Лабораторная работа № 7

Регулярные языки и Конечные автоматы

Теоретическая часть

Регулярные множества и регулярные выражения

Определение 1. Пусть Σ - конечный алфавит и ε – обозначает пустую цепочку. *Регулярное множество (или регулярный язык) в алфавите* Σ определяется рекурсивно следующим образом:

1. ∅ (пустое множество) – регулярное множество в алфавите Σ;
2. {ε} – регулярное множество в алфавите Σ;
3. {a} – регулярное множество в алфавите Σ для каждого $a \in \Sigma$;
4. если P и Q – регулярные множества в алфавите Σ, то таковы же и множества
   (a) $P \cup Q$ – объединение множеств,
   (b) PQ – множество, составленное из всевозможных сцеплений (конкатенаций) их элементов,
   (c) P^* – множество всевозможных сцеплений (конкатенаций) его элементов;
5. ничто другое не является регулярным множеством в алфавите Σ.

Определение 2. *Регулярные выражения в алфавите* Σ и регулярные множества, которые они *обозначают,* определяются рекурсивно следующим образом:

1. ∅ – регулярное выражение, обозначающее регулярное множество ∅,
2. ε – регулярное выражение, обозначающее регулярное множество {ε},
3. если a *in* Σ, то a – регулярное выражение, обозначающее регулярное множество {a},
4. если p и q – регулярные выражения, обозначающие регулярные множества P и Q соответственно, то
   (a) (p + q) – регулярное выражение, обозначающее $P \cup Q$;
   (b) (pq) – регулярное выражение, обозначающее PQ;
   (c) (p)* – регулярное выражение, обозначающее P^*;
5. ничто другое не является регулярным выражением.

Замечания.

1) $p+ = pp^*$,
2) будем устранять из регулярных выражений лишние скобки там, где это не может привести к недоразумениям. В этой связи предполагается, что наивысшим приоритетом обладает операция *, затем идет конкатенация и далее операция +,
3) в литературе вместо символа пустой цепочки ε часто используют символ λ.

Примеры регулярных выражений

1) (a+b)*abb обозначает множество всех цепочек, составленных из символов a и b и оканчивающихся цепочкой abb.
2) (00+11)*((01+10)(00+11)*(01+10)(00+ 1)*)* обозначает множество *всех* цепочек нулей и единиц, содержащих четное число нулей и четное число единиц.
3) 1*(01*0(01*01*0+1)*01*+ε) обозначает множество всех цепочек нулей и единиц, содержащих количество нулей кратное трем, и множество всех цепочек, составленных только из единиц.

В программировании регулярными выражениями описываются лексемы языка, а точнее категории лексем, такие, как ключевые слова, идентификаторы, константы, строковые литералы, символы пунктуации. Обобщением понятия лексемы является «токен» (token). Каждый токен можно представить в виде структуры, содержащей *идентификатор токена* (или идентификатор категории токена) и последовательность символов

лексемы, выделенной из входного потока. В качестве справочной информации, улучшающей диагностику, к описанию токена можно добавить *номер строки, начальную и конечную позицию* лексемы в пределах текущей строки. Например, для инструкции присваивания языка Pascal

d := b*b – 4*a*c

будет получена такая последовательность токенов:

< **Identifier**, "d" >
< **AssignOperator,** ":=" >
< **Identifier**, "b" >
< **MulOperator,** "*" >
< **Identifier**, "b" >
< **AddOperator,** "–" >
< **Number**, "4" >
< **MulOperator,** "*" >
< **Identifier**, "a" >
< **MulOperator,** "*" >
< **Identifier**, "c" >

В приведенном примере **Identifier, Number, AssignOperator, MulOperator, AddOperator** – это категории токенов.

