7. Формальные языки и формальные грамматики

Рассмотрим третий способ описания языков программирования. Этот раздел подготовлен в основном по главе 3 книги [1], оттуда же заимствованы некоторые примеры.

7.1. Формальные языки

Основными объектами теории формальных языков являются цепочки символов.

Алфавит (иногда также говорят **словарь**) — непустое множество Σ, **символы** (иногда **слова**) $a, b, c, \ldots$ — его элементы.

Цепочки (над алфавитом) — конечные упорядоченные последовательности символов.

Пример.
Алфавит $\Sigma = \{a, b, c\}$ и цепочки $\varepsilon, a, b, c, aa, ab, ac, ba, bb, bc, \ldots, babcca, \ldots$

Пустая цепочка обозначается ε. Длина цепочки — число содержащихся в ней символов: $|babcca| = 6$. Множество всех возможных цепочек над алфавитом Σ обозначается Σ^*.

Для цепочек символов естественным образом определена бинарная операция **конкатенации** (склеивания): $\alpha, \beta \in \Sigma^* \mapsto \alpha\beta \in \Sigma^*$. По определению $\alpha^n = \alpha\alpha\ldots\alpha$ и $a^n = aa\ldots a$. Иногда вместо $\alpha\beta$ пишут $\alpha \cdot \beta$.

Язык L **над словарем** Σ — некоторое множество цепочек символов, то есть подмножество множества Σ^*.

Существует и более абстрактное определение формального языка.

Полугруппа S — непустое множество с ассоциативной бинарной операцией:
$\forall s_1, s_2 \in S \mapsto s_1 \cdot s_2 \in S \mid (s_1 \cdot s_2) \cdot s_3 = s_1 \cdot (s_2 \cdot s_3)$.
(В общем случае $\cdot$ — не обязательно конкатенация.)

В полугруппе могут быть единица и нуль, то есть такие элементы, что ...

Пусть A — непустое множество. Множество A^* всех конечных упорядоченных последовательностей элементов из A с ассоциативной бинарной операцией называют **свободной полугруппой**.

Язык L — это подмножество свободной некоммутативной полугруппы A^*.

Такое определение неконструктивно: трудно ответить на вопрос, принадлежит ли некоторая цепочка символов тому или иному языку.

Используются различные **способы описания формальных языков**. Если язык состоит из конечного числа цепочек, то можно просто составить их полный список.

1. **Словесное описание** — перечисление свойств цепочек, принадлежащих данному языку.

2. **Алгебраическое описание** — указания, как с помощью алгебраических правил конструируются цепочки символов.

3. **Порождающие правила** — набор инструкций, по которым исходя из некоторого начального множества символов (может быть, только одного) строятся все цепочки языка. Этот способ наиболее распространен при описании языков программирования.

4. **Алгоритм распознавания** — последовательность действий, с помощью которых может быть проведен анализ цепочки символов и, как следствие, выясняется, принадлежит эта цепочка языку или нет. Удобно рассматривать некоторое распознающее устройство (автомат), которое переходит из одного состояния в другое в зависимости от того, какой символ из анализируемой цепочки поступает на его вход.

Примеры.

$\Sigma = \{a, b, c\}$. Все цепочки из трех символов этого алфавита легко выписать.

Натуральные числа: $1, 2, 3, 4, \ldots, 12403, \ldots$ Ясно, что в алфавите содержится десять арабских цифр, но первой цифрой цепочки не может быть нуль.

Цепочки из нулей и единиц, начинающиеся с единицы.

Слова из букв, входящих в заданное слово. (Студенты-математики на скучных лекциях играют в другие игры :-).

$\Sigma = \{a, b\}$; $L = \{a^k b^l a^m b^n \mid 0 \le k \le l \le m \le n\}$ — простой пример алгебраического описания языка.

$\Sigma = \{(,)\}$; L — множество цепочек, в которых для каждой левой скобки найдется правая скобка.

Множество правильных арифметических выражений. Здесь, конечно, нужно четко сформулировать, что такое арифметическое выражение и когда оно является правильным.

Пусть $\Sigma = \{a, b, c, \ldots 0, 1, 2, \ldots, begin, end, for, to, do, read, write, \ldots\}$. Язык программирования Паскаль (или ему подобный) можно задать, если перечислить правила записи основных конструкций языка со всеми необходимыми ограничениями. Полное словесное описание такого языка получится достаточно длинным.

Азбука Морзе. Або, бессарабка, вавилон, голова, догадка, ...

фао, вео, фа, ве, ве, ... — это цепочка символов из инструкции по сборке некоего устройства (кто был на лекции, тот знает, о чем идет речь).

Для описания языков программирования, как уже было сказано, удобно использовать порождающие правила. Металингвистические формы нотации Бэкуса-Наура фактически и являются такими правилами. Например,

<программа на Паскале> ::= [<заголовок>] [<объявления >] **begin** [<инструкции>] **end**.
<заголовок> ::= **program** <идентификатор> ;
.....
<тип объекта> ::= <имя типа> = **object** [(<имя другого типа>)] {<секция описания>} **end ;**
<секция описания> ::= [**private** | **protected** | **public**] [<поля>] [<методы>] [<свойства>]
<поля> ::= {<имена полей> **:** <тип поля> **;**}
<имена полей> ::= [{<идентификатор> **,** }] <идентификатор>
.....

7.2. Формальные грамматики

Грамматика языка — это набор средств для его описания. Различают **порождающие** и **распознающие** грамматики — это зависит от того, какая задача ставится на первый план: строить новые цепочки символов или определять, содержится некоторая цепочка в данном языке или нет.

Формальную грамматику $G = (T, N, S, P)$ образуют четыре компонента:

терминальный словарь T, содержащий символы языка $a, b, c, \ldots$;

нетерминальный словарь N, содержащий вспомогательные символы $A, B, C, \ldots$;

начальный символ S из множества N;

множество P **правил вывода** вида $\alpha \to \beta, \;\; \alpha, \beta \in V^*$, где $V = T \cup N$.

Греческими буквами обозначаются цепочки символов — терминальных и нетерминальных. При описании формальных языков (в том числе языков программирования) правила вывода часто сводятся к следующему: из цепочки α выводится цепочка β в результате замены некоторого нетерминального символа на какую-то другую цепочку символов. Этот принцип наглядно показан в предыдущем примере. В самом первом правиле вывода начальный нетерминальный символ грамматики языка <программа на Паскале> (напомним, что нетерминальным символом может быть длинное выражение на метаязыке) заменяется на правую часть БНФ. Нетерминальный символ <заголовок> во втором правиле вывода заменяется на ...и так далее.

Итак, исходя из начального символа с помощью правил вывода могут быть получены различные цепочки символов (терминальных и нетерминальных). Такие цепочки называют **выводимыми** в данной грамматике (или сентециальными формами). Сентециальные формы из терминальных символов называют предложениями. Предложения и образуют язык над грамматикой. Таким образом, **формальный язык** $L(G)$ — это множество всех цепочек из терминальных символов, выводимых в грамматике G.

Например, пусть $T = \{a, b\}, \;\; N = \{S\}, \;\; S = S, \;\; P = \{S \to ab;\; \gamma \to a\gamma b \;\; \forall \gamma \in V^*\}$. Легко видеть, что в языке, порождаемом такой грамматикой, содержатся предложения ab, $aabb$, $aaabbb$, ...

7.3. Классификация формальных грамматик

Ноам Хомский, известный американский лингвист, предложил в 1956 году ввести следующую классификацию формальных грамматик $G = (T, N, S, P)$.

Грамматики типа 0 (или **грамматики общего вида**) характеризуются тем, что правила вывода в них имеют вид $\alpha \to \beta$, где $\alpha \in V^*NV^*, \;\; \beta \in V^*$ (при ограничении $\alpha \neq \varepsilon$). Считается, что такие грамматики слишком свободны для описания языков программирования, но в то же время недостаточно мощны для описания естественных языков.

Грамматики типа 1 (или **контекстно-зависимые грамматики**) имеют правила вывода вида $\gamma_1 A \gamma_2 \to \gamma_1 \beta \gamma_2$, где $\gamma_1, \gamma_2 \in V^*$, $A \in N, \;\; \beta \in V^*$ (при ограничении $\beta \neq \varepsilon$). В этом случае замена A на β возможна только в контексте $\gamma_1 \gamma 2$. КЗ-грамматики хороши тем, что для них доказано утверждение: по некоторому алгоритму за конечное чис-

ло шагов можно установить, принадлежит цепочка терминальных символов данному языку или нет.

Грамматики типа 2 (или **контекстно-свободные грамматики**) наиболее широко применяются при описании языков программирования. Для них правила вывода имеют вид $A \to \beta$, где $A \in N,\ \beta \in V^*$. Нотация БНФ представляет собой список таких правил. Классический пример (который в дальнейшем будет использоваться в различных модификациях): $\{+, *, (,), i\},\ \{E, T, P\},\ E,\ \{E \to E + T,\ E \to T,\ T \to T * P,\ T \to P,\ P \to i,\ P \to (E)\}$. Эта грамматика позволяет строить арифметические скобочные выражения с операциями типа сложения и умножения, символ i рассматривается как некоторый идентификатор.

