**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ**

**Федеральное государственное бюджетное образовательное учреждение**

**высшего образования**

**«Чувашский государственный университет имени И.Н. Ульянова»**

**Факультет информатики и вычислительной техники**

**Кафедра вычислительной техники**

***ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ***

**Расчетно-графическая работа (часть 1)**

**Лексический анализ**

**Выполнил:**

студент группы ИВТ-41-21

Фамилия И.О.

**Руководитель:**

доцент Павлов Л.А.

Чебоксары 2024

**Оглавление**

[Задание к РГР 3](#_Toc170033152)

[Введение 4](#_Toc170033153)

[1. Описание языка 5](#_Toc170033154)

[2. Лексические классы языка 7](#_Toc170033155)

[3. Синтез конечных автоматов-распознавателей 8](#_Toc170033156)

[4. Структуры данных и алгоритмы 11](#_Toc170033157)

[5. Программная реализация лексического анализа 15](#_Toc170033158)

[Заключение 18](#_Toc170033159)

[Список использованной литературы 18](#_Toc170033160)

Задание к РГР

1. Разработать синтаксис простого учебного языка программирования, используя нотацию БНФ или РБНФ.

2. Разработать программный модуль лексического анализа (сканер).

Общие требования к учебному языку:

* язык должен быть со строгой явной статической типизацией, т.е. типы всех объектов должны быть объявлены в специальном разделе описаний и не допускается неявное преобразование типов;
* должны быть ключевые слова, обозначающие начало и конец программы;
* программа должна состоять из раздела описаний и раздела (последовательности) операторов;
* должно быть как минимум три простых предопределенных базовых типа: целый, вещественный, логический;
* для арифметических выражений должны быть определены как минимум операции сложения, вычитания, умножения, деления (унарный плюс и унарный минус – по желанию);
* для логических выражений должны быть определены операции дизъюнкции (ИЛИ), конъюнкции (И), отрицания (НЕ) и шесть операций отношения;
* обязательным оператором является оператор присваивания;
* текст программы должен допускать использование комментариев.

**Примечание.** В Вашей работе к общим требованиям добавляются производные типы и другие операторы в зависимости от номера варианта (например, вариант 99):

* производный тип – массив;
* оператор цикла с предусловием.

**Еще примечание.** Данный файл является шаблоном отчета по РГР. Поэтому далее будет рассмотрен простейший демо-язык, предусматривающий только два типа данных (целый и вещественный с фиксированной точкой), только оператор присваивания и только арифметические выражения. Нет логических выражений, управляющих операторов и производных типов.

Цель – показать обязательную структуру отчета, формы различных таблиц и т. п., минимизировать готовые решения, чтобы Вам было над чем работать.

Введение

***Цель работы*** – закрепление теоретических знаний, приобретение умений и практических навыков формального описания синтаксиса языка программирования и формирования перечня семантических соглашений. Изучение методов построения лексических анализаторов (сканеров), способов построения распознавателей для регулярных языков; получение практических навыков синтеза, детерминизации и минимизации распознающих конечных автоматов и их программной реализации, выбора структур данных для представления таблиц сканера; разработка лексического анализатора в соответствии с заданным вариантом.

Лексический анализ (сканирование) является первой фазой компиляции. Его основная задача состоит в предварительной обработке исходного текста программы, которая заключается в группировании символов входного потока в лексические единицы (*лексемы*). Для каждой лексемы сканер формирует выходной *токен* вида <*код\_токена*, *атрибут*> для последующих фаз компиляции. *Код\_токена* идентифицирует класс лексемы (*лексический класс*) и определяют работу синтаксического анализатора (рассматривается как терминал). Для удобства *код\_токена* будем представлять абстрактным именем (или специальным обозначением), выделенным жирным шрифтом, и ссылаться на токен по его имени (обозначению). *Атрибут* токена обеспечивает доступ к дополнительной информации о лексеме, если лексическому классу соответствует множество лексем, и определяет трансляцию токена (семантический анализ и генерация промежуточного кода).

Часто фазы лексического и синтаксического анализа объединяют в один проход. В этом случае лексический анализатор является подпрограммой синтаксического анализатора. Когда синтаксическому анализатору требуется очередной токен, он вызывает лексический анализатор, который формирует очередной токен и возвращает управление синтаксическому анализатору. В данной РГР, поскольку еще не рассматриваются последующие фазы компиляции, вместо процедур синтаксически управляемой трансляции реализовано сохранение токена. В результате исходная программа, интерпретируемая лексическим анализатором как последовательность лексем, полностью преобразуется в последовательность соответствующих токенов.

Лексический анализатор выполняет также и другие функции. В частности, он удаляет из текста исходной программы комментарии и не несущие смысловой нагрузки пробелы, символы табуляции и символы перевода строки. Еще одной задачей является согласование сообщений об ошибках компиляции и текста исходной программы (указать каким-либо образом позицию ошибки и ее характер в тексте программы). Кроме того, лексический анализатор должен строить различные таблицы, необходимые как для собственно лексического анализа, так и для последующих фаз компиляции.

В процессе проектирования сканера необходимо решить следующие задачи:

* определить формальную грамматику заданного языка;
* выделить конструкции и элементы заданного языка (лексемы), обрабатываемые на этапе лексического анализа, и определить для них соответствующие регулярные грамматики и/или регулярные выражения;
* синтезировать распознающий конечный автомат по регулярной грамматике (регулярному выражению), детерминизировать, минимизировать и программно реализовывать полученный конечный автомат;
* выбрать оптимальную (с точки зрения требуемой памяти и времени доступа) организацию структур данных лексического анализатора;
* разработать структуру лексического анализатора, определить его функции и алгоритмы и программно его реализовать.

# Описание языка

Синтаксис формального языка (назовем его языком *DemoLang*) представим в расширенной форме Бэкуса-Наура (РБНФ). Из существующих различных модификаций синтаксиса РБНФ используем следующий вариант.

Металингвистическая переменная (нетерминал) обозначается произвольной символьной строкой. Если нетерминал состоит из нескольких смысловых слов, то они записываются слитно или разделяются символом подчеркивания.

Терминальные символы изображаются цепочками символов, заключенными в одиночные (′) или двойные (″) кавычки. Открывающая кавычка должна быть точно такой же, как и закрывающая, и не должна встречаться внутри цепочки.

Левая и правая части правила разделяются метасимволом "=" (вместо "::=" в БНФ), альтернативные варианты разделяются метасимволом "**|**". Каждое правило заканчивается точкой.

