Классическая теория компиляторов

Лекция 2

ТЕОРИЯ ФОРМАЛЬНЫХ ЯЗЫКОВ. ГРАММАТИКИ

Фраза. Конечная последовательность слов языка. Фразы необходимо уметь строить и распознавать.

- ***Язык*** *L* – это множество цепочек конечной длины в алфавите Σ.

Способы задания языка

1. Полное перечисление
2. Описание языка

Пример.
A = {a, b, c}.

A*={e, a, b, c, aa, ab, ac, bb, bc, cc, …}.
e – пустая последовательность.

- Если $\sum$ – множество (алфавит или словарь), то $\sum^*$ – замыкание множества $\sum$ (*свободный моноид*, порожденный $\sum$), т.е. множество всех конечных последовательностей, составленных из элементов множества $\sum$, включая и *пустую* последовательность.
- $\sum+$ - *положительное замыкание* множества $\sum$ (*свободная полугруппу*, порожденную множеством $\sum$). В отличие от $\sum^*$ в $\sum+$ не входит e.

Грамматика

Грамматика - способ описания языка.

G = (N, Σ,P,S), где

- N – конечное множество нетерминальных символов (синтаксические переменные или синтаксические категории);
- Σ – конечное множество терминальных символов (слов) ($\Sigma \cap N = \varnothing$);
- P – конечное подмножество множества

$$(N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$$

- Элемент (α, β) множества P называется *правилом* или *продукцией* и записывается в виде $\alpha \to \beta$;
- S – выделенный символ из N ($S \in N$), называемый *начальным символом* ("начальная переменная" или "исходный символ").

Пример:

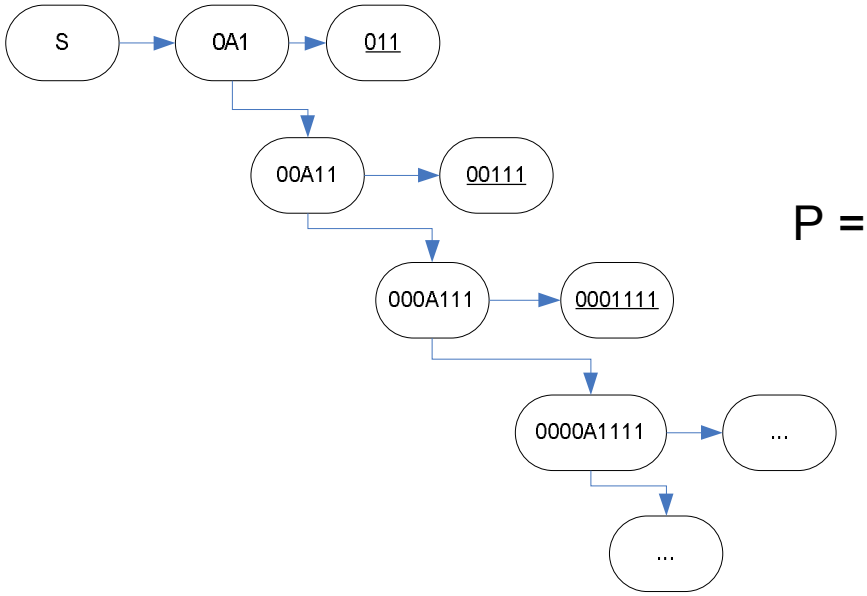
G = ({A,S},{0,1},P,S),

P = {S→0A1, 0A→00A1, A→1}

Задание языка

- *Язык, порождаемый грамматикой G* (обозначим его через L(G)) – это множество терминальных цепочек, порожденных грамматикой G.
- Язык L, порождаемый грамматикой, есть множество последовательностей (слов), которые:
  - состоят лишь из терминальных символов;
  - можно породить, начиная с символа S.

$$\mathbf{L(G)=\{\omega \mid \omega \in \Sigma^*, S \Rightarrow^* \omega\}}$$



$P = \{S \rightarrow 0A1,\ 0A \rightarrow 00A1,\ A \rightarrow 1\}$

Нормальная форма Бэкуса-Наура

НФБ (БНФ) служит для описания правил грамматики. Впервые применена Науром при описании синтаксиса языка Фортран.

БНФ является альтернативной, более упрощенной, менее строгой и потому более распространенной формой записи грамматики.

Пример:

- <число> ::= <чс>
- <чс> ::= <чс><цифра>|<цифра>
- <цифра> ::= 0|1|...|9

Расширенная БНФ

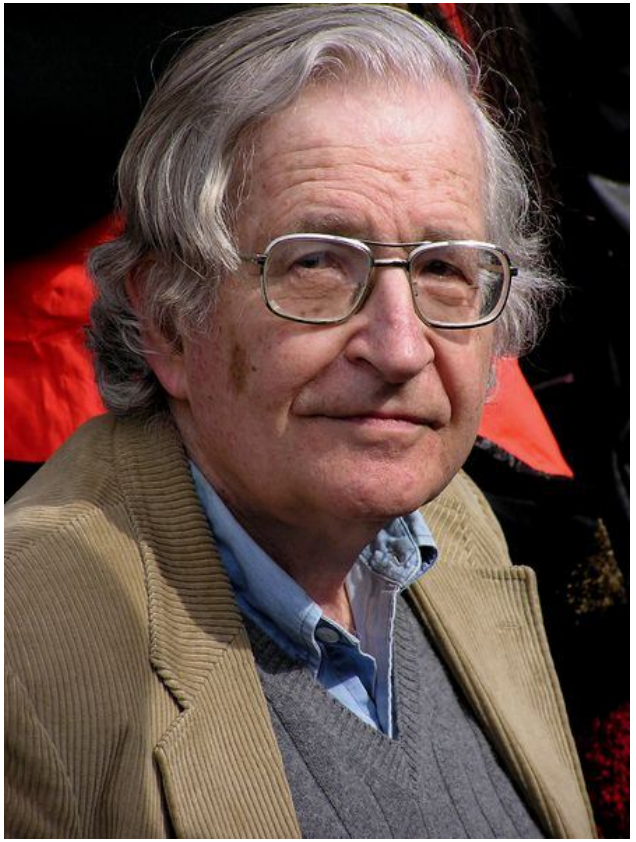
конкатенация, выбор, условное вхождение и повторение.