Грамматики типа 3 (или **автоматные** (**регулярные**) **грамматики**) отличаются тем, что правила вывода в них имеют вид $A \to \varepsilon,\ A \to a,\ A \to aB$, где $A, B \in N,\ a \in T$. Такие грамматики являются контекстно-свободными, но с ограниченными возможностями.

7.4. Представление вывода цепочек символов

Для цепочек символов вводится **отношение вывода**: $\alpha \Rightarrow \beta$ (из цепочки α выводится цепочка β).

Иногда в такие обозначения вносятся дополнительные элементы:
$\alpha \Rightarrow_G \beta$ — цепочка β выводится из цепочки α в грамматике G;
$\alpha \Rightarrow_{(m)} \beta$ — с помощью правила вывода с номером m (размеченный вывод);
$\alpha \Rightarrow^* \beta$ — β выводится из α каким-то способом;
$\alpha \Rightarrow^+ \beta$ — β выводится из α нетривиально, за конечное число шагов (≥ 1);
$\alpha \Rightarrow^k \beta$ — вывод длины k, то есть при использовании k правил вывода
и так далее.

Следовательно, формальный язык, порожденный грамматикой G,
$L(G) = \{\alpha \in T^* \mid S \Rightarrow^*_G \alpha\}$.

Пример. Пусть $T = \{a, b, c\},\ \ N = \{S\},\ \ S$ — начальный символ и
$P = \{(1)\ S \to aSa,\ \ (2)\ S \to bSb,\ \ (3)\ S \to c\}$.

В языке, порождаемом такой грамматикой, содержатся цепочки символов:
$S \Rightarrow_{(3)} c$,
$S \Rightarrow_{(1)} aSa \Rightarrow_{(3)} aca$,
$S \Rightarrow_{(1)} aSa \Rightarrow_{(1)} aaSaa \Rightarrow_{(3)} aacaa$,
$S \Rightarrow_{(2)} bSb \Rightarrow_{(1)} baSab \Rightarrow_{(2)} babSbab \Rightarrow_{(3)} babcbab$ и так далее.

Часто используется графическое представление вывода цепочек символов в грамматиках. **Дерево вывода** цепочки символов в КС-грамматике — это размеченное упорядоченное дерево, каждая вершина которого помечена символом из множества $T \cup N \cup \{\varepsilon\}$. При этом корень помечен начальным символом, внутренние узлы — нетерминальными символами, а листья — терминальными символами.

Пример. Рассмотрим множество правил вывода в грамматике некоторого примитивного языка программирования.

ПР → begin СО end
СО → ОП | ОП ; СО

ОП → ИД := АВ | for ИД from КО to КО do СО endfor | if УС then СО endif | write АВ
УС → АВ ЗН АВ
ЗН → = | <> | > | < | >= | <=
АВ → ИД | КО | АВ + АВ | АВ - АВ | АВ * АВ | АВ / АВ | (АВ) | read
ИД → БУ | ИД БУ | ИД ЦИ
КО → ЦИ | КО ЦИ
БУ → a | b | c | .....
ЦИ → 0 | 1 | 2 | .....

Нетрудно догадаться, что пары прописных букв обозначают нетерминальные символы: <программа>, <список операторов>, <оператор>, <идентификатор>, <арифметическое выражение>, <константа>, <условие> ... Смысл операторов языка программирования тоже легко понять.

Пусть поставлена **задача**: найти наибольшее из десяти натуральных чисел.

Составим **программу**, реализующую простой алгоритм поиска наибольшего числа, как цепочку терминальных символов. Для удобства чтения отдельные конструкции будем разделять пробелами или переносить на следующую строку (но это вовсе не обязательно с формальной точки зрения).

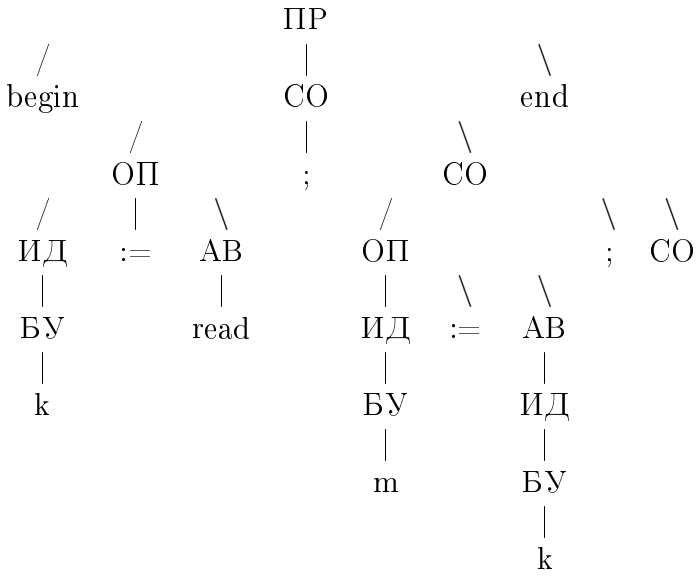
```
begin
k:=read; m:=k;
for j from 2 to 10 do
k:=read;
if k>m then m:=k endif
endfor
write m
end
```

Размеченный вывод программы начинается так:

ПР $\Rightarrow_{(1)}$ begin СО end $\Rightarrow_{(2)}$ begin ОП; СО end $\Rightarrow_{(3)}$ begin ИД := АВ; СО end ...

Отметим, что на каждом шаге вывода один из нетерминальных символов заменяется на соответствующую цепочку символов. Можно начинать всегда с левого нетерминального символа, всегда с правого нетерминального символа или с какого-то произвольно выбранного символа. Поэтому различают три способа вывода: левый, правый и произвольный.

Построим фрагмент дерева вывода программы (оно строится от корня к кроне, но рисовать это удобно сверху вниз):



Как уже было отмечено, одна и та же цепочка может быть выведена в грамматике разными способами: **левым выводом**, **правым выводом** или произвольным выводом. Наиболее важны левовыводимые и правовыводимые цепочки символов. Но дерево вывода для них — не обязательно одно и то же!

Пример. Вывод цепочки x+x*x в грамматике

(1) A → A + A
(2) A → A * A
(3) A → B
(4) B → x

1) $A \Rightarrow A+A \Rightarrow B+A \Rightarrow x+A \Rightarrow x+A*A \Rightarrow x+B*A \Rightarrow x+x*A \Rightarrow x+x*B \Rightarrow x+x*x$

2) $A \Rightarrow A*A \Rightarrow A*B \Rightarrow A*x \Rightarrow A+A*x \Rightarrow A+B*x \Rightarrow A+x*x \Rightarrow B+x*x \Rightarrow x+x*x$

Сечение дерева вывода — это такое множество **C** его вершин, что
1) никакие две вершины не лежат на одной ветви;
2) нельзя добавить к **C** вершину так, чтобы не нарушить условие 1).

В теории формальных языков и формальных грамматик доказываются два следующих важных утверждения.

Если $S \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n$ — вывод цепочки α_n, то можно построить дерево вывода, для которого α_n — крона и $\alpha_1, \ldots, \alpha_{n-1}$ — кроны некоторых сечений.

В КС-грамматике существует вывод $S \Rightarrow^* \alpha$ тогда и только тогда, когда существует дерево вывода с кроной α.

Второе утверждение представляется очень естественным. Будем им пользоваться (без доказательства).

8. Эквивалентные преобразования грамматик

Ясно, что один и тот же формальный язык может быть порожден различными (но похожими) формальными грамматиками. Одни из них, возможно, имеют преимущества

перед другими. Существуют относительно простые приемы улучшения формальных грамматик.

8.1. Неоднозначные грамматики

Грамматика называется **неоднозначной**, если в языке существует хотя бы одна цепочка, которая является кроной более чем одного дерева вывода.

Язык называется **неоднозначным**, если все порождающие его грамматики являются неоднозначными.

В теории формальных языков и формальных грамматик ставится ряд **проблем** — существенно важных вопросов.

Проблема называется **алгоритмически разрешимой**, если для ответа на соответствующий вопрос можно построить алгоритм, состоящий из конечного числа шагов.

Доказано, что *проблема однозначности произвольной грамматики алгоритмически не разрешима.*

Перечислим некоторые другие проблемы:

* Содержит ли язык хотя бы одну цепочку?
* Содержит ли язык бесконечное число цепочек?
* Содержит ли язык данную цепочку?

* Порождают ли две грамматики один и тот же язык?
* Порождает ли грамматика язык, содержащий все возможные терминальные цепочки?
* Пересекаются ли языки, порождаемые двумя грамматиками?