Квадратные скобки "[" и "]" означают, что заключенная в них синтаксическая конструкция может отсутствовать.

Фигурные скобки "{" и "}" означают нуль или более повторений заключенной в них синтаксической конструкции.

Курсивом выделены понятия, подчеркивающие семантический смысл синтаксической конструкции.

Ниже приведено описание синтаксиса языка *DemoLang*.

1. Программа = "**prog**" Имя\_*программы* "**;**" Блок.
2. Имя = Буква {Буква **|** Цифpа }.
3. Блок = "**var**" ОписПеременных "**start**" ПоследОператоров "**stop**".
4. ОписПеременных = ТипПеременных "**;**" { ТипПеременных "**;**" }.
5. ТипПеременных = СписокИмен "**:**" Имя\_*типа*.
6. СписокИмен = Имя { "**,**" Имя }.
7. ПоследОператоров = Оператор "**;**" { Оператор "**;**"}.
8. Оператор = Имя\_*переменной* "**:=**" Выражение.
9. Выражение = Терм { АддитОперация Терм }.
10. АддитОперация = "**+**" **|** "**–**".
11. Терм = Фактор { МультОперация Фактор }.
12. МультОперация = "**\***" **|** "**/**".
13. Фактор = Число **|** Имя\_*переменной* **|** "**(**" Выражение "**)**".
14. Число = Целое [ "**.**" Целое ].
15. Целое = Цифpа { Цифpа }.
16. Цифра = "0"**|**"1"**|**"2"**|**"3"**|**"4"**|**"5"**|**"6"**|**"7"**|**"8"**|**"9".

Определение нетерминала «Буква» здесь не приведено ввиду его очевидности – определяется выбранным алфавитом (обычно строчные и прописные буквы латинского алфавита).

**Внимание!** Вы обязательно в РГР должны предусмотреть возможность представления числа в формате с плавающей точкой!

***Краткая характеристика языка и семантические соглашения***:

* Язык удовлетворяет семантическим соглашениям, характерным для многих языков программирования (единственность именования различных объектов программы, необходимость описания идентификатора до его использования и т. п.).
* В ключевых словах и идентификаторах прописные и строчные буквы не различаются.
* Ключевые слова языка зарезервированы, их нельзя использовать в качестве идентификаторов.
* Отсутствуют именованные константы.
* Отсутствуют унарные операции + и –, т. е. для представления, например, отрицательного числа –8, надо использовать бинарную операцию 0 – 8.
* Базовые типы: *integer*, *real*.
* Отсутствуют производные типы, т. е. есть только базовые простые типы.
* Предусмотрен только один оператор – оператор присваивания.
* Отсутствуют логические выражения.
* Комментарий представляет собой любую последовательность символов, заключенную в ограничители – фигурные скобки "{" и "}".

В языке (в соответствии с приведенной выше грамматикой) определена следующая приоритетность арифметических операций (в порядке убывания приоритетов):

1. Мультипликативные операции: "**\***" **|** "**/**"

2. Аддитивные операции: "**+**" **|** "**–**"

Пример программы на языке *DemoLang*:

**prog** Test1;

**var**

i, j, k, l, m, n: integer;

x, y, z: real;

**start** {комментарий}

k := i \* (l + m) + n \* j – 48;

x := (y – z) \* 3.86 + x / 5.00;

**stop**

# Лексические классы языка

Перечень лексических классов (токенов) для языка *DemoLang* представлен в табл. 1 (в качестве формальных шаблонов токенов используются регулярные выражения).

Таблица 1. Таблица лексических классов

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| №  п/п | Токен | Имя (обозначение) | Код токена | Формальный шаблон | Значение атрибута |
|  | **prog** | **prog** | *lcProg* | prog | 0 |
|  | **start** | **start** | *lcStart* | start | 0 |
|  | **stop** | **stop** | *lcStop* | stop | 0 |
|  | **var** | **var** | *lcVar* | var | 0 |
|  | точка с запятой | **;** | *lcSemi* | **;** | 0 |
|  | двоеточие | **:** | *lcColon* | **:** | 0 |
|  | запятая | **,** | *lcComma* | **,** | 0 |
|  | откр. скобка | **(** | *lcOpPar* | **(** | 0 |
|  | закр. скобка | **)** | *lcClPar* | **)** | 0 |
|  | присваивание | **ass** | *lcAss* | **:=** | 0 |
|  | аддитивная  операция | **+** | *lcAdd* | **+ | –** | *opAdd* (+)  *opSub* (–) |
|  | мультипл.  операция | **\*** | *lcMult* | **\* | /** | *opMult* (\*)  *opDiv* (/) |
|  | идентификатор | **id** | *lcId* | ***l* (*l* | *d*)\***  *l* – буква, *d* – цифра | указатель на запись  в таблице символов |
|  | число | **num** | *lcNum* | ***d*+ | *d*+.*d*+**  *d* – цифра | указатель на запись  в таблице символов |
|  | пробел | **sp** |  | (**sp** | **tab** | **lf**)+  sp – пробел,  tab – табуляция,  lf – перевод строки | токен не формируется |
|  | комментарий | **com** |  | **{** **⊥** **}**  **⊥** – любой символ кроме **}**) | токен не формируется |
|  | маркер конца ввода | **eof** | *lcEof* |  | 0 |
|  | лексическая ошибка | **err** | *lcErr* |  | 0 |

Для лучшей читабельности для представления кодов токенов используются имена вместо безличных числовых кодов. Поскольку лексические классы «пробел» (**sp**) и «комментарий» (**com**) не формируют токен, нет необходимости в сопоставлении им кода токена (соответствующие ячейки таблицы пусты). Первые 4 строки соответствуют ключевым словам.

**Еще раз напоминаю!** Вы обязательно в РГР должны предусмотреть возможность представления числа в формате с плавающей точкой!

Наряду с токенами, определяемыми грамматикой языка *DemoLang*, добавлены служебные токены: «маркер конца ввода» **eof** (*lcEof*) и «лексическая ошибка» **err** (*lcErr*). Для этих токенов не предусмотрены лексемы языка, поэтому соответствующие формальные шаблоны пусты.

С помощью токена **err** сканер информирует парсер о наличии лексической ошибки (парсер должен прекратить работу). Токен **eof** предназначен для информирования парсера о завершении сканирования входного текста, т. е. генерируется маркер конца ввода.

**Примечание.** В моих методических работах и презентациях лекций маркер конца ввода обозначается символом ⊥. Очевидно, что имя токена **eof** в транслируемом тексте программы должно отсутствовать (автоматически генерируется сканером), но оно должно обязательно присутствовать в грамматике, используемой для генерации таблиц разбора в приложении TblGenerator.exe.

