**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-41-15

Фамилия И.О.

**Руководитель:**

доцент Павлов Л.А.

Чебоксары 2022

**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 (например, вариант 99):

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

**Примечание для студента.** Приведен пример оформления задания. В данном примере отчета по РГР в качестве языка взят модифицированный язык так называемых схем программ, где не детализируются выполняемые операции. Поэтому не все указанные выше требования выполнены (нет логических и арифметических операций, операций отношения и т.п.). Это, все-таки, пример оформления отчета для того, чтобы показать его структуру, а не полный шаблон какого-то языка, в котором студенту остается только внести небольшие изменения. Студент сам все это должен разрабатывать на основе знаний полученных на лекциях.

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 *MyLang*) 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 *MyLang*.

1. СхемаПрогр = "**scheme**" Идент "**;**" Блок "**endsch**".
2. Блок = СписОбъявл "**start**" ПоследОператоров "**;**" ЗаклОператор.
3. СписОбъявл = Объявление { Объявление }.
4. Объявление = [ Категория ] Тип СписокИдент "**;**".
5. Категория = "**const**" **|** "**funcional**".
6. Тип = Идент.
7. СписокИдент = Идент { "**,**" Идент }.
8. Идент = Буква {Буква **|** Цифpа }.
9. ПоследОператоров = Оператор { "**;**" Оператор }.
10. Оператор = [ Метка "**:**"] ОпБезМетки.
11. Метка = Целое.
12. ОпБезМетки = Присваивание **|** Условный.
13. Присваивание = Идент "**:=**" Терм ["**goto**" Метка ].
14. Терм = Идент [ "**(**" Аргументы "**)**" ].
15. Аргументы = Терм **|** Терм "**,**" Терм.
16. Условный = "**if**" Терм "**then**" Метка "**else**" Метка.
17. ЗаклОператор = [ Метка "**:**"] "**stop**".
18. Целое = Цифpа { Цифpа }.
19. 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, float, Boolean.
* В ключевых словах и идентификаторах прописные и строчные буквы не различаются.
* 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.
* В соответствии с правилами 4 и 5 все идентификаторы (кроме имени схемы) подразделяются на 3 категории: имя константы (ключевое слово **const**, не должно встречаться в левой части оператора присваивания), функциональный идентификатор (ключевое слово **funcional**) и имя переменной (если не указана категория).
* Если конструкция «Терм» (правило 14) представляет собой только идентификатор («Идент») без аргументов (конструкция «Аргумент»), то этот идентификатор может быть только именем переменной или константы, в противном случае – функциональный идентификатор. Тип терма определяется типом идентификатора «Идент», причем типы идентификатора и аргументов должны совпадать.
* В условном операторе (конструкция «Условный», правило 16) в качестве терма (конструкция «Терм») может быть только терм типа Boolean.
* Все операторы могут иметь метки для реализации переходов к ним из других операторов. В качестве метки обычно используются порядковый номер оператора в последовательности. Не может быть двух операторов с одинаковыми метками.
* Все операторы (кроме условного) могут иметь безусловный переход (конструкция «"**goto**" Метка») для реализации перехода к выполнению оператора с соответствующей меткой.
* Комментарий представляет собой любую последовательность символов, заключенную в фигурные скобки "{" и "}".

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

**scheme** Test1;

**const** float a;

**funcional** float g, h;

**funcional** Boolean p;

**funcional** integer m, n;

float x, y;

integer i, j;

**start** {комментарий}

y := a;

2: **if** p(m(g(i, j))) **then** 5 else 3;