Пример. Неоднозначную грамматику построить достаточно просто. Пусть

(1) ОП → if УС then ОП else ОП
(2) ОП → if УС then ОП
(3) ОП → БУ

Цепочка if УС then if УС then БУ else БУ выводится различными способами и может быть по-разному истолкована:
или if УС then (if УС then БУ) else БУ
или if УС then (if УС then БУ else БУ)

Чтобы устранить неоднозначность, можно в семантику языка добавить: ключевое слово else относится к ближайшему слева ключевому слову if

Также можно изменить грамматику. Если увеличить число правил вывода, то неоднозначности уже не будет:

(1) ОП → if УС then ОП
(2) ОП → if УС then ОП1 else ОП
(3) ОП → БУ
(4) ОП1 → if УС then ОП1 else ОП1
(5) ОП1 → БУ

8.2. Эквивалентные преобразования грамматик

Грамматики, порождающие один и тот же язык, называются **эквивалентными**. Известны алгоритмы, позволяющие эквивалентно перейти от одной грамматики к другой, но так, чтобы избавиться от тех или иных нежелательных свойств.

Нетерминальный символ называется **производящим**, если из него можно вывести терминальную цепочку.

Символ называется **недостижимым**, если он не может появиться ни в одной цепочке языка.

Символ называется **бесполезным**, если он непроизводящий или недостижимый.

Рассмотрим несколько полезных алгоритмов.

Алгоритм построения множества N_p производящих символов

1. $N_p^0 := \emptyset, \;\; i := 1$
2. $N_p^i := N_p^{i-1} \cup \{A \mid A \to \alpha, \; \alpha \in (N_p^{i-1} \cup T)^*\}$
3. Если $N_p^i <> N_p^{i-1}$, то ($i := i + 1$; перейти к 2)
4. $N_p := N_p^i$

Алгоритм построения множества N_r достижимых символов

1. $N_r^0 := \{S\}, \;\; i := 1$
2. $N_r^i = N_r^{i-1} \cup \{X \mid A \to \alpha X \beta \in P$ и $A \in N_r^{i-1}\}$
3. Если $N_r^i <> N_r^{i-1}$, то ($i := i + 1$; перейти к 2)
4. $N_r := N_r^i$

Алгоритм устранения бесполезных символов

1. Построить N_p для G
2. $P_1 := \{$правила из P, состоящие из символов $T \cup N_p\}; \;\; G_1 := (T, N_p, S, P_1)$
3. Построить N_r для G_1
4. $P' := \{$правила из P_1, состоящие из символов $N_r\}$;
$T' := T \cap N_r; \;\; N' := N_r \cap N; \;\; G' := (T', N', S, P')$

Задача. Устранить бесполезные символы в грамматике
$T = \{a, b, c\}, \;\; N = \{A, B, C, S\}, \;\; S, \;\; P = \{S \to aC, \; S \to A, \; A \to cAB,$
$B \to b, \; C \to a\}$

Эта задача решается следующим образом.

1) Очевидно, множество производящих символов $N_p = \{B, C, S\}$.
(Из A цепочку из только терминальных символов вывести нельзя: A всегда заменяется на цепочку, опять содержащую A).

2) Тогда правила вывода, в которых не содержится нетерминальный символ A, образуют множество $P_1 = \{S \to aC, \; B \to b, \; C \to a\}$.

3) В новой грамматике достижимые символы $N_r = \{a, C, S\}$.

4) Поэтому окончательно $T' = T \cap N_r = \{a\},\ \ N' = N_r \cap N = \{C, S\}$,
$P' = \{S \to aC,\ C \to a\}$.

Оказалось, что язык, порождаемый новой грамматикой, состоит только из одной цепочки символов: $L(G') = \{aa\}$.

8.3. Преобразование грамматики в неукорачивающую

Грамматика называется **неукорачивающей**, если в множестве правил вывода или не содержится ε-правил, или содержится только одно правило вывода $S \to \varepsilon$ и начальный символ S не содержится в правых частях остальных правил вывода.

Алгоритм построения множества N_ε укорачивающих нетерминальных символов

1. $N_\varepsilon^0 := \{A \in N \mid A \to \varepsilon\ \in P\},\ \ i := 1$
2. $N_\varepsilon^i := N_\varepsilon^{i-1} \cup \{A \in N \mid A \to \alpha\ \in P,\ \alpha \in (N_\varepsilon^{i-1})^*\}$
3. Если $N_\varepsilon^i <> N_\varepsilon^{i-1}$, то ($i := i + 1$; перейти к 2)
4. $N_\varepsilon := N_\varepsilon^i$

Алгоритм преобразования грамматики $G = (T, N, S, P)$ в неукорачивающую грамматику $G' = (T, N', S', P')$

1. Построить N_ε
2. $P' := \emptyset$
3. Если $A \to \alpha_0 B_1 \alpha_1 \dots B_k \alpha_k\ \in P$, где $k \geq 0$, $B_j \in N_\varepsilon$ и в цепочках $\alpha_j \in (T \cup N)^*$ не содержится символов из N_ε, то включить в P' все правила вывода вида
$A \to \alpha_0 C_1 \alpha_1 \dots C_k \alpha_k$, где $C_j = B_j$ или $C_j = \varepsilon$.
(Правило вывода $A \to \varepsilon$ в P' не включается.)
4. Если $S \in N_\varepsilon$, то $N' := N \cup \{S'\}$ и включить в P' правила вывода $S' \to S$ и $S' \to \varepsilon$, иначе $N' := N$ и $S' := S$.

Задача 1 [1]. Преобразовать грамматику в неукорачивающую:
$T = \{b, c\},\ \ N = \{S, A\}$,
$P = \{S \to cA,\ S \to \varepsilon,\ A \to cA,\ A \to bA,\ A \to \varepsilon\}$.

Ответ: $N' = \{S, A, S'\}$,
$P = \{S \to cA,\ S \to c,\ A \to cA,\ A \to c,\ A \to bA,\ A \to b,\ S' \to S,\ S' \to \varepsilon\}$.

8.4. Исключение цепных правил

Правило вывода называется **цепным**, если его правая часть состоит из только одного нетерминального символа ($A \to B$).

Алгоритм исключения цепных правил

1. Для каждого нетерминального символа A построить множество
$N_A = \{B \mid A \Rightarrow^* B\}$:

1.1. $N_A^0 := \{A\}; \quad i := 1.$

1.2. $N_A^i := N_A^{i-1} \cup \{C \mid B \to C \ \in P,\ B \in N_A^{i-1}\}.$

1.3. Если $N_A^i <> N_A^{i-1}$, то ..... , иначе $N_A := N_A^i$

2. Для каждого символа B из N_A, если $B \to \alpha \ \in P$ не цепное правило, то включить в P' правила вывода $A \to \alpha$.

Еще раз. Перебираются все нетерминальные символы. Для символа A просматриваются элементы B множества N_A.
Если правило вывода $B \to \alpha$ не цепное, то в P' добавляется $A \to \alpha$.

Задача 2 [1]. Исключить цепные правила:
$P = \{E \to E + T,\ E \to T,\ T \to T * P,\ T \to P,\ P \to i,\ P \to (E)\}$.

Ответ: $P' = \{E \to E + T,\ E \to T * P,\ E \to i,\ E \to (E),$
$T \to T * P,\ T \to i,\ T \to (E),\ \ P \to i,\ P \to (E)\}$.

8.5. Нормальная форма Хомского

Среди множества эквивалентных грамматик можно выбирать наиболее простые.

Грамматика $G = (T, N, S, P)$ называется грамматикой в нормальной форме Хомского, если в множестве P содержатся только правила вывода вида:
1) $A \to BC$, где $A, B, C \in N$;
2) $A \to a$, где $A \in N, \ \ a \in T$;
3) если $\varepsilon \in G(L)$, то $S \to \varepsilon$; причем S не встречается в правых частях правил.

Алгоритм преобразования грамматики к нормальной форме Хомского

0. Устранить бесполезные символы; преобразовать грамматику в неукорачивающую; устранить цепные правила.

1. Если S содержится в правых частях правил вывода, то вводится новый начальный символ S' и в P включается правило вывода $S' \to S$.

2. Если $S \to \varepsilon \ \in P$, то $S \to \varepsilon$ включается в P'.

3. Правила вывода вида $A \to BC$ и $A \to a$ переносятся из P в P'.

4. Если в P имеется правило вывода вида $A \to X_1 \dots X_k, \ \ k > 2$, то в N' вносятся новые символы $< X_2 \dots X_k >, \dots ,< X_{k-1}X_k >$ и в P' вносятся правила вывода $A \to X_1' < X_2 \dots X_k >$,
$< X_2 \dots X_k > \to X_2' < X_3 \dots X_k >, \ \ < X_{k-1}X_k > \to X_{k-1}' X\_k$,
где $X_i' = X_i$, если $X_i \in N$ (*нетерминальные символы остаются*)
или $X_i' =< X_i >$, если $X_i \in T$ (*вместо терминальных вводятся нетерминальные*)

5. Если в P имеется правило вывода вида $A \to X_1X_2$, где хотя бы один из символов X_1, X_2 — терминальный, то в P' включается правило вывода $A \to X_1'X_2'$, где...