На практике для описания токенов используют расширенный язык регулярных выражений. Например,

```
digit              ["0-9"]
token Decimal      '0' | ["1-9"]digit*;
token Octal        '0'["0-7"]*;
token Hexadecimal  '0' ('x'|'X') (digit | ["a-f"] | ["A-F"])+;
token Double       (((digit+ '.') | ('.' digit+))(('e'|'E') ('+'|'-')? digit+)?) |
                   (digit+ ('e'|'E') ('+'|'-')? digit+) | (digit+ '.' digit+ ('e'|'E') ('+'|'-')? digit+);
```

Кратко правила расширенного языка регулярных выражений можно сформулировать так.

1. В регулярном выражении любой символ соответствует самому себе, если только он не является метасимволом со специальным значением (такими метасимволами являются \, |, (,), [, {, *, +, ^, $, ? и .).
2. Можно "защитить" любой метасимвол, то есть заставить рассматривать его как обыкновенный символ, а не как команду, поставив перед метасимволом обратную косую черту \.
3. Символы могут быть сгруппированы в классы. Класс - это совокупность символов, заключенный в квадратные скобки [и]. Можно указывать как отдельные символы, так и их диапазон (диапазон задается двумя крайними символами, соединенными тире). Наример, [aeiou] обозначает гласную, а [A-Za-z]+ (метасимвол + означает утверждение: "один или более таких символов") обозначает слово, состоящее из букв.
4. Вместо символа альтернативы + используется символ | как разделитель. Чтобы было ясно, где начинается и где заканчивается набор альтернатив, их заключают в круглые скобки - иначе символы, расположенные справа и слева от группы, могут смешаться с альтернативами.
5. Чтобы указать на то, что тот или иной шаблон в строке может повторяться определенное количество раз, используются следующие квантификаторы:
   * - ноль или несколько совпадений,
   + - одно или несколько совпадений,
   ? - ноль совпадений или одно совпадение.

Конечные автоматы

Конечный автомат (КА) является распознавателем цепочек регулярного множества. Неформально КА можно определить, задав

- конечное множество его управляющих состояний,
- допустимые входные символы,
- начальное состояние и
- множество заключительных состояний, т. е. состояний, указывающих, что входная цепочка допускается,
- задается также функция переходов состояний, которая по данному «текущему» состоянию и «текущему» входному символу указывает все возможные следующие состояния.

В общем случае КА – это недетерминированное устройство в теоретико-автоматном смысле, т. е. оно переходит во все свои следующие состояния, если угодно, дублируя себя так, что в каждом из возможных следующих состояний находится один экземпляр этого устройства. Недетерминированный конечный автомат (НКА) допускает входную цепочку, если какой-нибудь из его параллельно работающих экземпляров достигает допускающего состояния. Недетерминизм КА не следует смешивать со «случайностью», при которой автомат может случайно выбрать одно из следующих состояний с фиксированными вероятностями, но этот автомат всегда имеется только в одном экземпляре.

Формальное определение НКА будет выглядеть так.

Определение 3. НКА – *это пятерка* $M = (Q, \Sigma, \delta, q0, F)$*, где*

1. Q - конечное множество *состояний;*
2. Σ конечное множество допустимых *входных символов,*
3. δ - отображение множества $Q \times \Sigma$ в множество $P(Q)$, определяющее поведение управляющего устройства; функцию δ иногда называют *функцией переходов*
4. $q0 \in Q$ – *начальное состояние* управляющего устройства;
5. $F \subseteq Q$ – множество *заключительных состояний.*

Определение 4. Если M – КА, то пара $(q, w) \in Q \times \Sigma^*$ называется *конфигурацией* автомата M. Конфигурация $(q0, w)$ называется *начальной,* а пара (q, ε), где $q \in F$, называется *заключительной* (или *допускающей).*

Обозначения для такта автомата

- Такт автомата представляется бинарным отношением $\vdash$
- Отношения $\vdash^+$ и $\vdash^*$ являются соответственно транзитивным и рефлексивно-транзитивным замыканием отношения $\vdash$
- Через $\vdash^k$ обозначается k-я степень отношения $\vdash$

