## Лексический анализатор

Первый этап перевода называется лексическим анализом или сканированием. Лексический анализатор (сканер) считывает поток символов, составляющих исходную программу, и группирует эти символы в значимые последовательности, называемые лексемами.

Лексема - это структурная единица языка, состоящая из элементарных символов языка и не содержащая других структурных единиц языка. Лексемы языка программирования - это идентификаторы, константы, ключевые слова языка, знаки операций и т. п.

На вход лексического анализатора поступает текст исходной программы, а выходная информация передаётся для дальнейшей обработки синтаксическому анализатору. Для каждой лексемы сканер строит выходной токен вида:

Первый компонент токена, , представляет собой абстрактный символ, использующийся во время синтаксического анализа, а второй компонент, значение атрибута, указывает на запись в таблице идентификаторов, соответствующую данному токену.

Предположим, например, что исходная программа содержит инструкцию присваивания:

Символы в этом присваивании могут быть сгруппированы в следующие лексемы и отображены в следующие токены:

1. представляет собой лексему, которая может отображаться в токен , где – абстрактный символ, обозначающий идентификатор, а указывает запись в таблице идентификаторов для a, в которой хранится такая информация как имя и тип идентификатора.
2. Символ присваивания представляет собой лексему, которая отображается в токен . Поскольку этот токен не требует значения атрибута, второй компонент данного токена опущен. В качестве имени токена может быть использован любой абстрактный символ, например, такой, как «assign», но для удобства записи в качестве имени абстрактного символа можно использовать саму лексему.
3. представляет собой лексему, которая отображается в токен , где указывает на запись в таблице идентификаторов для .
4. является лексемой, отображаемой в токен .
5. – лексема, отображаемая в токен , где указывает на запись в таблице идентификаторов для .
6. – лексема, отображаемая в токен .
7. – лексема, отображаемая в токен , где указывает на запись в таблице идентификаторов для .

Пробелы, разделяющие лексемы, лексическим анализатором пропускаются.

Представление инструкции присваивания после лексического анализа в виде последовательности токенов примет следующий вид:

С теоретической точки зрения лексический анализатор не является обязательной, необходимой частью транслятора. Его функции могут выполняться на этапе синтаксического разбора. Однако есть несколько причин, благодаря которым лексический анализ включают в состав практически всех трансляторов:

* упрощается работа с текстом исходной программы на этапе синтаксического разбора и уменьшается объём обрабатываемой информации, поскольку лексический анализатор структурирует поступающий на вход исходный код программы и удаляет всю незначащую информацию;
* для выделения в тексте и разбора лексем можно использовать простую, эффективную и теоретически хорошо проработанную методику анализа, в то время как на этапе синтаксического анализа конструкций исходного языка используются достаточно сложные алгоритмы разбора;
* сканер отделяет сложный по конструкции синтаксический анализатор от работы непосредственно с текстом исходной программы, структура которого может варьироваться в зависимости от версии входного языка – при такой конструкции транслятора при переходе от одной версии языка к другой достаточно только перестроить относительно простой сканер.

Функции, выполняемые лексическим анализатором, и состав лексем, которые он выделяет в тексте исходной программы, могут различаться в зависимости от версии транслятора. В основном лексические анализаторы выполняют исключение комментариев и незначащих пробелов из текста исходной программы, а также выделение лексем следующих типов: идентификаторов, строковых, символьных и числовых констант, ключевых (служебных) слов входного языка.

Лексический анализатор любого транслятора может быть реализован в виде детерминированного конечного автомата.

Для разрабатываемого транслятора необходимо создать два лексических анализатора, как для языка PascalABC, так и для языка C#. В данном курсовом проекте было принято решение разработать лексический анализатор общего назначения, где конкретную принадлежность к какому-либо языку программирования определяли бы только лексические правила.

Для того чтобы читать поток символов из строковых данных были разработаны следующие классы.

* IStrReader – интерфейс строкового буфера для чтения потока символов из строковых данных;
* StrReader – класса реализующий интерфейс строкового буфера для чтения потока символов из строковых данных;
* FileStrReader – класс реализующий интерфейс строкового буфера для чтения потока символов из текстового файла;

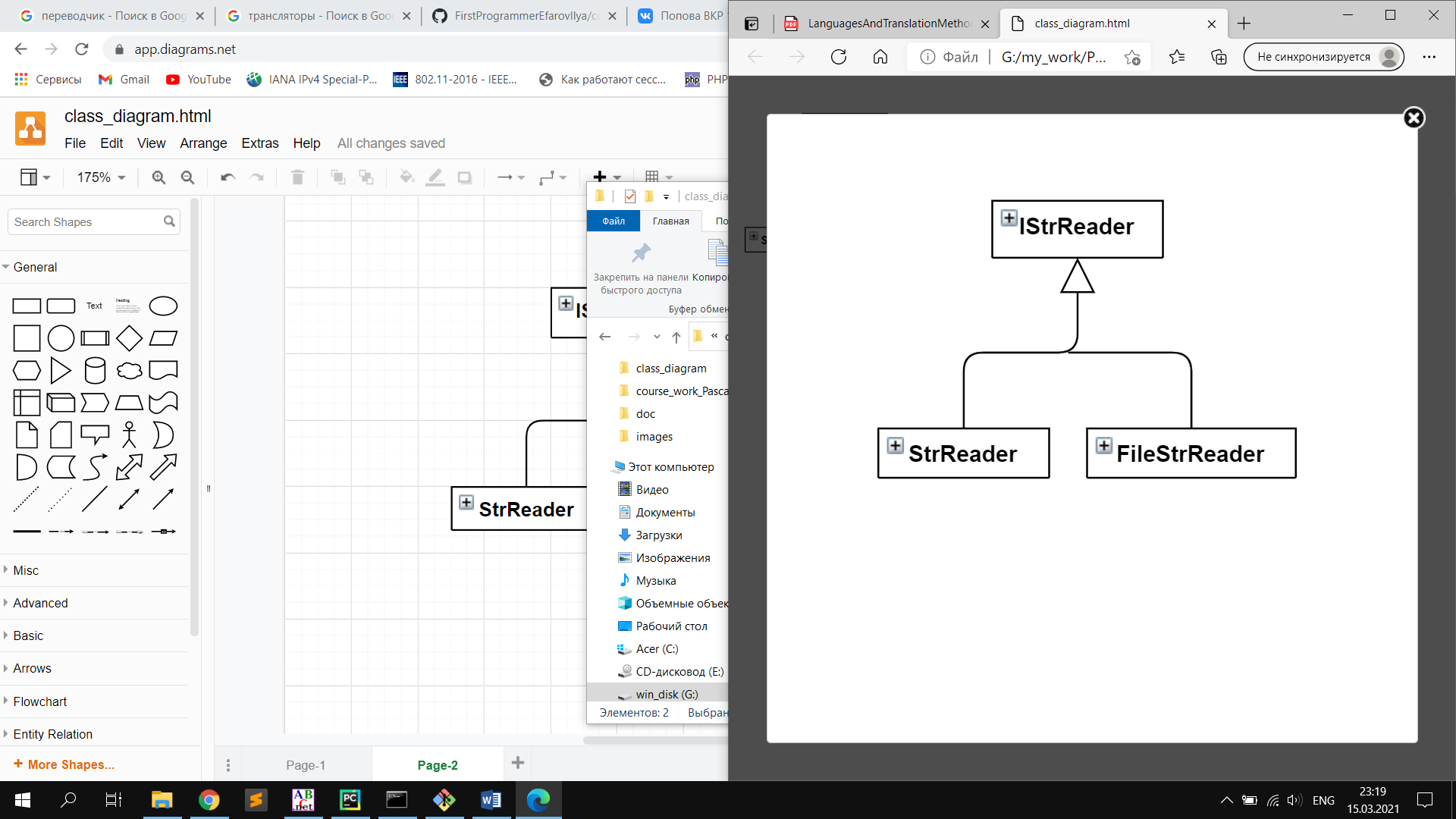


Рисунок 3.1.1 – Иерархия классов строкового буфера

Основные методы интерфейса IStrReader реализуемые его подклассами:

* reset() – Сбросить состояние буфера в начальное состояния. Переставить головку чтения в начало строковых данных;
* read(count: int): str – Считать строку указанного размера из буфера;

Данный интерфейс используется лексическим анализатором для того выполнять чтение последовательности символов данных, которую он впоследствии разбивает на отдельные лексемы.