6. Для новых нетерминальных символов $< X_i >$ в P' включаются правила вывода $< X_i > \to X_i$.

Задача 3 [1]. Преобразовать грамматику к нормальной форме Хомского:
$T = \{a, b\}, \quad N = \{S, A, B\}$,
$P = \{S \to A,\ S \to ABA,\ A \to aA,\ A \to a,\ A \to B,\ B \to bB,\ B \to b\}$.

8.6. Нормальная форма Грейбах

Грамматика $G = (T, N, S, P)$ называется грамматикой в нормальной форме Грейбах, если
1) в ней нет ε-правил вывода;
2) каждое правило вывода, отличное от $S \to \varepsilon$, имеет вид $A \to a\alpha$, где $a \in T, \ \ \alpha \in N^*$.

При приведении грамматики к нормальной форме Грейбах нужно устранить левую рекурсию.

Нетерминальный символ A называется **рекурсивным**, если в P содержится правило вывода вида $A \to \alpha A \beta$, где α, β — некоторые цепочки.

Алгоритм устранения левой рекурсии (пусть $N = \{A_1, \ldots, A_n\}$)

1. $i := 1$.

2. Правила вывода
$A_i \to A_i\alpha_1, \ldots, A_i \to A_i\alpha_m, A_i \to \beta_1, \ldots, A_i \to \beta_n$ заменяются на
$A_i \to \beta_1, \ldots, A_i \to \beta_n, A_i \to \beta_1 A_i', \ldots, A_i \to \beta_n A_i'$,
$A_i' \to \alpha_1, \ldots, A_i' \to \alpha_m, A_i' \to \alpha_1 A_i', \ldots, A_i' \to \alpha_m A_i'$,
где A_i' — новый нетерминальный символ.

3. Если $i = n$, то алгоритм завершен, иначе $i := i + 1; \ \ j := 1$.

4. Правила вывода вида $A_i \to A_j\alpha$ заменяются на
$A_i \to \beta_1\alpha, \ldots, A_i \to \beta_m\alpha$, где β_k берутся из правил вывода $A_j \to \beta_1, \ldots, A_j \to \beta_m$.

5. Если $j = i - 1$, то переходим к 2, иначе $j := j + 1$ и переходим к 4.

Более простой случай: в множестве правил вывода содержится:
$A \to A\alpha_1, \ldots, A \to A\alpha_m$ и $A \to \beta_1, \ldots, A \to \beta_n$.
Чтобы исключить левую рекурсию, введем A' и заменим правила вывода на
$A \to \beta_1, \ldots, A \to \beta_n, \ \ A \to \beta_1 A', \ldots, A \to \beta_n A'$,
$A' \to \alpha_1, \ldots, A' \to \alpha_m, \ \ A' \to \alpha_1 A', \ldots, A' \to \alpha_m A'$.

Задача 4 [1]. Устранить левую рекурсию:
$P = \{E \to E + T,\ E \to T,\ T \to T * P,\ T \to P,\ P \to i,\ P \to (E)\}$.

Задача 5 [1]. Устранить левую рекурсию в грамматике с правилами вывода:
$P = \{S \to AB,\ S \to a,\ A \to BS,\ A \to Sb,\ B \to SA,\ B \to BB,\ B \to a\}$.

Решается эта задача так (здесь $A_1 = S,\ A_2 = A,\ A_3 = B$).
1) $i = 1$: Для нетерминального символа S правил вывода вида $S \to S\alpha$ нет (в P сохраняются правила $S \to AB,\ S \to a$).

2) $i = 2;\ j = 1$: Для нетерминального символа A имеется правило вывода вида $A \to S\alpha$, а именно $A \to Sb$. Находим все правила вида $S \to \beta$ — это $S \to AB,\ S \to a$. Заменяем $A \to Sb$ на $A \to ABb,\ A \to ab$. Итак, после этого шага в множестве правил вывода содержится:
$S \to AB,\ S \to a,\ A \to BS,\ A \to ABb,\ A \to ab,\ B \to SA,\ B \to BB,\ B \to a.$

3) Так как $j = i - 1$, то переходим к шагу 2 алгоритма. Для нетерминального символа A нашлось правило вывода вида $A \to A\alpha$, а именно $A \to ABb$. Вводим новый нетерминальный символ A' и заменяем это правило (используем $A \to BS,\ A \to ab$ — правила вида $A \to \beta$) на $A \to BSA',\ A \to abA',\ A' \to Bb,\ A' \to BbA'$. Теперь все правила вывода:
$S \to AB,\ S \to a,\ A \to BS,\ A \to ab,\ A \to BSA',\ A \to abA',$
$A' \to Bb,\ A' \to BbA',\ B \to SA,\ B \to BB,\ B \to a.$

4) $i = 3;\ j = 1$: Есть правило вывода вида $B \to S\alpha$ — это $B \to SA$, а также правила вывода вида $S \to \beta$ — это $S \to AB,\ S \to a$. Поэтому это (первое) правило заменяем на $B \to ABA,\ B \to aA$.

5) $i = 3;\ j = 2$: Вместо $B \to ABA$ с учетом $A \to BS,\ A \to ab,\ A \to BSA',$
$A \to abA'$ записываем в P правила вывода $B \to BSBA,\ B \to abBA,\ B \to BSA'BA,$
$B \to abA'BA$.

И так далее, осталось совсем немного :-)

9. Синтаксически-ориентированная трансляция

Напомним хорошо известные всем программистам термины.

Трансляция — процесс перевода текста, написанного на одном языке (*исходный код*), в текст на другом языке.

Транслятор — компьютерная программа для перевода.

Компилятор переводит программу на языке высокого уровня целиком в программу на языке более низкого уровня (в *машинный, объектный или промежуточный код*).

Интерпретатор переводит отдельные операторы программы в машинный код и сразу передает его на выполнение.

Препроцессор — транслятор, который преобразует программу на языке высокого уровня с расширениями в программу на базовом языке.

Несколько слов о платформе .NET ...

Основная идея **синтаксически-ориентированной трансляции** (автор — Ноам Хомский): при обработке исходного текста использовать структуру предложения для выявления его смысла. В соответствии с этим получается, что *Синтаксис определяет семантику.*

Что такое **структура** сложного объекта? Это разбиение его на связанные между собой части, то есть описание отдельных частей объекта и связей между ними.

В теории формальных языков *структуру цепочки символов определяет ее дерево вывода.*

Пример. "Арифметическое" выражение `a%b#c?d`. Интуитивно ясно, что буквами обозначены какие-то операнды, а знаками — некоторые операции над ними (смысл этих операций в данном случае скрыт).

9.1. Атрибутные грамматики

Дональд Кнут (профессор Станфордского университета) предложил расширить понятие контекстно-свободной грамматики. Каждому символу (терминальному и нетерминальному) приписывается конечное число **атрибутов** — семантических параметров. К каждому синтаксическому правилу вывода добавляются **семантические правила**, по которым преобразуются значения атрибутов.

Смысл цепочки символов (предложения языка) вычисляется по построенному дереву вывода.

Удобно использовать объектную идеологию: каждый символ грамматики рассматривается как *класс*, а любое использование этого символа порождает *объект*.

Пример. Вещественные арифметические выражения

$$\text{АВ} \to \text{ВК} \mid (\text{АВ}) \mid \text{АВ} + \text{АВ} \mid \text{АВ} - \text{АВ} \mid \text{АВ} * \text{АВ} \mid \text{АВ}/\text{АВ}$$

Классы ВК (вещественная константа) и АВ (арифметическое выражение) имеют поле $Value$ (значение). В атрибутной грамматике правила вывода дополняются семантическими правилами, которые определяют действия над атрибутами:

$\text{АВ} \to \text{ВК}$	$\text{АВ}.Value := \text{ВК}.Value$
$\text{АВ}_0 \to (\text{АВ}_1)$	$\text{АВ}_0.Value := \text{АВ}_1.Value$
$\text{АВ}_0 \to \text{АВ}_1 + \text{АВ}_2$	$\text{АВ}_0.Value := \text{АВ}_1.Value + \text{АВ}_2.Value$
и так далее ...	

Семантические атрибуты подразделяются на **синтезированные** и **унаследованные**. Синтезированные атрибуты нетерминальных символов вычисляются в дереве вывода снизу вверх. Унаследованные атрибуты нетерминальных символов вычисляются в дереве вывода сверху вниз.

Пример. Объявление переменных в языке программирования.

ОП → ТП СИ | ОП ; ТП СИ
ТП → *real* | *int*
СИ → ИД | СИ , ИД
ИД → <буква>

(Описание Переменных, Тип Переменной, Список Идентификаторов, Идентификатор)

Для программиста смысл цепочки символов $real\ \ x, y;\ \ int\ \ k, l, m$, построенной по правилам вывода грамматики, вполне ясен.

