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 №1

Лексический анализ

Вариант 1 5

Выполнил:

студент группы ИВТ-41-21

Иванов В.С.

Руководитель:

доцент Павлов Л. А.

Чебоксары, 2025г.

Оглавление

[Задание к РГР 3](#_Toc133472392)

[Введение 4](#_Toc133472393)

[1. Описание языка 5](#_Toc133472394)

[2. Лексические классы языка 7](#_Toc133472395)

[3. Синтез конечных автоматов-распознавателей 8](#_Toc133472396)

[4. Структуры данных и алгоритмы 10](#_Toc133472397)

[5. Программная реализация лексического анализа 13](#_Toc133472398)

[Заключение 14](#_Toc133472399)

[Список использованной литературы 15](#_Toc133472400)

# Задание к РГР

1. Разработать синтаксис простого языка программирования, используя нотацию 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.
* производный тип – массив;
* цикл с постусловием repeat

# 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 ЛА обычно приходится решать следующие задачи:

* определить формальную грамматику заданного языка;
* выделить конструкции и элементы заданного языка (лексемы), обрабатываемые на этапе лексического анализа, и определять 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 *Program*) 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 "=", 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.

Ниже приведено описание синтаксиса языка *MyC* в расширенной форме Бэкуса-Наура (РБНФ).

1. Модуль = "Program" Идент ";" Блок "endProg".
2. Идент = Буква { Буква | Цифpа }.
3. Блок = { СписокОписаний } "begin" ПоследОператоров "end".
4. СписокОписаний = Описание { ";" Описание } ";".
5. Описание = Тип СписокИдент.
6. СписокИдент = Идент { "," Идент }.
7. Тип = ПростойТип | ТипМассив.
8. ПростойТип = Идент.
9. ТипМассив = ПростойТип "[" Целое "]".
10. ПоследОператоров = Оператор { ";" Оператор }.
11. Оператор = Присваивание | Цикл | Условие.
12. Присваивание = Переменная "=" Выражение.
13. Переменная = Идент | ИндексПеременная.
14. ИндексПеременная = Идент "[" Простое выражение "]"
15. Repeat = "repeat" ПоследОператоров "until" Выражение.
16. Выражение = ПростоеВыраж [ Отношение ПростоеВыраж ].
17. Отношение = "<" | "<=" | ">" | ">=" | "==" | "!=".
18. ПростоеВыраж = Терм { АддитОперация Терм }.
19. АддитОперация = "+" | "–" | "||".
20. Терм = Фактор { МультОперация Фактор }.
21. МультОперация = "\*" | "/" | "&&".
22. Фактор = Константа | Пеpеменная | "(" Выражение ")" | "not" Фактор.
23. Константа = ЧисловаяКонст.
24. ЧисловаяКонст = Целое [ "." Целое ] ["e" [ ("+" | "–" ) ] Целое ].
25. Целое = Цифpа { Цифpа }.
26. 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).

**Краткая характеристика языка и семантические соглашения:**

Язык представляет собой описание некоторой схемы программы, отражающей структуру программы, но не ее семантику вычислений.

Язык удовлетворяет семантическим соглашениям, характерным для многих языков программирования (единственность именования различных объектов программы, необходимость описания идентификатора до его использования и т.п.).

Идентификаторы предопределенных типов: Integer, Real, Boolean, String.

Ключевые слова языка зарезервированы, их нельзя использовать в качестве идентификаторов.

Идентификатор (конструкция «Идент») в правиле 1 – имя схемы, которое не должно встречаться ни в какой другой конструкции языка.

Комментарий представляет собой любую последовательность символов, заключенную в "/\*" и "\*/" или после "//" в пределах одной строки.