3: y := g(m(x, a), n(y, h(j)));

x:= h(x) **goto** 2; {конец тела цикла}

5: **stop**

**endsch**

# Лексические классы языка

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

Таблица 1. Лексические классы языка *MyLang*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Код | Токен | Имя  (обозначение) | Формальный  шаблон | Значение  атрибута |
|  | **const** | **cons** | ***const*** | 0 |
|  | **else** | **else** | ***else*** | 0 |
|  | **endsch** | **end** | ***endsch*** | 0 |
|  | **funcional** | **func** | ***funcional*** | 0 |
|  | **goto** | **goto** | ***goto*** | 0 |
|  | **if** | **if** | ***if*** | 0 |
|  | **scheme** | **sch** | ***scheme*** | 0 |
|  | **start** | **strt** | ***start*** | 0 |
|  | **stop** | **stop** | ***stop*** | 0 |
|  | **then** | **then** | ***then*** | 0 |
|  | точка с запятой | **;** | **;** | 0 |
|  | запятая | **,** | **,** | 0 |
|  | откр. скобка | **(** | **(** | 0 |
|  | закр. скобка | **)** | **)** | 0 |
|  | двоеточие | **:** | **:** | 0 |
|  | присваивание | **ass** | **:=** | 0 |
|  | идентификатор | **id** | ***l* (*l* | *d*)\***  *l* – буква, *d* – цифра | указатель на  запись в таблице |
|  | метка | **lbl** | ***dd*\***  *d* – цифра | указатель на  запись в таблице |
|  | пробел | **sp** | (**sp** | **tab** | **lf**)+  sp – пробел, tab – табуляция,  lf – перевод строки | токен не формируется |
|  | комментарий | **com** | **{** **⊥\*** **}**  **⊥** – 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.

**Примечание для студента.** 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 [4].

# 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*, 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.

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 *MyLang*, 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*).

**Примечание для студента**. Поскольку у Вас в языке обязательно будут операции отношения, которые обычно объединяются в один лексический класс (токен **rel**), в качестве значений атрибутов указываются коды соответствующих операций, определяемые в таблице лексических классов. То же самое касается и арифметических операций. Примеры определения соответствующих лексических классов приведены в методических указания к РГР [4]

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.

–1

**;**

<**;**, 0>, возврат 0

–2

**,**

<**,**, 0>, возврат 0

–4

**)**

<**)**, 0>, возврат 0

*а*

–3

**(**

<**(**, 0>, возврат 0

*б*

–6

**:**

<**ass**, 0>, возврат 0

**=**

–5

<**:**, 0>, возврат 1

**⊥**

–7

**⊥**

*l*

<**id**, *ns*>,

возврат 1

*l* | *d*

*в*

*d*

<**lbl**, *ns*>,

возврат 1

*d*

**⊥**

–8

*г*

–9

**⊥**

<**sp**, 0>,

возврат 1

sp | tab | lf

sp | tab | lf

–10

**}**

**{**

<**com**, 0>,

возврат 0

**⊥**

*е*

*д*

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

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, 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 | 5 |
| **;** | –1 | –5 | –7 | –8 | –9 | 5 |
| **,** | –2 | –5 | –7 | –8 | –9 | 5 |
| **(** | –3 | –5 | –7 | –8 | –9 | 5 |
| **)** | –4 | –5 | –7 | –8 | –9 | 5 |
| **:** | 1 | –5 | –7 | –8 | –9 | 5 |
| **=** | err(1) | –6 | –7 | –8 | –9 | 5 |
| ***l*** | 2 | –5 | 2 | –8 | –9 | 5 |
| ***d*** | 3 | –5 | 2 | 3 | –9 | 5 |
| **{** | 5 | –5 | –7 | –8 | –9 | 5 |
| **}** | err(1) | –5 | –7 | –8 | –9 | –10 |
| **sp** | 4 | –5 | –7 | –8 | 4 | 5 |
| **tab** | 4 | –5 | –7 | –8 | 4 | 5 |
| **lf** | 4 | –5 | –7 | –8 | 4 | 5 |
| любой другой  символ | err(1) | –5 | –7 | –8 | –9 | 5 |

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 новым конечным состоянием, соответствующим определенному виду лексических ошибок, а именно:

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

**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, 4].

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 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 состоит 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.