Вопрос. Какие семантические атрибуты нужно приписать нетерминальному символу ИД ?

Пример. Атрибутная грамматика: двоичные вещественные числа
$S \to L.L$
$L \to B \mid LB$
$B \to 0 \mid 1$

Атрибуты для нетерминальных символов:

B	Value	bit
L	Value	word,
	Length	byte
S	Value	real

Семантические правила:

$B \to 1$	$B.Value := 1$
$B \to 0$	$B.Value := 0$
$L \to B$	$L.Value := B.Value;\ \ L.Length := 1$
$L_0 \to L_1 B$	$L_0.Value := 2 \times L_1.Value + B.Value;$
	$L_0.Length := L_1.Length + 1$
$S \to L_1.L_2$	$S.Value := L_1.Value + L_2.Value \,/\, 2^{L_2.Length}$

9.2. Трансляция арифметических выражений

Для описания арифметических выражений можно использовать различные грамматики. Например,

1. $E \to i \mid (E) \mid E + E \mid E - E \mid E * E \mid E/E$

Эта грамматика является двусмысленной — по одной и той же цепочке можно построить несколько различных деревьев вывода. Но есть простой способ устранения двусмысленности. Нужно написать так:

2. $E \to E + T \mid T$
   $T \to i$

Здесь подразумевается, что порядок выполнения операций общепринятый — слева направо. Семантические правила:

$E_0 \to E_1 + T$	$E_0.Value := E_1.Value + T.Value$
$E \to T$	$E.Value := T.Value$
$T \to i$	$T.Value := i.Value$

3. $E \to E + T \mid E - T \mid T$
   $T \to i$

Здесь все операции имеют один и тот же приоритет.

4. $E \to E + T \mid E * T \mid T$
   $T \to i$

Получилось не очень хорошо — приоритет не соблюдается :-(

5. $E \to E + T \mid T$
   $T \to T * F \mid F$
   $F \to i$

Добавим еще вычитание, деление, возведение в степень и скобки:

6. $E \to E + T \mid E - T \mid T$
   $T \to T * P \mid T/P \mid P$
   $P \to Q \uparrow P \mid Q$
   $Q \to i \mid (E)$

9.3. Польская инверсная запись

ПОЛИЗ — бесскобочная постфиксная форма записи арифметических выражений (предложил Ян Лукасевич, 1925)

$x + y$ — инфиксная форма (знак операции пишется между операндами)

$+xy$ — префиксная форма (знак операции впереди)

$xy+$ — постфиксная форма (знак операции стоит последним)

В соответствии с правилами ПОЛИЗ
$a + b$ преобразуется в $ab+$
$a + b * c - d$ преобразуется в $abc * +d-$
$a + b * (c - d) + (f - e)$ преобразуется в $abcd - * + fe - +$

Цепочка символов, представляющая собой арифметическое выражение в ПОЛИЗ, обрабатывается слева направо:
операнды записываются в стек,
операции выполняются (операнды берутся из стека),
и результат — в стек.

В польской инверсной записи можно представить не только арифметические выражения. Например, оператор присваивания
$g := if\ a > c\ or\ d\ then\ e - f\ else\ q$ преобразуется в
$gac > d\ or\ ef - q\ ite\ :=$

Атрибутная грамматика перевода арифметических выражений в ПОЛИЗ (операции типа сложения $+$, операции типа умножения $*$) задается с помощью правил вывода

$E \to E + T \mid T$
$T \to T * F \mid F$
$F \to i \mid (E)$

Семантический атрибут нетерминальных символов E, T, F — символьная строка Str, операция над атрибутами — конкатенация (ее знак не пишется). Вот список правил вывода вместе с операциями над семантическими атрибутами:

$E \to E_1 + T$	$E.Str := E_1.Str\ T.Str +$
$E \to T$	$E.Str := T.Str$
$T \to T_1 * F$	$T.Str := T_1.Str\ F.Str *$
$T \to F$	$T.Str := F.Str$
$F \to i$	$F.Str := i$
$F \to (E)$	$F.Str := E.Str$

Семантика цепочки символов определяется так: сначала строится дерево вывода (от кроны к корню), а затем это дерево проходится в обратном направлении (от корня к кроне) и выписываются операции над семантическими атрибутами.

Пример. $(a + b) * (c + d)$ заменяется на $ab + cd + *$

Если дерево вывода строится снизу вверх, то семантику можно упростить!

9.4. Интерпретация и компиляция арифметических выражений

Интерпретация арифметических выражений может быть проведена на основе семантических правил изменения полей Value. Например,

$E \to E + T \mid T$
$T \to i$

$E_0 \to E_1 * T$	$E_0.Value := E_1.Value * T.Value$
$E \to T$	$E.Value := T.Value$
$T \to i$	$T.Value := i.Value$

При компиляции генерируется некоторое промежуточное представление исходного кода. Часто используется так называемый p-код — аналог команд языка Ассемблера.

Предположим, что компьютер имеет память для кода, память для данных и стек. Команды p-кода обычно выглядят так (операнды извлекаются из стека и результат операции записывается в стек):

LD a
ST a
ADD
SUB
MUL
DIV

Легко построить атрибутную грамматику, описывающую правила программирования в командах p-кода. Программировать также легко.

Например, арифметическое выражение $(a+b)*(c+d)$ (ему соответствует постфиксная запись на $ab+cd+*$) заменяется на следующий код:

LD a
LD b
ADD
LD c
LD d
ADD
MUL

10. Автоматы и преобразователи

Процесс распознавания и переработки цепочки символов можно рассматривать как процесс функционирования некоторого условного прибора. Автоматы и преобразователи используются для моделирования алгоритмов обработки цепочек символов.

Распознающий автомат имеет:
входную ленту,
устройство чтения,
устройство управления с конечной памятью,
вспомогательную память.

Текущее состояние распознавателя называют ***конфигурацией***.
Для каждой конфигурации определены:

состояние устройства управления (их конечное множество),
положение устройства чтения (какой символ читается),
состояние вспомогательной памяти (если она есть).

В множестве конфигураций выделяются начальная конфигурация и конечная конфигурация. Переход от одной конфигурации к другой осуществляется по тактам.

Распознаватель допускает входную цепочку, если он, обрабатывая эту цепочку начиная с начальной конфигурации, переходит в конечную конфигурацию за конечное число тактов.

Конечный автомат — простейший распознаватель без вспомогательной памяти. Используют следующее определение конечного автомата: $K = (Q, T, \delta, q_0, F)$, где

Q — конечное множество состояний устройства управления,

T — алфавит входных символов,

δ — функция переходов (отображение $Q \times T \to Q$),

$q_0 \in Q$ — начальное состояние,

$F \subset Q$ — множество заключительных состояний.

Если функция δ — однозначная, то КА называют детерминированным. Если функция δ — многозначная, то КА называют недетерминированным.

Конфигурация автомата $(q, w) \in Q \times T^*$, при этом начальная конфигурация (q_0, w) и конечная конфигурация $(q, \varepsilon) \mid q \in F$. Здесь w — цепочка символов, которые еще не были обработаны.

Переходы от конфигурации к конфигурации обозначаются знаком $\vdash$.

Распознаватель $K = (Q, T, \delta, q_0, F)$ допускает входную цепочку $w \in T^*$, если $(q_0, w) \vdash (q, \varepsilon),\ q \in F$.

Язык, определяемый конечным автоматом K,

$$L(K) = \{w \in T^* \mid (q_0, w) \vdash (q, \varepsilon),\ q \in F\}$$

Пример 1. Конечный автомат, допускающий цепочки из 0 и 1, в которых имеется подцепочка 11.

$K = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\})$

Функция переходов:

$\delta(q_0, 0) = \{q_0\}, \quad \delta(q_0, 1) = \{q_1\}, \quad \delta(q_1, 0) = \{q_0\},$
$\delta(q_1, 1) = \{q_2\}, \quad \delta(q_2, 0) = \{q_2\}, \quad \delta(q_2, 1) = \{q_2\}$

Часто функция переходов задается как таблица переходов или как диаграмма переходов.

Пример 2. Конечный автомат

$K = (\{q_0, q_1, q_2, q_3\}, \{a, b\}, \delta, q_0, \{q_3\})$

δ	a	b
q_0	q_1	q_0
q_1	q_1	q_2
q_2	q_3	q_0
q_3	q_3	q_3

определяет, содержится ли в цепочке символов подцепочка aba.

Пример 3. Распознаватель вещественного числа без знака вида $zz.zz$ имеет пять состояний, из которых четыре — заключительные. Правильные цепочки символов имеют вид: zz, $zz.$, $.zz$ и $zz.zz$.

Можно предусмотреть в конструкции такого автомата возможность отбрасывания лишних нулей (слева или справа).