*Пример схемы программы на языке MyC:*

**Program Test;**

**//a**

**real x, a, b, c;**

**real [3] array;**

**integer i;**

**string s;**

**bool m;**

**begin**

**x = 1.5;**

**c = 1.0;**

**s = "aaaa";**

**m = False;**

**i = 0;**

**repeat**

**array[i] = c / (i + 1) \* x + i;**

**i = i + 1;**

**until (i > 2 || i >= 100);**

**end**

**endProg**

# Лексические классы языка

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

Таблица 1. Лексические классы языка *MyC*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Код | Токен | Обозначение | Формальный шаблон | Значение атрибута |
|  | Program | Program | *Program* | 0 |
|  | endProg | endProg | *endProg* | 0 |
|  | begin | begin | *begin* | 0 |
|  | end | end | *end* | 0 |
|  | repeat | repeat | repeat | 0 |
|  | until | until | until | 0 |
|  | not | not | *not* | 0 |
|  | присваивание | = | *=* | 0 |
|  | точка с запятой | ; | *;* | 0 |
|  | запятая | , | *,* | 0 |
|  | открывающая квадратная скобка | [ | *[* | 0 |
|  | закрывающая квадратная скобка | ] | *]* | 0 |
|  | открывающая скобка | ( | *(* | 0 |
|  | закрывающая скобка | ) | *)* | 0 |
|  | точка | . | *.* | 0 |
|  | аддитивная операция | add | + | 0 |
| - | 1 |
| || | 2 |
|  | мультипликативная операция | mul | \* | 0 |
| / | 1 |
| % | 2 |
| && | 3 |
|  | операция  отношения | rel | < | 0 |
| <= | 1 |
| > | 2 |
| >= | 3 |
| != | 4 |
| == | 5 |
|  | идентификатор | id | *l* (*l* | *d*)\*  *l* – буква, *d* – цифра | указатель на  запись в таблице |
|  | число | num | *dd\**(1*|***.***dd\**2|(*ε|.dd\**)*e*(+*|*–*|ε*)*dd\**3 | указатель на  запись в таблице |
|  | строка | str | "\*" | указатель на  запись в таблице |
|  | пробел | sp | (*sp* | *tab* | *lf* )+  *sp* – пробел, *tab* – табуляция,  *lf* – перевод строки | токен не формируется |
|  | комментарий | com | (‘//’ 1\* *lf* | ‘/\*’ 2\* ‘\*/’)  ****2 – любой символ кроме ‘\*/’ | токен не формируется |

Поскольку в формальной грамматике для последующих фаз компиляции каждый лексический класс (токен) будет представлять собой терминал, для более компактной записи продукций для обозначения токена-терминала введено понятие имени (обозначения) токена. Первые 10 строк соответствуют ключевым словам.

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.

1. **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 в алфавитном порядке. При необходимости в структуру таблицы можно добавить поле для хранения имени (обозначения) соответствующего токена.

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*, , *k*0, *F*), где *K* – конечное множество состояний, *Т* – конечный входной алфавит, : *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.

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 *MyC*, 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 -21 автомата для токена **id** (рис. 1, *в*) определяется, является данный идентификатор ключевым словом (если является, формируются соответствующие токен и значение атрибута из таблицы ключевых слов) или нет (в этом случае формируется токен **id** с соответствующим значением атрибута). Для токенов **id**, **num** и **str** значениями атрибутов являются номера соответствующих строк таблицы символов (обозначение *ns*).

Аддитивные и мультипликативные операции и операции отношения объединяются в лексические классы (*add*, *mul* и *rel*), в качестве значений атрибутов указываются коды соответствующих операций, определяемые в таблице лексических классов.

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.