- Число = ["+"|"-"]НатЧисло["."[НатЧисло]][("e"|"E")["+"|"-"]НатЧисло].
- НатЧисло = Цифра{Цифра}.
- Цифра = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".

Идент = Буква{Буква|Цифра|"_"}.

ИЕРАРХИЯ ХОМСКОГО

- **Аврам Ноам Хомский** (*Avram Noam Chomsky), 1928 г.р.*
- МТИ
- Лингвист, политический публицист, философ.
- Его работы о порождающих грамматиках внесли значительный вклад в упадок бихевиоризма и содействовали развитию **когнитивных** наук.
- Известен своими радикально-левыми политическими взглядами, а также критикой внешней политики правительств США.
- Либертарный социалист и сторонник анархо-синдикализма.



«Синтаксические структуры», 1957

Виды грамматик

P: $(N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$

- *Грамматика типа 0*: Правила этой грамматики определяются в самом общем виде.

  P: $xUy \rightarrow z$

  Для распознавания языков, порожденных этими грамматиками, используются т.н. машины Тьюринга – очень мощные (на практике практически неприменимые) математические модели.
- *Грамматика типа 1*: Контекстно-зависимые (чувствительные к контексту)

  P: $xUy \rightarrow xuy$
- *Грамматика типа 2*: Контекстно-свободные. Распознаются стековыми автоматами (автоматами с магазинной памятью)

  P: $U \rightarrow u$
- *Грамматика типа 3*: Регулярные грамматики. Распознаются конечными автоматами

  P: $U \rightarrow a$ или $U \rightarrow aA$

$u \in (N \cup \Sigma)+$

$x, y, z \in (N \cup \Sigma)^*$

$A, U \in N$

$a \in \Sigma$

РЕГУЛЯРНЫЕ ГРАММАТИКИ

Пример 1. Идентификатор (в форме БНФ)

<идент> ::= <бкв>
<идент> ::= <идент> <бкв>
<идент> ::= <идент> <цфр>
<бкв> ::= a | b | ... | z
<цфр> ::= 0 | 1 | ... | 9

Пример 2. Арифметическое выражение (без скобок)

G = (N, ∑, P, S)
N={S, E, op, i}
∑={<число>, <идент>, +, -, *, /}
P={S → i
S → i E
E → op S
op → +
op → -
op → *
op → /
i → <число>
i → <идент>}

То же в виде БНФ:

<expr> ::= <число>|<идент>
<expr> ::= <expr><op><expr>
<op> ::= +|-|*|/
<число> ::= <цфр>
<число> ::= <число><цфр>

Пример 3:

Z ::= U0 | V1
U ::= Z1 | 1
V ::= Z0 | 0

Порождаемый язык состоит из последовательностей, образуемых парами 01 или 10, т.е.

$L(G) = \{ B^n \mid n > 0\}$,

где B = { 01, 10}.

Арифметическое выражение (со скобками)

G0=({E,T,F},{a,+,*,(,),#},P,E)

P = { E → E+T|T
T → T*F|F
F → (E)|a }

Это – уже КСГ

КОНЕЧНЫЕ АВТОМАТЫ

- Автомат – это формальная *воспринимающая* система (или акцептор). Правила автомата определяют принадлежность входной формы данному языку, т.е. автомат – это система, которая распознает принадлежность фразы к тому или иному языку.
- Автомат *эквивалентен данной грамматике*, если он воспринимает весь порождаемый ею язык и только этот язык.

Определение. Конечный автомат (КА) – это пятерка

КА = $(\sum, Q, q_0, T, P)$, где

$\sum$ – входной алфавит (конечное множество, называемое также входным словарем);

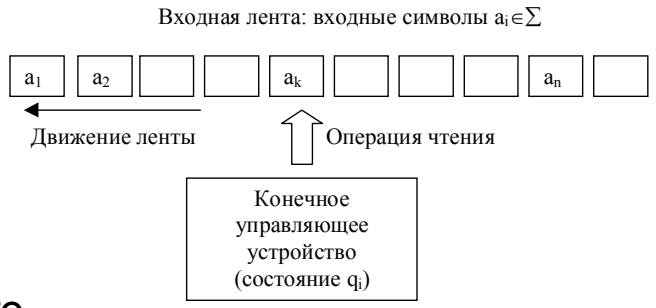
Q – конечное множество состояний;

q_0 – начальное состояние ($q_0 \in Q$);

T – множество терминальных (заключительных) состояний, $T \subset Q$;