Доказано: **множество языков, допускаемых конечными автоматами, совпадает с множеством языков, порождаемых автоматными грамматиками.**

Конечный преобразователь анализирует цепочку символов на входной ленте и записывает другую цепочку символов на выходной ленте. По определению $M = (Q, T, D, \delta, q_0, F)$, где

Q — конечное множество состояний устройства управления,
T — алфавит входных символов,
D — алфавит выходных символов,
δ — функция переходов (отображение $Q \times T \to Q$),
$q_0 \in Q$ — начальное состояние,
$F \subset Q$ — множество заключительных состояний.

Конфигурация конечного преобразователя $(q, x, y) \in Q \times T^* \times D^*$.

Цепочка символов $y \in D^*$ называется выходом для цепочки символов $x \in T^*$, если $(q_0, x, \varepsilon) \vdash (q, \varepsilon, y)$ для некоторого $q \in F$.

Автомат с магазинной памятью (МП-автомат) представляет собой конечный автомат, дополненный неограниченной памятью с доступом только к крайнему символу.

$P = (Q, T, \Gamma, \delta, q_0, Z_0, F)$, где

Q — конечное множество состояний устройства управления,
T — алфавит входных символов,
Γ — алфавит символов магазина,
δ — функция переходов (отображение $Q \times T \times \Gamma \to Q \times \Gamma^*$),
$q_0 \in Q$ — начальное состояние устройства управления,
$Z_0 \in \Gamma$ — начальный символ в магазине,
$F \subset Q$ — множество заключительных состояний.

Конфигурация МП-автомата: q — состояние устройства управления, x — необработанная часть входной цепочки и α — содержимое магазина.

Начальная конфигурация (q_0, w, Z_0), заключительная конфигурация (q, ε, α).

Такт работы $(q, x, \alpha) \vdash (q', x', \alpha')$.

МП-автомат допускает цепочку символов $w \in T^*$, если $(q_0, w, Z_0) \vdash^* (q, \varepsilon, \alpha)$ при некоторых $q \in F$ и $\alpha \in \Gamma^*$.

Язык, определяемый (допускаемый) МП-автоматом, образуют все распознаваемые им цепочки.

Пример 4. МП-автомат, допускающий язык $L = \{a^n b^n \mid n \geq 0\}$:
$Q = \{q_0, q_1, q_2\}, \quad T = \{a, b\}, \quad \Gamma = \{z, a\}, \quad \delta, \quad , q_0, \quad , z, \quad F = \{q_0\}$

$$\delta(q_0, a, z) = \{(q_1, +a)\},$$
$$\delta(q_1, a, a) = \{(q_1, +a)\},$$
$$\delta(q_1, b, a) = \{(q_2, -)\},$$
$$\delta(q_2, b, a) = \{(q_2, -)\},$$
$$\delta(q_2, \varepsilon, z) = \{(q_0, \varepsilon)\},$$

Идея следующая: если читается символ a, то он вносится в магазин (знак $+$); если читается символ b, то из магазина изымается один символ. Когда все символы прочитаны, а в магазине пусто, символов b было ровно столько, сколько символов a.

Во время работы МП-автомата операции над крайним символом в магазине не зависят от других символов в магазине!

Расширенные МП-автоматы допускают замену конечной цепочки крайних символов в магазине на другую конечную цепочку.

Множество языков, допускаемых автоматами с магазинной памятью, совпадает с множеством языков, порождаемых контекстно свободными-грамматиками.

Преобразователь с магазинной памятью (МП-преобразователь) представляет собой МП-автомат, имеющий устройство записи символов на выходную ленту.

$D = (Q, T, D, \Gamma, \delta, q_0, Z_0, F)$, где
Q — конечное множество состояний устройства управления,
T — алфавит входных символов,
D — алфавит выходных символов,
Γ — алфавит символов магазина,
δ — функция переходов (отображение $Q \times T \times \Gamma \to Q \times \Gamma^*$),
$q_0 \in Q$ — начальное состояние устройства управления,
$Z_0 \in \Gamma$ — начальный символ в магазине,
$F \subset Q$ — множество заключительных состояний.

Конфигурация МП-преобразователя: q — состояние устройства управления, x — необработанная часть входной цепочки, α — содержимое магазина и y — цепочка символов на выходной ленте.

Пример 5. Построим расширенный МП-преобразователь, переводящий арифметическое выражение в инфиксной форме в эквивалентную префиксную форму.

Конструкция автомата должна быть согласована с грамматикой. Пусть, как было раньше, грамматика следующая:
$\{E, T, P\}, \quad \{i, +, *, (,)\}, \quad E,$
$\{E \to E + T, \;\; E \to T, \;\; T \to T * P, \;\; T \to P, \;\; P \to (E), \;\; P \to i\}$
К сожалению, автомат получится недетерминированный — значения функции переходов выбираются из некоторых дополнительных соображений:

$$\delta(q, \varepsilon, E) = \{(q, E + T, +), \; (q, T, \varepsilon)\},$$
$$\delta(q, \varepsilon, T) = \{(q, T * P, *), \; (q, P, \varepsilon)\},$$
$$\delta(q, \varepsilon, P) = \{(q, (E), \varepsilon), \; (q, i, i)\},$$
$$\delta(q, a, a) = (q, \varepsilon, \varepsilon) \quad \forall a \in T$$

Преобразователь расширенный в том смысле, что символ в магазине может быть заменен на правую часть одного из соответствующих ему правил вывода.

Состояние автомата всегда одно и то же. Пусть на входной ленте $i*(i+i)$. Посмотрим, как идет анализ (магазин заполняется влево, выходные символы дописываются справа):

вход	магазин	выход
$i*(i+i)$	E	ε
$i*(i+i)$	T	ε
$i*(i+i)$	$T*P$	$*$
$i*(i+i)$	$P*P$	$*$
$i*(i+i)$	$i*P$	$*i$
$*(i+i)$	$*P$	$*i$
$(i+i)$	P	$*i$
$(i+i)$	(E)	$*i$
$i+i)$	$E)$	$*i$
$i+i)$	$E+T)$	$*i+$
$i+i)$	$T+T)$	$*i+$
$i+i)$	$P+T)$	$*i+$
$i+i)$	$i+T)$	$*i+i$
$+i)$	$+T)$	$*i+i$
$i)$	$T)$	$*i+i$
$i)$	$P)$	$*i+i$
$i)$	$i)$	$*i+ii$
$)$	$)$	$*i+ii$
ε	ε	$*i+ii$

Ясно, что действия были таковы: символы в магазине заменяются на правые части правил вывода так, чтобы в окне магазина был виден тот же символ, что на входной ленте, после чего эти одинаковые символы как бы вычеркиваются.

11. Стратегии разбора цепочек символов

При распознавании цепочек символов конструируются **синтаксические анализаторы** — устройства (или компьютерные программы), работающие на основе правил КС-грамматик.

Примем соглашение: входная цепочка анализируется слева направо.

Цепочка считается **разобранной**, если построено ее дерево вывода в заданной грамматике.

Пусть все правила вывода грамматики перенумерованы. Тогда разбор цепочки сводится к составлению последовательности номеров используемых правил вывода.

При разборе цепочек символов строятся МП-преобразователи, отображающие входные цепочки в последовательности правил вывода.

Основные **стратегии разбора**:
нисходящая (сверху вниз) — левый вывод;
восходящая (снизу вверх) — правый вывод.

Пример. Грамматика с правилами вывода

(1) $E \to E + T$
(2) $E \to T$
(3) $T \to T * P$
(4) $T \to P$
(5) $P \to i$
(6) $P \to (E)$

Пусть задана входная цепочка символов $i * (i + i)$

Левый разбор 23456124545 (левый вывод, список номеров слева направо)

$$E \Rightarrow T \Rightarrow T * P \Rightarrow P * P \Rightarrow i * P \Rightarrow i * (E) \Rightarrow i * (E + T) \Rightarrow i * (T + T) \Rightarrow$$

$$\Rightarrow i * (P + T) \Rightarrow i * (i + T) \Rightarrow i * (i + P) \Rightarrow i * (i + i)$$

Правый разбор 54542541632 (правый вывод, список номеров справа налево)

$$E \Rightarrow T \Rightarrow T * P \Rightarrow T * (E) \Rightarrow T * (E + T) \Rightarrow T * (E + P) \Rightarrow T * (E + i) \Rightarrow$$

$$\Rightarrow T * (T + i) \Rightarrow T * (P + i) \Rightarrow T * (i + i) \Rightarrow P * (i + i) \Rightarrow i * (i + i)$$

11.1. Нисходящий разбор

Пусть $G = (N, T, S, P)$ — КС-грамматика. **Левый анализатор** представляет собой МП-преобразователь
$M_l = (\{q\}, T, \{1, 2, \ldots, p\}, T \cup N, \delta, q, S, \{q\})$,
где
1) $\delta(q, a, a) = \{(q, \varepsilon, \varepsilon)\}$ для всех $a \in T$;
2) для правила вывода $A \to \alpha$ с номером i в множестве значений $\delta(q, \varepsilon, A)$ содержатся элементы (q, α, i).

(Магазин удобно заполнять влево.)