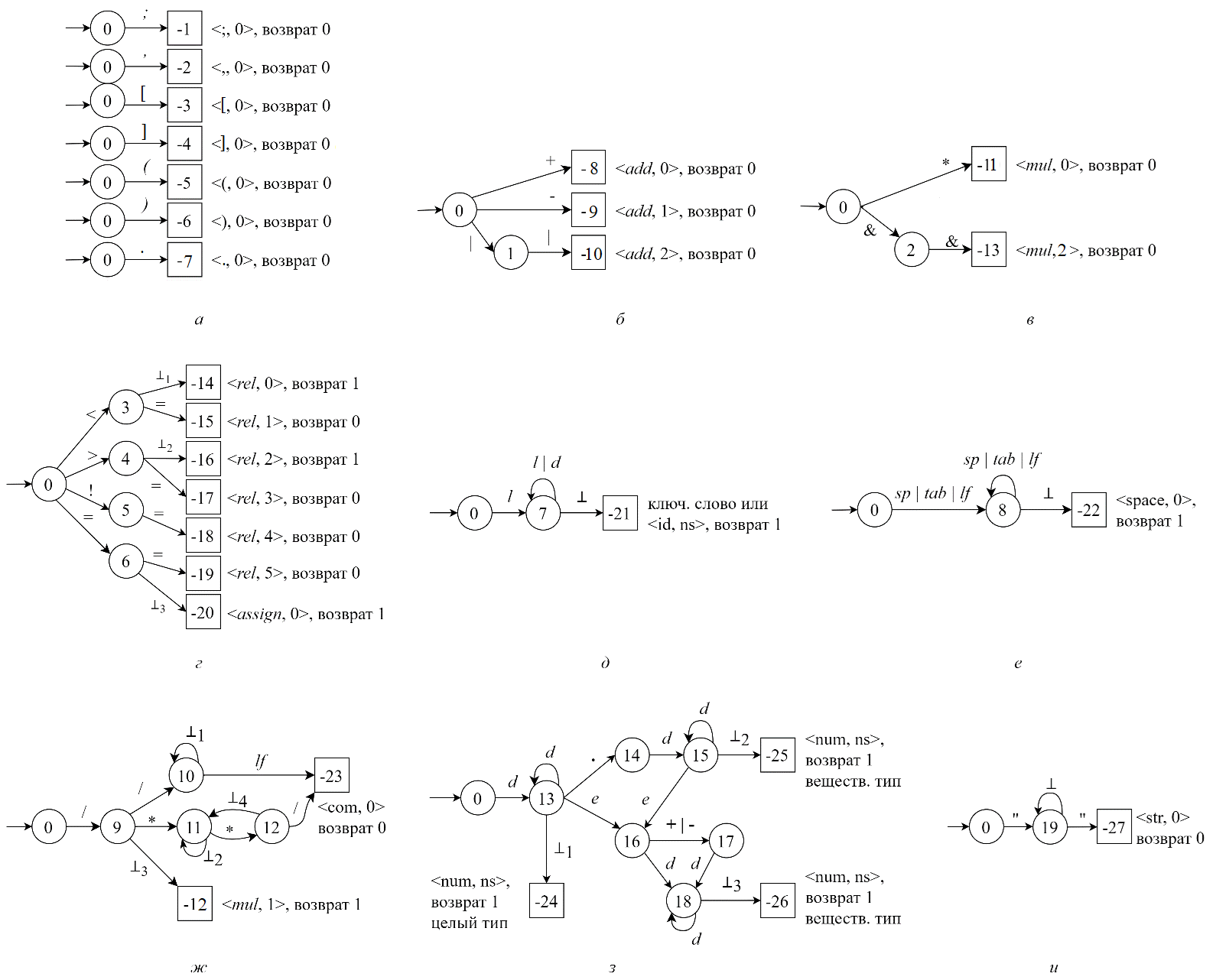


Рис. 1. Конечные автоматы-распознаватели для токенов:

а – символы-разделители и скобки; б – аддитивные операции; в – мультипликативные операции;

г – операции отношения и «присваивание; д – ключ. слово или «идентификатор»; е – «пробел»;

ж – «комментарий»; з – «число»; и – «строка»