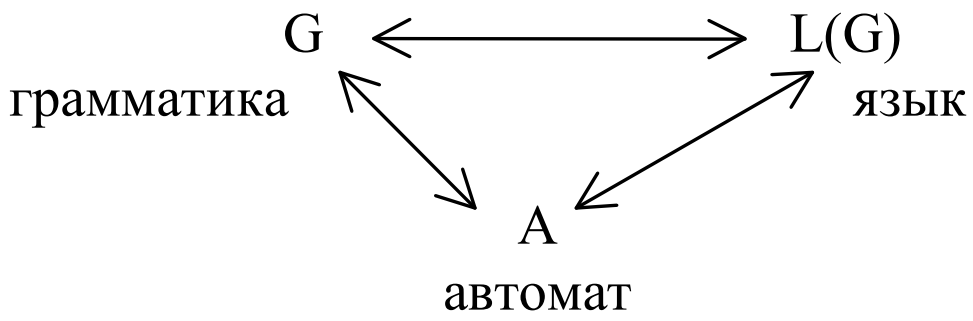
P – подмножество отображения вида $Q \times \sum \to Q$, называемое функцией переходов. Элементы этого отображения называются *правилами* и обозначаются как

$q_i a_k \to q_j$, где q_i и q_j – состояния, a_k – входной символ: $q_i, q_j \in Q$, $a_k \in \sum$.



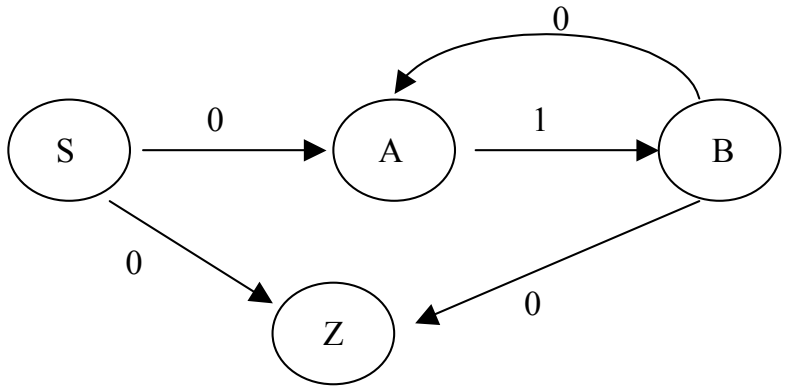
Автоматы и грамматика

Каждой грамматике можно поставить в соответствие эквивалентный ей автомат, и каждому автомату соответствует эквивалентная ему грамматика.



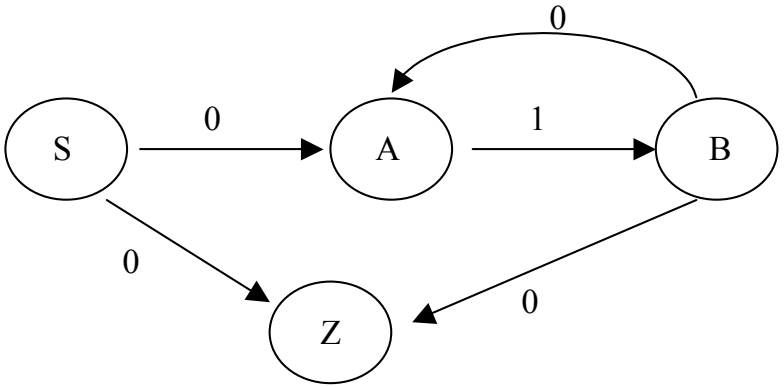
Задание автомата

$A = (\Sigma, Q, q_0, T, P)$
$\Sigma = \{0, 1\}$,
$Q = \{S, A, B, Z\}$,
$q_0 = S$,
$T = \{Z\}$,
$P = \{S0 \rightarrow Z, S0 \rightarrow A, A1 \rightarrow B, B0 \rightarrow Z, B0 \rightarrow A\}$



ДКА и НКА

- *Конфигурация конечного автомата* – это элемент множества $Q \times \sum^*$, т.е. последовательность вида $q\omega$, где $q \in Q$, $\omega \in \sum^*$.
- Если к любой конфигурации $q_i\omega$ применимо не более одного правила, то такой автомат называется ДКА. Иначе - НКА.
- НКА:
  - неоднозначность переходов;
  - наличием, в общем случае, более чем одного начальных состояний.



Пример.

S0→Z, **S0**→A, A1→B, **B0**→Z, **B0**→A

Реализация автомата

Пример 2. "Идентификатор"

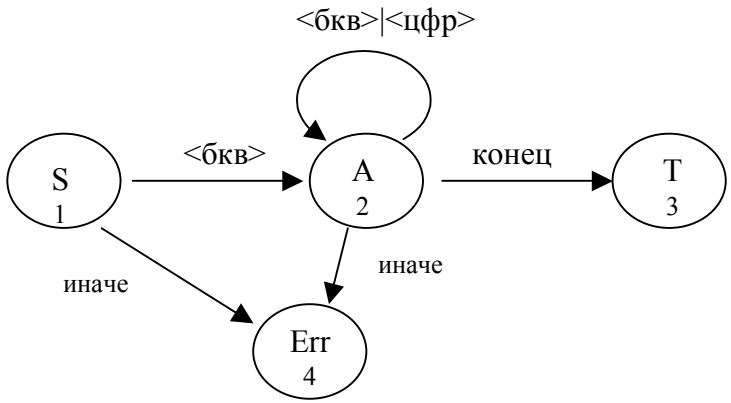
<идент> ::= <бкв>

<идент> ::= <идент> <бкв>

<идент> ::= <идент> <цфр>

<бкв> ::= a | b | ... | z

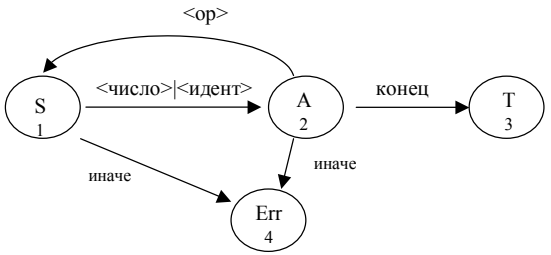
<цфр> ::= 0 | 1 | ... | 9



P:

	1	2	3
<бкв>	2	2	-
<цфр>	4	2	-
<конец>	4	3	-
<иначе>	4	-	-

Арифметическое выражение (без скобок)




```
char c;       //текущий исходный символ
int q;        //номер состояния
int a;        //входной текущий символ для автомата
q=0;          //начальное состояние автомата
while(1)      //бесконечный цикл
{      c = readchar();       //считывание входного символа
a = gettype(c);      //распознавание входного символа –
                             //отнесение его к одной из известных автомату
                             //категорий - <бкв>, <цфр>, <конец> или <иначе>
//Выполнение перехода
q = P[a, q];
//Обработка
if (q==3) return 1;  //нормальный выход из программы
if (q==4) return 0;  //выход по ошибке
}
```

ПОСТРОЕНИЕ ДКА ПО НКА

Для любого НКА можно построить эквивалентный ему конечный детерминированный автомат.

Теорема. Пусть НКА F= ($\sum$,Q,q0,T,P) допускает множество цепочек L. Определим ДКА F'= ($\sum$',Q',q0',T',P') следующим образом:

- Множество состояний Q' состоит из всех подмножеств множества Q.
  Q' = {[S1,S2,..,Sl]}, где S1,S2,..,Sl$\in$Q.
- $\sum$' = $\sum$.
- Отображение P' определяется как
  P'([S1,S2,..,Sm],x) = [R1,R2,..,Rm],
  где P({S1,S2,..,Sm },x) = { R1,R2,..,Rm }, Si,Ri$\in$Q, x$\in\sum$.
- Пусть q0={S1,S2,..,Sk}.
  Тогда q0'=[S1,S2,..,Sk].
- Пусть T={Sj, Sk,.., Sn}.
  Тогда T' = {t=[Sa,Sb,..,Sc] | $\exists$ Sb: Sb$\in$T}.
- Построенный таким образом ДКА будет эквивалентен в смысле "вход-выход" исходному НКА.

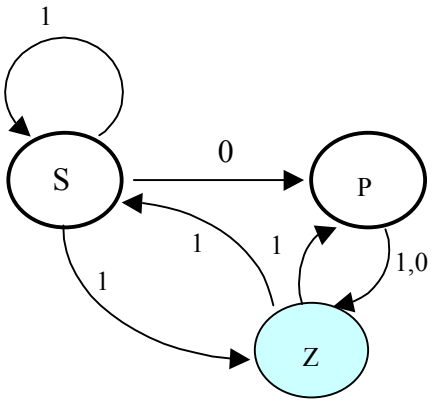
Пример

НКА

P = {S1→S, S1→Z,
S0→P, P1→Z,
P0→Z, Z1→P,
Z1→S}.

q_0 = {S, P}.

T = {Z}.



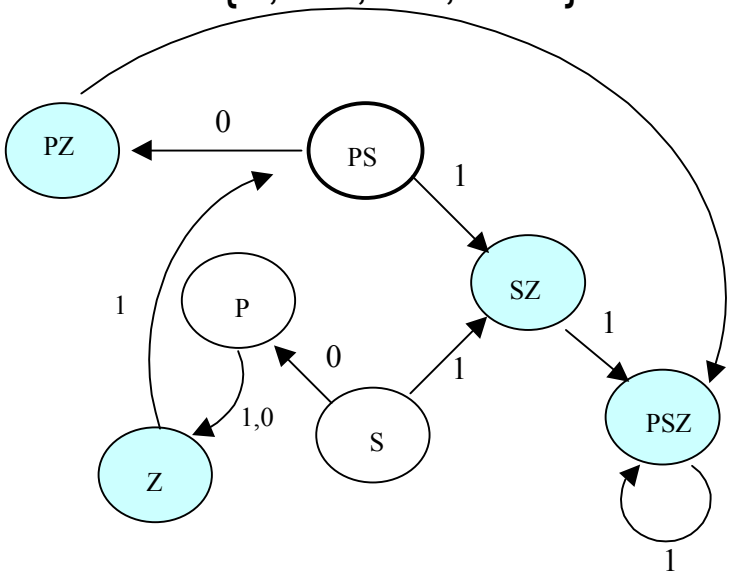
ДКА

Q = {S, P, Z, PS, SZ, PSZ, PZ} (2^{n-1} шт.).

P ={S1→SZ, S0→P, P1→Z, P0→Z,
Z1→PS, PS1→SZ, PS0→PZ,
SZ1→PSZ, PSZ1→PSZ,
PZ1→PSZ}.

