**Методический материал для лабораторной работы №2 . Регулярные грамматики и конечные автоматы. Регулярные выражения.**

1. **Распознающие модели**

В числе прочих задач компилятор должен определить принадлежность некоторого текста к конкретному языку. Распознаватель – это специальный алгоритм, позволяющий для некоторой цепочки символов определить, принадлежит ли она заданному языку. Это один из способов задания грамматики и соответственно языка. Распознаватель входит в состав компилятора.

Под распознавателем мы будем понимать обобщенный алгоритм, позволяющий определить некоторое множество (в нашем случае - язык) и использующий в своей работе следующие компоненты: 1) входную ленту, 2) управляющее устройство с конечной памятью и 3) дополнительную рабочую память. Обычно считается, что управляющее устройство может только читать информацию, записанную на входной ленте (чтение производится с помощью входной головки, указывающей на текущий символ) и продвигаться по ней вперед и, возможно, назад. Распознаватель также может изменять состояние памяти, которая может быть организована как конечный линейный список ячеек или как стек (в русской литературе называемый также магазином). В качестве примеров распознавателей можно назвать машину Тьюринга (для языков type-0 , type-1), конечные автоматы (для языков type-3 – регулярных языков) и магазинные автоматы (для языков type-2 – КС языки).

Язык определяется путем задания некоторого множества допустимых заключительных состояний распознавателя: если цепочка, поданная на входную ленту, позволяет распознавателю выполнить последовательность шагов и остановиться в заключительном состоянии, то цепочка принадлежит языку.

**Утверждение.** Язык распознающей системы Rs эквивалентен языку порождающей системы Gs: **L(Rs) = L(Gs).**

Из утверждения следует, *что класс языков, распознаваемый конечными автоматными, совпадает с классом языков, порождаемых автоматными грамматиками и наоборот.*

1. **Лексический анализ. Регулярные грамматики и конечные автоматы**

Рассмотрим методы и средства, которые обычно используются при построении лексических анализаторов. В основе таких анализаторов лежат регулярные грамматики, поэтому рассмотрим грамматики этого класса более подробно.

Лексический анализ (ЛА) - это первый этап процесса компиляции. На этом этапе символы, составляющие исходную программу, группируются в отдельные лексические элементы, называемые лексемами. Лексический анализ важен для процесса компиляции по нескольким причинам:

* замена в программе идентификаторов, констант, ограничителей и служебных слов лексемами делает представление программы более удобным для дальнейшей обработки;
* лексический анализ уменьшает длину программы, устраняя из ее исходного представления несущественные пробелы и комментарии;
* если будет изменена кодировка в исходном представлении программы, то это отразится только на лексическом анализаторе.

Выбор конструкций, которые будут выделяться как отдельные лексемы, зависит от языка и от точки зрения разработчиков компилятора. Обычно принято выделять следующие типы лексем: идентификаторы, служебные слова, константы и ограничители. Каждой лексеме сопоставляется пара (тип\_лексемы, указатель\_на\_информацию\_о\_ней) , называемая токеном. Таким образом, лексический анализатор - это транслятор, входом которого служит цепочка символов, представляющих исходную программу, а выходом - последовательность лексем.

Очевидно, что лексемы перечисленных выше типов можно описать с помощью регулярных грамматик.

Например, идентификатор (I):

I → a | b| ...| z | Ia | Ib |...| Iz | I0 | I1 |...| I9

целое без знака (N):

N→ 0 | 1 |...| 9 | N0 | N1 |...| N9

и т.д.

**Соглашение:** в дальнейшем, если особо не оговорено, под регулярной грамматикой будем понимать леволинейную грамматику.

Напомним, что грамматика G = (T, V, P, S) называется леволинейной, если

каждое правило из Р имеет вид A → Bt либо A → t, где A ∈ V, B ∈ V, t ∈ T.

**Соглашение:** предположим, что анализируемая цепочка заканчивается специальным символом ⊥ - признаком конца цепочки. Для грамматик этого типа существует алгоритм определения того, принадлежит ли анализируемая цепочка языку, порождаемому этой грамматикой

(алгоритм разбора):

(1) первый символ исходной цепочки a1a2...an⊥ заменяем нетерминалом A, для которого в грамматике есть правило вывода A → a1 (другими словами, производим "свертку" терминала a1 к нетерминалу A)

(2) затем многократно (до тех пор, пока не считаем признак конца цепочки) выполняем следующие шаги: полученный на предыдущем шаге нетерминал A и расположенный непосредственно справа от него очередной терминал ai исходной цепочки заменяем нетерминалом B, для которого в грамматике есть правил вывода B → Aai (i = 2, 3,.., n);

Это эквивалентно построению дерева разбора методом "снизу-вверх": на каждом шаге алгоритма строим один из уровней в дереве разбора, "поднимаясь" от листьев к корню.