Приведенные в таблице регулярные выражения не учитывают опережающее чтение дополнительных символов для распознавания токена. Поэтому эти символы будут учтены непосредственно при синтезе конечных автоматов-распознавателей.

**Примечание.** В приведенном языке каждое ключевое слово образует собственный лексический класс (токен). Возможны и другие подходы. Например, пусть в языке имеются логические операции **and** и **or**. Пусть каждая из этих операций образуют собственный лексический класс как ключевое слово. Тогда их нельзя включать в лексические классы «мультипликативные операции» и «аддитивные операции» соответственно. Рассмотрение логических операций отдельно от арифметических операций сделает более наглядной реализацию генерации промежуточного кода для логических выражений, хотя это и приведет к небольшому увеличению числа терминалов и числа продукций в формальной грамматике. Для упрощения формальной грамматики можно включить логические операции в соответствующие лексические классы. Тогда ключевые слова **and** и **or** не будут образовывать собственные токены, а будут включены в токены «мультипликативная операция» и «аддитивные операции» соответственно как отдельные операции с соответствующими значениями атрибута. Возникает ситуация, когда не каждому ключевому слову соответствует свой токен. Для того чтобы данная информация была доступна лексическому анализатору, в таблицу ключевых слов наряду с полем кода токена необходимо добавить также и поле для соответствующего значения атрибута, что приведет к увеличению размера необходимой памяти для хранения таблицы ключевых слов. Пример такой реализации приведен в [7].

# Синтез конечных автоматов-распознавателей

Прежде всего, следует решить вопрос распознавания ключевых слов. Одним из вариантов является создание специальной статической таблицы ключевых слов (тогда не надо для них строить автоматы). Для реализации бинарного поиска достаточно хранить список лексем, представляющих ключевые слова, в алфавитном порядке. При необходимости (в случае длинных ключевых слов) в структуру таблицы можно добавить поле для хранения более короткого имени (обозначения) соответствующего токена.

**Примечание.** Создание таблицы ключевых слов не самый эффективный метод. Посудите сами. Пусть *m* – длина лексемы. Тогда конечный автомат распознает лексему за время *O*(*m*). Далее, если лексема представляет собой идентификатор, для выяснения, является ли лексема ключевым словом, реализуется бинарный поиск в таблице ключевых слов, что потребует времени *O*(log *n*), где *n* – число ключевых слов. В итоге для распознавания ключевого слова потребуется время *O*(*m* log *n*). Если же ключевые слова включить в конечный автомат-распознаватель, то потребуется время *O*(*m*).

Для всех остальных токенов (включая и лексические классы «комментарий» и «пробел») следует построить распознающие автоматы.

Распознавателем регулярного языка является конечный детерминированный автомат *M* = (*K*, *T*, δ, *k*0, *F*), где *K* – конечное множество состояний, *Т* – конечный входной алфавит, δ: *K* × *T* → *K* – функция переходов, *k*0∈*K* – начальное состояние автомата, *F*⊆ *K* – множество конечных состояний.

При синтезе автоматов будет учитываться опережающее чтение дополнительных символов и возврат этих символов во входной поток. Данная информация указывается в конечных состояниях автоматов, а именно: формируемый токен с соответствующим значением атрибута и число символов, возвращаемых во входной поток. Синтез выполняется по соответствующим регулярным выражениям (см. табл. 1). Конечные состояния автоматов обозначим отрицательными числами [2].

После синтеза автоматов для отдельных токенов (подмножеств токенов) можно выполнить их объединение в один автомат. Такое объединение легко реализуется, если первые читаемые символы для всех исходных автоматов различны. Поэтому начальные состояния всех автоматов обозначаются состоянием 0, для остальных состояний используется сквозная нумерация (отдельно для внутренних и отдельно для конечных состояний).

Графы синтезированных автоматов, распознающих токены языка *DemoLang*, представлены на рис. 1. Конечные состояния автоматов изображены в виде прямоугольников. Дуги, помеченные символом ⊥, означают переход в соответствующее состояние при чтении любого другого входного символа. Для конечного состояния указаны формируемый токен с соответствующим значением атрибута и число символов, возвращаемых во входной поток при выполнении опережающего чтения. Поскольку ключевые слова удовлетворяют правилам образования идентификаторов, в конечном состоянии –11 автомата для токена **id** (рис. 1, *д*) определяется, является данный идентификатор ключевым словом (если является, формируются соответствующие токен и, если есть, значение атрибута из таблицы ключевых слов) или нет (в этом случае формируется токен **id** с соответствующим значением атрибута). Для токенов **num** и **id** значениями атрибутов устанавливаются указатели на соответствующие записи таблицы символов (обозначение *pnt*), а для токенов **+**, **\*** – соответствующие коды операций (обозначение *op*).

Поскольку первые читаемые символы в начальном состоянии 0 для всех исходных автоматов различны, легко реализуется их объединение в один автомат. В этом случае автоматы, изображенные на рис. 1, можно рассматривать как соответствующие фрагменты одного автомата, реализующего лексический анализ. Этот автомат после распознавания токена в каком-либо конечном состоянии и передачи его синтаксическому анализатору прекращает работу. После запроса следующего токена автомат начинает его распознавание с начального состояния 0. Исключением являются токены «пробел» и «комментарий», которые не передается синтаксическому анализатору. После их распознавания автомат переводится в состояние 0, т. е. начинается распознавание следующего токена с начального состояния (причем без возврата управления парсеру).

–1

**;**

<**;**, 0>, возврат 0

–2

**,**

<**,**, 0>, возврат 0

*а*

*г*

–10

**:**

<**ass**, 0>, возврат 0

**=**

–9

<**:**, 0>, возврат 1

**⊥**

–11

**⊥**

*l*

<**id**, *pnt*>,

возврат 1

*l* | *d*

–14

**⊥**

<**sp**, 0>,

возврат 1

sp | tab | lf

sp | tab | lf

–15

**}**

**{**

<**com**, 0>,

возврат 0

**⊥**

*з*

*ж*

–4

**)**

<**)**, 0>, возврат 0

–3

**(**

<**(**, 0>, возврат 0

*в*

–7

**\***

<**\***, *opMult*>, возврат 0

–8

**/**

<**\***, *opDiv*>, возврат 0

*б*

–5

**+**

<**+**, *opAdd*>, возврат 0

–6

**–**

<**+**, *opSub*>, возврат 0

*е*

<**num**, *pnt*>,

возврат 1,

целый тип

*d*

–12

*d*

**.**

*d*

*d*

**⊥**

**⊥**

<**num**, *pnt*>,

возврат 1,

веществ. тип

–13

*д*

Рис. 1. Конечные автоматы-распознаватели для токенов:  
*а* – символы-разделители и скобки; *б* – «аддитивная операция»; *в* – «мультипликативная операция»;  
*г* – «двоеточие» и «присваивание»; *д* – «идентификатор»; *е* – «число»;  
*ж* – «пробел»; *з* – «комментарий»

**Примечание.** Обратите внимание на состояние 7. Переход из этого состояния возможен только после чтения символа «**}**». Если символа нет (пропущен правый ограничитель комментария), то это лексическая ошибка, которая обнаруживается только после завершения сканирования текста в состоянии 7. Поэтому при разработке алгоритма работы сканера необходимо учесть этот факт, т. е. необходимо реализовать переход из состояния 7 не по входному символу, а по достижению конца текста.

