя условия, циклы, операции чтения и записи и т.д., а также для распознавания блока данных, который может содержать в себе объявление переменных, массивов и записей различных типов, нам необходим список ключевых слов. Этот список будет содержать все необходимые для дальнейшего анализа команды, с которыми мы сможем сравнивать исходную последовательность.

Также необходимо помнить, что команды в языке Pascal отделяются точками с запятой, а в конце блока ставится точка. Кроме того, в выражениях могут присутствовать знаки сравнения, арифметические операции, знаки присваивания и т.д. Для их распознавания создадим отдельный список.

Как мы сказали ранее, в программе могут быть объявлены переменные, которые мы будем классифицировать отдельным образом, представляя их идентификаторами.

Для возможности создания арифметических выражений введем отдельную лексическую единицу "число".

Реализуем наш лексический анализатор с помощью конечных автоматов [2]. Для этого нам потребуются регулярные выражения, которые помогут в классификации наших лексических единиц.

Прежде, чем мы перейдем к определению регулярных выражений, необходимо сказать, что программа, которую будет разбирать наш анализатор является независимой от отступов. Команды могут отделяться друг от друга любым количеством одиночных пробелов, табулированных

пробелов, переносов строки или не отделяться вовсе. Также после отступов могут следовать открывающиеся скобки, которые в этом случае должны быть идентифицированы. Построим недетерминированный конечный автомат (затем его детерминируем и минимизируем) для обработки этой ситуации. Недетерминированный конечный автомат представлен на рисунке 1.

Регулярное выражение: (' '|'\n'|'\t'|'(')* другой символ.

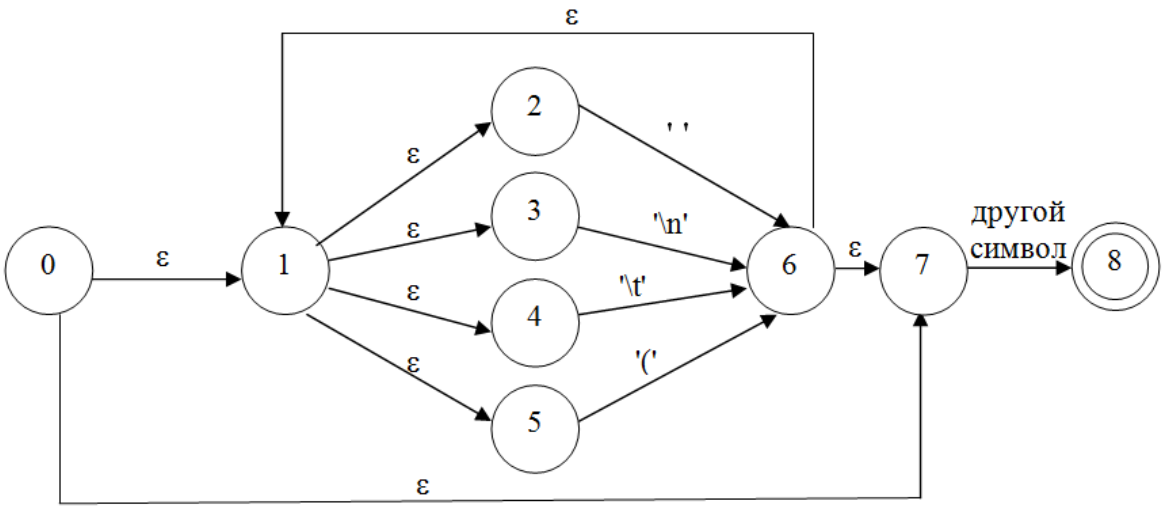

Рисунок 1 – Недетерминированный конечный автомат для обработки отступов

Преобразуем недетерминированный конечный автомат к детерминированному. Построим таблицу со всеми возможными переходами (Таблица 1). Для этого определим ε-closure(Sn).

Таблица 1 – Таблица переходов для обработки отступов

	' '	'\n'	'\t'	'('	Другой символ
{0,1,2,3,4,5,7}	{1,2,3,4,5,6,7}	{1,2,3,4,5,6,7}	{1,2,3,4,5,6,7}	{1,2,3,4,5,6,7}	{8}
{1,2,3,4,5,6,7}	{1,2,3,4,5,6,7}	{1,2,3,4,5,6,7}	{1,2,3,4,5,6,7}	{1,2,3,4,5,6,7}	{8}
{8}	-	-	-	-	-

По представленной таблице построим детерминированный конечный автомат (Рисунок 2).

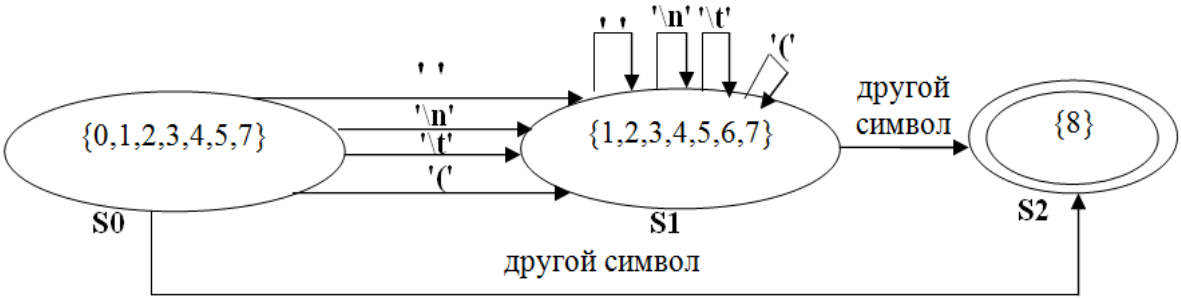

Рисунок 2 – Детерминированный конечный автомат для обработки отступов

По схеме видно, что состояние S0 является избыточным и его нужно удалить, так как все его переходы уже задействованы в состоянии S1. Минимизированный конечный автомат представлен на рисунке 3.