**Примечание для студента**. У Вас обязательно будут числовые константы, для которых необходима динамическая таблица. Тогда можно предложить следующую структуру таблицы констант, в которой элемент таблицы состоит из полей *Lex* – лексема константы, *Tip* – тип константы. Тип числовой константы определяется в процессе лексического анализа. На последующих фазах компиляции при каждом обращении к константе будет выполняться преобразование символьного представления константы в ее значение, что приведет к некоторому снижению эффективности компиляции. В таблицу числовых констант можно добавить поле для хранения значения константы. В этом случае выполняется единственное преобразование символьного представления константы 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 константе 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.

В таблице идентификаторов наряду с лексемой могут храниться тип идентификатора, точность, длина, 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 *MyLang* идентификатор может быть именем программы, функциональным 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 – функциональный идентификатор, 3 – имя типа, 4 – имя константы, 5 – имя переменной. В языке *MyLang* имеют место предопределенные идентификаторы имен типов (integer, float, Boolean), 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:

*Lex* – лексема идентификатора,

*Cat* – категория идентификатора,

*Tip* – тип идентификатора,

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

**Примечание.** Естественным названием поля типа является слово Type. Однако используется слово Tip, т.к. программная реализация планируется в среде Delphi, где слово Type является зарезервированным ключевым словом и его нельзя использовать в качестве идентификатора.

**Примечание для студента**. Поля *Cat* и *Tip* 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.

**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.

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

**Примечание для студента.** 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 для таблицы идентификаторов (для таблицы констант (меток) – какой хотите).

**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 **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, 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.

**Примечание для студента**. В автомат желательно включить пропуск пробелов (комментариев), чтобы не передавать токен «пробел» («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» («комментарий»). Студент должен модифицировать алгоритм самостоятельно и включить в записку модифицированный алгоритм.

**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 |

# Программная реализация лексического анализа

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 для студента**. 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.

Сканер реализован в виде функции

**function** Scaner: tToken,

выходным параметром которой является формируемый токен. Тип tToken определен как

**type** tToken = **record** Code, Attr: byte; **end**,

где Code – код токена, Attr – атрибут токена.

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 лексического анализатора рассматриваются как символьные константы типа char. Следует заметить, что 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 системе программирования есть соответствующая функция).

Таблица переходов автомата задается константой-массивом

TP: **array** [1..14, 0..5] **of** integer.

Для определения номера строки, соответствующей входному символу, предусмотрена функция

**function** NomStrOfTP (Sym: char): integer,

которая возвращает номер строки для символа Sym.

Исходный текст тестовой программы размещается в визуальном компоненте типа TMemo 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.

Таблица ключевых слов (KeyWords) представляет собой массив-константу

KeyWords: **array** [1.. MaxKeys] **of** tElemKey,

элемент массива определяется как

tElemKey = **record** Lex, Name: string; **end**,

где строковые поля Lex и Name предназначены для хранения лексемы и ее обозначения соответственно.

Таблица меток (Lbls) представляет собой массив

Lbls: **array** [1.. MaxLbl] **of** tElemLbl,

элемент массива определяется как

tElemLbl = **record** Lex: string; Other: Byte; **end**,

где строковое поле Lex предназначено для хранения лексемы.

Таблица идетификаторов (Idents) представляет собой массив

Idents: **array** [1.. MaxId] **of** tElemIde,

элемент массива определяется как

tElemIde = **record** Lex: string; Cat, Tip, Other: Byte; **end**,

где строковое поле Lex предназначено для хранения лексемы, поле Cat – категория лексемы, поле Tip –тип идентификатора.

В tElemLbl и tElemIde поле Other носит фиктивный характер (поэтому и его тип Byte взят произвольно), главная цель – показать, что элемент имеет сложную структуру и в последующем будет меняться (не затрагивая другие поля).

**Примечание.** В массиве Idents для поля типа Tip указан тип Byte. Это временно, поскольку тип этого поля будет зависеть от принятой системы типов, которая будет разрабатываться в РГР № 2 при решении задач семантического анализа (проверки типов).