В общем случае после объединения автоматов в один автомат, полученный автомат может иметь неполную функцию переходов δ: *K* × *T* → *K*. Все элементы множества *K* × *T*, на которых функция переходов δ не определена (за исключением конечных состояний), соответствуют лексической ошибке. Таблица переходов объединенного автомата представлена в табл. 2.

Таблица 2. Таблица переходов объединенного конечного автомата

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| №  п/п | Входные  символы | Состояния автомата (за исключением конечных состояний) | | | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
|  | **;** | –1 | –9 | –11 | –12 | err(2) | –13 | –14 | 7 |
|  | **:** | 1 | –9 | –11 | –12 | err(2) | –13 | –14 | 7 |
|  | **,** | –2 | –9 | –11 | –12 | err(2) | –13 | –14 | 7 |
|  | **.** | err(1) | –9 | –11 | 4 | err(2) | –13 | –14 | 7 |
|  | **(** | –3 | –9 | –11 | –12 | err(2) | –13 | –14 | 7 |
|  | **)** | –4 | –9 | –11 | –12 | err(2) | –13 | –14 | 7 |
|  | **+** | –5 | –9 | –11 | –12 | err(2) | –13 | –14 | 7 |
|  | **–** | –6 | –9 | –11 | –12 | err(2) | –13 | –14 | 7 |
|  | **\*** | –7 | –9 | –11 | –12 | err(2) | –13 | –14 | 7 |
|  | **/** | –8 | –9 | –11 | –12 | err(2) | –13 | –14 | 7 |
|  | **=** | err(1) | –10 | –11 | –12 | err(2) | –13 | –14 | 7 |
|  | ***l*** | 2 | –9 | 2 | –12 | err(2) | –13 | –14 | 7 |
|  | ***d*** | 3 | –9 | 2 | 3 | 5 | 5 | –14 | 7 |
|  | **{** | 7 | –9 | –11 | –12 | err(2) | –13 | –14 | 7 |
|  | **}** | err(1) | –9 | –11 | –12 | err(2) | –13 | –14 | –15 |
|  | **sp** | 6 | –9 | –11 | –12 | err(2) | –13 | 6 | 7 |
|  | **tab** | 6 | –9 | –11 | –12 | err(2) | –13 | 6 | 7 |
|  | **lf** | 6 | –9 | –11 | –12 | err(2) | –13 | 6 | 7 |
|  | любой  другой | err(1) | –9 | –11 | –12 | err(2) | –13 | –14 | 7 |

В данной таблице строкам соответствуют входные символы, а столбцам – состояния автомата (конечные состояния не указаны, поскольку для них функция переходов не определена). В ячейке на пересечении строки и столбца записывается состояние, в которое должен перейти автомат, если в данном состоянии он считал данный входной символ. Автомат является частичным, поскольку на некоторых элементах множества *K* × *T* функция переходов не определена. Автомат доопределен новыми конечными состояниями, соответствующими определенному виду лексических ошибок, а именно:

err(1) – лексема не может начинаться с данного символа (состояние –16);

err(2) – неправильный формат числа (состояние –17);

err(3) – пропущен конец комментария (состояние –18, в таблице переходов не указано).

**Примечание**. Исходя из того, что областью определения функции переходов δ является множество *K* × *T*, логичнее, чтобы в таблице переходов строки соответствовали состояниям, а столбцы – входным символам (в большинстве публикаций так и делается). Однако для удобства представления таблицы (как в нашем случае) можно воспользоваться транспонированием матрицы, заменив строки на столбцы, а столбцы на строки.

# Структуры данных и алгоритмы

Исходный текст обрабатываемой грамматики представляет собой текстовый файл. Для облегчения реализации опережающего чтения и возврата символов во входной поток используется входной буфер, из которого лексический анализатор может выполнять чтение и в который может возвращать прочитанные символы путем простого перемещения указателя. Использование входного буфера повышает также эффективность анализатора, так как считывание блока символов обычно существенно более эффективно, чем посимвольное считывание. Используется схема буферизации, рекомендованная в [1; 7].

Буфер рассматривается как массив символов. Для распознавания лексемы требуется два указателя (индекса): *f* указывает на позицию первого символа лексемы, *r* перемещается в процессе распознавания по символам лексемы, начиная с позиции *f* (рис. 2). После того, как автомат, распознав лексему, переходит в конечное состояние, указатель *r* будет указывать на последний символ лексемы или на последний дополнительно прочитанный символ. В последнем случае дополнительно прочитанные символы возвращаются во входной поток соответствующим уменьшением значения указателя *r*. В итоге *r* будет указывать на последний символ лексемы. Для распознавания следующей лексемы указатель *f* устанавливается в позицию, непосредственно следующую за *r*.



Рис. 2. Использование указателей при распознавании лексемы

Для каждой лексемы сканер формирует выходной токен вида <*код\_токена*, *атрибут*> для последующих фаз компиляции. Код\_токена идентифицирует лексический класс лексемы и определяют работу синтаксического анализатора (рассматривается как терминал). Для удобства код\_токена будем представлять абстрактным именем (или специальным обозначением) и ссылаться на токен по его имени (обозначению). Атрибут токена обеспечивает доступ к дополнительной информации о лексеме, если лексическому классу соответствует множество лексем, и определяет трансляцию токена (семантический анализ и генерация промежуточного кода). Кодировка токенов и их обозначения реализуются в соответствии с табл. 1.