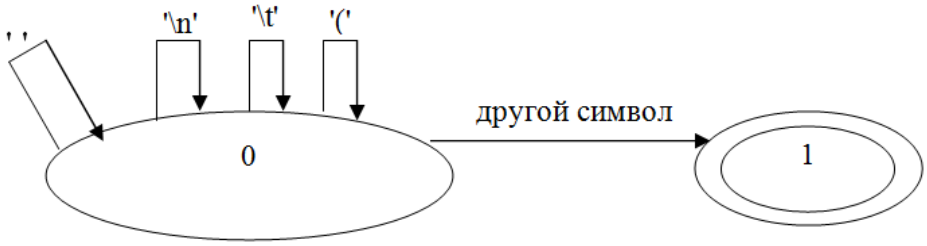

Рисунок 3 – Минимизированный конечный автомат для обработки отступов

Определим регулярные выражения и построим недетерминированные конечные автоматы (затем их детерминируем и минимизируем) к следующим лексемам:

1) Идентификатор - последовательность латинских букв и цифр, начинающаяся с латинской буквы. Недетерминированный конечный автомат представлен на рисунке 4.

Регулярное выражение: [A-Za-z][A-Za-z0-9]* другой символ

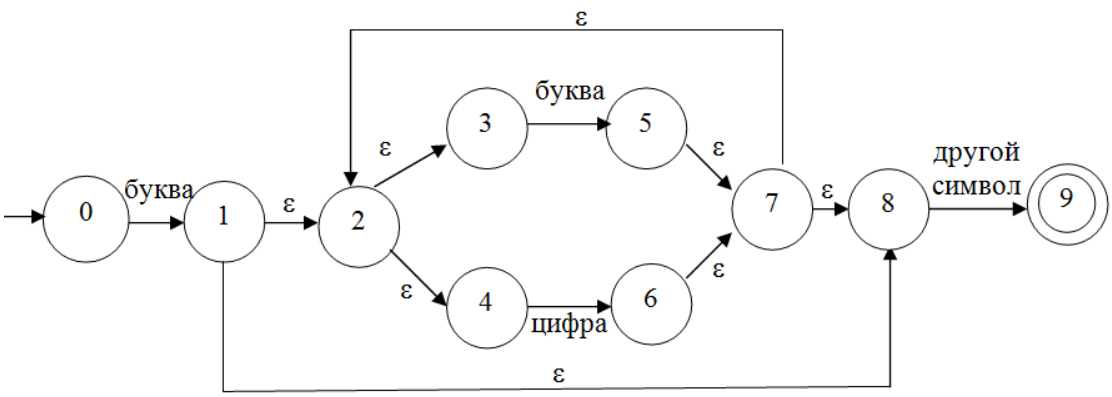

Рисунок 4 – Недетерминированный конечный автомат для идентификатора

Преобразуем недетерминированный конечный автомат к детерминированному. Построим таблицу со всеми возможными переходами (Таблица 2). Для этого определим ε-closure(Sn).

Таблица 2 – Таблица переходов для идентификатора

	буква	цифра	Другой символ
{0}	{1,2,3,4,8}	-	-
{1,2,3,4,8}	{5,7,2,3,4,8}	{6,7,2,3,4,8}	{9}
{5,7,2,3,4,8}	{5,7,2,3,4,8}	{6,7,2,3,4,8}	{9}
{6,7,2,3,4,8}	{5,7,2,3,4,8}	{6,7,2,3,4,8}	{9}
{9}	-	-	-

По представленной таблице построим детерминированный конечный автомат (Рисунок 5).

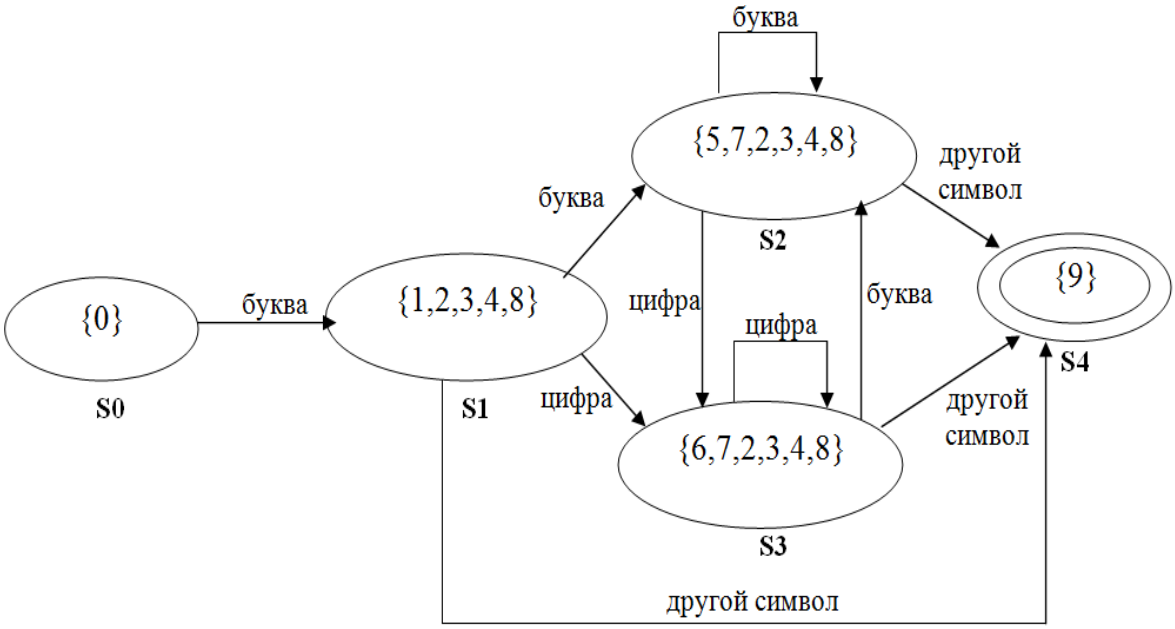

Рисунок 5 – Детерминированный конечный автомат для идентификатора

По схеме видно, что конечным состоянием является только состояние S4, а неконечные состояния S0, S1, S2, S3. Из состояний S1, S2, S3 можно попасть в конечное по входному сигналу "другой символ". Таким образом группу неконечных состояний можно разбить на две. Минимизированный конечный автомат представлен на рисунке 6.