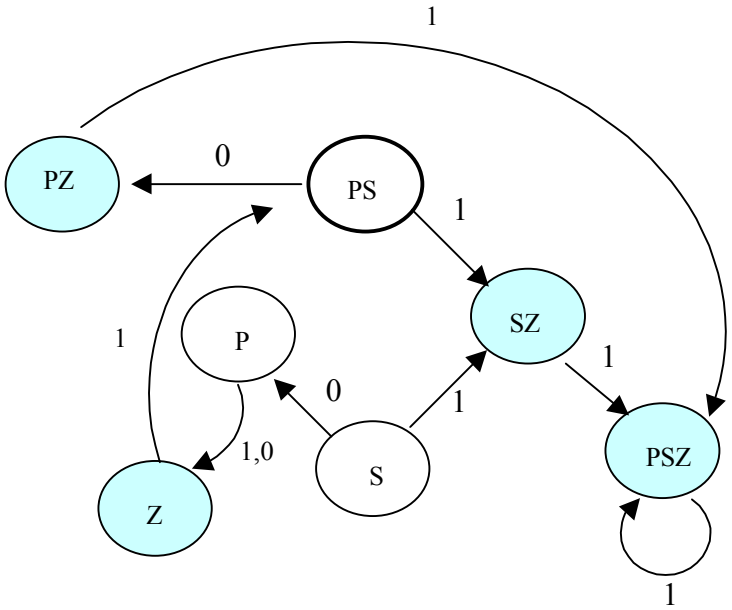
q_0 = SP.

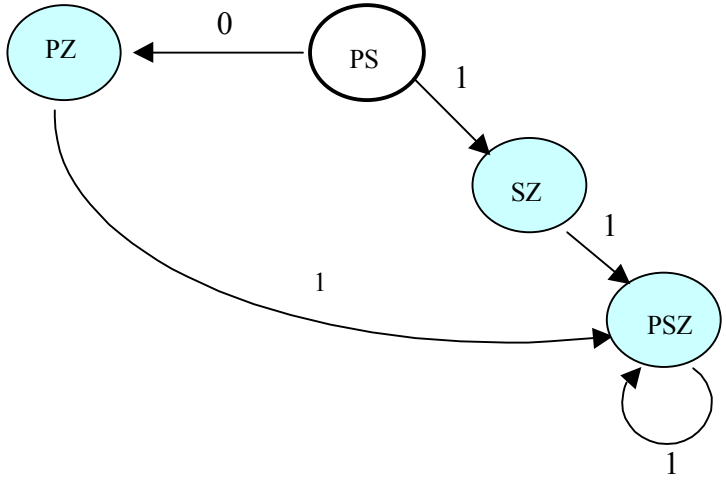
T = {**Z**, P**Z**, S**Z**, PS**Z**}.



Упрощение автомата

- Устранение недостижимых вершин



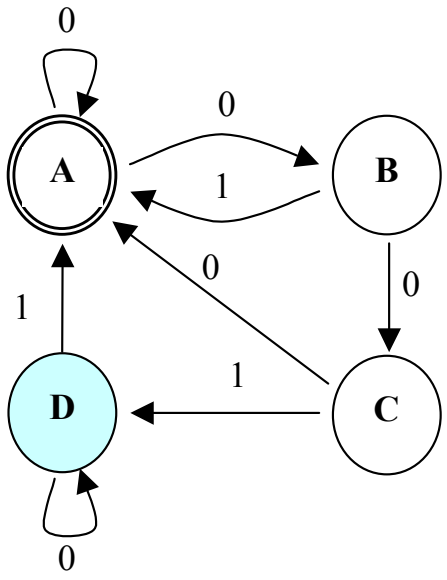


Замечание о начальных состояниях в НКА.
Когда имеется несколько н.с., работа автомата заключается в том, что переход по входному символу осуществляется одновременно *из всех* начальных состояний.
В этом - суть процедуры объединения всех н.с. в одно.

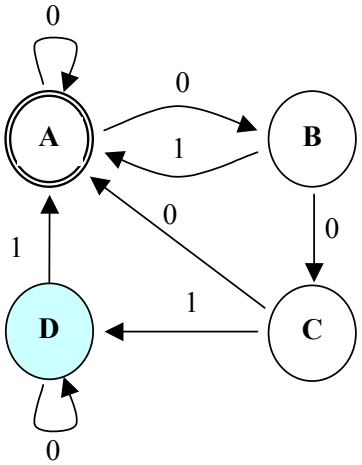
Конечные состояния

Смысл к.с. заключается в определении условия завершения работы автомата.

1. Работа автомата может быть завершена при попадании его в одно из заключительных состояний (такие состояния назовем поглощающими). Это - ПЛА.
2. Более сложное условие завершения работы: работа автомата заканчивается тогда, когда входная последовательность исчерпана и при этом автомат находится в одном из терминальных состояний. Это - НЛА.



Условия завершения работы



№	Правила перехода	Правила грамматики
1	A0→A	A→0A
2	A1→B	A→1B
3	B0→C	B→0C
4	B1→A	B→1A
5	C0→A	C→0A
6	C1→D	C→1**D**
7	D0→D	D→0**D**
8	D1→A	D→1A

Эти правила грамматики не могут породить язык.

В них отсутствуют *терминальные подстановки*

Для $A_{ПЛА}$, (попадание в T):

- Правило №6: C→1**D** преобразуется в терминальную подстановку C→1 (т.к. D∈T)

Для $A_{НЛА}$: добавление правил подстановки.

