#0 Ман по граммаматикам

Алфавит – это непустое конечное множество элементов. Элементы алфавита называются символами.

Цепочка – всякая конечная последовательность символов алфавита.

Если x и y – цепочки, то их *конкатенацией* (или *катенацией*) xy является цепочка, полученная путем дописывания символов цепочки y вслед за символами цепочки x.

Множества цепочек в алфавитах будем обозначать как A, B, C,... .

Произведение AB двух множеств цепочек A и B определяется как $AB = \{xy \mid x \in A, y \in B\}$.

$\{\Lambda\}$ – множество, состоящее из пустого символа Λ.

Степени цепочки x определяются следующим образом:

$x^0 = \Lambda, x^1 = x, x^2 = xx, x^3 = xxx, ...$

Степени алфавита A:

$A^0 = \{\Lambda\}, A^1 = A, A^n = AA^{n-1}, (n > 0)$.

Итерация A^* множества A: $A^* = A^0 \cup A^+$

Усеченная итерация A^+ мнжества A:

$A^+ = A^1 \cup A^2 \cup ... \cup A^n \cup ...$

Продукцией или *правилом* подстановки называется упорядоченная пара (U, x), которая обычно записывается $U ::= x$, где U - символ, x - непустая конечная цепочка символов. U называется *левой частью*, x - *правой частью* продукции.

Далее вместо термина «продукция» будем использовать термин «правило».

Грамматикой $G[x]$ называется конечное, непустое множество правил, z - символ, который должен встретиться в левой части, по крайней мере, одного правила. Он называется *начальным символом*, *аксиомой* или *помеченным символом*.

Все символы, которые встречаются в левых и правых частях правил, образуют *словарь* V, Если из контекста ясно, какой символ является символом z, то вместо $G[z]$ будем писать G.

В заданной грамматике G символы, которые встречаются в левой части называются *нетерминальными* или *синтаксическими* единицами языка. они образуют множество нетерминальных символов V_N. Символы, которые не входят в V_N, называются *терминальными символами* (или *терминалами*). Они образуют V_T.

$V = V_N \cup V_T$

Как правило, нетерминалы будем заключать в угловые скобки <>.

Множество правил $U ::= x, U ::= y, ..., U ::= z$ с одинаковыми левыми частями будем записывать $U ::= x / y / ... / z$.

Пример.

Грамматика **G1 <число>** записывается следующим образом:

<число> ::= <чс>

<чс> ::= <чс><цифра> / <цифра>

<цифра> ::= 0 / 1 / 2 / 3 / 4 / 5 / 6 / 7 / 8 / 9

Эта форма записи называется нормальной формой Бэкуса (НБФ) или формой Бэкуса-Наура.

Пусть G - грамматика. Мы говорим, что цепочка v непосредственно порождает цепочку w, и обозначаем это $v \Rightarrow w$, если для некоторых цепочек x и y можно написать $v = xUy$, $w = xuy$, где $U ::= u$ - правило грамматики G. Мы также говорим, что w *непосредственно выводима из* v или что w *непосредственно редуцируется (приводится) к* v

Цепочки x и y могут, конечно, быть пустыми. Следовательно, для любого правила $U ::= u$ грамматики G имеет место $U \Rightarrow u$.

Говорят, что v *порождает* w или w *приводится к* v, что записывается как $v \Rightarrow +w$, если существует последовательность непосредственных выводов

$v = u0 \Rightarrow u1 \Rightarrow u2 \Rightarrow ... \Rightarrow un = w, n > 0$.

Эта последовательность называется *выводом длины* n. Говорят, что w является словом для v. И, наконец, пишут $v \Rightarrow *w$, если $v \Rightarrow +w$ или $v \Rightarrow w$.

Заметим, что пока в цепочке есть хотя бы один нетерминал, из нее можно вывести новую цепочку.

Терминал (конечный, заключительный) – символ, который не встречается в левой части ни одного из правил.

Пусть $G[z]$ - грамматика. Цепочка x называется *сентенциальной формой*, если x выводится из z, т. е. $z \Rightarrow *x$.

Предложение – это сентенциальная форма, состоящая из терминальных символов.

Язык $L(G[z])$ - это множество предложений.

Хомский впервые в 1956г. описал формальный язык. С тех пор теория формальных языков быстро прогрессировала. Хомский определил четыре основных класса языков в терминах грамматик, являющихся упорядоченной четверкой (V,T,P,z), где V - алфавит, $T \subseteq V$ - алфавит терминальных символов, P - конечный набор правил подстановки, z - начальный символ, $z \in (V-T)$.

Язык, порожденный грамматикой – это множество терминальных цепочек, которые можно вывести из z.

Различие четырех типов грамматик заключается в форме правил подстановки, допустимых в P.