Говорят, что автомат M *допускает* цепочку w, если $(qo, w) \vdash^* (q, \varepsilon)$ для некоторого $q \in F$. *Языком, определяемым (распознаваемым, допускаемым) автоматом* M (обозначается $L(M)$), называется множество входных цепочек, допускаемых автоматом M, т. е.
$L(M) = \{w \mid w \in \Sigma^* \text{ и } (qo, w) \vdash^* (q, \varepsilon) \text{ для некоторого } q \in F\}$

Примеры конечных автоматов

1) Пусть $M = (\{p, q, r\}, \{0, 1\}, \delta, p, \{r\})$ – КА, где δ задается таблицей.

Состояния	**Вход**	
	0	1
p	$\{q\}$	$\{p\}$
q	$\{r\}$	$\{p\}$
q	$\{r\}$	$\{r\}$

M допускает все цепочки нулей и единиц, содержащие два стоящих рядом нуля.

2) Пусть $M = (\{q0, q1, q2, q3, qf\}, \{1, 2, 3\}, \delta, q0, \{qf\})$ – КА, где δ задается таблицей.

Состояния	**Вход**		
	1	2	3
q0	{*q0, q1*}	{*q0, q2*}	{*q0, q3*}
q1	{*q1, qf*}	{*q1*}	{*q1*}
q2	{*q2*}	{*q2, qf*}	{*q2*}
q3	{*q3*}	{*q3*}	{*q3, qf*}
qf	∅	∅	∅

M допускает цепочки в алфавите {1, 2, 3}, у которых последний символ цепочки уже появился в ней раньше.

Часто бывает удобно графическое представление конечного автомата.

Определение 5. Пусть *M* - НКА. *Диаграммой* (или *графом переходов)* автомата *M* называют неупорядоченный помеченный граф, вершины которого помечены именами состояний и в котором есть дуга *(p, q),* если существует такой символ $a \in \Sigma$, что $q \in \delta(p, a)$. Кроме того, дуга *(p, q)* помечается списком, состоящим из таких *a,* что $q \in \delta(p, a)$.

Частным случаем НКА является детерминированный конечный автомат (ДКА).

Определение 6. Пусть *M* – НКА. Назовем автомат *M детерминированным,* если множество $\delta(q, a)$ содержит не более одного состояния для любых $q \in Q$ и $a \in \Sigma$. Если $\delta(q, a)$ всегда содержит точно *одно* состояние, то автомат *M* назовем *полностью определенным.*

Один из наиболее важных результатов теории КА состоит в том, что класс языков, определяемых НКА, совпадает с классом языков, определяемых полностью ДКА. Существует теорема, которая гласит, что «любой недетерминированный конечный автомат может быть преобразован в детерминированный так, чтобы их языки совпадали».

Примечание. В работе [1] приводятся следующие алгоритмы:
1. Построение НКА по регулярному выражению.
2. Преобразование НКА в ДКА.
3. Минимизация количества состояний ДКА.

Связь между конечными автоматами и регулярными языками устанавливается с помощью теоремы Клини, которая гласит, что «класс языков, допускаемых конечными автоматами, совпадает с классом регулярных языков».

Практическая часть

Построение диаграмм переходов

Теория конечных автоматов и регулярных языков применяется при построении программы лексического анализа. Лексический анализ образует первый этап процесса компиляции. На этом этапе терминальные символы, составляющие исходную программу, считываются и группируются в отдельные лексемы. Какие объекты считать лексемами, зависит от определения языка программирования. Определения лексем могут быть представлены либо в виде регулярных выражений, либо в виде правил праволинейной (или автоматной) грамматики. Во втором

случае требуется преобразование праволинейной грамматики к регулярным выражениям. Процесс лексического анализа можно промоделировать с помощью КА-преобразователя.