Добавляются все терминальные правила, т.е. правила, в правой части которых отбрасываются терминальные состояния:

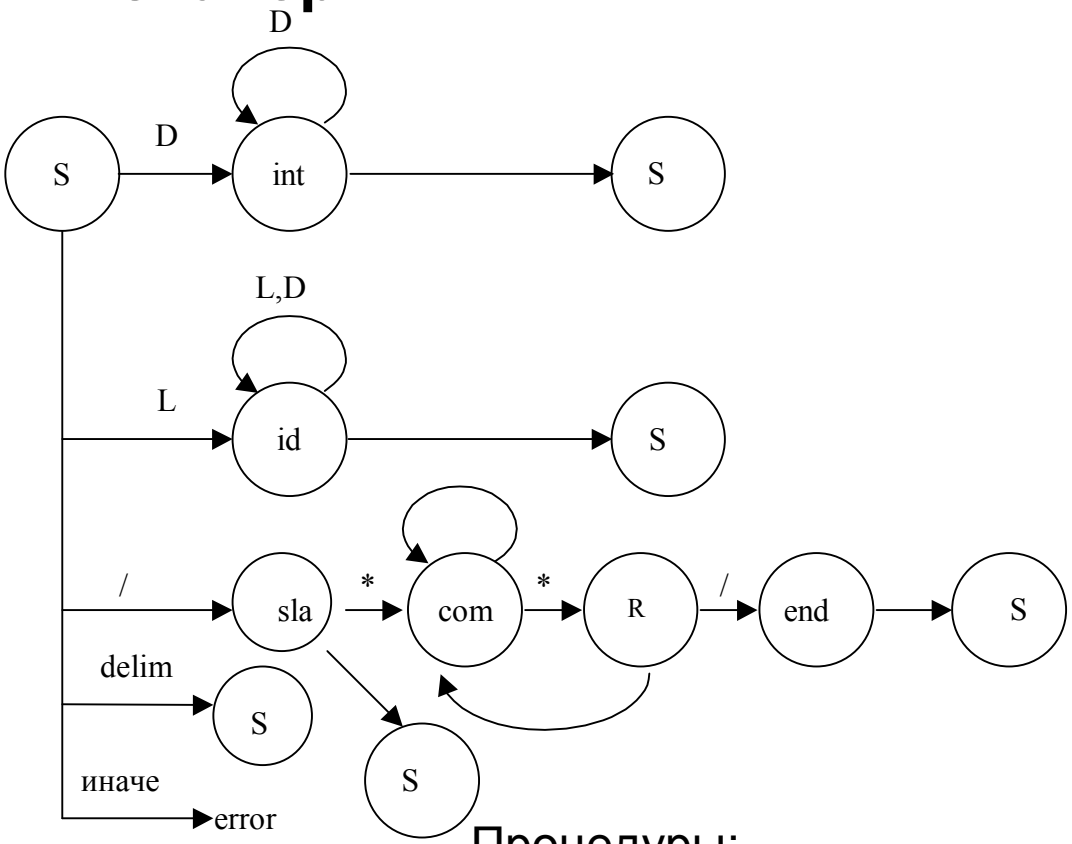
C→1 (из правила №6) и

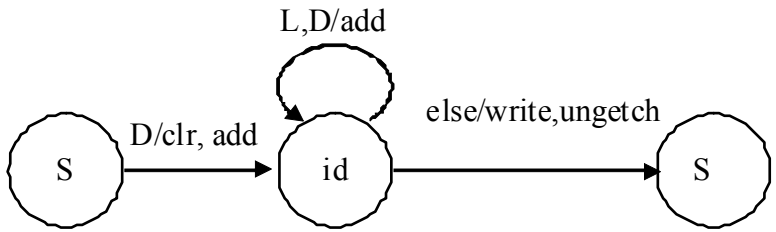
D→0 (из правила №7).

Сканер

Особенности автомата

- При переходе в S иногда требуется возвращать обратно считываемый символ
- Автомат в таком виде бесполезен, он не выполняет никаких полезных процедур.
- Автомат необходимо дополнить *процедурной* частью.





Процедуры:

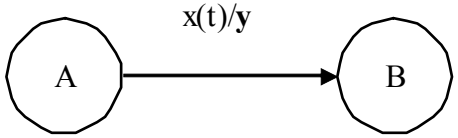
- Clr
- Add
- Ungetch
- Write
- ...

Автоматы, выполняющие действия

- Автомат Мили

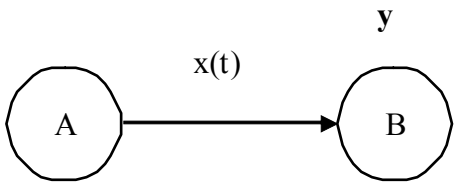
  Функция выходов λ: $Q \times X \rightarrow Y$

  $y(t) = \lambda(q(t), x(t))$



- Автомат Мура

  $y(t) = \mu(q(t))$



Все не так просто

- Проблемы диагностики на этапе ЛА
  - Баланс скобок
  - Типы лексем
  - ...
- Области видимости лексем
- Метки

```
1.    …
2.    int A=1, B=0;
3.    …
4.    void f(int A)
5.    {
6.    …
7.       X=A+1;
8.       B = X;
9.     …
10.   }
```

ТАБЛИЦА СИМВОЛОВ

- Полнота информации
- Быстрый поиск

№	Имя	вид	тип	размер	значение	уровень	…
1	ABC	идент.				0	
2	"qwerty"	константа	строка	6		0	
…	…	…					
N	103	константа	число	2	103	2	

ХЕШ-ФУНКЦИИ

Задача: по имени найти местоположение элемента

- Поиск в неупорядоченном массиве. N/2 сравнений
- Поиск в упорядоченном массиве. log N сравнений

ХЕШ-АДРЕСАЦИЯ

- Идеальный вариант – уметь по имени сразу определить адрес элемента.
- “Хорошая" процедура поиска - та, которая сможет определить хотя бы приблизительный адрес элемента.
- Хеш-адресация – это метод преобразования имени в индекс элемента в таблице.