Лексема или токен реализуется классом Token, который состоит из двух полей kind (вид лексемы) и value (значение лексемы). Поле kind может использоваться в качестве идентификатора класса лексем или названия класса лексем, а поле value может использоваться в качестве идентификатора лексемы среди лексем того же класса или фактическим значением лексемы, выделенной из последовательности символов.

Лексический анализатор описывается с помощью интерфейса ILexer. Основные методы и свойства данного интерфейса:

* data\_reader – свойство ссылка на экземпляр класса реализующего интерфейс IStrReader;
* num\_column – свойство номер текущего просматриваемого столбца в анализируемом тексте;
* num\_line – свойство номер текущей просматриваемой строки в анализируемом тексте;
* \_tokens – метод для выполнения лексического анализа и выделения лексем в анализируемом тексте. Все токены возвращаемые данным методом имеют вид ;
* tokens - метод для выполнения лексического анализа и выделения лексем в анализируемом тексте. Все токены возвращаемые данным методом имеют вид . Данный метод использует в своей работе, метод \_tokens, но токены, полученные от него заносит в таблицу лексем и вместо них возвращает в качестве токена позицию в таблице лексем;
* new\_id\_token – метод занесения токена в таблицу лексем и преобразование данного токена в токен, который представляет собой позицию в таблице лексем;
* lexemes – свойство для доступа к таблице лексем. Таблица лексем представляет собой двумерный массив, где индексом строки является , а индексом элемента в строке является . Таблиц идентификаторов, как явной структуры в разрабатываемом трансляторе не присутствует, но в не явном виде её подобие содержится в таблице лексем, в качестве строки идентификаторов программы;
* kinds – свойство для преобразования в ;
* kind\_ids – свойство для преобразования в ;

Класс Lexer является реализацией интерфейса лексического анализатора ILexer. Данный класс реализует все основные методы и свойства ILexer. Lexer в лексическом анализе для выделения лексем использует функции и классы стандартной библиотеки регулярных выражения языка Python.

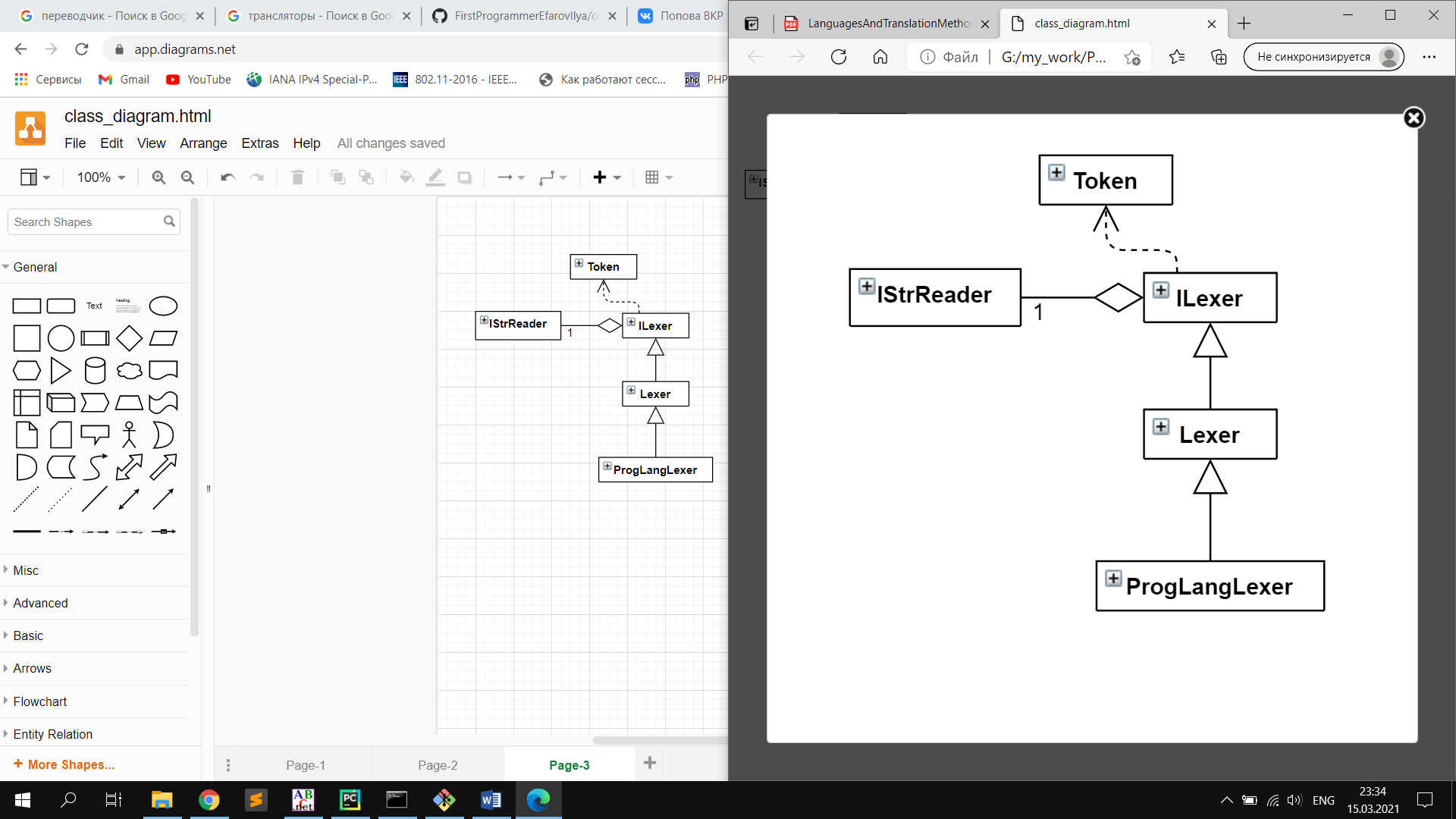


Рисунок 3.1.2 – Иерархия классов лексического анализатора

Лексические правила для правильного выделения токенов задаются в Lexer с помощью свойства specification. Это список кортежей, где первый элемент является названием токена, а второй элемент является регулярным выражением для выделения значения токена из текста. Пример значения для поля specification находится в приложении А.

Lexer решает базовую задачу лексического анализа – выделение токенов из текста, но этого недостаточно. Для лексического анализатора языков программирования важной особенностью является возможность отличия ключевых слов от идентификаторов, возможность распознавания лексем, которые необходимо отбросить, например, комментариев, возможность преобразования регистров в случае, когда язык программирования является не чувствительным к регистру, возможность распознавания многострочных лексем. Все выше описанные особенности реализуются с помощью класса ProgLangLexer, который является наследником класса Lexer и расширяет его метод \_tokens, реализацией дополнительных проверок и алгоритмов разбора лексем.

Для реализации возможности отличия ключевых слов от идентификаторов ProgLangLexer имеет свойство keywords, которое в качестве значения принимает список ключевых слов языка программирования. Также данный класс имеет в наличии свойство id\_kind для указания идентификаторов и свойство keword\_kind для указания ключевых слов.

Для выделения многострочных конструкций используется свойство multitokens. В качестве значения оно принимает список граничных символов с помощью, которых лексический анализатор понимает, где конец, а где начало многострочных конструкций.