Пример. Для грамматики
(1) $E \to E + T$, (2) $E \to T$, (3) $T \to T * P$,
(4) $T \to P$, (5) $P \to i$, (6) $P \to (E)$

левый анализатор
$M_l = (\{q\}, \{i, +, *, (,)\}, \{1, 2, 3, 4, 5, 6\}, \{i, +, *, (,), E, T, P\}, \delta, q, E, \{q\})$,

$\delta(q, a, a) = (q, \varepsilon, \varepsilon)$ для всех $a \in \{i, +, *, (,)\}$,
$\delta(q, \varepsilon, E) = \{(q, E + T, 1), (q, T, 2)\}$,
$\delta(q, \varepsilon, T) = \{(q, T * P, 3), (q, P, 4)\}$,
$\delta(q, \varepsilon, P) = \{(q, i, 5), (q, (E), 6)\}$.

11.2. Восходящий разбор

Пусть $G = (N, T, S, P)$ — КС-грамматика. **Правый анализатор** представляет собой расширенный МП-преобразователь
$M_r = (\{q\}, T, \{1, 2, \ldots, p\}, T \cup N \cup \{z\}, \delta, q, z, \{q\})$,
где
1) $\delta(q, a, \varepsilon) = \{(q, a, \varepsilon)\}$ для всех $a \in T$;

2) для правила вывода $A \to \alpha$ с номером i в множестве значений $\delta(q, \varepsilon, \alpha)$ содержатся элементы (q, A, i);
3) $\delta(q, \varepsilon, Sz) = \{(q, \varepsilon, \varepsilon\}$.

(Магазин удобнее заполнять вправо, но будем заполнять его влево :-))

Пример. Для грамматики
(1) $E \to E + T$, (2) $E \to T$, (3) $T \to T * P$,
(4) $T \to P$, (5) $P \to i$, (6) $P \to (E)$

правый анализатор $M_r =$
$= (\{q\}, \{1, 2, 3, 4, 5, 6\}, \{i, +, *, (,)\}, \{i, +, *, (,), E, T, P, z\}, \delta, q, z, \{q\})$,

$\delta(q, a, \varepsilon) = (q, a, \varepsilon)$ для всех $a \in \{i, +, *, (,)\}$,
$\delta(q, \varepsilon, E + T) = (q, E, 1)$,
$\delta(q, \varepsilon, T) = (q, E, 2)$,
$\delta(q, \varepsilon, T * P) = (q, T, 3)$,
$\delta(q, \varepsilon, P) = (q, T, 4)$,
$\delta(q, \varepsilon, i) = (q, P, 5)$,
$\delta(q, \varepsilon, (E)) = (q, P, 6)$,
$\delta(q, \varepsilon, Ez) = (q, \varepsilon, \varepsilon)$.

11.3. Нисходящий разбор с возвратами

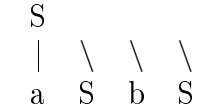
Раньше рассматривались *недетерминированные* правый и левый синтаксические анализаторы — функция переходов была многозначной. Ее значение выбиралось среди всех возможных из каких-то дополнительных соображений.

При полном разборе цепочки символов нужно просматривать все возможные переходы от конфигурации к конфигурации. Должен быть предусмотрен возврат назад, если дальнейшие построения не имеют смысла.

В общем случае при построении дерева вывода выбираются *активные вершины*. Если в активной вершине находится нетерминальный символ, то он разворачивается. Если в активной вершине находится терминальный символ, то он сравнивается с текущим входным символом (цепочка символов анализируется слева направо). При совпадении активного терминального символа с текущим входным символом переходят к следующему символу входной цепочки (он становится текущим). Если нетерминальные символы в активных вершинах могут быть развернуты различными способами, то пробуют последовательно все варианты. Если возникает ошибка, то возвращаются назад и пробуют другой способ.

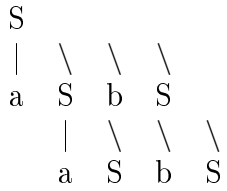
Пример. Разбор цепочки символов *aacbac* в грамматике с правилами вывода (1) $S \to aSbS$, (2) $S \to aS$, (3) $S \to c$

Первая активная вершина S. Разворачиваем ее (выбрали первое правило вывода из P, может быть, получится):

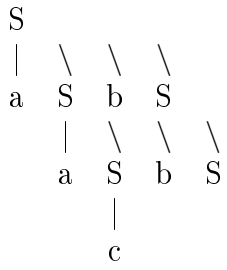


Следующая активная вершина a. Терминальный символ совпадает с текущим входным символом. Объявляем активной вершиной S и разворачиваем этот нетерминальный

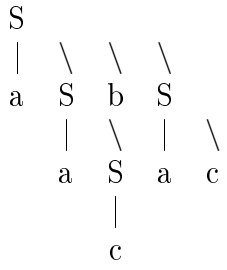
символ (по первому правилу вывода):



Опять в активной вершине терминальный символ a, он совпадает со вторым входным символом. Переходим к следующей вершине S и разворачиваем ее. Ясно, что первые два правила вывода не подходят — разворачиваем по третьему правилу. Получилось



Что теперь ни делай с последней активной вершиной S — будет ошибка. Нужно вернуться назад. Правильное дерево вывода



Можно показать, что нисходящий разбор всегда будет завершен успешно, если грамматика не содержит левой рекурсии.

Чтобы автомат мог моделировать алгоритм нисходящего разбора с возвратами, нужно предусмотреть в его конструкции *два магазина и счетчик*. В магазин L_1 записываются проделанные выборы и обработанные входные символы, в магазин L_2 вносятся символы левовыводимой цепочки — активные вершины, счетчик i хранит номер обрабатываемой позиции входной цепочки.

Предусматривается три состояния левого анализатора с возвратами: q — нормальное состояние, b — состояние возврата и e — заключительное состояние. Конфигурацию автомата образуют четыре параметра:
s — состояние,
i — значение счетчика,
α — содержимое магазина L_1 (удобно считать, что вершина этого магазина справа),
β — содержимое магазина L_2 (удобно считать, что вершина этого магазина слева).

Рассмотрим знакомый **пример**. Пусть грамматика содержит правила вывода
(1) $E \to E + T$, (2) $E \to T$, (3) $T \to T * P$,
(4) $T \to P$, (5) $P \to i$
Проведем анализ цепочки символов $i + i * i$.

В первый магазин будем записывать **альтернативы** — условные обозначения правил вывода грамматики — E_1, E_2, T_1, T_2, P_1.

В строках таблицы записаны состояния левого анализатора с возвратами.

состояние s	счетчик i	магазин L_1	магазин L_2	комментарий
q	1	ε	E	нач. конфигурация
q	1	E_1	$E+T$	разрастание дерева
q	1	E_1T_1	$E+T*P$	разрастание дерева
q	1	$E_1T_1P_1$	$E+T*i$	разрастание дерева
q	2	$E_1T_1P_1i$	$E+T*$	сравн. симв.: совп.
b	2	$E_1T_1P_1i$	$E+T*$	сравн. симв.: не совп.
b	1	$E_1T_1P_1$	$E+T*i$	возврат
b	1	E_1T_1	$E+T*P$	возврат
b	1	E_1T_2	$E+P$	смена альтернативы

и так далее ...

11.4. Восходящий разбор с возвратами

Восходящий разбор с возвратами выполняется в обратном порядке: дерево вывода строится от листьев к корню. Все начинается с просмотра входной цепочки символов. Если находится группа символов, совпадающая с правой частью одного из правил вывода, то проводится **сворачивание** — данная группа символов заменяется на левую часть этого правила вывода. Так делается до тех пор, пока не будет получен начальный нетерминальный символ грамматики.

Если цель не достигнута, то нужно вернуться назад (отменить сворачивание) и попробовать свернуть другую группу символов.

Установлено, что для успешного завершения разбора: 1) в грамматике не должно быть циклов, то есть нетривиальных выводов нетерминальных символов из самих себя; 2) в грамматике не должно быть пустых правил вывода (с пустыми цепочками в правой части).

Рассмотрим простой пример. Пусть в грамматике содержатся правила вывода:
(1) $S \to AB$, (2) $A \to ab$, (3) $B \to aba$
Проведем восходящий разбор с возвратами цепочки символов $ababa$.

Заменим группу символов ab на нетерминальный символ A, затем вторую такую же группу снова на A. Получим AAa. Ошибка: в этой цепочке символов ничего не сворачивается, начальный символ не достигнут. Делаем шаг назад, к цепочке: $Aaba$. Пробуем по-другому — три последних символа заменяем на B. Вот теперь получилось — AB сворачивается в S.

Правый анализатор с возвратами конструируется тем же образом, что и левый анализатор. В магазин L_1 помещаются терминальные и нетерминальные символы, из которых выводится группа символов, расположенных в левой части входной цепочки. Сначала этот магазин пуст, но затем туда переносятся по одному символы из входной цепочки. Если можно свернуть то, что в окне магазина — эта группа символов заменяется на их свертку. В магазине L_2 хранятся номера правил вывода, использованных при сворачивании, и туда же помещается специальный символ s в знак того, что в

первый магазин был перенесен символ из входной цепочки. При возврате выполняются очевидные действия ....