**При работе этого алгоритма возможны следующие ситуации:**

(1) прочитана вся цепочка; на каждом шаге находилась единственная нужная "свертка"; на последнем шаге свертка произошла к символу S. Это означает, что исходная цепочка a1a2...an⊥ ∈ L(G).

(2) прочитана вся цепочка; на каждом шаге находилась единственная нужная "свертка"; на последнем шаге свертка произошла к символу, отличному от S. Это означает, что исходная цепочка a1a2...an⊥ ∉ L(G).

(3) на некотором шаге не нашлось нужной свертки, т.е. для полученного на предыдущем шаге нетерминала A и расположенного непосредственно справа от него очередного терминала ai исходной цепочки не нашлось нетерминала B, для которого в грамматике было бы правило вывода B → Aai. Это означает, что исходная цепочка a1a2...an⊥ ∉ L(G).

(4) на некотором шаге работы алгоритма оказалось, что есть более одной подходящей свертки, т.е. в грамматике разные нетерминалы имеют правила вывода с одинаковыми правыми частями, и поэтому непонятно, к какому из них производить свертку. Это говорит о недетерминированности разбора.

**Допустим, что разбор на каждом шаге детерминированный.**

Для того, чтобы быстрее находить правило с подходящей правой частью, зафиксируем все возможные свертки (это определяется только грамматикой и не зависит от вида анализируемой цепочки). Это можно сделать в виде таблицы, строки которой помечены нетерминальными символами грамматики, столбцы - терминальными. Значение каждого элемента таблицы - это нетерминальный символ, к которому можно свернуть пару "нетерминал-терминал", которыми помечены соответствующие строка и столбец.

Например, для **грамматики G** = ({a, b, ⊥}, {S, A, B, C}, P, S), такая таблица

будет выглядеть следующим образом:

P: S → C⊥

C → Ab | Ba

A → a | Ca

B → b | Cb

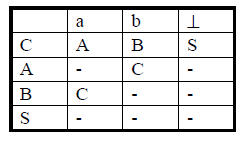


Рис 1 . Таблица для свертки

Знак "-" в таблице для свертки ставится в том случае, если для пары "терминал-нетерминал" свертки нет.

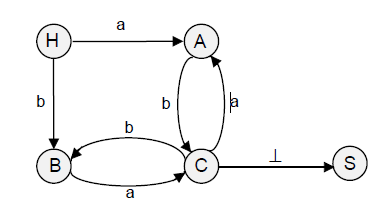
**Но чаще информацию о возможных свертках представляют в виде диаграммы состояний (ДС) - неупорядоченного ориентированного помеченного графа, который строится следующим образом:**

(1) строят вершины графа, помеченные нетерминалами грамматики (для каждого нетерминала - одну вершину), и еще одну вершину, помеченную символом, отличным от нетерминальных (например, H). Эти вершины будем называть состояниями. H - начальное состояние.

(2) соединяем эти состояния дугами по следующим правилам:

a) для каждого правила грамматики вида W → t соединяем дугой состояния H и W (от H к W) и помечаем дугу символом t;

б) для каждого правила W → Vt соединяем дугой состояния V и W (от V к W) и помечаем дугу символом t;

****

**Рис. 2 Диаграмма состояний для грамматики G**

**Алгоритм разбора по диаграмме состояний:**

(1) объявляем текущим состояние H;

(2) затем многократно (до тех пор, пока не считаем признак конца цепочки) выполняем следующие шаги: считываем очередной символ исходной цепочки и переходим из текущего состояния в другое состояние по дуге, помеченной этим символом. Состояние, в которое мы при этом попадаем, становится текущим.