Говорят, что G - это (по Хомскому) грамматика типа 0 или грамматика с фразовой структурой, если правила имеют вид:

$u ::= v$, где $u \in V^+$ и $v \in V^*$

То есть левая часть, может быть тоже последовательностью символов, а правая часть может быть пустой. Языки этого класса могут служить моделью естественных языков.

Грамматика типа 1 (контекстно-чувствительные, или контекстно-зависимые языки) имеют правила подстановки вида:

$xUy ::= xuy$, где $U \in (V-T)$, $u \in V^+$ и $x, y \in V^*$

«Контекстно-чувствительная» отражает тот факт, что U можно заменить на u лишь в контексте $x...y$.

Грамматика называется контекстно-свободной – типа 2 (КС-грамматика), если все ее правила имеют вид:

$U ::= u$, где $U \in (V-T)$, и $u \in V^*$

Грамматика типа 2 является хорошей моделью для языков программирования.

Регулярная грамматика (тип 3 или автоматная грамматика, А-грамматика.) – это грамматика, правила которой имеют вид:

$U ::= N$ или $U ::= WN$, где $N \in T$ а $U, W \in (V-T)$

Регулярные грамматики играют основную роль, как в теории языков, так и в теории автоматов.

В реальных языках программирования отдельные подмножества можно отнести к третьему классу.

Иерархия языков по Хомскому включающая, т.е. все грамматики типа 3 являются грамматиками типа 2, все грамматики типа 1 являются грамматиками типа **О** и т.д. Иерархия грамматик соответствует иерархии языков. Например, если язык можно генерировать с помощью грамматики типа 2, то его называют языком типа 2. Эта иерархия опять включающая. Включения так же справедливы в том, что существуют языки, которые относятся к типу **i**, но не к типу **(i+1)(0<=i=>2)**.

1. Интерпретаторы и компиляторы.

Любую программу, которая переводит произвольный текст на некотором входном языке в текст на другом языке, называют транслятором. В частности, исходным текстом может быть **входная программа**. Транслятор переводит ее в выходную или объектную программу. Программа, полученная после обработки транслятором, либо непосредственно исполняется на машине, либо подвергается обработке другим транслятором.

Обычно процессы трансляции и исполнения программы разделены по времени. Сначала вся программа транслируется, а затем исполняется. Трансляторы, работающие в таком режиме, называются трансляторами **компилирующего** типа. Принцип, альтернативный компилированию, реализуется в программах, обычно называемых **интерпретаторами**, которые незамедлительно выполняют текущие предложения языка программирования.

Назначение компилятора заключается в том, чтобы из исходного кода выработать выполнимый. В некоторых случаях компилятор выдает только код строки (ассемблера) для определенной машины, а окончательная выработка машинного кода осуществляется другой программой, называемой ассемблером. Код ассемблера аналогичен машинному коду, но в нем для команд и адресов используется мнемоника, метки и могут применяться в качестве адресов перехода. Ассемблер выполняет довольно простую задачу. Компилятор должен транслировать сложные конструкции языка высокого уровня в относительно простой машинный код или код сборки соответствующей машины.

2. Основные фазы трансляции

В состав любого компилятора входят три основных компонента:

- лексический анализатор (блок сканирования)
- синтаксический анализатор
- генератор кода машинных команд

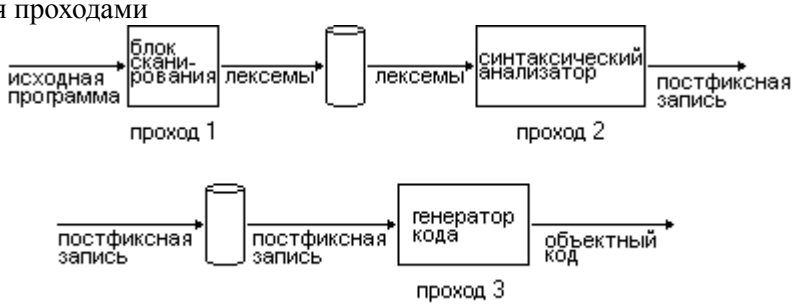
На фазе лексического анализа исходный текст программы разбивается на единицы, называемыми **лексемами**. Такими текстовыми единицами являются слова (например, IF, DO, BEGIN, и др.), имена переменных, константы и знаки операций. Далее эти слова рассматриваются как неделимые образования, а не как группы отдельных символов. После разбиения программы на лексемы следует фаза **синтаксического**

анализа, называемая **грамматическим разбором**, на котором проверяется правильность следования операторов.

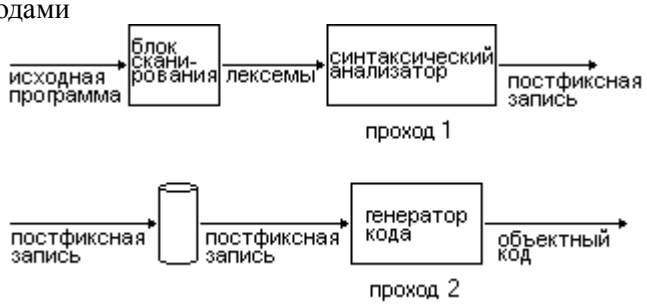
Последним выполняется процесс **генерирования кода**, который использует результаты синтетического анализа и создает программу на машинном языке, пригодную к выполнению.