Свойство skip\_kind предназначено для указания лексем, которые необходимо отбрасывать. Все токены, которые содержат в поле kind название из свойства skip\_kind будут мгновенно отброшены.

Для указания чувствительности к регистру используется логическое свойство case\_sensitive. Если его значение True, то язык программирования считается чувствительным к регистру.

Пример вывода результатов лексического анализа текста программы на PascalABC с помощью разработанного лексического анализатора представлен в приложении Б.

## Синтаксический анализатор

Синтаксический анализатор – это часть транслятора, которая отвечает за идентификацию и проверку синтаксических конструкций входного языка. Синтаксический анализатор получает строку токенов от лексического анализатора и проверяет, может ли эта строка токенов порождаться грамматикой входного языка. Другая функция синтаксического анализатора является генерация сообщений обо всех выявленных ошибках, причём довольно внятных и полных, и кроме того, синтаксический анализатор должен уметь обрабатывать обычные, часто встречающиеся ошибки и продолжать работу с оставшейся частью программы. В случае корректной программы синтаксический анализатор строит дерево разбора и передаёт его следующей части транслятора для дальнейшей обработки.

Перед синтаксическим анализатором стоят две основные задачи: проверить правильность конструкций программы, которая представляется в виде уже выделенных слов входного языка, и преобразовать её в вид, удобный для дальнейшей семантической (смысловой) обработки и генерации кода. Одним из способов такого представления является дерево синтаксического разбора.

Основой для построения распознавателей контекстно-свободных языков являются автоматы с магазинной памятью – МП-автоматы – односторонние недетерминированные распознаватели с линейно-ограниченной магазинной памятью.

МП-автомат в отличие от обычного конечного автомата имеет стек (магазин), в который можно помещать специальные «магазинные» символы (обычно это терминальные и нетерминальные символы грамматики языка). Переход МП-автомата из одного состояния в другое зависит не только от входного символа, но и от одного или нескольких верхних символов стека. Таким образом, конфигурация автомата определяется тремя параметрами: состоянием автомата, текущим символом входной цепочки (положением указателя в цепочке) и содержимым стека.

Синтаксический анализатор любого транслятора может быть реализован в виде детерминированного автомата с магазинной памятью.

В данном курсовом проекте для реализации синтаксического анализатора для разрабатываемого транслятора использовались алгоритмы восходящего синтаксического анализа. Рассмотрим их далее.

Восходящий синтаксический анализ соответствует построению дерева разбора для входной строки, начиная с листьев (снизу) и идя по направлению к корню (вверх).

Можно рассматривать восходящий синтаксический анализ, как процесс «свёртки» строки к целевому символу грамматики. На каждом шаге свёртки определённая подстрока, соответствующая телу правила, заменяется нетерминалом из левой части этого правила.

Ключевые решения, принимаемые в процессе восходящего синтаксического анализа, – когда выполнять свёртку и какое правило применять.

Рассмотрим базовый алгоритм восходящего синтаксического анализа - алгоритм «сдвиг-свёртка», в котором для хранения символов грамматики используется стек, а для хранения оставшейся непроанализированной части входной строки – входной буфер.

Будем использовать символ для маркирования дна стека и правого конца входной строки. При рассмотрении восходящего анализа удобно располагать вершину стека справа. Изначально стек пуст, а во входном буфере находится строка :

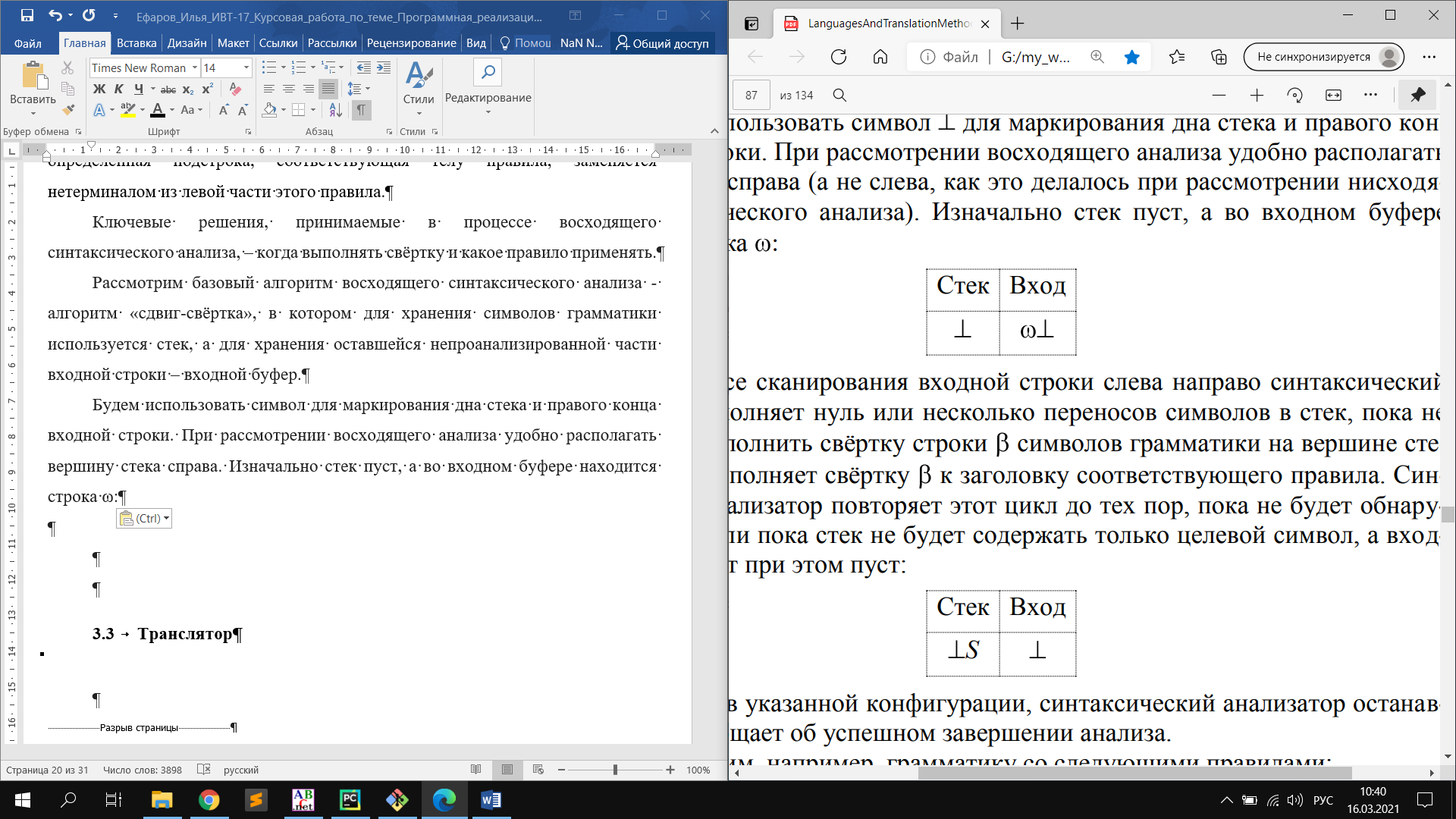


Рисунок 3.2.1 – Начальное состояние МП-автомата в алгоритме «сдвиг-свёртка»

В процессе сканирования входной строки слева направо синтаксический анализатор выполняет ноль или несколько переносов символов в стек, пока не будет готов выполнить свёртку строки символов грамматики на вершине стека. Затем он выполняет свёртку к заголовку соответствующего правила. Синтаксический анализатор повторяет этот цикл до тех пор, пока не будет обнаружена ошибка или пока стек не будет содержать только целевой символ, а входной буфер будет при этом пуст:

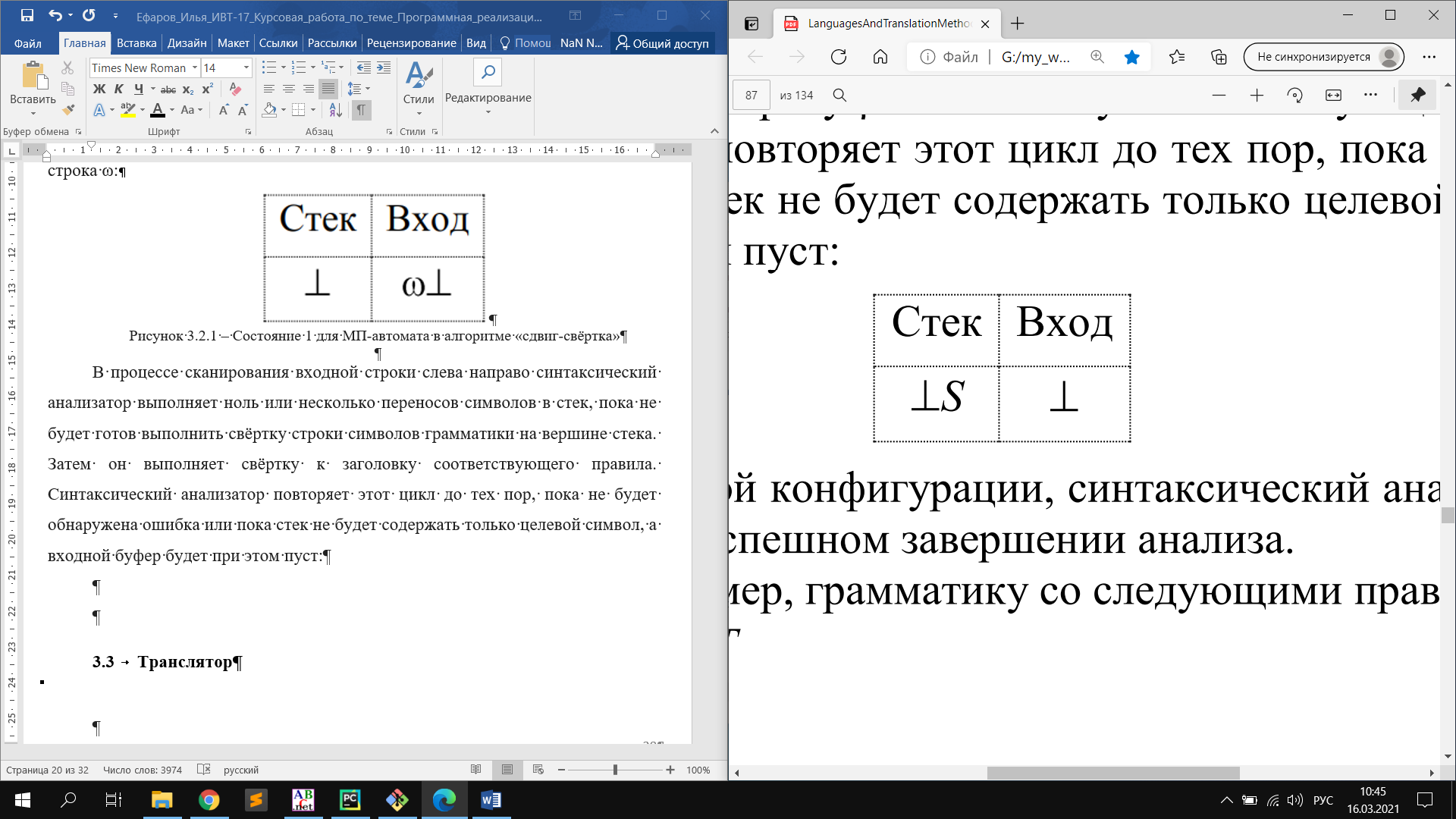


Рисунок 3.2.2 – Конечное состояние МП-автомата в алгоритме «сдвиг-свёртка»

Достигнув указанной конфигурации, синтаксический анализатор останавливается и сообщает об успешном завершении анализа.

Рассмотрим, например, грамматику со следующими правилами:

В таблице 3.2.1 показаны действия анализатора, выполняющего синтаксический анализ строки по алгоритму «сдвиг-свёртка».

Анализатор, работающий по данному алгоритму, может выполнять четыре действия:

1. Сдвиг. Перенос очередного входного символа на вершину стека;
2. Свёртка. Правая часть сворачиваемой строки должна располагаться на вершине стека. Определяется левый конец строки в стеке и принимается решение о том, каким нетерминалом будет заменена строка.
3. Принятие. Объявление об успешном завершении синтаксического анализа;
4. Ошибка. Обнаружение синтаксической ошибки и выполнение восстановления после ошибки.

Таблица 3.2.1 - Действия анализатора, выполняющего синтаксический анализ строки .

|  |  |  |
| --- | --- | --- |
| **Стек** | **Вход** | **Действие** |
| ⊥ |  | Сдвиг |
|  |  | Свёртка ) |
|  |  | Свёртка ) |
|  |  | Сдвиг |
|  |  | Сдвиг |
|  |  | Свёртка ) |
|  |  | Свёртка ) |
|  |  | Свёртка ) |
|  |  | Принятие |

Для создания синтаксического анализатора в данной курсовой работе использовалась реализация алгоритма «сдвиг-свёртка» основанная на использовании LR(1) - грамматики и её модификации LALR(1) – грамматики.

Контекстно-свободная грамматика обладает свойством LR(k), k ≥ 0, если на каждом шаге вывода для однозначного решения вопроса о выполняемом действии в алгоритме «сдвиг-свёртка» достаточно знать содержимое верхней части стека и рассмотреть первые k символов от текущего положения во входной цепочке символов. Название «LR(k)» имеет определённый смысл: «L» (от англ. left) обозначает порядок чтения входной цепочки символов слева направо, «R» (от англ. right) означает, что в результате работы распознавателя получается правосторонний вывод. Вместо «k» в названии стоит число, которое показывает, сколько символов входной цепочки надо рассмотреть, чтобы принять решение о сдвиге или свёртке.