Все работает достаточно просто. Рассмотрим тот же самый пример: (1) $E \to E+T$, (2) $E \to T$, (3) $T \to T*P$,
(4) $T \to P$, (5) $P \to i$
Проведем анализ цепочки символов $i+i*i$.

В таблице описаны конфигурации автомата по тактам:

состояние s	счетчик i	магазин L_1	магазин L_2	комментарий
q	1	ε	ε	нач. конфигурация
q	2	i	s	перенос
q	2	P	$s5$	сворачивание
q	2	T	$s54$	сворачивание
q	2	E	$s542$	сворачивание
q	3	$E+$	$s542s$	перенос
q	4	$E+i$	$s542ss$	перенос
q	4	$E+P$	$s542ss5$	сворачивание
q	4	$E+T$	$s542ss54$	сворачивание
q	4	E	$s542ss541$	сворачивание
q	5	$E*$	$s542ss541s$	перенос
q	6	$E*i$	$s542ss541ss$	перенос
q	6	$E*P$	$s542ss541ss5$	сворачивание
q	6	$E*T$	$s542ss541ss54$	сворачивание
q	6	$E*E$	$s542ss541ss542$	сворачивание

а дальше не сворачивается :-(Возврат!

и так далее ... :-)

11.5. Алгоритм Эрли

Существуют и другие алгоритмы разбора цепочек символов. Рассмотрим один из них — алгоритм Эрли. Проведем разбор цепочки символов $a_1a_2\dots a_n$ в грамматике $G=(T,N,S,P)$.

Ситуацией называется конструкция вида

$$[A \to X_1\dots X_k \bullet X_{k+1}\dots X_m, i],$$

если в P есть правило вывода $A \to X_1\dots X_m$
(k,i — целые числа от 0 до m, $\bullet$ — метасимвол "толстая точка").
Нужно построить **список разбора**, состоящий из списков ситуаций $I_0, I_1, \dots I_n$.

Построение списка разбора сводится к двум этапам.
I. Составляется список I_0.
II. По спискам $I_0,\dots I_{j-1}$ составляется список I_j, $j=1..n$.

Этап I.
1. Для каждого правила вывода $S \to \alpha$
ситуация $[S \to \bullet\alpha, 0]$ вносится в I_0.

2. Это действие повторяется, пока появляются новые ситуации:
1) если $[B \to \gamma\bullet, 0] \in I_0$ и $[A \to \alpha \bullet B\beta, 0] \in I_0$,
то ситуация $[A \to \alpha B \bullet \beta, 0]$ вносится в I_0;
2) если $[A \to \alpha \bullet B\beta, 0] \in I_0$ и $B \to \gamma \in P$,
то ситуация $[B \to \bullet\gamma, 0]$ вносится в I_0.

Этап II.

1. Для каждой ситуации $[B \to \alpha \bullet a_j\beta, i]$ из I_{j-1}
ситуация $[B \to \alpha a_j \bullet \beta, i]$ вносится в I_j.

2. Это действие повторяется, пока появляются новые ситуации:
1) если $[A \to \alpha\bullet, i] \in I_j$ и $[B \to \alpha \bullet A\beta, k] \in I_i$,
то ситуация $[B \to \alpha A \bullet \beta, k]$ вносится в I_j;
2) если $[A \to \alpha \bullet B\beta, i] \in I_j$ и $B \to \gamma \in P$,
то ситуация $[B \to \bullet\gamma, j]$ вносится в I_j.

По списку разбора можно построить правый разбор цепочки символов следующим образом.

Если в I_n нет ситуации вида $[S \to \alpha\bullet, 0]$,
то выдается сообщение об ошибке;
иначе $\pi := \varepsilon$ и выполняется рекурсивно процедура $R([S \to \alpha\bullet, 0], n)$.

Процедура $R([A \to \beta\bullet, i], j)$ вписывает *слева* в глобальную переменную π номера правил вывода:

Дальнейшие шаги алгоритма следующие.

1. Если $A \to \beta$ — правило вывода с номером p, то p вписывается в π.

2. Если $\beta = X_1X_2 \dots X_m$, то $k := m$ и $l := j$.

3. Повторяется, пока $k \neq 0$:

1) если $X_k \in T$, то $k := k - 1$, $l := l - 1$;

2) если $X_k \in N$, то в списке I_l нужно найти ситуацию $[X_k \to \gamma\bullet, r]$, для которой в списке I_r имеется ситуация вида $[A \to X_1X_2 \dots X_{k-1} \bullet X_k \dots X_m, i]$;

3) выполнить $R([X_k \to \gamma\bullet, r], l)$;

4) $k := k - 1,\ \ l := r$.

Рассмотрим пример. Пусть правила вывода грамматики
(1) $E \to T + E$, (2) $E \to T$, (3) $T \to P * T$,
(4) $T \to P$, (5) $P \to a$, (6) $P \to (E)$
(здесь символ a используется вместо старого i)

Этап I.

Шаг 1. Составляем список I_0.
Так как $E \to T + E \in P$, то в I_0 вносится ситуация $[E \to \bullet T + E, 0]$;
так как $E \to T \in P$, то в I_0 вносится ситуация $[E \to \bullet T, 0]$.
Шаг 2. 1) Ситуаций вида $[B \to \gamma\bullet, 0]$ в I_0 нет.
2) Так как $[E \to \bullet T + E, 0] \in I_0$ и $T \to P * T \in P$,
то в I_0 вносится ситуация $[T \to \bullet P * T, 0]$;
так как $[E \to \bullet T + E, 0] \in I_0$ и $T \to P \in P$,
то в I_0 вносится ситуация $[T \to \bullet P, 0]$.
В I_0 имеется также подходящая ситуация $[E \to \bullet T, 0]$, но список I_0 не пополняется — новых ситуаций не образуется.

2) На предыдущем подшаге в I_0 появились ситуации, удовлетворяющие условию подшага 2:
так как $[T \to \bullet P * T, 0] \in I_0$ и $P \to a \in P$,
то в I_0 вносится ситуация $[P \to \bullet a, 0]$;
так как $[T \to \bullet P * T, 0] \in I_0$ и $P \to (E) \in P$,
то в I_0 вносится ситуация $[P \to \bullet(E), 0]$.
Из $[T \to \bullet P, 0] \in I_0$ ничего нового не получается.
Итак, в список I_0 было внесено всего 6 ситуаций.

Этап II.
$j = 1$. Составляем список I_1. Имеем $a_1 = a$.
Шаг 1. Ищем в I_0 ситуации с символом a после толстой точки.
Так как $[P \to \bullet a, 0] \in I_0$, то в I_1 вносим $[P \to a\bullet, 0]$.
Шаг 2. 1) Так как $[P \to a\bullet, 0] \in I_1$ и $[T \to \bullet P * T, 0] \in I_0$,
то в I_1 вносится $[T \to P \bullet * T, 0]$;
а в силу $[T \to \bullet P, 0] \in I_0$ еще и $[T \to P\bullet, 0]$.
1) В I_1 появилась ситуация $[T \to P\bullet, 0]$, поэтому подшаг повторяется.
В I_1 вносятся ситуации $[E \to T \bullet + E, 0]$ и $[E \to T\bullet, 0]$.
1) Хотя подходящая ситуация $[E \to T\bullet, 0]$ и была внесена в I_1, но в I_0 нет ситуаций, в которых выводится цепочка символов с парой $\bullet E$. Этот подшаг не дает новых ситуаций.
2) Условия этого подшага не выполняются — новых ситуаций не вносится в I_1.
Получилось, что в I_1 содержится 5 ситуаций.

Этап II.
$j = 2$. Составляем список I_2. Имеем $a_2 = *$.
Укажем схематично, как выполняются шаги алгоритма.
Шаг 1. $[T \to P \bullet * T, 0] \in I_1 \to [T \to P * \bullet T, 0] \in I_2$;
Шаг 2. 1) —
2) $[T \to P * \bullet T, 0] \in I_2$:
$T \to P * T \in P \to [T \to \bullet P * T, 2] \in I_2$,
$T \to P \in P \to [T \to \bullet P, 2] \in I_2$;
2) $[T \to \bullet P * T, 2] \in I_2$ (или $[T \to \bullet P, 2] \in I_2$) :
$P \to a \in P \to [P \to \bullet a, 2] \in I_2$,
$P \to (E) \in P \to [P \to \bullet(E), 2] \in I_2$

und so weiter :-)

Этап II.
$j = 3$. Составляем список I_3 ...

Этап II.
$j = 4$. Составляем список I_4 ...

Этап II.
$j = 5$. Составляем список I_5 ...

Этап II.
$j = 6$. Составляем список I_6 ...

Этап II.
$j = 7$. Составляем список I_7 ...