Взаимодействие трех компонентов может осуществляться различными способами.

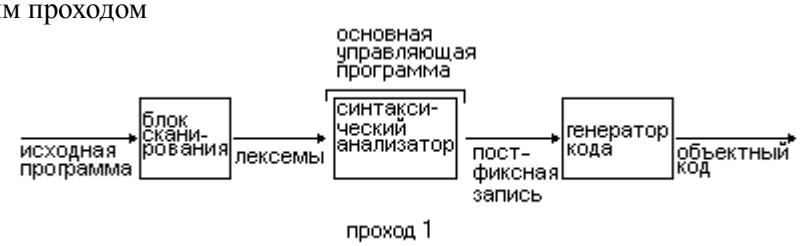
1) компилятор с тремя проходами



2) компилятор с двумя проходами



3) компилятор с одним проходом



В первом варианте блок сканирования считывает исходную программу и представляет ее в виде **файла лексем.** Синтаксический анализатор читает этот файл и выдает новое представление программы в постфиксной форме. Этот файл считывается генератором кода, который создает объектный код программы. Компилятор такого вида называется трехпроходным, так как программа прочитывается трижды. Такая организация не способна придать компилятору высокую скорость выполнения.

Преимущества: относительная независимость каждой фазы компилирования и гибкость компилятора.

Высокая скорость компилирования может быть достигнута при использовании компилятора с однопроходной структурой. В этом случае синтаксический анализатор выступает в роли **основной управляющей программы**, вызывая блок сканирования и генератор кода, организованные в виде подпрограмм. Синтаксический анализатор постоянно обращается к блоку сканирования, получая от него лексему за лексемой до тех пор, пока не построит новый элемент постфиксной записи, после чего он обращается к генератору кода, который создает объектный код для этого фрагмента программы.

Такая программа отличается эффективностью, так как программа просматривается однажды и в выполнении не участвуют операции обращения к файлам. Однако, такой организации свойственны недостатки.

* Неоптимальность создаваемой объектной программы.

* Трудность в решении проблемы перехода по метке.

* Во время обработки предложения «GOTO метка» могут возникнуть осложнения, так как «метка» еще не встречалась в программе.

* Поскольку однопроходной компилятор должен полностью размещаться в памяти, его реализация сопровождается повышенными требованиями к ресурсу памяти.

Структура двухпроходного компилятора занимает промежуточное положение между рассмотренными вариантами. В этом случае синтаксический анализатор, вызывая блок сканирования, получает лексему за лексемой и строит файл постфиксной записи программы.

Преимущества: небольшое время выполнения, легко разрешить проблему перехода по метке и задачу оптимизации программы.

Компилирование программы включает **анализ** - определение предусмотренного результата действия

программы и последующий **синтез -** генерирование эквивалентной программы в машинном коде.

В процессе анализа компилятор должен выяснить, является ли выходная программа недействительной в каком либо смысле (т.е. принадлежит ли она к языку, для которого написан данный компилятор), и если она окажется недействительной - выдать соответствующее сообщение программисту. Этот аспект компиляции называется **обнаружением ошибок.**

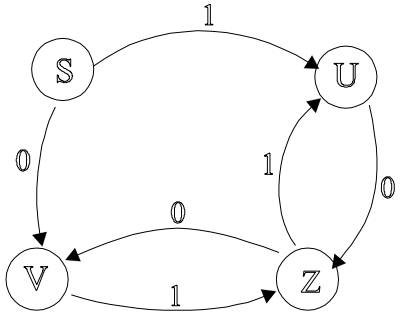
3. Регулярные выражения и конечные автоматы

Диаграммы состояний.

Рассмотрим регулярную грамматику $G[z]$.

Z :: = U0 / V1
U :: = Z1 / 1
V :: = Z0 / 0
L (G) = { B^n | n >0}, где B= {01, 10}

Легко видеть, что порождаемый ею язык состоит из последовательностей, образуемых парами 01 или 10. Чтобы облегчить распознавание предложений грамматики G , нарисуем диаграмму состояний.



В этой диаграмме каждый нетерминал грамматики G представлен узлом или состоянием; кроме того, есть начальное состояние **S**, предполагается, что G не содержит **S.**

Каждому правилу **Q ::= T** в G соответствует дуга с пометкой **T**, направленная от начального состояния **S** к состоянию **Q**, Каждому правилу **Q ::=RT** соответствует дуга с пометкой **T**, направленная от состояния **R** к состоянию **Q.**

Чтобы распознать или разобрать цепочку x , используем диаграмму состояний следующим образом:

1. Первым текущим состоянием считаем начальное состояние. Начинаем с самой левой литеры в цепочке x повторяем шаг 2 до тех пор, пока не будет достигнут правый конец x .