**При работе этого алгоритма возможны следующие ситуации** (аналогичные ситуациям, которые возникают при разборе непосредственно по регулярной грамматике):

* прочитана вся цепочка; на каждом шаге находилась единственная дуга, помеченная очередным символом анализируемой цепочки; в результате последнего перехода оказались в состоянии S. **Это означает, что исходная цепочка принадлежит L(G).**
* прочитана вся цепочка; на каждом шаге находилась единственная "нужная" дуга; в результате последнего шага оказались в состоянии, отличном от S. **Это означает, что исходная цепочка не принадлежит L(G).**
* на некотором шаге не нашлось дуги, выходящей из текущего состояния и помеченной очередным анализируемым символом. **Это означает, что исходная цепочка не принадлежит L(G).**
* на некотором шаге работы алгоритма оказалось, что есть несколько дуг, выходящих из текущего состояния, помеченных очередным анализируемым **символом, но ведущих в разные состояния. Это говорит о недетерминированности разбора. Анализ этой ситуации будет приведен ниже.**

**Диаграмма состояний** **определяет конечный автомат**, построенный по регулярной грамматике, который допускает множество цепочек, составляющих язык, определяемый этой грамматикой. Состояния и дуги Диаграммы состояний - это графическое изображение функции переходов конечного автомата из состояния в состояние приусловии, что очередной анализируемый символ совпадает с символом-меткой дуги.

Среди всех состояний выделяется начальное (считается, что в начальный момент своей работы автомат находится в этом состоянии) и конечное (если автомат завершает работу переходом в это состояние, то анализируемая цепочка им допускается).

**Определение: конечный автомат (КА) - это пятерка (K, T, F, H, S), где**

**K** - конечное множество состояний;

**T** - конечное множество допустимых входных символов;

**F** - отображение множества K × T → K, определяющее поведение автомата;

отображение F часто называют функцией переходов;

**H** ∈ K - начальное состояние;

**S** ∈ K - заключительное состояние (либо конечное множество заключительных состояний).

**F(A, t) = B означает, что из состояния A по входному символу t происходит переход в состояние B.**

Определение: конечный автомат допускает цепочку a1a2...an, если F(H,a1) = A1; F(A1,a2) = A2; . . . ; F(An-2,an-1) = An-1; F(An-1,an) = S, где ai ∈ T, Aj ∈ K, j = 1, 2 , ... ,n-1; i = 1, 2, ... ,n; H - начальное состояние, S - одно из заключительных состояний.

**Определение:** множество цепочек, допускаемых конечным автоматом, составляет определяемый им язык.

***Класс языков, распознаваемый конечными автоматными, совпадает с классом языков, порождаемых автоматными грамматиками и наоборот.***

**Для более удобной работы с диаграммами состояний введем несколько соглашений:**

a) если из одного состояния в другое выходит несколько дуг, помеченных разными символами, то будем изображать одну дугу, помеченную всеми этими символами;

b) непомеченная дуга будет соответствовать переходу при любом символе, кроме тех, которыми помечены другие дуги, выходящие из этого состояния.

c) введем состояние ошибки (ER); переход в это состояние будет означать, что исходная цепочка языку не принадлежит.

По диаграмме состояний легко написать анализатор для регулярной грамматики.

***Например, для грамматики G = ({a,b, ⊥}, {S,A,B,C}, P, S), где***

***P: S → C⊥***

***С → Ab | Ba***

***A → a | Ca***

***B → b | Cb***

**Код анализатора для грамматики G на языке СИ**

#include <stdio.h>

int scan\_G()

{

/\* множество состояний \*/

enum state {H, A, B, C, S, ER};

enum state CS; /\* CS - текущее состояние \*/

FILE \*fp;

int c;

CS = H; /\* текущее состояние = начальное состояние \*/

fp = fopen("data","r"); /\* открываем файл и считываем первый символ \*/

c = fgetc(fp);

do

{

switch(CS)

{

/\* начальное состояние \*/

case H:

if(c == 'a')

{

c = fgetc(fp);

CS = A;

}

else if(c == 'b')

{

c = fgetc(fp);

CS = B;

}

else CS = ER;

break;

/\* состояние А \*/

case A:

if(c == 'b')

{

c = fgetc(fp);

CS = C;

}

else CS = ER;

break;

/\* состояние В \*/

case B:

if(c == 'a')

{

c = fgetc(fp);

CS = C;

}

else CS = ER;

break;

/\* состояние С \*/

case C:

if(c =='a')

{

c = fgetc(fp);

CS = A;

}

else if(c == 'b')

{

c = fgetc(fp);

CS = B;

}

else if(c == '⊥') CS = S;

else CS = ER;

break;

}

}while(CS != S && CS != ER);

if(CS == ER) return -1;

else return 0;

}

**Регулярные выражения**

**Регулярные выражения** представляют собой еще один способ определения регулярных языков. Регулярные выражения **определяют** точно те же **языки**, что и конечные автоматы. Но в отличие от автоматов, они позволяют определять допустимые цепочки **декларативным способом**. Поэтому регулярные выражения используются в качестве входного языка во многих системах, обрабатывающих цепочки символов, например:

* в утилите grep, использующейся в операционной системе GNU/Linux для поиска строк по

заданному регулярному выражению;

* в генераторах лексических анализаторов Lex и Flex как формальное описание лексем

(ключевых слов, идентификаторов, операторов, числовых констант) языка.