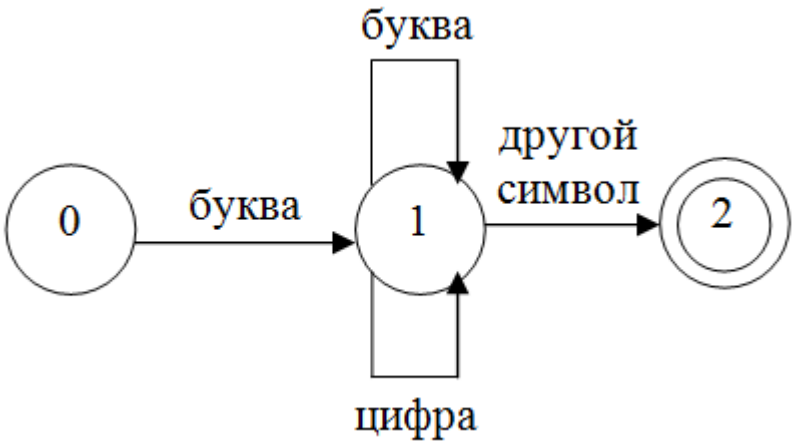

Рисунок 6 – Минимизированный конечный автомат для идентификатора

2) Число - последовательность цифр, целая часть от дробной отделяется точкой, причем дробная часть может отсутствовать. Недетерминированный конечный автомат представлен на рисунке 7.

Регулярное выражение: $[0\text{-}9]^{+}(.[0\text{-}9]^{+})?$ другой символ

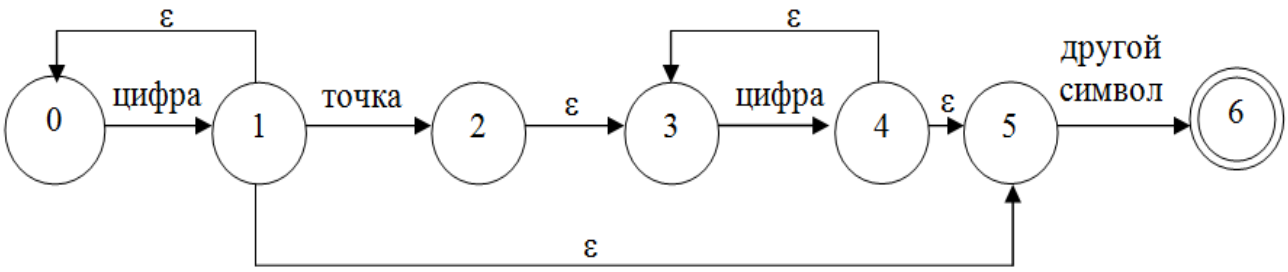

Рисунок 7 – Недетерминированный конечный автомат для числа

Построим таблицу со всеми возможными переходами (Таблица 3).

Таблица 3 – Таблица переходов для числа

	цифра	точка	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}	-	-	-

По представленной таблице построим детерминированный конечный автомат (Рисунок 8).

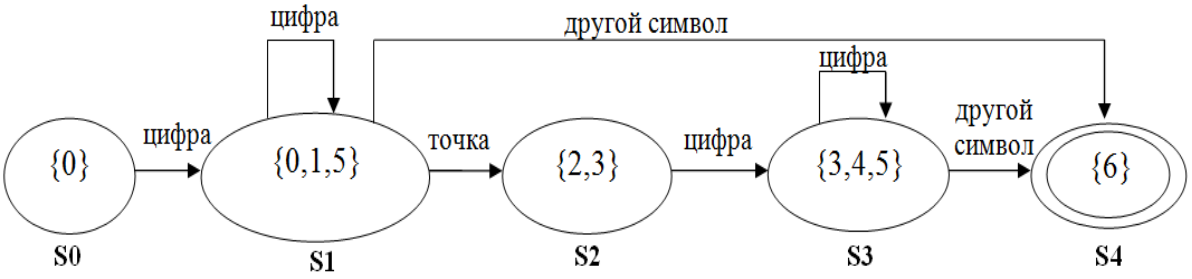

Рисунок 8 – Детерминированный конечный автомат для числа

По схеме видно, что конечным состоянием является только состояние S4, а неконечные состояния S0, S1, S2, S3. Из состояний S1 и S3 можно попасть в конечное по входному сигналу "другой символ". Таким образом группу неконечных состояний можно разбить на две. Минимизированный конечный автомат представлен на рисунке 9.

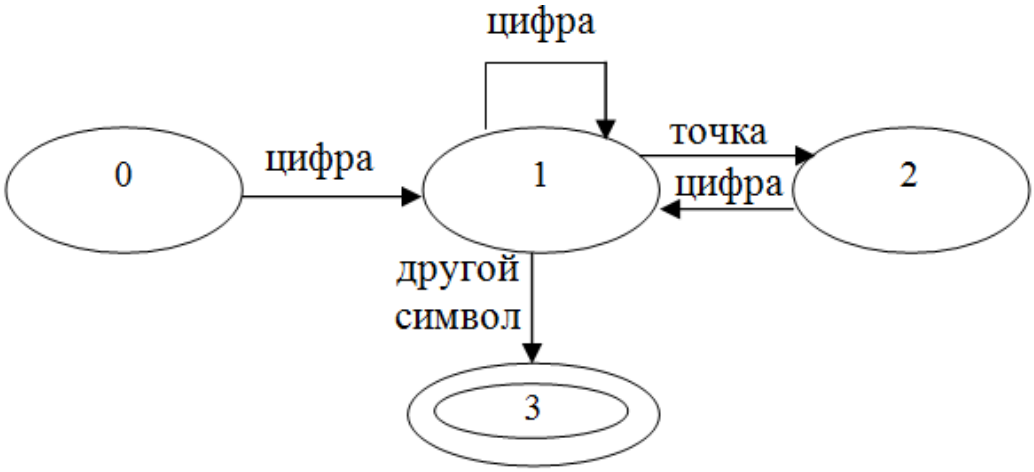

Рисунок 9 – Минимизированный конечный автомат для числа

Для определения ключевых слов и специальных символов конечные автоматы не строятся, так как в этом случае достаточно сравнивать входную последовательность с составленными словарями, содержащими грамматику языка.

Реализуем наш лексический анализатор на универсальном языке программирования СИ [7]. Программу для анализа будем получать из текстового файла. Полученные токены будем сохранять в текстовый файл и в массив, а также сохранять их код и строку, на которой встречалась анализируемая лексема.

Реализовывать наше получение токенов будем с помощью функции getToken, в которой последовательно проверяется каждый возможный класс лексем.

В самом начале мы игнорируем все отступы или классифицируем открывающуюся круглую скобку, если такие символы есть.

Далее мы проверяем, достигли ли мы конца файла анализа. Если достигли, то в этом случае лексический разбор окончен.

Если же нет, то далее мы с помощью вызова функций по условиям проверяем последовательность на принадлежность к классу "число", "слово", в котором заложена проверка на токен "идентификатор", а также происходит сравнения слова со словарем ключевых слов. В конце происходит проверка на принадлежность к классу специальных символов. В этой функции последовательность сравнивают со вторым созданным словарем.