2. Сканируем следующую литеру сроки x , продвигаемся по дуге помеченной этой литерой, переходя к следующему состоянию.

Если при каком-то повторении шага 2 такой дуги не окажется, то цепочка x не является предложением. Если мы достигаем конца x , то x – предложение тогда и только тогда, когда последнее текущее состояние есть Z.

Последовательность действий соответствует алгоритму восходящего разбора. На каждом шаге (кроме первого) основой является имя текущего состояния, за которым следует входной символ. Символ, к которому приводится основа, будет именем следующего состояния.

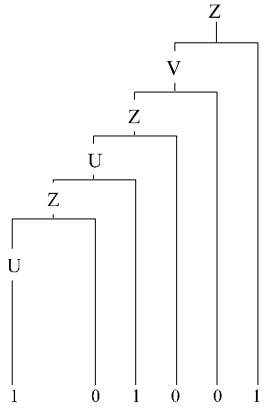
Пример:

Проведем разбор предложения 101001

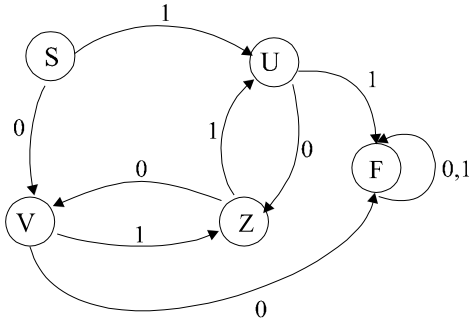
шаг	Текущее состояние	Остаток цепочки x
1	S	101001
2	U	01001
3	Z	1001
4	U	001
5	Z	01
6	V	1
7	Z	

В данном примере разбор выглядит простым благодаря простому характеру правил. Нетерминалы встречаются лишь как первые символы правой части. На первом шаге первый символ предложения всегда приводит к нетерминалу. На каждом последующем шаге первые два символа **UT** сентенциальной формы **UTt** приводят к нетерминалу **V**, при этом используется правило **V ::= UT**. При выполнении этой редукции имя текущего состояния **U**, а имя следующего текущего состояния **V**. Так как каждая правая часть единственна, то единственным оказывается символ, к которому она приводится.

Синтаксические деревья для предложения регулярных грамматик всегда имеют подобный вид:



Чтобы избавиться от проверки на каждом шаге, есть ли дуга с соответствующей пометкой, можно добавить еще одно состояние **F** (НЕУДАЧА) и добавлять все необходимые дуги от всех состояний к **F**. Добавляется так же дуга, помеченная всеми литерами из **F** в **F**.



4. Детерминированный конечный автомат.

Определение: Детерминированный автомат с конечным числом состояний (КА) – это пятерка (K, V_T, M, S, Z).

1) K – алфавит элементов, называемых состояниями;

2) V_T – входной алфавит (литеры, которые могут встретится в цепочке или предложении);

3) M – отображение (или функция) множества $K \acute{} V_T$ в множестве K (если $M(Q,T) = R$, то это значит, что из состояния Q при входной литере T происходит переключение в состояние R);

4) $S \in K$ – начальное состояние;

5) Z – множество заключительных состояний, $Z \in K$.

Можно формально определить, как работает КА с входной цепочкой t. Определим:

$M(Q, L) = Q$ при любом состоянии Q. Если на входе пустой символ, состояние не изменится.

$M(Q, \mathrm{Tt}) = M(M(Q,T),t)$ для всех $t \in V_T$ и $T \in V_T$.

Говорят, что КА допускает цепочку t (цепочка считается допускаемой), если $M(S,t) = P$, где $P \in Z$.

Такие автоматы называются детерминированными, т.к. на каждом шаге входная литера однозначно определяет следующие текущее состояние.

Пример: Рассмотренной ранее диаграмме состояний соответствует КА

M(S,0) = V, M(U,0) = Z, M(F,0) = F, M(S,1) = U,
M(V,0) = F, M(Z,0) = V, M(F,1) = F, M(V,1) = Z,
M(Z,1) = 0, M(V,1) = F.

Если предложение x принадлежит грамматике G, то оно так же допускается КА, соответствующим грамматике G. Для любого КА существует грамматика G, порождающая только те предложения, которые являются цепочками допускаемыми КА.

5. Методы приведения недетерминированного конечного автомата к детерминирован-ному

Если грамматика G содержит два правила с одинаковыми правыми частями, то отображение M оказывается неоднозначным. Автомат, построенный по такой диаграмме, называется недетерминированным конечным автоматом и определяется следующим образом:

Недетерминированным КА (НКА) называется пятерка (K, V_T, M, S, Z), где

1. K – алфавит состояний;
2. V_T – входной алфавит;
3. M – отображение $K \times V_T$ в подмножество множества K;

4. $S \subseteq K$ – множество начальных состояний;

5. $Z \subseteq K$ – множество заключительных состояний.