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

Таблица 2. Таблица переходов объединенного конечного автомата

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Входные  символы | Состояния автомата | | | | | | | | | | | | | | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| ; | -1 | Err | Err | -14 | -16 | Err | -20 | -21 | -22 | -12 | 10 | 11 | 11 | -24 | Err | -25 | Err | Err | -26 | 19 |
| , | -2 | Err | Err | -14 | -16 | Err | -20 | -21 | -22 | -12 | 10 | 11 | 11 | -24 | Err | -25 | Err | Err | -26 | 19 |
| [ | -3 | Err | Err | -14 | -16 | Err | -20 | -21 | -22 | -12 | 10 | 11 | 11 | -24 | Err | -25 | Err | Err | -26 | 19 |
| ] | -4 | Err | Err | -14 | -16 | Err | -20 | -21 | -22 | -12 | 10 | 11 | 11 | -24 | Err | -25 | Err | Err | -26 | 19 |
| ( | -5 | Err | Err | -14 | -16 | Err | -20 | -21 | -22 | -12 | 10 | 11 | 11 | -24 | Err | -25 | Err | Err | -26 | 19 |
| ) | -6 | Err | Err | -14 | -16 | Err | -20 | -21 | -22 | -12 | 10 | 11 | 11 | -24 | Err | -25 | Err | Err | -26 | 19 |
| . | -7 | Err | Err | -14 | -16 | Err | -20 | -21 | -22 | -12 | 10 | 11 | 11 | 14 | Err | -25 | Err | Err | -26 | 19 |
| + | -8 | Err | Err | -14 | -16 | Err | -20 | -21 | -22 | -12 | 10 | 11 | 11 | -24 | Err | -25 | 17 | Err | -26 | 19 |
| *-* | -9 | Err | Err | -14 | -16 | Err | -20 | -21 | -22 | -12 | 10 | 11 | 11 | -24 | Err | -25 | 17 | Err | -26 | 19 |
| *|* | 1 | -10 | Err | -14 | -16 | Err | -20 | -21 | -22 | -12 | 10 | 11 | 11 | -24 | Err | -25 | Err | Err | -26 | 19 |
| \* | -11 | Err | Err | -14 | -16 | Err | -20 | -21 | -22 | 12 | 10 | 12 | 11 | -24 | Err | -25 | Err | Err | -26 | 19 |
| / | 9 | Err | Err | -14 | -16 | Err | -20 | -21 | -22 | 10 | 10 | 11 | -23 | -24 | Err | -25 | Err | Err | -26 | 19 |
| & | 2 | Err | -13 | -14 | -16 | Err | -20 | -21 | -22 | -12 | 10 | 11 | 11 | -24 | Err | -25 | Err | Err | -26 | 19 |
| < | 3 | Err | Err | -14 | -16 | Err | -20 | -21 | -22 | -12 | 10 | 11 | 11 | -24 | Err | -25 | Err | Err | -26 | 19 |
| > | 4 | Err | Err | -14 | -16 | Err | -20 | -21 | -22 | -12 | 10 | 11 | 11 | -24 | Err | -25 | Err | Err | -26 | 19 |
| ! | 5 | Err | Err | -14 | -16 | Err | -20 | -21 | -22 | -12 | 10 | 11 | 11 | -24 | Err | -25 | Err | Err | -26 | 19 |
| = | 6 | Err | Err | -15 | -17 | -18 | -19 | -21 | -22 | -12 | 10 | 11 | 11 | -24 | Err | -25 | Err | Err | -26 | 19 |
| *l* | 7 | Err | Err | -14 | -16 | Err | -20 | 7 | -22 | -12 | 10 | 11 | 11 | -24 | Err | -25 | Err | Err | -26 | 19 |
| *d* | 13 | Err | Err | -14 | -16 | Err | -20 | 7 | -22 | -12 | 10 | 11 | 11 | 13 | 15 | 15 | 18 | 18 | 18 | 19 |
| *space* | 8 | Err | Err | -14 | -16 | Err | -20 | -21 | 8 | -12 | 10 | 11 | 11 | -24 | Err | -25 | Err | Err | -26 | 19 |
| *tab* | 8 | Err | Err | -14 | -16 | Err | -20 | -21 | 8 | -12 | 10 | 11 | 11 | -24 | Err | -25 | Err | Err | -26 | 19 |
| *line feed* | 8 | Err | Err | -14 | -16 | Err | -20 | -21 | 8 | -12 | -23 | 11 | 11 | -24 | Err | -25 | Err | Err | -26 | 19 |
| *"* | 19 | Err | Err | -14 | -16 | Err | -20 | -21 | -22 | -12 | 10 | 11 | 11 | -24 | Err | -25 | Err | Err | -26 | -27 |
| *e* | 7 | Err | Err | -14 | -16 | Err | -20 | -21 | -22 | -12 | 10 | 11 | 11 | 16 | Err | 16 | Err | Err | -26 | 19 |
| *др. симв.* | Err | Err | Err | -14 | -16 | Err | -20 | -21 | -22 | -12 | 10 | 11 | 11 | -24 | Err | -25 | Err | Err | -26 | 19 |