**Выбор структуры данных для представления таблицы символов.**

Поскольку ключевые слова удовлетворяют правилам образования идентификаторов, для определения, является данный идентификатор ключевым словом или нет, выполняется поиск лексемы в таблице ключевых слов. Никаких других операций для этой таблицы нет. Таким образом, таблица ключевых слов является статической, и для нее необходима только операция поиска. Элемент таблицы ключевых слов состоит из 2-х полей: *LexName* – лексема ключевого слова,  
*LexCode* – код лексического класса (код токена).

Для хранения чисел и идентификаторов необходимы динамические таблицы, поскольку в них выполняется поиск лексемы, и если поиск безуспешный, лексема добавляется в таблицу. В данной работе числа и идентификаторы будут храниться в одной таблице символов.

**Примечание**. На практике в ряде случаев имеет смысл числа и идентификаторы хранить в разных таблицах, если структуры этих таблиц отличаются. Можно рассмотреть следующую структуру таблицы чисел, в которой элемент таблицы состоит из полей *LexNum* – лексема числа, *TypeNum* – тип числа. Тип числа определяется в процессе лексического анализа. На последующих фазах компиляции при каждом обращении к числу будет выполняться преобразование символьного представления числа в его значение, что приведет к некоторому снижению эффективности компиляции. В таблицу чисел можно добавить поле для хранения значения числа. В этом случае выполняется единственное преобразование символьного представления числа в ее значение, а на последующих фазах компиляции это значение будет использоваться без всякого преобразования, что улучшит производительность. При этом надо иметь в виду следующее.

Если выбранная среда программирования не предусматривает некоторого универсального типа (например, тип Variant в Delphi), то придется для каждого типа предусмотреть собственное поле для хранения значения, что приведет к дополнительным затратам памяти.

Если же есть универсальный тип, то для хранения значения достаточно одного поля. В этом случае при каждом обращении к элементу универсального типа обычно происходит преобразование универсального типа в нужный тип. Насколько отличается эффективность такого преобразования от обычного преобразования символьного представления в соответствующий тип, требует дополнительного анализа среды программирования.

Другой вариант – использовать нетипизированные указатели. В этом случае также достаточно одного поля, которое является указателем на область памяти, где хранится значение числа. Это также приводит к дополнительным затратам памяти для хранения указателей.

В таблице символов наряду с лексемой могут храниться тип элемента, точность, размер, адрес памяти, число измерений и значения граничных пар (для массивов) и т.п., которые определяются на последующих фазах компиляции.

Идентификаторы могут представлять различные объекты. В языке *DemoLang* идентификатор может быть именем программы, именем типа или именем переменной. Поэтому используется понятие категории идентификатора. Примем следующие обозначения категорий: *catProgName* – имя программы, *catTypeName* – имя типа, *catVarName* – имя переменной. Поскольку в РГР предполагается, что идентификаторы и числа будут храниться в одной таблице символов, добавлена категория *catConst* – число. Добавлена также специальная категория *catNoCat* – отсутствие категории (устанавливается, когда категория еще неизвестна).

В языке *DemoLang* имеют место предопределенные идентификаторы для именования базовых типов (*integer*, *real*), которые не являются ключевыми словами. Эти идентификаторы заносятся в таблицу символов заранее с указанием категории, типа и размера. К базовым типам добавим также тип *void*, означающий отсутствие типа.

Таблица символов *TblIde* организована как односвязный список элементов (узлов) [3; 4] со следующей структурой узла:

*Lex* – лексема идентификатора (числа),

*Cat* – категория идентификатора (или число),

*Type* – тип идентификатора (числа),

*Width* – размер типа в байтах,

*Other* – другие поля, которые не используются лексическим анализатором, и могут быть легко детализированы в процессе разработки других фаз компиляции,

*NextEl* – указатель на следующий элемент таблицы символов.

**Примечание**. Поля *Cat* и *Type* можно объединить в одно поле, используя соответствующую кодировку для представления категорий как некоторые специальные типы.

Осталось определиться с хранением лексем. Хранить лексемы непосредственно в таблицах неэффективно, поскольку они обычно имеют разную длину, следовательно, придется устанавливать максимальную длину лексемы и размер поля таблицы выбирать из этой максимальной длины, что приведет к излишним затратам памяти. Поэтому в таблицах лучше хранить указатели на область памяти, где находится лексема.

Поле *Type*, в свою очередь, также состоит из полей:

*TypeCode* – код типа данных (в языке *DemoLang* это коды для *integer*, *real*, *void*),

*Width* – размер типа в байтах.

Целью такого представления поля *Type* является сокращение числа атрибутов при реализации синтаксически управляемой трансляции (вместо нескольких атрибутов будет один атрибут *type*). Более того, для разных типов могут понадобиться и другие сведения о типе. Для типа диапазон нужны значения *Low* (нижняя граница диапазона) и *High* (верхняя граница диапазона). Для типа массив надо знать число размерностей, данные о каждой размерности. Для типа запись понадобится ссылка на данные о полях записи и т. д. Представление всех этих сведений в качестве отдельных атрибутов существенно усложнит семантические правила.

**Выбор метода поиска в таблицах.**

После распознавания лексемы как токен «число» (**num**) или «идентификатор» (**id**) осуществляется ее поиск в таблице символов, если поиск безуспешный, лексема добавляется в таблицу. В результате возвращаемый поиском указатель на запись в таблице символов включается в токен в качестве значения атрибута. Поскольку таблица символов представляет собой связный список, возможен только последовательный поиск, алгоритм которого имеет следующий вид:

**function** *SearchLex* (*Lex*, *Cat*, *Type*)

**begin**

*p* **:=** *PntFirst* //первый элемент списка

**while** *p* ≠ **null** **do** //пока не дошли до конца списка

**begin**

**if** *Lex* **=** *p***.***Lex* **then** //найдено, выход из цикла

Break

*p* **:=** *p***.***NextEl* //следующий элемент списка

**end**

**if** *p* = **null** **then** //лексема не найдена, добавить элемент в табл. символов

**begin**

*AddElemTblIde*(*Lex*, *Cat*, *Type*) // добавить элемент в табл. символов

*SearchLex* **:=** *PntFirst* //добавление всегда в начало списка

**end** **else** *SearchLex* **:=** *p* //лексема уже есть в таблице

**end**

Функция *SearchLex* возвращает указатель на запись в таблице символов, соответствующую лексеме, задаваемой входным параметром *Lex*. Другими входными параметрами функции являются категория (*Cat*), тип (*Type*). Указатель *PntFirst* указывает на первый элемент списка.