Отличия:

1. Отображение M дает не единственное, а (возможно пустое) множество состояний

2. Может быть несколько начальных состояний.

Как и ранее, $M(Q,\Lambda) = \{Q\}$, $M(Q,Tt)$ есть объединение множеств $M(P,t)$, где $P \in M(Q,T)$.

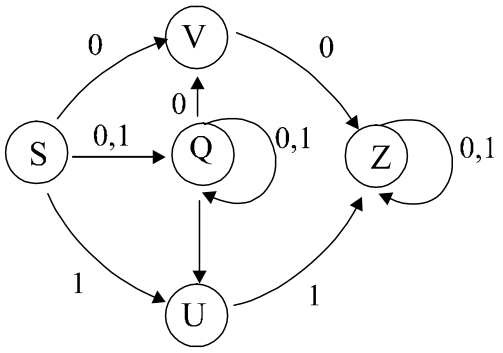
Цепочка t допускается автоматом, если найдется состояние P, такое, что $P \in M(S,t)$ и $P \in Z$.

Пример:

Рассмотрим регулярную грамматику $G[z]$:

Z :: = U1 | V0 | Z0 | Z1
U :: = Q1 | 1
V :: = Q0 | 0
Q ::= Q0 | Q1 | 0 | 1

Диаграмма состояний и ее НКА имеют вид:



NKAF = ({S,Q,V,U,Z}, {0,1}, M, {S}, {Z})

M(S,0)={V,Q}, M(S,1)={U,Q}, M(V,0)={Z}, M(V,1)={ Λ }=$\varnothing$, . . .

Состояние НЕУДАЧА представлено подмножеством $\varnothing$.

Покажем способ построения КА из НКА, при котором как бы параллельно проверяются все возможные пути разбора и отбрасываются тупиковые. Если в НКА имеются, к примеру, выбор из трех состояний x, y, z, то в КА будет одно состояние $[xyz]$, которое представляет все три.

Теорема.

Пусть НКА F =(K ,V_T , M , S , Z) допускает множество цепочек L. Определим КА F'=(K' ,V_T , M' , S' , Z') следующим образом:

1. Алфавит состояний K' состоит из всех подмножеств множества K. Обозначим элемент множества K' через $[s_1, s_2, ..., s_i]$, где $s_1, s_2, ..., s_i \in K$. Положим, что $\{s_1, s_2\} = \{s_2, s_1\} = [s_1, s_2]$.

2. Множества входных литер V_T для F и F' одни и те же.

3. Отображение M' определим как:

$M'([s_1, s_2, ..., s_i], T) = [r_1, r_2, ..., r_j]$, где $M(\{s_1, s_2, ..., s_i\}, T) = \{r_1, r_2, ..., r_j\}$.

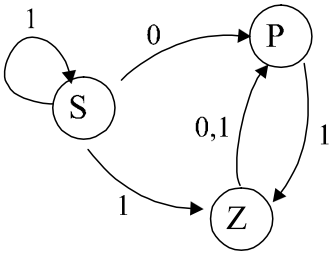
4. Пусть $S = \{s_1, s_2, ..., s_n\}$, тогда $S' = [s_1, s_2, ..., s_n]$.

5. Пусть $Z = \{s_j, s_k, ..., s_l\}$, тогда $Z' = [s_j, s_k, ..., s_l]$.

Утверждается, что F и F' допускают одно и тоже множество цепочек.

Пример:

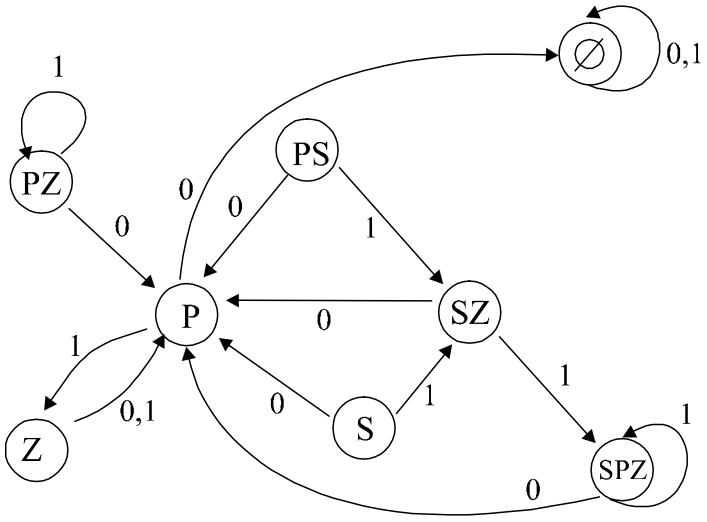
НКА



Начальное состояние S.

Заключительное состояние Z.

