**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**

**Выполнил:**

студент группы ИВТ-41-20

Галкин Д.С.

**Руководитель:**

доцент Павлов Л.А.

Чебоксары 2023

**Оглавление**

[Задание к РГР 3](#_Toc425271714)

[Введение 4](#_Toc425271715)

[1. Описание языка 5](#_Toc425271716)

[2. Лексические классы языка 7](#_Toc425271717)

[3. Синтез конечных автоматов-распознавателей 8](#_Toc425271718)

[4. Структуры данных и алгоритмы 10](#_Toc425271719)

[5. Программная реализация лексического анализа 13](#_Toc425271720)

[Заключение 16](#_Toc425271721)

[Список использованной литературы 17](#_Toc425271722)

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.

К общим требованиям добавляются производные типы и другие операторы в зависимости от номера варианта 41:

* производный тип – запись;
* оператор варианта типа case.

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 *Rat*) 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.

Ниже приведено описание синтаксиса языка *Rat*.

1. Прогр = "**funMain**" Идент Блок "**End**".
2. Блок = СписОбъявл ПоследОператоров "**;**".
3. СписОбъявл = Объявление {Объявление}.
4. Идент = Буква {Буква **|** Цифpа }.
5. Объявление = Тип СписокИдент | Структура "**;**".
6. Тип = "**float | bool | int**".
7. Структура = "**struct**" Идент "**{**" СписОбъявл "**}**".
8. СписокИдент = Идент { "**,**" Идент }.
9. ПоследОператоров = Оператор { "**,**" Оператор }.
10. Оператор = Присвоение | Переключатель.
11. Присвоение = Переменная "**=**" Выражение "**;**".
12. Переменная = [Идент | Идент.Идент].
13. Переключатель = "**when**""**(**" Идент"**)**" "**head**" СравнПереключ {СравнПереключ} ";" "**footer**".
14. СравнПереключ = {Цифра} “->” {Цифра}.
15. Выражение = ПростоеВыраж [Отношение ПростоеВыраж].
16. Отношение = "<" | "<=" | ">" | ">=" | “==” | “!=”.
17. ПростоеВыраж = Переменная [Отношение | Операция] Переменная { [Отношение | Операция] Переменная }.
18. Операция = [ПростОперация | ЛогичОперация].
19. ПростОперация = ["\*" | "." | "+" | "-"].
20. ЛогичОперация = [Отриц | “and” | “or” ].
21. Отриц = "!" "(" Переменная ")".
22. Целое = Цифpа { Цифpа }.
23. ЧисловаяКонст = Целое ["." Целое] ["e" ["-" | "+"] Целое].
24. 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: int, float, bool.
* В ключевых словах и идентификаторах прописные и строчные буквы не различаются.
* Ключевые слова языка зарезервированы, их нельзя использовать в качестве идентификаторов.
* В соответствии с правилами 4 “Идент” является только именем переменной.
* Правило 13 (переключатель) принимает переменную типа (int, float), сравнивает значение с списком значений и возвращает результат доступных типов.
* Комментарий представляет собой любую последовательность символов, заключенную в фигурные скобки "//" и "//".

Пример программы (а точнее, схемы программы) на языке *Rat*:

**funMain**

**int a;**

**float b;**

**bool z;**

**struct example {**

**int a;**

**float b;**

**}**

**a = 1;**

**b = 1 + 2;**

**z = (1 / 2) and !(1 + 1 == 2);**

**a = example.a;**

**switch(a) head**

**1 ->** "**1**";

2 **->** "**2**" ;

**footer**

**End**

# Лексические классы языка

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

Таблица 1. Лексические классы языка *Rat*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Код | Токен | Имя  (обозначение) | Формальный  шаблон | Значение  атрибута |
|  | funMain | **fnM** | ***funMain*** | 0 |
|  | End | **end** | ***End*** | 0 |
|  | struct | **struct** | ***struct*** | 0 |
|  | and | **and** | ***and*** | 0 |
|  | or | **or** | ***or*** | 0 |
|  | when | **when** | ***when*** | 0 |
|  | head | **head** | ***head*** | 0 |
|  | footer | **footer** | ***footer*** | 9 |
|  | not | **!** | ***!*** | 0 |
|  | точка | **.** | ***.*** | 0 |
|  | точка с запятой | **;** | **;** | 0 |
|  | запятая | **,** | **,** | 0 |
|  | следование | **->** | **->** | -> |
|  | откр. скобка | **(** | **(** | 0 |
|  | закр. скобка | **)** | **)** | 0 |
|  | присваивание | **assign** | **=** | 0 |
|  | идентификатор | **id** | ***l* (*l* | *d*)\***  *l* – буква, *d* – цифра | указатель на  запись в таблице |
|  | пробел | **sp** | (**sp** | **tab, nl**)+  sp – пробел, tab – табуляция,  nl – перевод строки | токен не формируется |
|  | комментарий | **com** | **/.** **⊥\*** **./**  **⊥** – любой символ кроме "./" или "/." | 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 5 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.