  H(NAME)->address
- Простейший вариант хеш-функции – это использование внутреннего представления первой литеры имени.

H(“Пушкин»)

Коллизии

- Проблемы начинаются тогда, когда результаты хеширования совпадают для двух и более различных имен. Эта ситуация называется *коллизией*.
- Хорошая хеш-функция распределяет адреса равномерно, так что коллизии возникают не слишком часто.

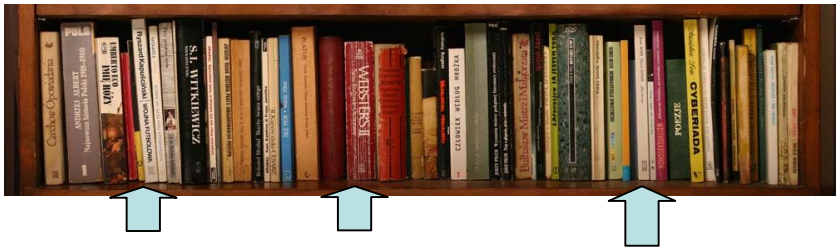
H(«Пушкин»)

РЕХЕШИРОВАНИЕ

- Пусть хешируется имя S и при этом обнаруживается коллизия.
- Пусть h - значение хеш-функции.
- Сравниваем S с элементом по адресу

  $(h+p_1)$ mod N, где N – длина таблицы.
- Если опять возникает коллизия, то выбираем для сравнения элемент с адресом $(h+p_2)$ mod N и т.д.
- Это будет продолжаться до тех пор, пока не встретится элемент $(h+p_i)$ mod N, который либо пуст, либо содержит S, либо снова является элементом h (p_i=0, и тогда считается, что таблица полна).

Способы рехеширования заключаются в выборе чисел $p_1,p_2,..,p_n$.



Коэффициент загрузки таблицы lf:

lf = n/N,

- где n – текущее количество элементов в таблице, N – максимально возможное число элементов.

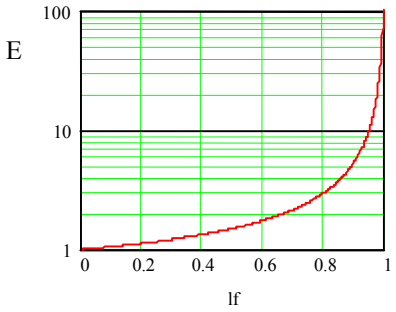
Линейное рехеширование

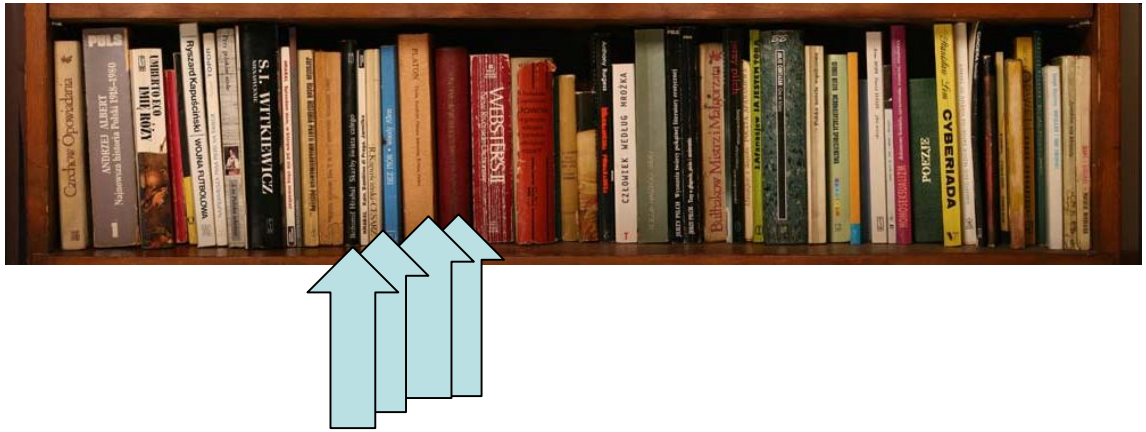
- $p_i = i$ (т.е. $p_1=1$, $p_2=2$,..,$p_n=n$).

Среднее число сравнений E при коэффициенте загрузки lf

$$E(lf) = (1-lf/2)/(1-lf)$$

- E(0.1)=1.06
- E(0.5)=1.5
- E(0.9)= 5.5





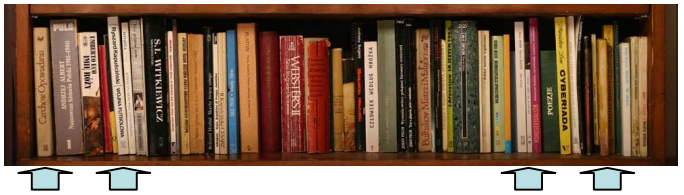
Прочие методы рехеширования

Случайное рехеширование

Пусть максимальное число элементов в таблице кратно степени двойки, т.е. $N=2^k$.