При невозможности проанализировать следующую лексему лексический анализатор возвращает ошибку и заканчивает работу.

Фрагменты Реализации нашего лексического анализатора приведены в приложении А.

СИНТАКСИЧЕСКИЙ АНАЛИЗ

Вторым этапом построения нашего компилятора является процесс сопоставления полученной классифицированной последовательности лексем языка с его формальной грамматикой [5]. Для этого напишем программу, результатом работы которой будет полное дерево синтаксического анализа, которое в дальнейшем мы преобразуем в абстрактное синтаксическое дерево.

Синтаксический анализатор будет строиться при помощи метода рекурсивного спуска. Он использует алгоритм нисходящего обхода, то есть раскрывает правила формальной грамматики, начиная со стартового символа [4]. Реализация этого метода основана на взаимном вызове процедур, где каждая процедура соответствует одному из правил грамматики.

На этапе лексического анализа мы распознали все числа и идентификаторы. Их определениям в правилах синтаксического анализатора будут соответствовать значения num и id соответственно.

Зададим правила, по которым будет работать наш синтаксический анализатор:

1) Так как мы реализуем грамматику для языка Pascal, начнем задание правил грамматики с команды **"PROGRAM -> program id ; BLOCK ."** . Это правило говорит о том, что программа начинается с объявления заголовка программы, за которым следует блок исполнения;

2) Определим составляющие элементы блока программы. Грамматика языка Pascal должна содержать объявление типов записей и массивов, а также функций и процедур. Кроме того в программе можно объявить встроенные типы данных. После этого начинается основной блок программы, состоящий из вызовов тех или иных команд. Правило для этого случая будет выглядеть так **"BLOCK -> TYPES VARS FUNCTS PRCDS INSTS"**.

3) Правило TYPES в нашей грамматике представлено объявлением записей или массивов. Инструкция "e" говорит о том, что объявление типов может отсутствовать. Правило, соответствующее грамматике языка Pascal

выглядит так: **" TYPES -> type id = (array [num : {id | num}] of TYPE; | record id : TYPE {; id : TYPE} end;) | e "**.

4) Правило VARS должно включать в себя объявление переменных целочисленного, вещественного, логического или своего типа. При этом таких объявлений может быть больше одного или вообще не быть. Правило, соответствующее этом описанию выглядит так: **"VARS -> var DCLR ; {DCLR ;} | e"**.

5) Правило DCLR должно описывать объявление переменной (одной или нескольких) и типа. Его описание выглядит так "**DCLR -> id { , id} : TYPE**".

6) Правило TYPE содержит все необходимые для реализации подмножества языка типы данных: **"TYPE -> integer | boolean | real | id"**.

7) Для объявления функций и процедур зададим по два правила для большего понимания. Их объявление соответствует синтаксису языка Pascal:

Объявление функций: **"FUNCTS -> FUNCT {FUNCT} | e"**
" FUNCT -> function id (DCLR {; DCLR}) : TYPE ; BLOCK ;";

Объявление процедур: "**PRCDS -> PRCD {PRCD} | e"**

" PRCD -> procedure id {(DCLR {; DCLR}) | e} ; BLOCK ;".

8) Основной блок (INSTS) программы должен начинаться с лексемы begin и заканчиваться лексемой end. Между ними может располагать любое количество возможных в языке действий, в том числе и то же самое объявление блока INSTS. Возникает рекурсивная ситуация. Для ее избавления опишем возможные действие в отдельном правиле (INST). Полученные правила выглядит так: **"INSTS -> begin INST ; {INST ;} end"** **"INST -> INSTS | ASSIGN | EXECUTE | IF | WHILE | FOR | WRITE | READ | e"**.

9) Правила, описанные в INST соответствуют формальной грамматики языка Pascal [6]. Эти правила описывают присваивание идентификатору какого-либо выражения, или вызов функции или процедуры. Также они

описывают условия и циклы, а также блоки чтения и записи. Формальное описание этих правил представлено в приложении Б.

10) Во всех описанных выше действиях мы используем так называемые выражения (EXPR), необходимые для присваивания, проверки условий или циклов и т.п. Это выражение может быть логического типа или представлять собой любую последовательность арифметических операций с другими такими же выражениями (EXPR). Так как в этом случае снова получается левая рекурсия, введем для ее избавления несколько правил:

- Правило TERM представляет собой произведение или частное факторов (FACTOR);

- Правило FACTOR представляет собой число, идентификатор, выражение в скобках, элемент массива, поле записи или вызов процедуры или функции.

Таким образом, правило для выражений сводится к анализу суммы или разности термов.

Полный список правил, заложенных в работу синтаксического анализатора приведен в приложении Б.

Также нам необходимо указать несколько семантических правил, соблюдение которых упростит реализацию и понимание программы:

1) Все переменные, типы, функции и процедуры должны быть объявлены до главного блока;

2) Объявление должно происходить в следующем порядке: типы, переменные, функции, процедуры;

3) Все типы должны являться стандартными (integer, real, boolean) или быть определены пользователем.

Опишем наш синтаксический анализатор, реализующий описанную выше грамматику.

Для работы нашего анализатора необходим массив токенов, полученный на этапе лексического анализа.

Процесс выполнения анализа начинается с функции Program(), которая проверяет наличие заголовка и вызывает функцию Block(). Последняя же последовательно вызывает функции Types(), Vars(), Functs(), Prcds(), Insts().

Каждая из этих функций проверяет токены на соответствие правилам и, в зависимости от результата, вызывает другие функции в соответствии с правилами или возвращает ошибку.

Ниже представлена реализация функции For(), которая проверяет последовательность токенов на соответствие грамматике цикла FOR в языке Pascal.

```
void For(){
   fprintf(xml_file,"<INST>\n");
   WriteCurrTokenToFile();
   if(!verifyToken(ID))
      SyntaxError(ID, "ID is missing");
   if(!verifyToken(ASSIGN))
      SyntaxError(ASSIGN, "Assign sign is missing");
   if(!Expr())
      SyntaxError(curr_token.code, "Initial value is missing");
   if(!verifyToken(TO)&&!verifyToken(DOWNTO))
      SyntaxError(TO, "TO or DOWNTO keyword is missing");
   if(!Expr())
      SyntaxError(curr_token.code, "Final value is missing");
   if(!verifyToken(DO))
      SyntaxError(DO, "DO keyword is missing");
   Inst();
}
```

В процессе работы анализатор строит полное синтаксическое дерево разбора, которое представлено в файле формата XML. Это дерево содержит последовательности токенов, объединенных в правила, соответствующих грамматике. Пример такого дерева представлен на рисунке 10.