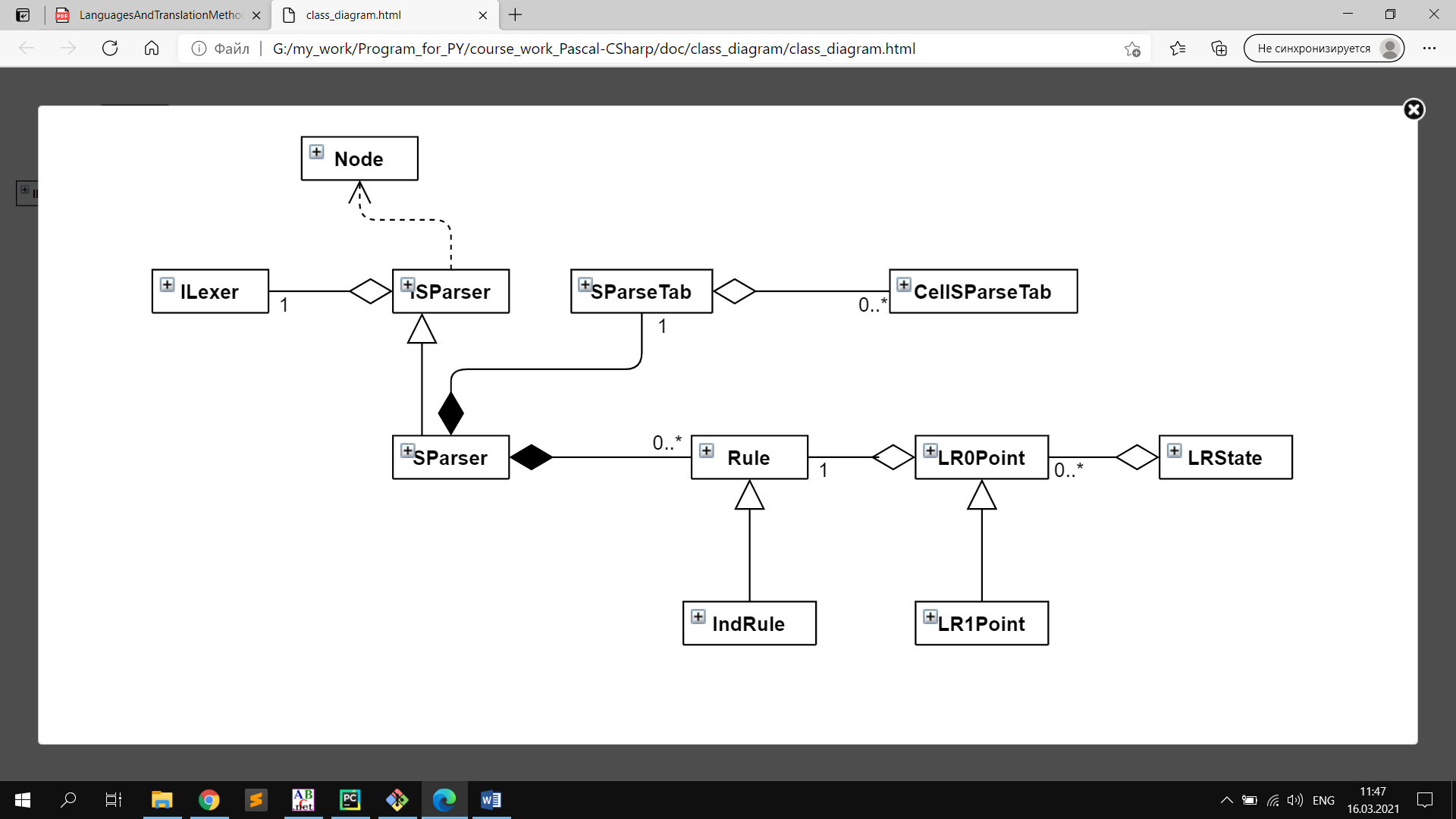


Рисунок 3.2.3 – Диаграмма классов синтаксического анализатора

Интерфейс разрабатываемого синтаксического анализатора представлен классом ISParser. Данный синтаксический анализатор в своей работе для получения токенов входного текста использует лексический анализатор - экземпляр класса, реализующего интерфейс ILexer.

Абстрактное синтаксическое дерево, которое является результатом работы синтаксического анализатора строится с помощью узлов, которые создаются с помощью класса Node. Класс Node имеет следующие свойства:

* kind – вид узла;
* value – токен, который хранится в данном узле;
* parent – родительский узел;
* childs – дочерние узлы;

Как говорит теория, LR-анализатор принимает решение о выборе «сдвиг-свёртка», поддерживая состояния, которые отслеживают, где именно в процессе синтаксического анализа он находится. Состояния представляют собой множества пунктов (ситуаций).

LR(0)-пункт грамматики – это правило этой грамматики с точкой в некоторой позиции правой части. Следовательно, правило даёт четыре пункта:

Правило генерирует единственный пункт .

Можно сказать, что пункт указывает какую часть правила, мы уже просмотрели в данной точке в процессе синтаксического анализа.

Один набор множеств LR(k)-пунктов, именуемый каноническим набором, обеспечивает основу для построения детерминированного конечного автомата, который используется для принятия решений в процессе синтаксического анализа. Такой автомат называется LR(k)-автоматом. Каждое состояние LR(k)-автомата представляет собой множество пунктов в каноническом наборе.

Для построения канонического набора определяется расширенная грамматика и два множества: и . Если – грамматика с целевым символом , то расширенная грамматика представляет собой с новым целевым символом и дополнительным правилом . Назначение этого нового правила – указать синтаксическому анализатору, когда следует прекратить анализ и сообщить о принятии входной строки; т.е. принятие осуществляется тогда и только тогда, когда синтаксический анализатор выполняет свёртку с использованием правила .

Для реализации правила грамматики был разработан класс Rule. Данный класс имеет следующие свойства:

* key – нетерминальный символ левой части правила;
* value – список символов правой части правила;

Для реализации правила грамматики, которому может быть назначен индекс был разработан класс IndRule, который является наследником класса Rule. Он имеет одно дополнительное поле index для хранения индекса правила.

LR(0)-пункт реализуется с помощью класса LR0Point. Данный класс имеет следующие свойства:

* rule – экземпляр класса Rule;
* iptr – позиция точки в правиле. Если iptr равен -1, то он никуда не указывает, если он равен длине списка value класса Rule, то указывает на конец правила, иначе он указывает на позицию перед некоторым символов в левой части правила;

Можно хранить в состоянии LR-анализатора больший объём информации, который позволит отбрасывать некорректные свёртки. Разделяя при необходимости состояния, можно добиться того, что каждое состояние будет точно указывать, какие входные символы могут следовать за основой , для которой возможна свёртка в . Дополнительная информация вносится в состояние путём такого переопределения пунктов, чтобы они включали в качестве второго компонента терминальный символ. Общим видом пункта становится , где – правило, a – терминал или маркер конца входной строки . Такой объект называется LR(1)-пунктом. Здесь 1 означает длину второго компонента, именуемого предпросмотром (англ. lookahead) пункта. Предпросмотр не влияет на пункт вида , где не равно , но пункт приводит к свёртке в соответствии с правилом , только если очередной входной символ равен .

Таким образом, свёртка в соответствии с продукцией применяется только при входном символе , для которого является LR(1)- пунктом из состояния на вершине стека.

LR(1)-пункт реализуется с помощью класса LR1Point, который является наследником класса LR0Point. Данный класс расширяет его функциональность свойством lookahead, которое по своей сути представляет список символов предпросмотра.

Рассмотрим алгоритм построения множества «замыкание пунктов» или .

Пусть – множество пунктов грамматики , тогда представляет собой множество пунктов, построенное из согласно двум правилам.

1. Изначально в добавляются все пункты из .

2. Если входит в , и существует правило , то в добавляется пункт для каждого терминала , если такого пункта ещё нет во множестве замыкания. Это правило применяется до тех пор, пока не останется пунктов, которые могут быть добавлены в .

Множество – множество терминальных символов, которыми начинаются цепочки, выводимые из в грамматике , т.е. ;

Построения множества в разрабатываемом трансляторе выполняется с помощью процедуры first\_term, а построение множества LR1-пунктов с помощью процедуры closure\_LR1. Алгоритм работы данных процедур основывается на общеизвестном алгоритме BFS или «поиск в ширину» и алгоритмах описанных выше.

Канонический набор LR(k)-пунктов, который используется LR(k)-автоматом для выполнения синтаксического анализа, состоит из множества LR(k)-состояний автомата, которые в свою очередь являются набором LR(k) –пунктов.

LR-состояние LR-автомата в данном курсовом проекте реализуется с помощью класса LRState. Основные свойства данного класса:

* index – номер LR-состояния LR-автомата;
* lrpoints – список LR(k) – пунктов;
* goto – ассоциативный массив переходов, где ключом является символ перехода, а значением LR-состояние, в которое можно выполнить переход по данному символу;
* rgoto – ассоциативный массив обратных переходов, где ключ является символ перехода, а значение список LR-состояний из которых можно перейти по данному символу перехода в данное состояние;

Рассмотрим алгоритм построения множества «переходов» или .

Алгоритм множества переходов достаточно прост. определяется, как замыкание множества всех пунктов , таких что находится в .

В разрабатываемом синтаксическом анализаторе роль множества реализуется через ранее описанные свойства класса LRState goto и rgoto. Важно понять, как правильно заполнить подобные структуры данных.