**Примечание.** Как видите, в этой демо-работе использована не самая эффективная реализация поиска, поскольку порядок следования элементов в таблице произвольный, вследствие чего безуспешный поиск всегда приводит к просмотру всей таблицы. Если Вы для представления таблицы символов также будете использовать связный список, то придется применить последовательный поиск. Однако можно повысить его эффективность за счет улучшения безуспешного поиска. Для этого надо поддерживать в списке алфавитный порядок следования лексем, т. е. при добавлении нового элемента следует найти место вставки и в соответствующую позицию реализовать добавление (все это мы изучали в САОД [3; 4]).

Для еще более эффективной реализации можно применить поиск с хешированием. Самой простой реализацией является открытое хеширование, эквивалентное комбинации разных методов поиска: хеширование используется на первом шаге, а другие методы используются для поиска среди лексем с одинаковыми хеш-значениями. Как минимум, следует в хеш-таблице хранить указатели на связные списки упорядоченных по лексемам элементов, образующих коллизии.

Процедура *AddElemTblIde* добавляет новый элемент в начало списка в соответствии с очевидным алгоритмом:

**procedure** *AddElemTblIde*(*Lex*, *Cat*, *Type*, *Width*)

**begin**

*New*(*p*) //новый элемент списка

*p***.***Lex* **:=** *Lex*

*p***.***Cat* **:=** *Cat*

*p***.***Type* **:=** *Type*

*p***.***NextEl* **:=** *PntFirst*

*PntFirst* **:=** *p*

**end**

Для таблицы ключевых слов, поскольку она статическая и упорядочена в алфавитном порядке, используется бинарный поиск.

**Примечание.** Здесь не приводится алгоритм бинарного поиска специально, а то Вам будет нечего делать. По заданию Вы должны обосновать применение и применить бинарный поиск для таблицы ключевых слов и поиск с хешированием или поиск в упорядоченном связном списке для таблицы символов.

**Выбор метода реализации конечного автомата**.

Рассмотрим два очевидных метода реализации автомата. Первый метод заключается в явном хранении таблицы переходов как двумерной таблицы, проиндексированной состояниями и символами. Она используется для реализации перехода в состояние, определяемое текущим состоянием и очередным входным символом. Достоинство этого метода заключается в том, что получается очень маленький программный код. Недостаток тоже очевиден – большие затраты памяти для хранения таблицы переходов. Для больших таблиц и жестких требований к объему памяти можно использовать методы сжатия таблиц переходов (один из таких методов рассмотрен в [1]). Второй метод предполагает использование операторов выбора вида **switch** (**case**): внешний оператор **switch** реализует выбор состояний автомата, для каждого состояния требуется оператор **switch** для выбора входного символа. Основной недостаток метода – относительно большой программный код, который к тому же зависит от функций переходов автомата – любое изменение функции переходов приводит к необходимости изменения программного кода. Исходя из изложенного, выбор осуществляется в пользу первого метода реализации автомата. Алгоритм работы автомата имеет следующий вид:

*S* **:=** 0 //перевод автомата в начальное состояние

**while** *S* ∉ *F* **do** //пока не достигли конечного состояния

**begin**

*Sym* **:=** *NextSym* //чтение очередного входного символа

*S* **:=** *JumpTbl* [*S*, *Sym*] //переход в следующее состояние

**end**

**if** *S* ∈ *ES* //состояние ошибки?

**then** *Lex\_Error* //лексическая ошибка

**else** *Lex\_Accept* //лексема распознана и принята

В алгоритме используются следующие обозначения: *S* –состояние автомата, *Sym* – очередной прочитанный символ, функция *NextSym* возвращает очередной символ из входной строки (в соответствии с рис. 2 это символ, на который указывает *r*), *F* – множество конечных состояний (включая и состояния ошибок), *JumpTbl* – таблица переходов, *ES* – множество состояний лексической ошибки, *Lex\_Error* – процедура формирования сообщения о лексической ошибке, процедура *Lex\_Accept* фиксирует распознавание лексемы, формирует токен, при необходимости возвращает дополнительно прочитанные символы во входной поток, устанавливает указатель *f* (см. рис. 2) на начало следующей лексемы.

**Примечание**. В автомат необходимо включить пропуск пробелов и комментариев, чтобы не передавать токен «пробел» или «комментарий» синтаксическому анализатору. Для этого следует добавить внешний цикл, выход из которого реализуется только в том случае, если достигнутое автоматом конечное состояние не является конечным состоянием для токена «пробел» или «комментарий». Студент должен модифицировать алгоритм самостоятельно и включить в записку модифицированный алгоритм.

Не забудьте также про необходимость реализации перехода из внутреннего состояния токена «комментарий» (в нашем автомате это состояние 7) не по входному символу, а по достижению конца текста для формирования соответствующего сообщения о лексической ошибке.

**Перечень лексических ошибок**.

Лексические ошибки, обнаруживаемые сканером, и их коды представлены в табл. 3.

Таблица 3. Перечень лексических ошибок и их кодов

|  |  |
| --- | --- |
| Код ошибки | Название ошибки |
| LexErr\_1 | Лексема не может начинаться с данного символа |
| LexErr\_2 | Неправильный формат числа |
| LexErr\_3 | Пропущен конец комментария |

# Программная реализация лексического анализа

***Выбор среды разработки***

В качестве среды разработки и языка программирования выбрана система программирования Delphi 7. Этот выбор сделан исходя из следующих соображений.

Прежде всего Delphi предназначен для профессиональных разработчиков, желающих очень быстро разрабатывать приложения. Delphi производит небольшие по размерам высокоэффективные исполняемые модули (.exe и .dll). С другой стороны небольшие по размерам и быстро исполняемые модули означают, что требования к клиентским рабочим местам существенно снижаются.

Преимущества Delphi по сравнению с аналогичными программными продуктами:

* быстрота разработки приложения (RAD – Rapid Application Development);
* высокая производительность разработанного приложения;
* низкие требования разработанного приложения к ресурсам компьютера;
* наращиваемость за счет встраивания новых компонент и инструментов в среду Delphi;
* возможность разработки новых компонентов и инструментов собственными средствами Delphi (существующие компоненты и инструменты доступны в исходных кодах);
* удачная проработка иерархии объектов.

Основным конкурентом Borland Delphi 7 является RAD-среда Borland C++ Builder, технология работы с которой полностью совпадает с технологией, принятой в Delphi 7. Только в Delphi программный код пишется на языке программирования Паскаль, точнее на его объектно-ориентированной версии ObjectPascal (с версии 7 язык называется Delphi), а не на языке C++.