Фрагменты реализации нашего синтаксического анализатора приведены в приложении В.

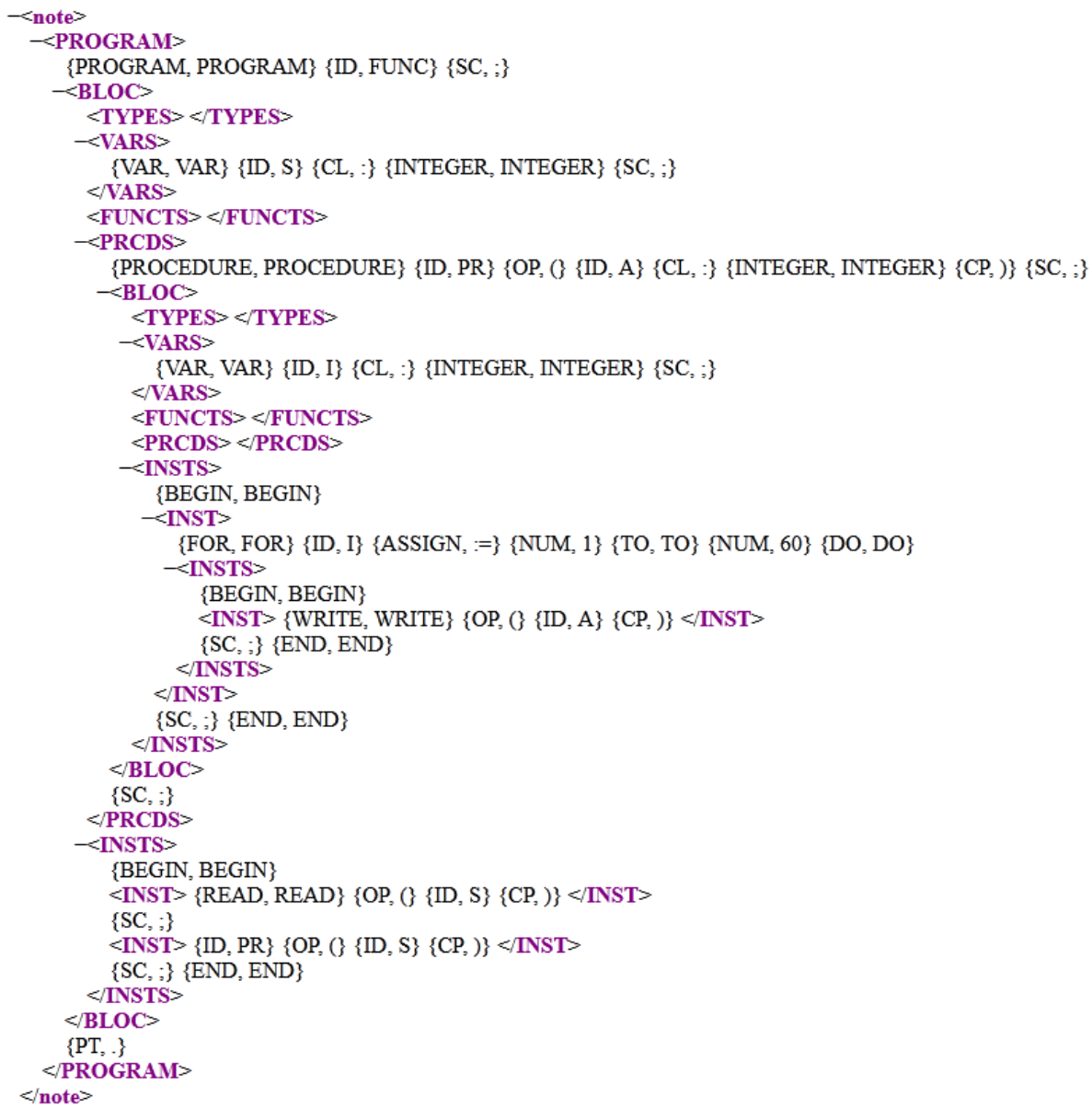

Рисунок 10 – Синтаксическое дерево разбора

АБСТРАКТНОЕ СИНТАКСИЧЕСКОЕ ДЕРЕВО

Реализуем наше абстрактное синтаксическое дерево при помощи полученного при лексическом анализе массива токенов. Абстрактное синтаксическое дерево — дерево, отражающее структурные связи между существенными элементами исходного выражения, но не отражающее вспомогательные языковые средства [1].

То есть нам необходимо построить такое дерево, в котором внутренние вершины сопоставлены (помечены) с операторами языка программирования, а листья — с соответствующими операндами.

Для построения дерева нам необходимо задать синтаксически-управляемое определение. Оно использует контекстно-свободную грамматику для определения синтаксической структуры входа [3]. С каждым символом грамматики оно связывает набор атрибутов, а с каждой продукцией - набор семантических правил для вычисления значений атрибутов, связанных с используемыми в продукциях символами.

Определим синтезируемые атрибуты с помощью грамматических правил, заданных на этапе синтаксического анализа.

Так как вся наша программа пишется в едином блоке, то в корень этого дерева нам необходимо поместить тип "BLOCK" и значение, равное названию программы.

В блоке могут содержаться объявления типов, переменных, функций и процедур. Каждое из этих объявления имеет собственное представление в дереве.

Объявление переменных является бинарной операцией в дереве, где узлом является ND_VAR_TYPE, а левым и правым узлами сама переменная и ее тип.

Объявление записи состоит из узла ND_VAR_RECORD, от которого отходят названии и объявления переменных.

Объявление массива состоит из узла ND_VAR_ARRAY, от которого отходят узел инициализации, содержащей диапазон и тип, а также название массива.

Процедура содержит узел ND_VAR_PROCEDURE, который включает название процедуры и ее параметры (название и тип каждого). Объявление функции происходит аналогично, но дополнительно указывается тип функции.

Кроме того, в блоке могут содержаться инструкции (команды) для выполнения. Эти инструкции могут быть циклами, условиями, командами чтения или записи, а также присваиванием или вызовом процедуры.

Листьями нашего дерева могут являться либо числовые значения, либо значения идентификатора. Это последняя проверка в алгоритме построения нашего дерева, которая осуществляется в функции Node *primary().