В данной таблице строкам соответствуют допустимые 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 состояниеми, соответствующим определенному виду лексических ошибок. Коды этих ошибок представлены в табл. 3.

1. **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.

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. Позиция (индекс) ключевого слова в таблице соответствует коду его токена. Элемент таблицы ключевых слов состоит из 2-х полей: *lex* – лексема ключевого слова, *name* – обозначение токена (нужно только для отображения результатов).

Для хранения идентификаторов, чисел и строк необходимы динамические таблицы, поскольку в них выполняется поиск лексемы, и если поиск безуспешный, лексема добавляется в таблицу. Структуры этих таблиц различны.

Для числовых констант и строк, используется динамическая таблица со следующей структурой:

Элемены таблицы состоят из полей *lex* – лексема константы, *type* – тип константы. Тип числовой константы определяется в процессе лексического анализа. Cимвольное представление константы преобразуется в ее значение, а на последующих фазах компиляции это значение будет использоваться без всякого преобразования.

В таблице идентификаторов наряду с лексемой хранятся тип идентификатора, точность, длина, адрес памяти, число измерений и т.п., которые определяются на последующих фазах компиляции. Идентификаторы могут представлять различные объекты. В языке *MyC* идентификатор может быть именем программы, именем типа, именем переменной. Поэтому используется понятие категории идентификатора, для ее хранения предусмотрено специальное поле. Примем следующую кодировку категорий: 1 – имя программы, 2 – имя типа, 3 – имя переменной. В языке *MyC* имеют место предопределенные идентификаторы имен типов (Integer, Float, Boolean, String), которые не являются ключевыми словами. Эти идентификаторы заносятся в таблицу идентификаторов заранее с указанием категории и типа.

Таким образом, можно предложить следующую структуру таблицы идентификаторов:

* l*ex* – лексема идентификатора,
* *cat* – категория идентификатора,
* *other* – другие поля, которые не используются лексическим анализатором, и могут быть легко детализированы в процессе разработки других фаз компиляции.

**Выбор метода поиска в таблицах.**

После распознавания лексемы как токен «метка» или «идентификатор» осуществляется ее поиск в соответствующей таблице, если поиск безуспешный, лексема добавляется в таблицу. В результате возвращаемый поиском номер строки таблицы включается в токен в качестве значения атрибута. Поскольку эти таблицы относительно небольшие, можно ограничиться рассмотрением бинарного и последовательного методов поиска. Бинарный поиск требует упорядоченности таблиц. При вставке в таблицу лексемы в соответствии с принятым порядком номера строк лексем, следующих за вставленной лексемой, изменятся (увеличатся на единицу). Это недопустимо, поскольку старые номера уже использовались в качестве значений атрибутов ранее распознанных токенов. Можно, конечно, применить простейшую технологию индексации поиска, что потребует наряду с основной неупорядоченной таблицей создание дополнительной индексной таблицы, в которой лексемы упорядочены и имеется дополнительное поле для номера строки, соответствующей лексеме в основной таблице. Очевидно, что потребуются дополнительные затраты памяти, при этом для небольших таблиц существенного повышения эффективности поиска не следует ожидать. Поэтому в качестве метода поиска выбирается последовательный поиск, алгоритм которого имеет следующий вид:

**function** *SearchLex*(*Lex*)

**begin**

*Tbl* [*n* + 1] := *Lex* //запись лексемы в конец таблицы

*i* := 1

**while** *Lex* <> *Tbl* [*i*] **do** *i* := *i* + 1 //поиск лексемы

**if** *i* > *n* **then** //лексема не найдена

**begin**

*n* := *n* + 1 //увеличение текущего размера таблицы

*SearchLex* := *n*

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

**end**

Функция *SearchLex* возвращает номер строки в таблице, соответствующей лексеме, задаваемой входным параметром *Lex*. Предполагается, что поиск выполняется в таблице *Tbl* с текущим размером *n* (число содержащихся в таблице лексем). Для простоты таблица *Tbl* рассматривается как массив лексем (при программной реализации следует учесть, что таблица представляет собой массив указателей на лексемы). В самом начале поиска производится запись лексемы в конец таблицы, что позволяет, с одной стороны, использовать этот элемент в качестве «сторожа» («часового») при поиске, а с другой стороны, если поиск безуспешен, лексема уже оказывается добавленной в таблицу и достаточно только скорректировать размер таблицы.

**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 инструкций типа **switch**-**case**: внешняя инструкция **switch** реализует выбор стояний автомата, для каждого состояния требуется инструкция **case** 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

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

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

**begin**

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

**if** *S*  *F* **then**

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

**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), *TP* – 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.

**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 | Ожидается символ '|' |
| LexErr\_3 | Ожидается символ '&' |
| LexErr\_4 | Ожидается символ '=' |
| LexErr\_5 | Ожидается число |
| LexErr\_6 | Ожидается число, '+' или '-' |

1. **Программная реализация лексического анализа**

В качестве языка программирования выбран язык JacaScript. Этот выбор сделан исходя из следующих соображений.

* Динамическая типизация: JavaScript позволяет изменять тип данных переменной на лету, что делает процесс разработки более гибким.
* Простота: JavaScript имеет простой синтаксис
* Кроссплатформенность: программу можно запустить и отлаживать в любом браузере на любых операционных системах

**Инструкция пользователя и результаты тестирования и отладки**

Разработанное приложение работает в любых операционных системах. Для его запуска необходимо открыть файл index.html в браузере.

Интерфейс и результаты работы приложения на тестовом примере представлены на рис. 3.

В верхней части расположены кнопки для загрузки файла и его сохранения.

Для запуска лексического анализа служит кнопка «Сканер». Результаты сканирования отражаются в правом текстовом редакторе. Если при сканировании обнаружена ошибка, там же отображается сообщение об ошибке и ее код. Сканер прекращает работу сразу же после первой обнаруженной ошибки.

Левый текстовый редактор представляет область ввода и редактирования текста программы.

В статусной строке показывается позиция курсора в области редактирования (номер строки и позиция в строке).