Можно перечислить некоторые недостатки языка С++ по сравнению с Delphi:

1. Надо делать много инициализации (регистрировать класс окна, организовывать цикл обработки сообщений, создавать оконную функцию, пиктограмму и прочее) и частично быть системным программистом. На Delphi-же системное программирование уже встроено и инициализация работает по умолчанию, поэтому программист главный упор делает на своих алгоритмах, а не на организации вспомогательных работ.

2. Значительно большая, по сравнению с Delphi, сложность языка, даже, несмотря на компактность кода, возникают сложности в его восприятии.

3. Одна особенность, на взгляд некоторых программистов, языка С++ очень портит этот язык, а именно: он чувствителен к регистру символов, т. е. переменная *A* и переменная *a* – это разные переменные.

4. В Delphi классы (объекты) могут располагаться только в динамической памяти, а в C++ в любой памяти (статическая, стек, динамическая). Это добавляет безопасности программирования в Delphi.

На выбор среды повлияло также то, что имеется опыт разработки других приложений в среде Delphi, а также отсутствие у меня других систем программирования (включая и более поздние версии Delphi).

**Примечание**. Одним из пунктов обоснования выбора можно включить необходимость (или желание) получения опыта разработки в среде и более глубокого изучения ее возможностей в связи с предполагаемым практическим применением на будущем месте работы.

В данном разделе для простоты используется концепция процедурного программирования. Студенту же рекомендуется вести разработку программы в парадигме ООП.

***Глобальные структуры данных***

Чтобы сделать исходный код программы более ясным и читабельным, для представления различных значений именами (вместо применения безличных числовых кодов) использованы перечислимые типы.

*Лексические классы* (*коды токенов*):

tLexCode = (lcProg, lcStart, lcStop, lcVar, lcSemi, lcColon, lcComma, lcOpPar, lcClPar, lcAss,

lcAdd, lcMult, lcId, lcNum, lcErr, lcEof).

*Категории идентификаторов* *или число*:

tCatId = (catNoCat, catProgName, catTypeName, catVarName, catConst).

*Коды* *типов данных*:

tTypesCode = (typeVoid, typeInt, typeReal).

*Операции для лексических классов lcAdd*, *lcMult*:

tOperCode = (opAdd, opSub, opMult, opDiv).

*Размеры предопределенных типов* представлены именованными константами:

wInt = 4; //целый

wReal = 8; //вещественный

*Элемент типа данных*

tType = **record**

TypeCode: tTypesCode; //код типа данных

Width: integer; //размер типа

**end**

*Таблица символов* TblIde представляет собой односвязный список элементов со следующей структурой узла (таблица задается указателем на первый элемент списка):

tElemTblIde = **record**

Lex: string; //лексема идентификатора или числа

Cat: tCatId; //категория идентификатора или число

Typ: tType; //тип (**type** – ключевое слово, поэтому поле названо Typ)

NextEl: tPntElTblIde; //след. элемент таблицы символов

**end**

Вместо отдельной таблицы ключевых слов реализована таблица лексических классов, в которой позиции ключевых слов известны, т. е. таблица ключевых слов является подтаблицей таблицы лексических классов. Это решение связано с тем, что на следующих фазах при синтаксически управляемой трансляции понадобится сопоставлять лексические классы с соответствующими элементами таблиц разбора.

*Таблица лексических классов* LexClass представляет собой массив-константу

LexClass: **array** [tLexCode] **of** tElLexClass, элемент массива определяется как

tElLexClass = **record**

LexName: string; //обозначение лексического класса

LexCode: tLexCode; //код лексического класса (токена)

**end**

Как видно из типа tLexCode ключевыми словами являются первые 4 элементов массива.

*Токен*:

tToken = **record**

Code: tLexCode; //код токена (лексического класса)

Attr: Pointer; //атрибут токена (нетипизированный указатель)

Name: string; //обозначение токена (для визуализации)

LexBeg, LexEnd: Word; //для локализации ошибок

**end**

Для синтаксически управляемой трансляции необходимы только поля Code и Attr. Поле Name добавлено для визуализации обозначения токена при демонстрации работы сканера. Поля LexBeg и LexEnd указывают позиции в текстовом файле начала и конца лексемы соответственно, предназначены для визуального выделения ошибочной лексемы в исходном тексте транслируемой программы.

***Глобальные процедуры и функции***

**procedure** AddElemTblIde (Lex: string; Cat: tCatId; Typ: tTypes; Width: Byte) добавляет элемент в таблицу символов.

**procedure** AddBaseTypes добавляет в таблицу символов предопределенные базовые типы.

**function** Scaner: tToken – реализация сканера (цель данной части РГР), выходным параметром является формируемый токен.

***Локальные для сканера структуры данных***

Чтобы не привязываться к внутренней кодировке символов для букв, цифр и символов, которые не рассматриваются индивидуально, соответствующие элементы представлены как константы-множества:

Letters = ['A' .. 'Z', 'А' .. 'Я', 'Ё']

Digits = ['0' .. '9']

остальные символы алфавита рассматриваются как символьные константы типа char. В множество Letters включены только прописные буквы. Поскольку прописные и строчные буквы не должны различаться, перед сравнением букв и строк производится их преобразование в прописные (в языке программирования есть соответствующая функция). Из множества Letters следует, что разрешены идентификаторы, включающие в себя буквы кириллицы.

Таблица переходов автомата задается константой-массивом

JumpTbl: **array** [1 .. 19, 0 .. 7] **of** integer,

где строки 1 .. 19 соответствуют входным символам, а столбцы 0 .. 7 – внутренним состояниям автомата (см. табл. 2).

Исходный текст тестовой программы размещается в визуальном компоненте типа TRichEdit в свойстве Lines. Данный компонент позволяет набирать текст исходной программы, загружать программу из текстового файла, сохранять набранный (отредактированный) текст как текстовый файл (методы LoadFromFile и SaveToFile). У этого компонента имеется свойство Text, в котором исходный многострочный текст представляется как одна строка, что делает его удобным для использования в качестве буфера ввода.

***Локальные для сканера процедуры и функции***

**function** NumRowOfJT (Sym: char): Word возвращает номер строки в таблице переходов, соответствующей входному символу Sym.

**function** SearchLex(Lex: string; Cat: tCatId; Typ: tTypes; Width: Byte): tPntElTblIde реализует поиск лексемы Lex в таблице символов (в случае безуспешного поиска – добавляет новую запись в таблицу символов) и возвращает указатель на соответствующую запись.

**function** SearchLexKey(Lex: string): tLexCode реализует поиск лексемы Lex ключевого слова и возвращает соответствующий код лексического класса или *lcErr* в случае безуспешного поиска.