В качестве промежуточного шага при создании лексического анализатора рекомендуется преобразовать регулярные выражения в стилизованные блок-схемы, называемые диаграммами переходов. Вершины диаграммы переходов соответствуют состояниям анализатора, а дуги соответствуют переходам из одного состояния в другое. Состояния помечают целыми числами, а дуги помечают входными символами. Некоторые состояния являются допускающими, или конечными; конечные состояния на диаграмме изображаются как двойные кружки. Одно состояние является стартовым, или начальным; оно указывается при помощи дуги, помеченной словом **start** и входящей в состояние ниоткуда. Если необходимо вернуться в предыдущее состояние, то возле такого состояния дополнительно ставится символ '*'. Метка **other** на дуге диаграммы означает появление любого символа, не указанного другими исходящими дугами.

В качестве простого примера на рис. 1 показана диаграмма переходов для класса лексем **relop**.

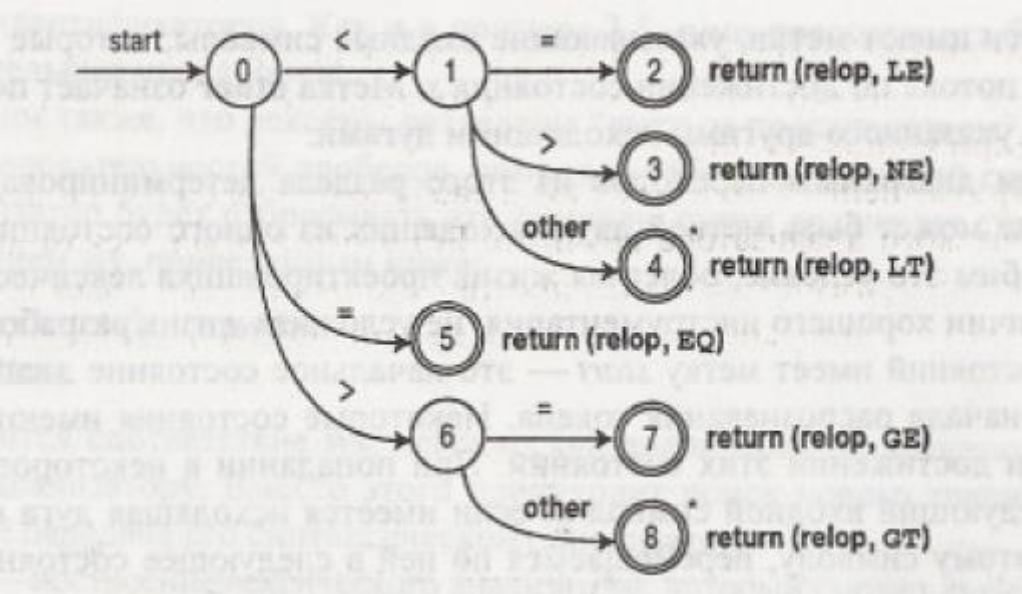

Рисунок 1 - Диаграмма переходов для класса лексем **relop**

Следует заметить, что рис. 1 представляет собой часть более сложной диаграммы переходов. Вообще говоря, может существовать ряд диаграмм переходов, каждая из которых определяет группу лексем. Если при перемещении по диаграмме переходов происходит сбой, указатель текущего входного символа возвращается к тому месту, где он был в стартовом состоянии данной диаграммы, и выполняется переход к следующей диаграмме. Поскольку указатель на начало лексемы (назовем его **token_beginning**) и указатель на текущий входной символ (назовем его **token_forward)** в стартовом состоянии указывают на одно и то же место, такой возврат осуществляется очень просто - возвратом в позицию, которая определяется указателем **token_beginning**. Если сбой происходит во всех диаграммах, вызывается программа обработки и восстановления ошибки.