**Алгебра регулярных выражений состоит** из констант (ε, ) и переменных для обозначения языков и операторов для задания трех операций: объединение ( + или | ), конкатенация ( . ) и

итерация (\*).

**ПРАВИЛО:** Константы ε и ∅ являются регулярными выражениями, определяющими языки { ε } и ∅, соответственно, т.е. L(ε) = { ε } и L(∅) =∅.

**ПРАВИЛО:** Если а – произвольный символ, то регулярное выражение, определяющее язык {a},

состоит просто из символа a.

**ПРАВИЛО:** **Для группировки** констант и переменных в регулярных выражениях **используются**

**круглые скобки ( ).**

**ПРАВИЛО:** Если E и F – регулярные выражения, **то E | F (или E + F)** – регулярное выражение,

**определяющее объединение языков L(E) и L(F).**

**ПРАВИЛО:** Регулярное выражение, **определяющее конкатенацию языков L(E) и L(F)**, записывается в виде **E.F или просто EF**.

**ПРАВИЛО: Итерация E\*** – это регулярное выражение, определяющее объединение конкатенаций E c самим собой 0 или более раз.

**Пример.** **Напишем регулярное выражение для множества цепочек из чередующихся нулей и единиц.**

Сначала построим регулярное выражение для языка, состоящего из одной-единственной цепочки 01. Затем используем оператор итерации (\*) для того, чтобы построить выражение для всех цепочек вида 0101...01. 0 и 1 — это выражения, обозначающие языки {0} и {1}, соответственно. Если соединить эти два выражения, то получится регулярное выражение 01 для языка х {01}. Как правило, если мы хотим написать выражение для языка, состоящего из одной цепочки, то используем саму эту цепочку как регулярное выражение. Далее, для получения всех цепочек, состоящих из нуля или нескольких вхождений 01, используем регулярное выражение (01)\*. Заметим, что выражение 01 заключается в скобки, чтобы не путать его с выражением 01\*. Причина такой интерпретации состоит в том, что операция **\*** имеет высший приоритет по сравнению с операцией конкатенации, и поэтому аргумент **оператора итерации** выбирается до выполнения любых конкатенаций.

Однако L((01)\*) – не совсем тот язык, который нам нужен. Он включает только те цепочки из чередующихся нулей и единиц, которые начинаются с 0 и заканчиваются 1. Мы должны также учесть возможность того, что вначале стоит 1 и/или в конце 0. Одним из решений является построение еще трех регулярных выражений, описывающих три другие возможности. Итак, (10)\* представляет те чередующиеся цепочки, которые начинаются символом 1 и заканчиваются символом 0, 0(10)\* можно использовать для цепочек, которые начинаются и заканчиваются символом 0, а 1(01)\* – для цепочек, которые и начинаются, и заканчиваются символом 1. Полностью это регулярное выражение имеет следующий вид: (01)\* | (10)\* | 0(10)\* | 1(01)\* Заметим, что оператор | используется для объединения тех четырех языков, которые вместе дают все цепочки, состоящие из чередующихся символов 0 и 1.

**Пример. Целочисленная константа без знака представляет собой 0 или цифру, отличную от нуля, за которой может следовать последовательность других цифр.** Регулярное выражение будет иметь такой вид: 0 | (1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9) (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)\*. Для удобства можно ввести сокращенное обозначение диапазона значений, и тогда вместо перечисления всех цифр от 0 до 9 можно просто записать как [0…9], а все регулярное выражение – 0 | [1…9][0…9]\*.

**Пример. Вещественное число с фиксированной точкой без знака можно записать так:**

(0 | [1…9] [0…9]\*) (ε | . [0…9]\*)

В данном случае регулярное выражение состоит из двух частей: регулярного выражения для целого без знака, за которым может следовать точка и дробная часть или ничего (ε).