# Лексические классы языка

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 *Rat*, 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 –7 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 **lbl** и **id** значениями атрибутов являются номера соответствующих строк таблицы символов (обозначение *ns*).

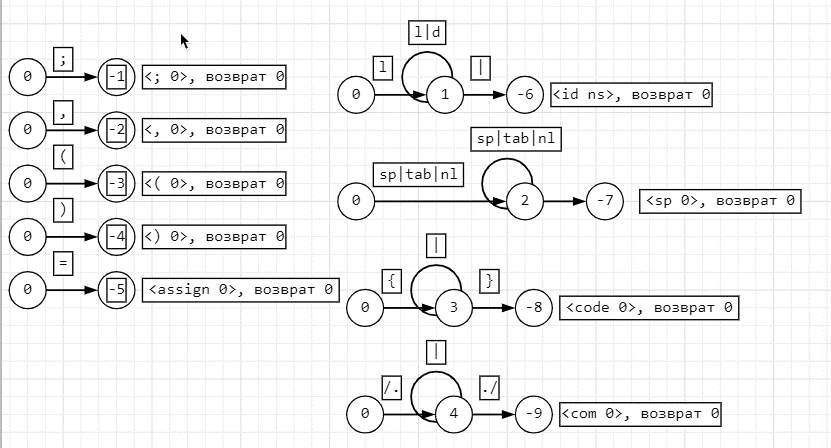


Рис. 1. Конечные автоматы-распознаватели для токенов:

а – символы-разделители и скобки; б – «двоеточие» и «присваивание»;

в – «идентификатор»; г – «пробел»; в – «комментарий»

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 (за исключением конечных состояний) | | | | |
| 0 | 1 | 2 | 3 | 4 |
| **;** | -1 | -6 | -7 | 3 | 4 |
| **,** | -2 | -6 | -7 | 3 | 4 |
| **(** | -3 | -6 | -7 | 3 | 4 |
| **)** | -4 | -6 | -7 | 3 | 4 |
| **=** | -5 | -6 | -7 | 3 | 4 |
| **l** | 1 | 1 | -7 | 3 | 4 |
| **d** | 1 | 1 | -7 | 3 | 4 |
| **/.** | 4 | -6 | -7 | 3 | 4 |
| **./** | err(1) | -6 | -7 | 3 | -9 |
| **{** | 3 | -6 | -7 | 3 | 4 |
| **}** | err(1) | -6 | -7 | -8 | 4 |
| **sp** | 2 | -6 | 2 | 3 | 4 |
| **tab** | 2 | -6 | 2 | 3 | 4 |
| **lf** | 2 | -6 | 2 | 3 | 4 |
| Любой другой символ | err(1) | -6 | -7 | 3 | 4 |

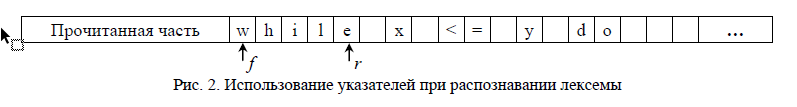
В данной таблице строкам соответствуют допустимые входные символы, а столбцам – со стояния автомата (конечные состояния не указаны, поскольку для них функция переходов не определена). В ячейке на пересечении строки и столбца записывается состояние, в которое должен перейти автомат, если в данном состоянии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

err(1) – лексема не может начинаться с данного символа (номер состояния –11).

# Структуры данных и алгоритмы

Исходный текст обрабатываемой грамматики представляет собой текстовый файл. Для об- легчения 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 вход- ного буфера повышает также эффективность анализатора, так как считывание блока символов обычно существенно более эффективно, чем посимвольное считывание. Используется схема бу- феризации, рекомендованная в [1, 4].