Если проверка вернула положительный результат, то запустятся проверки на следующие токены, чтобы определить, является ли текущая анализируемая последовательность элементом массива, структуры или вызовом процедуры. Если все описанные выше проверки закончатся неудачно, то функция просто вернет значение идентификатора в узле на место листа предыдущего сохраненного узла. Если текущий токен не является идентификатором, то числовое значение в узле будет записано на месте листа предыдущего сохраненного узла.

Для хранения всех узлов и ветвей нашего дерева создадим структуру Node, в которой будем хранить тип текущего узла, все возможные значения которого будут храниться в перечисляемом типе NodeKind, а также указатели на следующий за текущим узел, узлы левого и правого поддерева, и три специальных указателя на узлы cond, then, els, которые хранят в себя информацию об узле условия. Для хранения текущего значения в нашей структуре объявим массив типа char.

Имена узлов в нашем дереве будем получать, используя коды, хранящиеся в нашем перечисляемом типе, с помощью специального массива NodeKind_list, который содержит объявления массивов, процедур, типов, команд чтения, записей, циклов, логических и арифметических операций и так далее. Полный список возможных условий приведен в приложении Г.

Каждую из команд в нашем дереве можно представить как отдельный узел с разным количеством потомков. Например, для объявления условия (IF) необходим потомок, отвечающий за само условие. Этот потомок может быть выражением содержащим других потомков, причем некоторые могут быть унарными (например, команда not для отрицания или число в выражении). Также нужен потомок, отвечающий за действие, который может содержать в себе длинную последовательность инструкций. Наконец, в зависимости от кода исходной программы, узел IF может содержать потомок else, который также может содержать последовательность инструкций.

В целом, узлы дерева будут содержать левый и правый потомок, так как большинство инструкций бинарные. Ниже приведен пример функции, строящий узел для знаков плюс или минус. Эта функция в зависимости от текущего токена строит связи между текущим узлом и узлом, полученным после выполнения умножения между двумя другими узлами.

```
static Node *add(void) {
   Node *node = mul();

   for (;;){
      if (consume(PLUS))
         node = new_binary(ND_ADD, node, mul());
      else if (consume(MINUS))
         node = new_binary(ND_SUB, node, mul());
      else
         return node;
   }
}
```

На рисунке 11 представлен полный вывод абстрактного синтаксического дерева, которое содержит объявления переменных

различных типов, условия, объединенные логической операцией, математическое выражение, а также цикл и операции ввода/вывода.

Реализация абстрактного синтаксического дерева приведена в приложении Г.

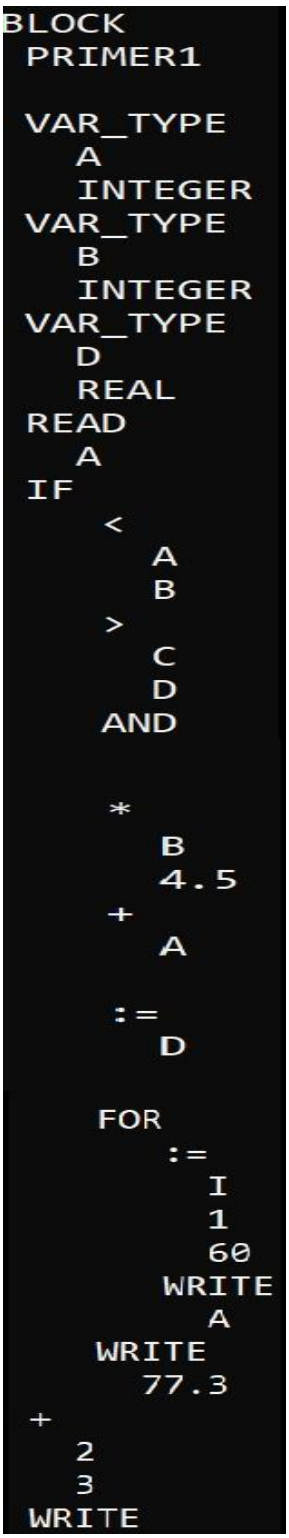

Рисунок 11 – Абстрактное синтаксическое дерево

ЗАКЛЮЧЕНИЕ

В настоящей работе был описан и реализован до этапа построения абстрактного синтаксического дерева включительно компилятор для подмножества языка Pascal. Мы выяснили, что анализ включает в себя несколько этапов предобратки получаемого кода.

Мы разработали лексический анализатор на основе конечных автоматов, разбивающий полученный код на последовательность токенов, синтаксический анализатор методом рекурсивного спуска, который в соответствии с грамматикой языка Pascal проверяет последовательность токенов на корректность и строит полное дерево разбора. Также было построено абстрактное синтаксическое дерево, которое оставляет в процессе разбора только значащие единицы языка.

Таким образом, можно сказать, что все поставленные перед нами задачи были выполнены и цель курсового проекта была достигнута.

СПИСОК ЛИТЕРАТУРЫ

1. Абстрактное синтаксическое дерево (AST) [Электронный ресурс]. - Режим доступа: https://eugenezolotarev.ru/programming/creating-language/g-02-ast (дата обращения: 12.04.2022).

2. Конечные автоматы [Электронный ресурс]. - Режим доступа: https://ps-group.github.io/compilers/fsm (дата обращения: 12.04.2022).

3. Синтаксически управляемые определения [Электронный ресурс]. - Режим доступа: https://studfile.net/preview/9023758/page:12/ (дата обращения: 12.04.2022).

4. Синтаксические анализаторы. Нисходящие анализаторы. [Электронный ресурс]. - Режим доступа: https://ps- https://intuit.ru/studies/courses/26/26/lecture/805?page=2 (дата обращения: 12.04.2022).

5. Структура компиляторов. Общая схема работы компиляторов. [Электронный ресурс]. - Режим доступа: https://studfile.net/preview/2910570/ (дата обращения: 12.04.2022).

6. ЯЗЫК ПРОГРАММИРОВАНИЯ PASCAL [Электронный ресурс]. - Режим доступа: https://phys.bspu.by/static/um/inf/prg/lecpdf/pascal1s.pdf (дата обращения: 12.04.2022).

7. Язык Си [Электронный ресурс]. - Режим доступа: https://prog-cpp.ru/c/ (дата обращения: 12.04.2022).