Задача построения множества на самом деле расшифровывается, как задача построения всех возможных LR-состояний для LR-автомата для указанной грамматики.

В разрабатываемом трансляторе реализуется процедура goto\_LR1Point, которая по входным LR(1)-пунктам и значению символа перехода, строит множество LR(1)-пунктов, в котором точка находится за символом перехода. Данная процедура используется в дальнейшем для построения все возможных LR-состоний LR-автомата.

Построение всех возможных LR-состояний в данной курсовой работе реализуется с помощью процедуры create\_LR1States. На вход данная процедура принимает единственный LR(1)-пункт, который содержит правило вывода целевого нетерминального символа грамматики языка.

Алгоритм create\_LR1States:

1. Построить по целевому пункту «замыкание пунктов» или . Именно это множество пунктов будет первым LR-состоянием LR(1)-автомата.
2. Построить с помощью процедуры goto\_LR1Point все возможные переходные наборы LR(1)-пунктов.
3. По переходным наборам построить множества «замыкание пунктов» или . Эти множества будут следующими построенными LR-состояниями LR(1)-автомата. Заполнить массивы goto и rgoto LR-состояний.
4. Повторить алгоритм начиная с пункта 2 для всех построенных состояний на данном шаге. Продолжать пока наборы LR(1)-пунктов построенных с помощью goto\_LR1Point не окажутся пустыми.

Рассмотрим рисунок, на котором изображён пример LR(1)-автомата построенного с помощью грамматики:

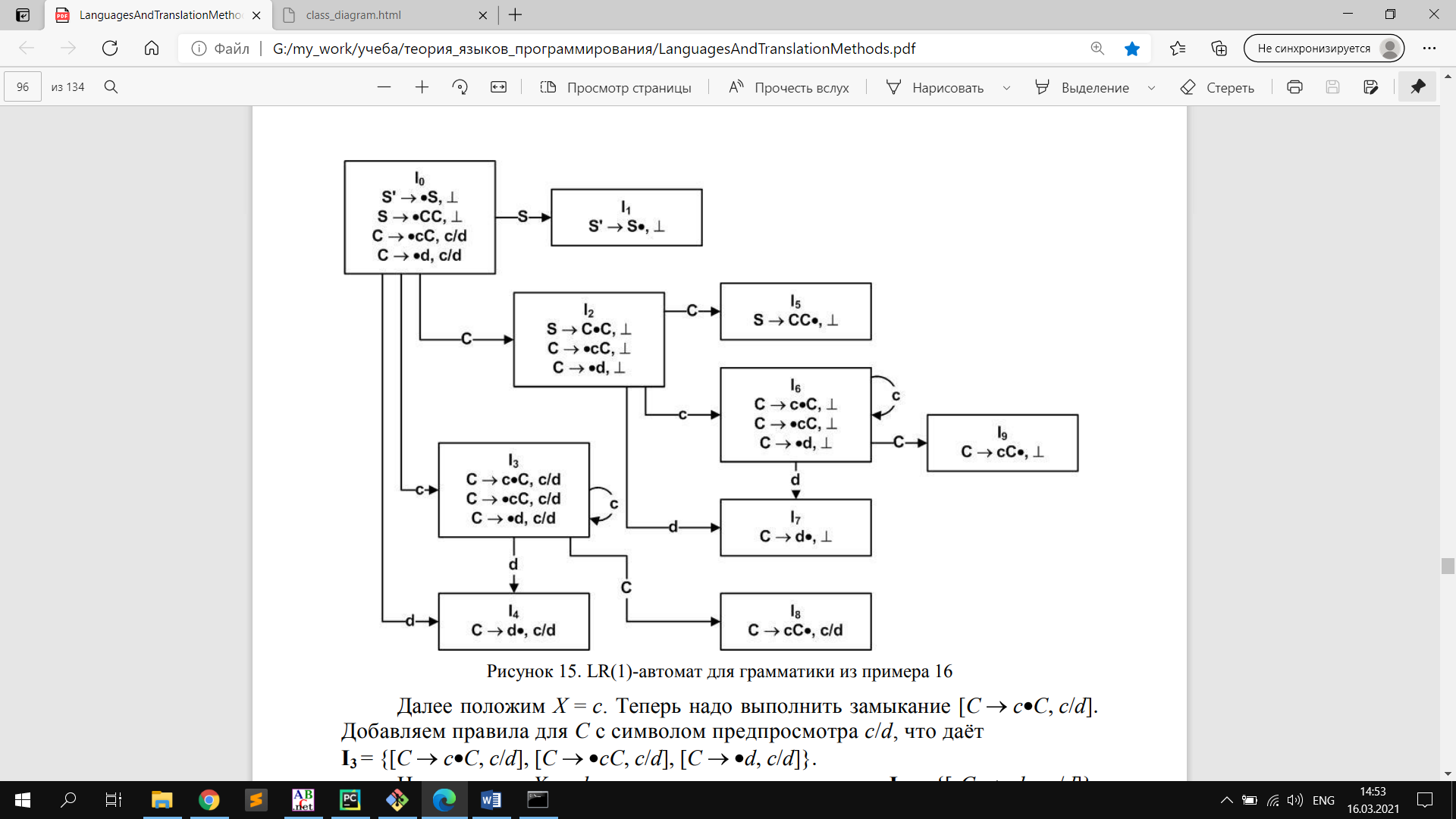


Рисунок 3.2.4 – Пример LR(1)-автомата

Имея список всех возможных LR-состояний можно построить с его помощью каноническую таблицу синтаксического анализа с помощью, которой можно быстро и качественно производить анализ синтаксиса любого современного языка программирования.

В теории алгоритм построения канонической таблицы LR(1)-анализа имеет вид. Действие синтаксического анализа для состояния i определяется следующим образом:

1. если входит во множество пунктов и , установить равным «сдвиг ». Здесь должно быть терминалом;
2. если входит в и , то установить равным «свёртка »;
3. если входит в , то установить равным «принятие».

Таблица 3.2.2 - Каноническая таблица LR(1)-анализа для LR(1)-автомата из рисунка 3.2.4.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Состояние** | **ACTION** | | | **GOTO** | |
| c | d |  | S | C |
| 0 | s3 | s4 |  | 1 | 2 |
| 1 |  |  | acc |  |  |
| 2 | s6 | s7 |  |  | 5 |
| 3 | s3 | s4 |  |  | 8 |
| 4 | r1 | r2 |  |  |  |
| 5 |  |  | r1 |  |  |
| 6 | s6 | s7 |  |  | 9 |
| 7 |  |  | r3 |  |  |
| 8 | r2 | r2 |  |  |  |
| 9 |  |  | r2 |  |  |

В данном курсовом проекте каноническая таблица LR-анализа реализуется с помощью класса SParseTab, а ячейка данной таблицы реализуется с помощью класса CellSParseTab.

Идея распознавателей для LALR(k)-грамматик (англ. lookahead LR – LR с предпросмотром) – сократить множество рассматриваемых пунктов до множества пунктов LR(0)-грамматики и упростить построение таблицы синтаксического анализа. Все LALR(k)-грамматики для всех k ≥ 1 образуют класс LALR-грамматик.

На практике используются только LALR(1)-грамматики, так как применяемый для них метод разрешения конфликтов («сдвиг-свёртка» или «свёртка-свёртка») трудно распространить на LALR(k)-грамматики для k > 1.

Сравнивая размеры синтаксических анализаторов, можно сказать, что количество состояний в LALR-таблицах обычно составляет несколько сотен для языков наподобие Си. Каноническая LR-таблица для языка такого типа обычно содержит несколько тысяч состояний.

Преобразование LR(1)-автомата в LALR(1)-автомат производится с помощью нахождения LR-состояний у которых LR(1)-пункты имеют схожие ядра и выполнения слияния найденных состояний. Два LR(1) – пункта имеют схожее ядро если правила, которые содержатся в них идентичны и позиция точек совпадают, а значение символов предпросмотра не учитывается. При слиянии двух LR(1) –пунктов новый LR(1)-пункт содержит в своём списке символов предпросмотра все символы предпросмотра сливаемых пунктов.