Реализация диаграмм переходов

Один из способов применения набора диаграмм переходов для реализации лексического анализатора состоит в использовании переменной **state**, хранящей номер текущего состояния в диаграмме переходов. Инструкция **switch**, построенная на основе значения переменной **state**, дает код для каждого из возможных состояний, где и находятся действия, выполняемые в том или ином состоянии. Зачастую код состояния сам представляет собой инструкцию **switch** или множественное ветвление, которые путем чтения и исследования очередного входного символа определяют следующее состояние, в которое должен быть выполнен переход.

На рис. 2 приведен набросок функции `getRelop()` на языке C++, работа которой состоит в моделировании диаграммы переходов, представленной на рис. 1, и возврате указателя на объект типа `TOKEN`, который состоит из идентификатора категории токена (в данном случае `RELOP`) и значения токена (в данном случае – кода для одного из шести возможных операторов сравнения: EQ, NE, LT, LE, GT, GE).

```
TOKEN *getRelop()
{
	TOKEN *retToken = new TOKEN(RELOP);
	while (1)
	{
		switch (state)
		{
		case 0:
			{
				c = nextChar();
				if (c == '<') state = 1;
				else if (c == '=') state = 5;
				else if (c == '>') state = 6;
				else fail();
				break;
			};
		case 1:
			...
		case 8:
			retract();
			(*retToken).value = GT;
			return (retToken);
		}
	}
}
```

Рисунок 2 – Набросок реализации диаграммы переходов для класса лексем **relop**

Функция `getRelop()`сначала создает новый объект типа `TOKEN`, и инициализирует его первый компонент (идентификатор категории токена) членом перечисляемого типа по имени `RELOP`.

Функция `nextChar()`получает очередной символ из входного потока и присваивает его переменной `c`. Затем выполняется проверка соответствия `c` трем ожидаемым символам, и в каждом случае выполняется переход в состояние, определяемое диаграммой перехода на рис. 1.

Если считанный символ не принадлежит множеству символов, с которых может начинаться оператор сравнения, вызывается функция `fail()`. Какие именно действия выполняет эта функция, зависят от глобальной стратегии лексического анализатора по восстановлению после ошибок.

В состоянии 8 сначала вызывается функция `retract()`, поскольку это состояние помечено звездочкой и требуется вернуть указатель входного потока на одну позицию назад. Так как состояние 8 представляет распознанную лексему '>', второй компонент объекта типа `TOKEN` (с именем `value`) устанавливается равным члену перечисляемого типа по имени `GT` .

Задание

1. Изучить упрощенную лексику языка С.
2. Составить регулярные выражения для каждой категории лексем.
3. Составить программу лексического анализа в соответствии с изложенной методикой.
4. Программа должна:
   a. Читать файл исходного текста программы на языке С.
   b. Игнорировать директивы компилятора.

c. Игнорировать комментарии.
d. Выделять лексемы и направлять их в выходной поток в формате XML. Узел XML-документа должен включать:
    i. Категорию лексемы.
    ii. Посимвольное изображение лексемы.
    iii. Номер строки исходного файла.
    iv. Стартовую позицию первого символа лексемы от начала текущей строки.
    v. Позицию последнего символа лексемы или длину лексемы.

5. Упрощенная лексика языка С описана в файле **Лексика.doc**.
6. Варианты входных данных находятся в папке **Варианты**.
7. Пример входных данных и соответствующие выходные данные находятся в папке **Пример**.

Рекомендуемая литература

1. БЕЛОУСОВ А.И., ТКАЧЕВ С.Б. Дискретная математика: Учеб. Для вузов / Под ред. В.С. Зарубина, А.П. Крищенко. – М.: Изд-во МГТУ им. Н.Э. Баумана, 2001.
2. БАКНЕЛЛ Дж. Фундаментальные алгоритмы и структуры данных в Delphi. Библиотека программиста. – М.: ООО «ДиаСофтЮП»; СПб.: Питер, 2006. – 557 с.