ПРИЛОЖЕНИЕ А

(обязательное)

ФРАГМЕНТЫ РЕАЛИЗАЦИИ ЛЕКСИЧЕСКОГО АНАЛИЗАТОРА ДЛЯ ПОДМНОЖЕСТВА ЯЗЫКА PASCAL

```
typedef struct {
    token_code code;
    char name[lenName];
    char value[lenValue];
    int line;
} Token;

void lexicalAnalysis(){
    curr_line = 1;
    curr_column = 1;
    nbr_tokens = 1;
    MALLOC(tokens, 1, Token);

    next_char();
    do{
        if(error_count!=0) break;
        getToken();
    }while (curr_token.code != EndOfFile);
}

void WriteCurrTknToFile(){
    if(error_count==0)
        fprintf(out_file,"CODE: %d, %s_TOKEN, %s , line: %d\n",
            curr_token.code, curr_token.name, curr_token.value,
curr_token.line);
}

char next_char(){
    curr_column++;
    prev_char = curr_char;
    curr_char = fgetc(prog_file);
    if( curr_char == '\n'){
        curr_column = 1;
        curr_line++;
    }
    return curr_char;
}
```

```
void LexError(char *message){
   error_count+=1;
   printf("Lexical Error: Line %d, Column %d:\n%s\n",
         curr_line, curr_column, message);
}

void getToken(){
   memset(curr_token.name, '\0', lenName);
   memset(curr_token.value, '\0', lenValue);

   ignoreWhiteSpaces();

   if(curr_char == EOF){
      curr_token.code = EndOfFile;
      strcpy(curr_token.name, "EOF");
      strcpy(curr_token.value, "EndOfFile");
      curr_token.line = curr_line;
      WriteCurrTknToFile();

      tokens[nbr_tokens-1] = curr_token;
      REALLOC(tokens, ++nbr_tokens, Token);
   }

   else if( isNumber() ){
      WriteCurrTknToFile();
   }
   else if( isWord() ){
      WriteCurrTknToFile();
   }
   else if( isSpecial() ){
      WriteCurrTknToFile();
   }
   else{
      LexError("Invalid Token");
      next_char();
   }
}
bool isWord(){
   if( !isalpha(curr_char) || error_count!=0)
      return false;

   char word[MAXCHAR];
   memset(word,'\0', sizeof(word));
```

```
        int i = 0, j = 0, cmp = 0;
        word[0] = toupper(curr_char);

        while( isalnum( word[++i] = curr_char = toupper(next_char()) ) );

        if(i > MAXCHAR){
           LexError("Word exceeded MAXCHAR limit");
           return true;
        }

        word[i] = '\0';

        while( (cmp = strcmp(word , keywords_list[j++])) != 0 && j < NBRKEYWORDS );

        if(cmp == 0){ // is keyword
           curr_token.code = j-1;
           strcpy(curr_token.name, keywords_list[j-1]);
           strcpy(curr_token.value, word);
        }

        else{
           curr_token.code = ID;
           strcpy(curr_token.name, "ID");
           strcpy(curr_token.value, word);
        }
        if (curr_char=='\n')
           curr_token.line = curr_line-1;
        else
           curr_token.line = curr_line;
        tokens[nbr_tokens-1] = curr_token;
        REALLOC(tokens, ++nbr_tokens, Token);
        return true;
      }
```

ПРИЛОЖЕНИЕ Б

(обязательное)

ФОРМАЛЬНАЯ ГРАММАТИКА ДЛЯ ПОДМНОЖЕСТВА ЯЗЫКА PASCAL

PROGRAM -> program id ; BLOCK .
BLOCK -> TYPES VARS FUNCTS PRCDS INSTS
TYPES -> type id = (array [num : {id | num}] of TYPE; | record id : TYPE {; id : TYPE} end;) | e
VARS -> var DCLR ; {DCLR ;} | e
DCLR -> id { , id} : TYPE
TYPE -> integer | boolean | real | id
FUNCTS -> FUNCT {FUNCT } | e
FUNCT -> function id (DCLR {; DCLR}) : TYPE ; BLOCK ;
PRCDS -> PRCD {PRCD } | e
PRCD -> procedure id {(DCLR {; DCLR}) | e} ; BLOCK ;
INSTS -> begin INST ; {INST ;} end
INST -> INSTS | ASSIGN | EXECUTE | IF | WHILE | FOR | WRITE | READ | e
ASSIGN -> id {[EXPR] { . id | e } | . id | e} := EXPR
EXECUTE -> id (EXPR {, EXPR})
IF -> if COND then INST {else INST | e}
WHILE -> while COND do INST
FOR -> for id := EXPR {to | downto} EXPR do INST
WRITE -> write (EXPR {, EXPR})
READ -> read (id {[EXPR] { . id | e } | . id | e} {, id {[EXPR] { . id | e } | . id | e}})
COND -> {(| e} {EXPR | e} {= | <> | < | > | <= | >= | not } EXPR {) | e} {{and | or | xor} COND | e}
EXPR -> TERM {{+ | -}TERM } | {true|false}
TERM -> FACTOR {{* | /} FACTOR }
FACTOR -> id | {-|+|e} num | (EXPR) | id [EXPR] { . id | e } | id . id | EXECUTE

ПРИЛОЖЕНИЕ В

(обязательное)