КА



Начальное состояние S или PS .

Заключительное состояние Z , ZP , SZ , SPZ .

Состояния PS и PZ можно исключить, т.к. нет путей, к ним ведущих. Построенный автомат не является минимальным, возможно построить автомат с меньшим числом состояний.

Для построения КА из НКА воспользуемся следующим приемом.

Строим матрицу переходов для НКА.

	0	**1**
S	**P**	**S ,Z**
P		**Z**
Z	**P**	**P**

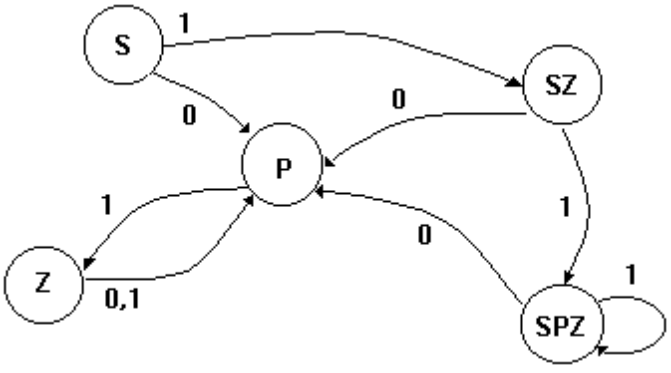
Вводим новое состояние **SZ**

	0	**1**
S	**P**	**S Z**
S Z	**P**	**S Z, P**
P		**Z**
Z	**P**	**P**

Вводим еще одно новое состояние **SPZ**

	0	**1**
S	**P**	**S Z**
S Z	**P**	**S PZ**
S ZP SPZ	**P**	**S PZ**
P		**Z**
Z	**P**	**P**

Больше неоднозначных переходов нет, строим граф КА.



6. Автоматная грамматика

Регулярная грамматика (тип 3 или автоматная грамматика, А-грамматика.) – это грамматика, правила которой имеют вид:

$U ::= N$ или $U ::= WN$, где $N \in T$ а $U, W \in (V - T)$

Регулярные грамматики играют основную роль, как в теории языков, так и в теории автоматов.

Регулярная грамматика Материал из Википедии — свободной энциклопедии

В информатике, регулярная грамматика — формальная грамматика типа 3 по иерархии Хомского. Регулярные грамматики определяют в точности все регулярные языки, и поэтому эквивалентны конечным автоматам и регулярным выражениям. Регулярные грамматики являются подмножеством контекстно-свободных.

Задание набором правил

Регулярная грамматика может быть задана набором правил как левая или правая регулярная грамматика.

правая регулярная грамматика - все правила могут быть в одной из следующих форм:

A → a
A → aB
A → ε

левая регулярная грамматика - все правила могут быть в одной из следующих форм:

A → a
A → Ba
A → ε

где

заглавные буквы (A, B) обозначают нетерминалы из множества N
строчные буквы (a, b) обозначают терминалы из множества Σ
ε - пустая строка, т.е. строка длины 0

Классы правых и левых регулярных грамматик эквивалентны - каждый в отдельности достаточен для задания всех регулярных языков. Любая регулярная грамматика может быть преобразована из левой в правую, и наоборот.

Пример

Правая регулярная грамматика G, заданная N = {S, A}, Σ = {a, b, c}, P состоит из следующих правил:

S → aS
S → bA
A → ε
A → cA

и S является начальным символом. Эта грамматика описывает тот же язык, что и регулярное выражение a*bc*.

Ограниченность

Любая контекстно-свободная грамматика может быть легко преобразована в вид, в котором правила состоят только из лево-регулярных или право-регулярных (для контекстно-свободных грамматик допустимо наличие тех и других одновременно). Следовательно, такие грамматики могут выразить все контекстно-свободные языки. Регулярные грамматики могут содержать либо лево-регулярные правила, либо право-регулярные, но не оба вида одновременно. Поэтому они могут описать лишь подмножество языков, называемых регулярными языками.

Например, контекстно-свободный язык строк вида aibi, i≥0 задается грамматикой G, где N = {S, A}, Σ = {a, b}, P состоит из правил

S → aA
A → Sb
S → ε

и S является начальным символом. Обратите внимание на то, что данная грамматика содержит одновременно лево-регулярные и право-регулярные правила, и следовательно не является регулярной.

Классификация грамматик Материал из Википедии — свободной энциклопедии

Согласно Хомскому, формальные грамматики делятся на четыре типа. Для отнесения грамматики к тому или иному типу необходимо соответствие *всех* её правил (продукций) некоторым схемам.

Тип 0 — неограниченные

Грамматика с фразовой структурой G — это алгебраическая структура, упорядоченная четвёрка (V_T, V_N, P, S), где:

- V_T — *алфавит* (множество) терминальных символов — *терминалов*,
- V_N — *алфавит* (множество) нетерминальных символов — *нетерминалов*,

- $V = V_T \cup V_N$ — *словарь* G, причём $V_T \cap V_N = \emptyset$
- P — *конечное множество продукций* (правил) грамматики, $P \subseteq V^+ \times V^*$
- S — *начальный символ* (*источник*).