Алгоритм работы автомата есть не что иное, как алгоритм лексического анализа, и реализован рассмотренной выше функцией Scaner. Специальное хранение множества конечных состояний не требуется, поскольку они закодированы отрицательными числами. Поэтому условие *S* ∉ *F* в алгоритме реализуется условием *S* >= 0. После распознавания лексемы или возникновения ошибки (т. е. после выхода из цикла) с помощью инструкции **case** реализуется распознавание всех конечных состояний и для каждого конечного состояния реализуется свой программный код для формирования токена или сообщения об ошибке.

***Инструкция пользователя и результаты тестирования и отладки****.*

Разработанное приложение работает под ОС Windows, не требует какой-либо специальной установки и представляет собой исполняемый файл **SUT.exe**. После запуска приложения формируется подпапка **TESTS,** которая является папкой по умолчанию для загрузки и сохранения тестовых программ как текстовых файлов.

Интерфейс и результаты работы приложения на тестовом примере представлены на рис. 3. Группа элементов «Текст тестовой программы» включает в себя область ввода и редактирования текста программы и кнопки для загрузки и сохранения теста программы в текстовый файл. Нажатие кнопок откроет стандартное окно диалога для выбора папок и имен файлов. В статусной строке группы показывается позиция курсора в области редактирования (номер строки и позиция в строке) и информация о том, были ли внесены в грамматику изменения.

Для запуска лексического анализа служит кнопка «Сканер». Результаты сканирования отражаются в элементе «Сформированные токены», а в элементе «Обнаруженные ошибки» формируется сообщение об успехе или неуспехе сканирования. Если при сканировании обнаружена ошибка, в элементе «Обнаруженные ошибки» отображается сообщение об ошибке и ее код. Сканер прекращает работу сразу же после первой обнаруженной ошибки.

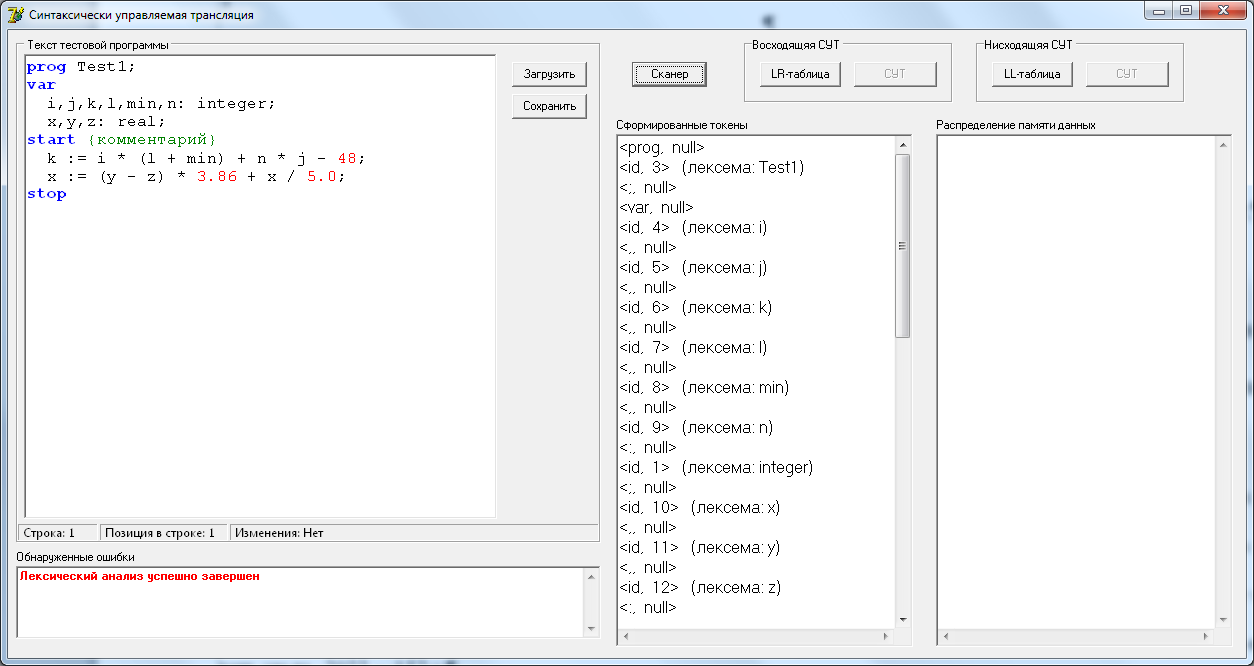


Рис. . Интерфейс приложения

Заключение

В процессе выполнения расчетно-графической работы

* разработан язык программирования *DemoLang*, синтаксис описан с помощью РБНФ;
* изучены методы построения лексических анализаторов;
* изучены способы построения распознавателей для регулярных языков;
* разработаны структуры данных для представления таблиц сканера;
* получены практические навыки синтеза, детерминизации и минимизации распознающих конечных автоматов и их программной реализации;
* получены практические навыки разработки языка программирования и модуля лексического анализа.

Список использованной литературы

1. *Ахо, А*. Компиляторы: принципы, технологии и инструментарий / А. Ахо, М. Лам, Р. Сети, Д. Ульман. – 2-е изд.– М.: Вильямс, 2008. – 1184 с.

2. *Павлов, Л.А*. Лексический и синтаксический анализ: учеб. пособие / Л.А. Павлов. – Чебоксары: Изд-во Чуваш. ун-та, 2022. – 152 с.

3. *Павлов, Л.А*. Структуры и алгоритмы обработки данных: учеб. пособие / Л.А. Павлов, Н.В. Первова. – 2-е  
изд. – Чебоксары: Изд-во Чуваш. ун-та, 2018. – 254 с.

4. *Павлов, Л.А*. Структуры и алгоритмы обработки данных: учебник / Л.А. Павлов, Н.В. Первова. – 3-е изд. – СПб.: Лань, 2021. – 256 с.

5. *Свердлов*, *С.З*. Языки программирования и методы трансляции: Учебное пособие / С.З. Свердлов.– СПб.: Питер, 2007.– 638 с.

6. *Свердлов*, *С.З*. Языки программирования и методы трансляции: учеб. пособие / С.З. Свердлов. – 2-е изд. – СПб.: Лань, 2019.– 564 с.

7. Теория языков программирования и методы трансляции: Метод. указания к выполнению расчетно-графических работ / Сост. Л.А. Павлов. – Чебоксары: Изд-во Чуваш. ун-та, 2015. – 48 с.