ФРАГМЕНТЫ РЕАЛИЗАЦИИ СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА ДЛЯ ПОДМНОЖЕСТВА ЯЗЫКА PASCAL

```
bool verifyToken(token_code code_to_test){
   if ( (curr_token = next_token()).code != code_to_test ){
      previous_token();
      return false;
   }
   WriteCurrTokenToFile();
   return true;
}

Token previous_token(){
   return curr_token = tokens[--currTokenNbr];
}

Token next_token(){
   prev_token = curr_token;
   return curr_token = tokens[currTokenNbr++];
}

void WriteCurrTokenToFile(){
   fprintf(xml_file,"{%s, %s}\n", curr_token.name, curr_token.value);
}

void SyntaxError(token_code code, char *message){
   printf("Syntax Error: Code %d\nLine %d: %s\n",
         code, curr_token.line, message);
}

void Program(){
   currTokenNbr = 0;
   MALLOC(IDs, 1, ID_struct);
   nbr_IDs = 1;
   fprintf(xml_file,"<PROGRAM>\n");
   bool r = verifyToken(PROGRAM);
   if(r){
      strcpy(IDs[nbr_IDs-1].name, curr_token.value);
      IDs[nbr_IDs-1].type = TPROG;
      IDs[nbr_IDs-1].line = curr_token.line;
```

```
            strcpy(IDs[nbr_IDs-1].block, "PROGRAM");
            REALLOC(IDs, ++nbr_IDs, ID_struct);
         }
         else
            SyntaxError(PROGRAM, "Program keyword is missing");
         if(!verifyToken(ID))
            SyntaxError(ID, "Program ID is missing");
         if(!verifyToken(SC))
            SyntaxError(SC, "SemiColon is missing");
         strcpy(curr_block, "PROGRAM");
         Block();
         if (!verifyToken(PT)){
            SyntaxError(PT, "The end's Point is missing");
         }
         fprintf(xml_file,"</PROGRAM>\n");
      }

      void Block(){
         fprintf(xml_file,"<BLOC>\n");
         Types();
         Vars();
         Functs();
         Prcds();
         Insts();
         fprintf(xml_file,"</BLOC>\n");
      }


      bool isNewType(){
         if(!verifyToken(ID))
            return false;
         for(int i = 0; i<=nbr_IDs-1;i++){
            if(          (IDs[i].type==TTYPE)          &&          (strcmp(IDs[i].name,
curr_token.value)==0) )
               return true;
         }
         return false;
      }


      bool Inst(){
         switch((curr_token = next_token()).code){
            case BEGIN  :
               previous_token();
               Insts();
```

```
            return true;
        case ID         :Assign_Execute();break;
        case IF         :If();break;
        case ELSE       :If();break;
        case WRITE      :Write();break;
        case READ       :Read();break;
        case WHILE      :While();break;
        case FOR        :For();break;
        default:
            previous_token();
            return false;
    }
    fprintf(xml_file,"</INST>\n");
    return true;
}

void While(){
    fprintf(xml_file,"<INST>\n");
    WriteCurrTokenToFile();
    Cond();
    if(!verifyToken(DO))
        SyntaxError(DO, "DO keyword is missing");
    Inst();
}


void Write(){
    bool r = true;
    fprintf(xml_file,"<INST>\n");
    WriteCurrTokenToFile();
    if(!verifyToken(OP))
        SyntaxError(OP, "Opening parenthesis is missing");
    do{
        r = Expr();
    }while(verifyToken(COMMA));
    if(!r)
        SyntaxError(COMMA, "Comma not needed");
    if(!verifyToken(CP))
        SyntaxError(CP, "Closing parenthesis is missing");
}
```

ПРИЛОЖЕНИЕ Г

(обязательное)

ФРАГМЕНТЫ РЕАЛИЗАЦИИ ПОСТРОЕНИЯ АБСТРАКТНОГО СИНТАКСИЧЕСКОГО ДЕРЕВА

```
const char* NodeKind_list[] =
{"PROCEDURE","ELEMENT","FUNCTION","DIAPASON","INIT","VAR_ARRAY","VAR_RECORD","ELEM_RECORD","ELEM_MAS","VAR_TYPE",":","WRITE","READ","NOT","XOR","OR","AND","FOR","WHILE","+", "-", "*", "/", "=", "<>", "<","<=",">",">=", ":=", "VAR", "EXPR_STMT", "IF", "BLOCK"};
struct Node{
    NodeKind kind;
    Node *next;
    Node *lhs;
    Node *rhs;
    Node *cond;
    Node *then;
    Node *els;
    char name[lenValue];
};
static Node *new_node(NodeKind kind){
  Node *node = calloc(1, sizeof(Node));
  node->kind = kind;
  return node;
}
static Node *new_binary(NodeKind kind, Node *lhs, Node *rhs){
  Node *node = new_node(kind);
  node->lhs = lhs;
  node->rhs = rhs;
  printTree(count_space);
  printf(" %s\n",NodeKind_list[node->kind]);
  printTree(count_space);
  printf("  %s\n",node->lhs->name);
  printTree(count_space);
  printf("  %s\n",node->rhs->name);
  return node;
}
static Node *new_unary(NodeKind kind, Node *expr) {
 Node *node = new_node(kind);
 node->lhs = expr;
 printTree(count_space);
```

```
  printf(" %s\n",NodeKind_list[node->kind]);
  printTree(count_space);
  printf("   %s\n",node->lhs->name);
  return node;
}
static Node *new_var_node(){
        Node *node = new_node(ND_VAR);
        strcpy(node->name, val);
        return node;
}
Node *progr(void){
    currTokenNbr=0;
    Node head = {};
        Node *cur = &head;
    expect(PROGRAM);
    if (consume(ID)){
        strcpy(val, tokens[--currTokenNbr].value);
        strcpy(cur->name, val);
        currTokenNbr++;  }
    expect(SC);
    cur->kind=ND_BLOCK;
    consume(BEGIN);
    printf("%s\n",NodeKind_list[cur->kind]);
    printf(" %s\n",cur->name);
    printf("\n");
        while (tokens[currTokenNbr].code!=EndOfFile) {
        cur->next = stmt();
        cur=cur->next;
        }
        return head.next;
}
static Node *primary(void){
    Node *node, *node1;
    if (consume(OP)){
        node = expr();
        expect(CP);
        return node;}
    Token *tok = consume_indent();
    if (tok){
        strcpy(val, tok->value);
        node = new_var_node();}
    if (consume(OB)){
        Node *temp = expr();
        node = new_binary(ND_ELEM_MAS, node, temp);
```

```
        expect(CB);
        if (!consume(PT)) return node;
        else{
            strcpy(val, tokens[currTokenNbr].value);
            Node *temp = new_var_node();
            node = new_binary(ND_ELEM_RECORD, node, temp);
            currTokenNbr++;
            return node;}
    }else if (consume(PT)){
        strcpy(val, tokens[currTokenNbr].value);
        Node *temp = new_var_node();
        node = new_binary(ND_ELEM_RECORD, node, temp);
        currTokenNbr++;
        return node;
    }else if (consume(OP)){
        while (consume(ID) || consume(NUM)){
            count_space+=3;
            currTokenNbr--;
                Node head = {};
                Node *cur = &head;
                while (tokens[currTokenNbr].code!=CP) {
                    strcpy(val, tokens[currTokenNbr].value);
                    Node *temp1 = expr();
                    cur->next = temp1;
                    if (consume(COMMA))cur = cur->next;
                }
                if (consume(CP)){
                    cur = &head;
                    while(cur->next!=NULL){
                        node1 = new_unary(ND_VAR_TYPE, cur->next);
                        cur = cur->next;
                    }
                }
        }
        consume(SC);
        count_space-=3;
        return node1;
    }
else if (tok){return node;}
    strcpy(val, expect_number()->value);
    return new_var_node();
}
```