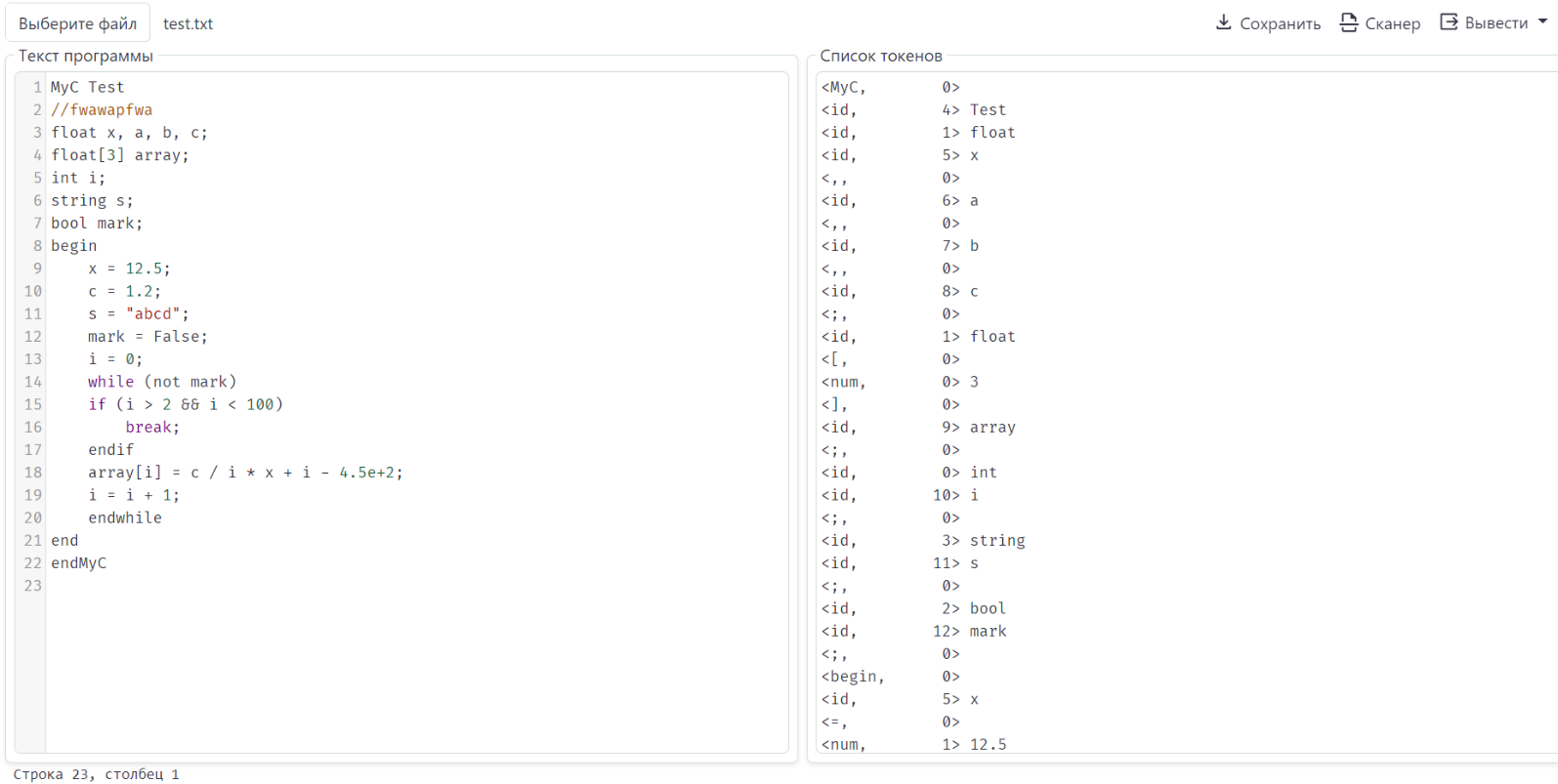


Рис. 3. Интерфейс приложения

# Заключение

В процессе выполнения расчетно-графической работы

* разработан формальный язык описания контекстно-свободного языка программирования;
* 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. Структуры и алгоритмы обработки данных: Учебное пособие / Л.А. Павлов. Чебоксары: Изд-во Чуваш. ун-та, 2008. 252 с.
3. Свердлов, С.З. Языки программирования и методы трансляции: Учебное пособие / С.З. Свердлов.– СПб.: Питер, 2007.– 638 с.
4. 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.