В разрабатываемом трансляторе преобразование полученного раннее LR(1)-автомата в LALR(1)-автомат производится с помощью процедуры states\_LR1\_to\_LALR1.

И уже по полученному LALR(1)-автомату транслятор данной курсовой работы выполняет построение ранее описанной канонической таблицы синтаксического анализа.

Далее сгенерированную таблицу разрабатываемый транслятор записывает в бинарный файл для последующего использования. Запись в файл является достаточно важным этапом потому, что генерация новой канонической таблицы синтаксического анализа процедура довольно трудоёмкая и долгая, а выполнить чтение её из бинарного файла можно в тысячу раз быстрее.

Рассмотрим пример использования канонической таблицы синтаксического анализа для выполнения анализа синтаксиса.

Пусть нам дана грамматика со следующими правилами:

Тогда по данным правилам можно построить следующую каноническую таблицу LALR(1) – анализа.

Таблица 3.2.3 - Каноническая таблица LALR(1)-анализа.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Состояние** | **ACTION** | | | | | | | | **GOTO** | | |
| + | - | \* | / | ( | ) | id |  | E | T | F |
| 0 |  |  |  |  | s4 |  | s3 |  | 1 | 2 | 5 |

Продолжение таблицы 3.2.3

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | s6 | s7 |  |  |  |  |  | acc |  |  |  |
| 2 | r3 | r3 | s12 | s13 |  | r3 |  | r3 |  |  |  |
| 3 | r8 | r8 | r8 | r8 |  | r8 |  | r8 |  |  |  |
| 4 |  |  |  |  | s4 |  | s3 |  | 10 | 2 | 5 |
| 5 | r6 | r6 | r6 | r6 |  | r6 |  | r6 |  |  |  |
| 6 |  |  |  |  | s4 |  | s3 |  |  | 8 | 5 |
| 7 |  |  |  |  | s4 |  | s3 |  |  | 9 | 5 |
| 8 | r1 | r1 | s12 | s13 |  | r1 |  | r1 |  |  |  |
| 9 | r2 | r2 | s12 | s13 |  | r2 |  | r2 |  |  |  |
| 10 | s6 | s7 |  |  |  | s11 |  |  |  |  |  |
| 11 | r7 | r7 | r7 | r7 |  | r7 |  | r7 |  |  |  |
| 12 |  |  |  |  | s4 |  | s3 |  |  |  | 14 |
| 13 |  |  |  |  | s4 |  | s3 |  |  |  | 15 |
| 14 | r4 | r4 | r4 | r4 |  | r4 |  | r4 |  |  |  |
| 15 | r5 | r5 | r5 | r5 |  | r5 |  | r5 |  |  |  |

По данной таблице можно выполнить следующий разбор.

Таблица 3.2.4 - Действия LALR(1)-анализатора для строки .

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№** | **Стек** | **Символы** | **Вход** | **Действие** |
| 1 | 0 |  |  | Сдвиг |
| 2 | 0 3 |  |  | Свёртка по |
| 3 | 0 5 |  |  | Свёртка по |
| 4 | 0 2 |  |  | Сдвиг |
| 5 | 0 2 12 |  |  | Сдвиг |
| 6 | 0 2 12 3 |  |  | Свёртка по |
| 7 | 0 2 12 14 |  |  | Свёртка по |
| 8 | 0 2 |  |  | Свёртка по |
| 9 | 0 1 |  |  | Сдвиг |
| 10 | 0 1 6 |  |  | Сдвиг |

Продолжение таблицы таблицы 3.2.4

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 11 | 0 1 6 3 |  |  | Свёртка по |
| 12 | 0 1 6 5 |  |  | Свёртка по |
| 13 | 0 1 6 8 |  |  | Свёртка по |
| 14 | 0 1 |  |  | Принятие |

Интерфейс ISParser диктует необходимость реализации следующих методов:

* parse(): Node – Выполняет синтаксический анализ входящего потока токенов и выполняет построение синтаксического дерева разбора, где узлами дерева являются объекты класса Node.
* lexer(): ILexer – Свойство доступа к лексическому анализатору. Даёт возможность установить или получить лексический анализатор для парсера.

Класс Sparser реализует интерфейс ISParser. Опишем некоторые основные методы для работы с данным классом:

* rules(): list(Rule, …) – получить список правил грамматики языка;
* parse\_rules(str) –преобразовать текстовую спецификацию грамматики языка, в список грамматических правил для синтаксического анализатора. Спецификация грамматики языка - это текстовое описание грамматических правил в форме описания, которая соответствует форме Бэкуса-Наура. Пример спецификации находится в приложении В;
* term\_segreg() – получить или назначить кортеж символов, которые являются обособлением терминальных символов (лексем) в тексте спецификации к синтаксическому анализатору;
* tokens() – назначить или получить лексемы, которые являются терминальными символами грамматики языка;
* goal\_nterm(): str – получить или назначить целевой символ грамматики языка;
* end\_term(): str – получить или назначить конечный символ грамматики языка;
* empty\_term() – получить или назначить символ пустой цепочки для грамматики языка;
* create\_sparse\_tab() – метод для генерации канонической таблицы LALR(1)-анализа по полученным ранее rules, term\_segreg, tokens, goal\_nterm, end\_term и empty\_term. Здесь создаётся расширенная грамматика по отношению к заданной. Строится LR(1)-состояния для LR(1)-автомата. Конечный LR(1)-автомат преобразуется в LALR(1)-автомат и уже на его основе генерируется каноническая таблица синтаксического анализа;
* parse(): Node – реализация метода parse интерфейса ISParser;
* write\_stab\_to\_file(str) – записать каноническую таблицу LALR(1)- анализа в бинарный файл;
* read\_stab\_from\_file(str) – считать каноническую таблицу LALR(1) – анализа из бинарного файла.

Одним из самых важных методов который реализует класс Sparser является метод parse. Он работает согласно хорошо известному алгоритму, который гласит, что при синтаксическом анализе основной упор необходимо делать на работу с канонической таблицей синтаксического анализа, то есть выполнять все действия, которые она диктует в своих ячейках. Пример использования данного алгоритма был описан ранее в таблице 3.2.4.

Но есть ряд деталей в алгоритме метода parse, на которые необходимо обратить внимание.

1. В качестве символьного буфера используется список объектов класса Node. Это значит, что при каждом сдвиге анализатор создаёт новый объект Node и помещает в его поле value токен полученный от лексического анализатора;
2. При свёртке анализатор создаёт новый узел Node и выполняет извлечение определённого количество узлов из буфера узлов согласно правилу, по которому выполняется свёртка. Извлечённые узлы анализатор помещает, как дочерние узлы для вновь созданного узла Node, то есть добавляет данные объекты в список childs созданного узла. Для каждого из дочерних узлов созданный узел устанавливается в поле parent, то есть становится их родителем;
3. В случае выполнения принятия анализатор извлекает последний узел из буфера узлов и возвращает его из метода, как корень синтаксического дерева разбора;
4. В случае возникновения ошибок при анализе данный метод выбрасывает исключения, которые должны быть обработаны в процедуре, которая данный метод вызвала.

Пример текстового вывода синтаксического дерева разбора, построенного разработанным синтаксическим анализатором, находится в приложении Г.

Далее разработанные трансляторы обоих языков, как PascalABC, так и C# производят анализ построенного синтаксическим анализатором дерева разбора и выполняют генерацию кода на целевом языке программирования.