Здесь V^* — множество всех строк над алфавитом V, а V^+ — множество непустых строк над алфавитом V.

К типу 0 по классификации Хомского относятся неограниченные грамматики — грамматики с фразовой структурой, то есть все без исключения формальные грамматики. Правила можно записать в виде:

$\alpha \to \beta$,

где α — любая непустая цепочка, содержащая хотя бы один нетерминальный символ, а β — любая цепочка символов из алфавита.

Практического применения в силу своей сложности такие грамматики не имеют.

Тип 1 — контекстно-зависимые

К этому типу относятся контекстно-зависимые (КЗ) грамматики и неукорачивающие грамматики. Для грамматики $G(V_T,V_N,P, S)$, $V=V_T \cup V_N$ все правила имеют вид:

- $\alpha A \beta \to \alpha \gamma \beta$, где $\alpha, \beta \in V^*$, $\gamma \in V^+$, $A \in V_N$. Такие грамматики относят к контекстно-зависимым.
- $\alpha \to \beta$, где $\alpha, \beta \in V^+$, $1 \le |\alpha| \le |\beta|$. Такие грамматики относят к неукорачивающим.

Эти классы грамматик эквивалентны. Могут использоваться при анализе текстов на естественных языках, однако при построении компиляторов практически не используются в силу своей сложности. Для контекстно-зависимых грамматик доказано утверждение: по некоторому алгоритму за конечное число шагов можно установить, принадлежит цепочка терминальных символов данному языку или нет.

Тип 2 — контекстно-свободные

К этому типу относятся контекстно-свободные (КС) грамматики. Для грамматики $G(V_T,V_N,P, S)$, $V=V_T \cup V_N$ все правила имеют вид:

- $A \to \beta$, где $\beta \in V^+$ (для неукорачивающих КС-грамматик, $\beta \in V^*$ для укорачивающих), $A \in V_N$. То есть грамматика допускает появление в левой части правила только нетерминального символа.

КС-грамматики широко применяются для описания синтаксиса компьютерных языков.

Тип 3 — регулярные

К третьему типу относятся регулярные грамматики (автоматные) — самые простые из формальных грамматик. Они являются контекстно-свободными, но с ограниченными возможностями.

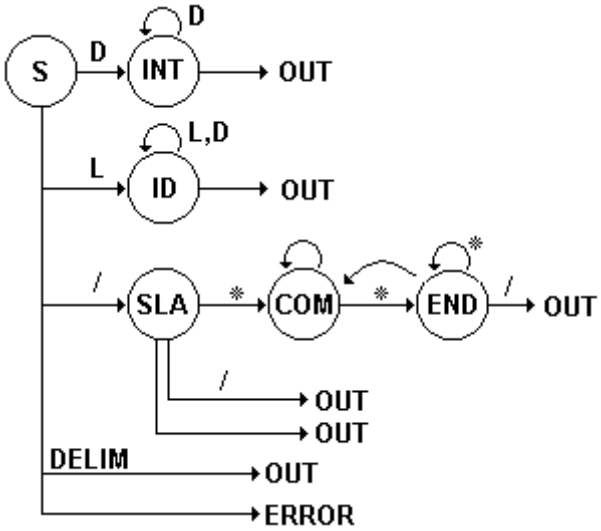
Все регулярные грамматики могут быть разделены на два эквивалентных класса, которые для грамматики вида III будут иметь правила следующего вида:

- $A \to B\gamma$ или $A \to \gamma$, где $\gamma \in V_T^*$, $A, B \in V_N$ (для леволинейных грамматик).
- $A \to \gamma B$; или $A \to \gamma$, где $\gamma \in V_T^*$, $A, B \in V_N$ (для праволинейных грамматик).

Регулярные грамматики применяются для описания простейших конструкций: идентификаторов, строк, констант, а также языков ассемблера, командных процессоров и др.

7. Диаграмма состояний сканера.

Начнем реализацию сканера с того, что нарисуем диаграмму состояний, показывающую, как ведется разбор символа.



Метка D используется вместо любой из меток 0,1,2....., 9, т.е. D представляет класс цифр (для упрощения диаграммы)

Метка L представляет класс букв A, B,.....Z.

DELIM представляет класс разделителей состоящих из одной литеры +, -, *, (,). Заметим, что литера / не принадлежит к классу разделителей, т.к. обрабатывается особым образом.

Непомеченные дуги будут выбраны, если сканируемая (обозреваемая) литера не совпадает ни с одной из литер, которыми помечены другие дуги.