Буфер рассматривается как массив символов. Для распознавания лексемы требуется два ука- зателя (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, распо- знав лексему, переходит в конечное состояние, указатель *r* будет указывать на последний символ лексемы или на последний дополнительно прочитанный символ. В последнем случае дополни- тельно 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*.



Для каждой лексемы сканер формирует выходной токен вида <*код\_токена*, *атрибут*> для последующих фаз компиляции. Код\_токена идентифицирует лексический класс лексемы и опре- деляют работу синтаксического анализатора (рассматривается как терминал). Для удобства *код\_токена* будем представлять абстрактным именем (или специальным обозначением) и ссы- латься на токен по его имени (обозначению). *Атрибут* токена обеспечивает доступ к дополни- тельной информации о лексеме, если лексическому классу соответствует множество лексем, и определяет трансляцию токена (семантический анализ и генерация промежуточного кода). Коди- ровка 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 лек- семы 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. По- зиция (индекс) ключевого слова в таблице соответствует коду его токена. Элемент таблицы клю чевых слов состоит из 2-х полей: *Lex* – лексема ключевого слова, *Name* – обозначение токена (нужно только для отображения результатов).

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

Можно предложить следующую структуру таблицы меток, в которой элемент таблицы со-стоит из полей *Lex* – лексема метки и *Other* – другие поля, которые не используются лексическим анализатором, и могут быть легко детализированы в процессе разработки других фаз компиляции.

В таблице идентификаторов наряду с лексемой могут храниться тип идентификатора, точ-ность, длина, адрес памяти, число измерений и значения граничных пар (для массивов) и т.п., ко-торые определяются на последующих фазах компиляции. Идентификаторы могут представлять различные объекты. В языке *Rat* идентификатор может быть именем программы, функцио-нальным идентификатором, именем типа, именем переменной или именем константы. Поэтому используется понятие категории идентификатора, для ее хранения необходимо предусмотреть специальное поле. Примем следующую кодировку категорий: 1 – имя программы, 2 – функцио-нальный идентификатор, 3 – имя типа, 4 – имя константы, 5 – имя переменной. В языке *Rat* имеют место предопределенные идентификаторы имен типов (integer, float, Boolean), которые не являются ключевыми словами. Эти идентификаторы заносятся в таблицу идентификаторов зара-нее с указанием категории и типа.

Таким образом, можно предложить следующую структуру таблицы идентификаторов:

*Lex* – лексема идентификатора,

*Cat* – категория идентификатора,

*Tip* – тип идентификатора,

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

Осталось определиться с хранением лексем. Хранить лексемы непосредственно в таблицах неэффективно, поскольку они обычно имеют разную длину, следовательно, придется устанавли-вать 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 об-ласть памяти, где находится лексема.

**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 ре-зультате 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 таб-лиц. При вставке в таблицу лексемы в соответствии с принятым порядком номера строк лексем, следующих за вставленной лексемой, изменятся (увеличатся на единицу). Это недопустимо, по-скольку 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 ко-торой 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 ожи-дать. 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, задавае-мой 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 таб-лицы, что позволяет, с одной стороны, использовать этот элемент в качестве «сторожа» («часово-го») при поиске, а с другой стороны, если поиск безуспешен, лексема уже оказывается добавлен-ной в таблицу и достаточно только скорректировать размер таблицы.

Для таблицы ключевых слов, поскольку она статическая и упорядочена в алфавитном поряд-ке, применим бинарный поиск

**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 симво-лами. 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 13

использовать методы сжатия таблиц переходов (один из таких методов рассмотрен в [1]). Второй метод предполагает использование инструкций типа **case** языка Паскаль: внешняя инструкция **case** реализует выбор стояний автомата, для каждого состояния требуется инструкция **case** для выбора входного символа. Основной недостаток метода – относительно большой программный код, кото-рый к тому же зависит от функций переходов автомата – любое изменение функции переходов приводит к необходимости изменения программного кода. Исходя из изложенного, выбор осу-ществляется в пользу первого метода реализации автомата. Алгоритм работы автомата имеет сле-дующий 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 про-читанный символ, функция *NextSym* возвращает очередной символ из входной строки (в соответ-ствии 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 дополни-тельно прочитанные символы во входной поток, устанавливает указатель *f* (см. рис. 2) на начало следующей лексемы.

**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 лексическо-го 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.: Пи-тер, 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.