**ПРИЛОЖЕНИЕ А**

(обязательное)

Пример спецификации (лексических правил) лексического анализатора

SPECIFICATION = [  
 ('SKIP', r'[\s\t]+'),  
 ('ID', r'[\_A-Za-zА-Яа-я&][\_A-Za-zА-Яа-я\d]\*'),  
 ('NUM', r'\d+(\.\d+)?((e|E)(\+|-)?\d+)?'),  
 ('HEX\_NUM', r'\$[A-Fa-f0-9]+'),  
 ('STR', r"'[^']\*'"),  
 ('RANGE', r'\.\.'),  
 ('PTR', r'[\^@]'),  
 ('LAMBDA', r'->'),  
 ('COND', r'\?\[?'),  
 ('LINE\_COMMENT', r'//.\*'),  
 ('OP\_ASN', r'[-\+\\*/:]='),  
 ('OP\_ARTHM', r'(\\*\\*|[-\\*\+/])'),  
 ('OP\_COMP', r'(<>|[<>]?=|[<>])'),  
 ('DIRECTIVE', r'{\$.\*}'),  
 ('DELIM', r'[:;,()\.\[\]]'),  
]

Краткое описание некоторых элементов спецификации:

"""  
ID: r'''  
 [\_A-Za-zА-Яа-я] # первый символ идентификатора  
 [\_A-Za-zА-Яа-я\d]\* # последующие символы  
 ''',  
  
NUM: r'''  
 \d+ # целая часть  
 (\.\d+)? # вещественная часть  
 ((e|E)(\+|-)?\d+)? # десятичный порядок  
 '''  
  
STR: r"  
 '[^']\*' # строковый литерал  
 "  
"""

**ПРИЛОЖЕНИЕ Б**

(обязательное)

Пример вывода результатов лексического анализа текста программы на языке PascalABC

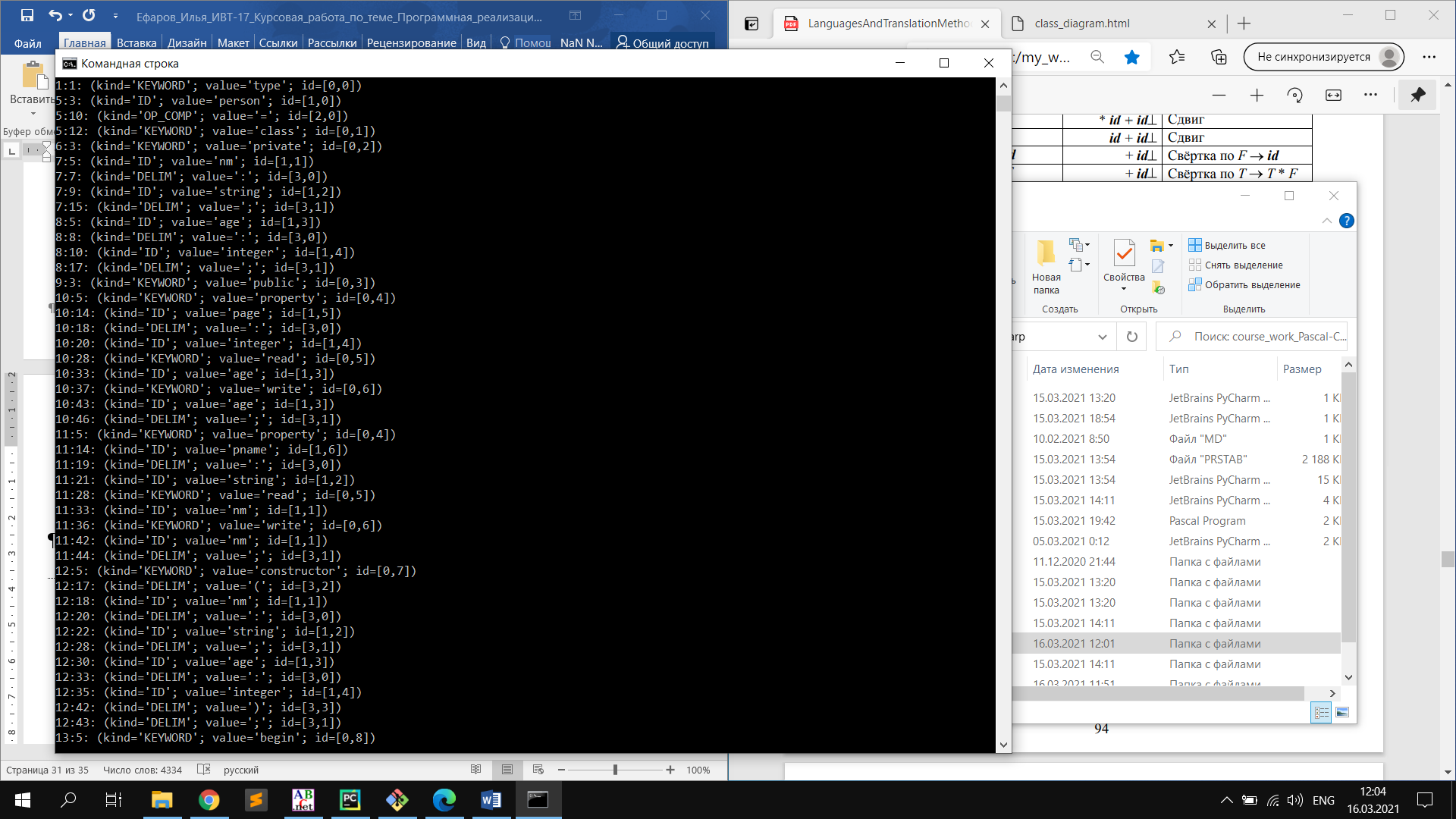


Рисунок Б.1 - Пример вывода результатов лексического анализа текста программы на языке PascalABC

**ПРИЛОЖЕНИЕ В**

(обязательное)

Пример спецификации грамматики языка для синтаксического анализатора

GOAL\_NTERM = "E"  
 END\_TERM = '⊥'  
 EMPTY\_TERM = 'ε'  
 RULES = """  
 E -> E '+' T |  
 E '-' T |  
 T;  
 T -> T '\*' F |  
 T '/' F |  
 F;  
 F -> '(' E ')' |  
 ID  
 """  
 TOKENS = ('ID',)

С помощью переменной GOAL\_NTERM указывается целевой символ грамматики языка. С помощью END\_TERM конечный символ грамматики языка. С помощью EMPTY\_TERM символ пустой цепочки. С помощью TOKENS терминальные символы в грамматике языка. Также терминальными символами могут быть лексемы, которые обособляются в спецификации ординарными кавычками вида «'». Конец любого правила в спецификации, кроме последнего помечается с помощью точки с запятой «;» и символа переноса строки «\n». Множественность вариантов исхода для символа с правой стороны правила обозначается с помощью символа «|» и символа переноса строки «\n».

**ПРИЛОЖЕНИЕ Г**

(обязательное)

Пример текстового вывода синтаксического дерева разбора, построенного синтаксическим анализатором, для текста программы на языке PascalABC

node : program

node : section\_var

var

node : vars

node : vars\_decl

node : list\_ids2

node : list\_ids

p1

,

p2

,

pres

:

integer

;

node : funcs

node : func

node : func\_hdr

procedure

foo

node : func\_params

(

node : fparams1

var

node : var\_decl

p

:

integer

)

node : method\_body

;

node : block

begin

node : stmts

node : asgn

p

:=

node : add

p

+

1001

;

end

;

node : block

begin

node : stmts

node : stmts

node : stmts

node : stmts

node : stmts

Продолжение приложения Г

node : asgn

p1

:=

12

;

node : asgn

p2

:=

100

;

node : asgn

pres

:=

node : add

p1

+

p2

;

node : if

if

node : rel

pres

>

200

then

node : asgn

pres

:=

node : add

pres

-

p2

else

node : asgn

pres

:=

node : add

pres

+

p1

;

node : call

writeln

(

pres

)

;

end

.