Дуги, ведущие в OUT говорят от том, что обнаружен конец символа и необходимо покинуть сканер. Одна из проблем, которая возникает при переходе в OUT, состоит в том, что в этот момент не всегда сканируется литера, следующая за распознанным символом. Так, литера выбрана при переходе в OUT из INT, но она еще не выбрана, если только что распознан разделитель (DELIM). Когда потребуется следующий символ, необходимо знать, сканировалась уже его первая литера ил нет. Это можно сделать, используя, например переключатель . В дальнейшем будем считать, что перед выходом из сканера следующая литера всегда выбрана.

Замечание.

Мы составили детерминированную диаграмму, т.е. мы сами определили, что начинается с литеры / : SL (/), SLSL (//) или комментарий, поскольку всегда идем в общее для них состояние SLA.

Теперь перейдем к рассмотрению диаграммы состояний с семантическими процедурами.

Для работы сканера потребуются следующие переменные и подпрограммы:

1. CHARACTER CHAR, где CHAR – глобальная переменная, значение которой всегда будет сканируемая литера исходной программы.

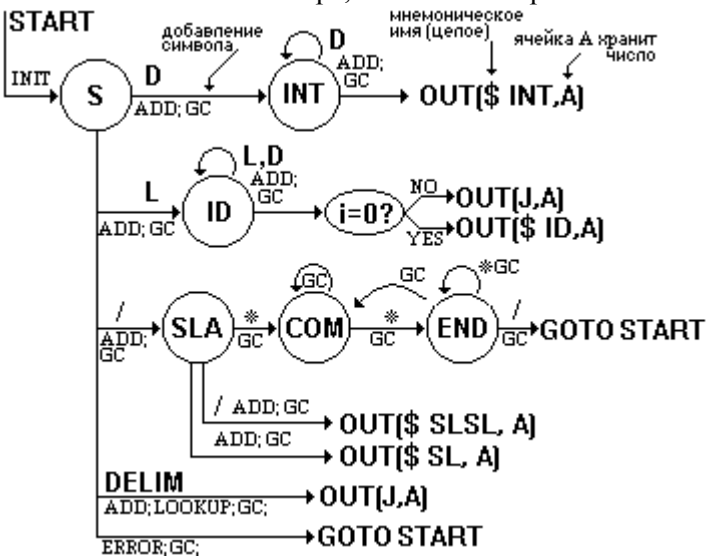
2. INEGER CLASS, где CLASS – содержит целое число, которое характеризует класс литеры находящейся в CHAR. Будем считать, что класс D (цифра)=1, класс L (буква)=2, класс, содержащий литеру / , =3, класс DELIM=4.

3.STRING A – ячейка, которая будет содержать цепочку (строку) литер, составляющих символ.

4. GETCHAR – процедура, задача которой состоит в том, чтобы выбрать следующую литеру исходной программы и поместить ее в CHAR, а класс литеры в CLASS. Кроме того, GETCHAR, когда это необходимо, будет читать и печать следующую строку исходной программы и выполнять другие подобные мелочи.

Гораздо предпочтительнее использовать для таких целей отдельную процедуру, чем повторять группу команд в тех местах, где необходимо сканировать литеру.

5. GETNOBLANK. Эта программа проверяет содержимое CHAR и если там пробел, то повторно вызывает GETCHAR до тех пор, пока в CHAR не окажется литера, отличная от пробела.



GC – вызов процедуры GETCHAR.

Выражение OUT(C,D) означает возврат к программе, которая вызвала сканер, с двумя величинами C и D в качестве результата.

Под первой дугой, ведущей к начальному состоянию S, записана команда INIT, которая указывает на необходимость выполнения последовательных операций и начальных установок (инициализация).

В данном случае выполняются команды:

GETNOBLANK;

A:= ' ';

Тем самым в CHAR заносится литера отличная от пробела, и производится начальная установка ячейки A, в которой будет хранится символ. Команда ADD означает, что литера CHAR добавляется к символу в A.

Команда LOOCUP осуществляет поиск символа набранного в A, по таблице служебных слов и разделителей. Если символ является служебным словом или разделителем, то соответствующий индекс заносится в глобальную переменную J, в противном случае туда записывается 0.

При обнаружении комментария или ошибки мы не выходим из сканера, а начинаем сканировать следующий символ.

Пример.

Последим за разбором символа // . Начнем с инициализации (INIT). Затем переходим в состояние S. Проверяем CHAR, находим, что там /, и по соответствующей дуге переходим в состояние SLA. В момент перехода добавляем / в A (команда ADD) и вызываем GETCHAR, чтобы сканировать следующую литеру в CHAR. и выбираем дугу, помеченную /. В момент перехода по дуге добавляем литеру из CHAR к цепочке в A и сканируем следующую литеру (GC), к программе, заканчиваем работу возвратом к программе, которая вызвала сканер, с парой величин

К моменту выхода литера следующего символа находится в CHAR.

Процедура имеет два параметра – внутренне представление получаемого символа, второй цепочка литер, составляющих символ.