В разделе 4 говорилось, что хранить лексемы непосредственно в массиве неэффективно, а лучше хранить указатели на область памяти, где находится лексема. Представленная выше организация таблиц вроде бы противоречит сказанному. Однако это не так. В Delphi строка типа string автоматически представляется как указатель на строку, поэтому и массив строк формируется как массив указателей на строки.

Здесь использованы статические массивы с предельно допустимым размером (MaxLbl, MaxId), поскольку тестовые программы относительно небольшие. Для работы с данными больших размеров (что характерно для текстов реальных программ) такие массивы не самый лучший способ представления таблиц.

Перейдем к рассмотренным в разделе 4 алгоритмам. Алгоритм поиска лексемы реализован одноименной функцией (с соответствующей модификацией для разных типов таблиц)

**function** SearchLex (Lex: string): byte.

Поскольку предельные размеры таблиц ограничены, функция осуществляет также соответствующий контроль и возвращает нулевое значение в случае переполнения какой-либо из таблиц.

Для статической таблицы ключевых слов используется бинарный поиск, который реализован функцией

**function** SearchKeyWord (Lex: string): byte.

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.

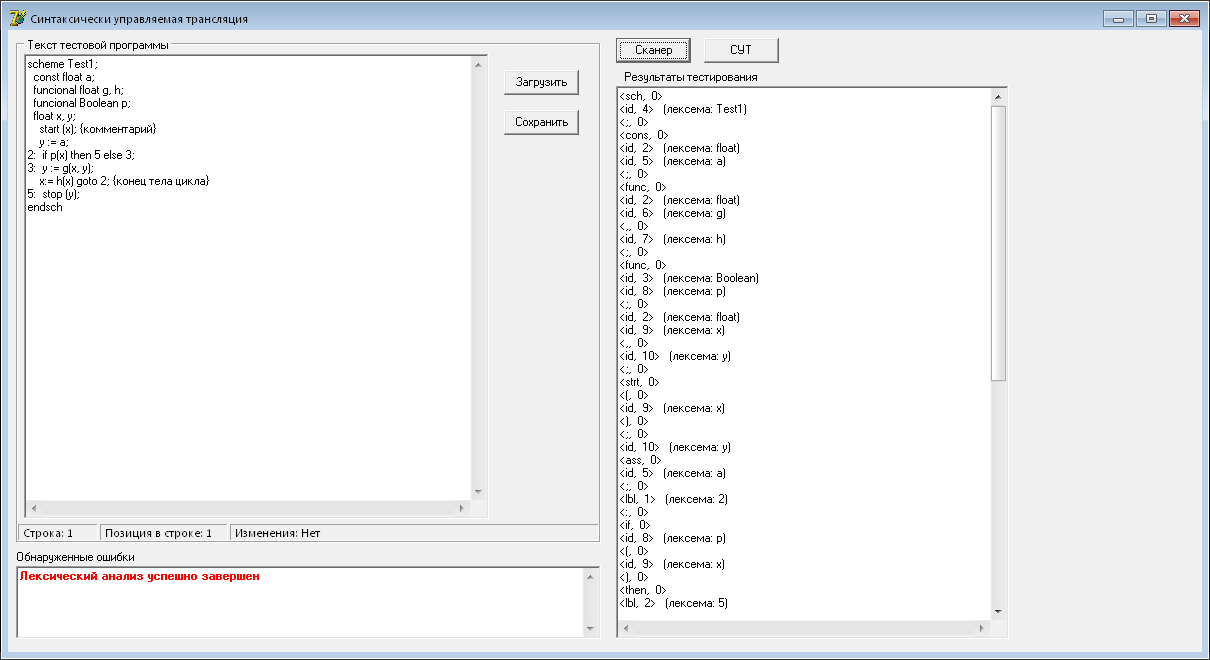


Рис. . Интерфейс приложения

Заключение

В процессе выполнения расчетно-графической работы

* разработан 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.