Вычисление p_i:

1. R := 1
2. Вычисляем p_i:
3. R := R*5
4. выделяем младшие k+2 разрядов R и помещаем их в R
5. p_i := R>>1 (сдвигаем R вправо на 1 разряд)
6. Перейти к П.2



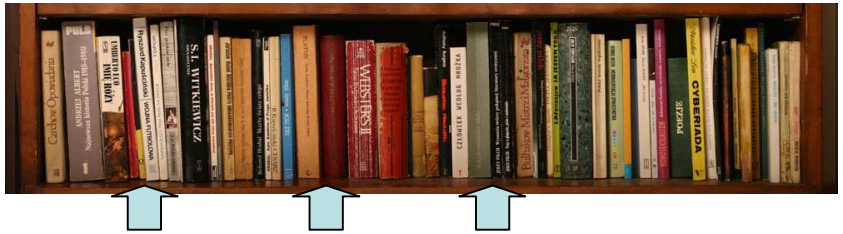
4 1 3 2

Среднее число сравнений E при коэффициенте загрузки lf

- E = -(1/lf)log2(1-lf)

Рехеширование сложением

- pi = i(h+1), где h – исходный хеш-индекс.
- Этот метод хорош для N, являющихся простыми числами.



1 2 3

ХЕШ-ФУНКЦИИ

Пусть имя - множество литер (символов)

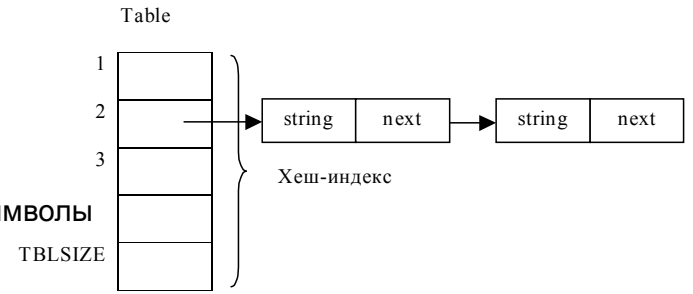
1. Берется S', являющееся результатом суммирования литер из исходного символа S (либо простое суммирование, либо поразрядное исключающее ИЛИ):
2. Вычисляется окончательный индекс. При этом возможны самые различные варианты. В частности:
3. Умножить S' на себя и использовать n средних битов в качестве значения h (для $N=2^n$);
4. Использовать какую-либо логическую операцию (например, исключающее ИЛИ) над некоторыми частями S';
5. Если $N=2^n$, то разбить S' на n частей и просуммировать их. Использовать n крайних правых битов результата;
6. Разделить S' на длину таблицы, и остаток использовать в качестве хеш-индекса.

Хорошая хеш-функция – это та, которая дает как можно более равномерное и случайное распределение индексов по именам.

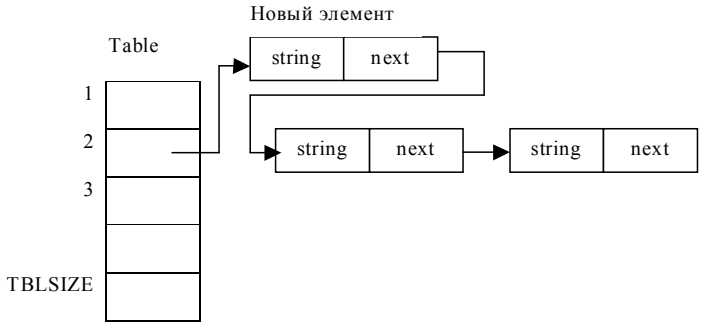
Иными словами, для "похожих" или "близких" имен хеш-функция должна выдавать сильно разнящиеся индексы, т.е. не допускает группировки имен в таблице символов.

Пример программы

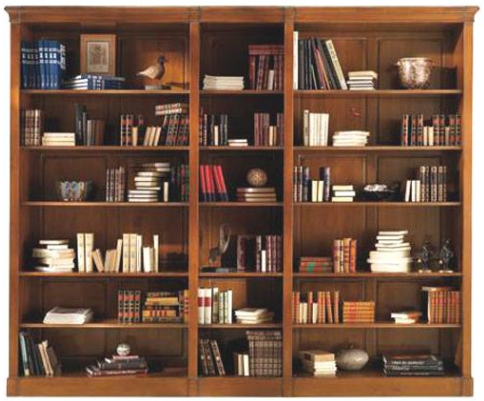
```
struct name                  //элемент таблицы символов
{       char *string;        //имя
        name *next;          //ссылка на следующий элемент
        double value;        //значение
};
const TBLSIZE = 23; // Количество "ящиков", по которым раскладываем символы
name *table[TBLSIZE];
name *findname(char *str, int ins)
// Функция поиска в таблице имени name
// Если ins!=0, то происходит добавление элемента в таблицу
{  int hi = 0;      //Это, фактически, наш хеш-индекс
   char *pp = str;
   // Суммируем по исключающему ИЛИ каждый символ строки.
   // Далее сдвигаем для того, чтобы индекс был лучше
   // (исключаем использование только одного байта)
   while (*pp)
   { hi<<=1; // На самом деле здесь лучше применить циклический сдвиг
     hi^= *pp++;
   }
   if (hi<0) hi = -hi;
   hi %= TBLSIZE; //Берем остаток
   //Нашли список, теперь используем метод линейного рехеширования
   for(name *n=table[hi];n;n=n->next)
      if(strcmp(str,n->string)==0) return n; //Нашли. Возвращаем адрес
   if(ins==0) return NULL; //Ничего не нашли
   //Добавляем имя
   name *nn = new name;
   nn->string = new char[strlen(str)+1];
   strcpy(nn->string, str);
   nn->value = 0;
   nn->next = table[hi];
   table[hi] = nn;
   return nn;
}
```



Исходное состояние



Процесс добавления нового элемента



Заключение о хеш-функциях

- БД. Пользовательские функции хеширования
- Зависимость качества хеширования от корпуса текстов
- Равномерность заполнения хеш-списков