

# Основные определения

- Аксиоматика: “символ” и “множество”
- Алфавит – непустое конечное множество символов, будет обозначаться символом  $V$
- Цепочка символов в алфавите  $V$  – любая конечная последовательность символов этого алфавита
- Цепочка, которая не содержит ни одного символа, называется пустой цепочкой, для её обозначения используется символ  $\varepsilon$ , который по предположению сам в алфавит  $V$  не входит
- Длина цепочки – число составляющих её символов, например, если  $\alpha = abcdefgh$ , то длина  $\alpha$  равна 8  
Длина цепочки  $\alpha$  обозначается как  $|\alpha|$   
Длина пустой цепочки  $\varepsilon$  равна 0, то есть  $|\varepsilon| = 0$

# Основные определения

- Если  $\alpha$  и  $\beta$  – цепочки, то цепочка  $\alpha\beta$  (результат приписывания цепочки  $\beta$  в конец цепочки  $\alpha$ ) называется конкатенацией (или сцеплением) цепочек  $\alpha$  и  $\beta$
- Конкатенацию  $\alpha\beta$  можно считать двуместной операцией над цепочками  $\alpha$  и  $\beta$  ( $\alpha \cdot \beta = \alpha\beta$ ), например, если  $\alpha = ab$  и  $\beta = cd$ , то  $\alpha \cdot \beta = abcd$
- Для любой цепочки  $\alpha$  всегда  $\alpha \cdot \varepsilon = \varepsilon \cdot \alpha = \alpha$
- Конкатенация некоммутативна, то есть  $\alpha \cdot \beta \neq \beta \cdot \alpha$   
Например, если  $\alpha = ab$  и  $\beta = cd$ , имеем  $\beta \cdot \alpha = cdab$
- Конкатенация цепочек ассоциативна, для любых цепочек  $\alpha$ ,  $\beta$  и  $\gamma$  всегда верно:  $\alpha \cdot \beta \cdot \gamma = (\alpha \cdot \beta) \cdot \gamma = \alpha \cdot (\beta \cdot \gamma)$

# Основные определения

- Цепочки символов можно разбивать на подцепочки
- Цепочки символов можно заменять на другие цепочки
- В результате замены (подстановки) цепочек получаются новые цепочки символов
- Цепочку  $\gamma = abcd$  можно разбить (в том числе) на три подцепочки:  $\alpha = a$ ,  $\omega = b$  и  $\beta = cd$ , то есть исходную цепочку можно представить как  $\gamma = \alpha\omega\beta$
- Если для цепочки  $\omega$  выполнить подстановку  $u = aba$ , получится новая цепочка  $\gamma = aabacd$  ( $\gamma = \alpha u \beta$ )
- Любая подстановка выполняется с помощью операций разбиения и конкатенации

# Основные определения

- Обращением (реверсом) цепочки  $\alpha$  называется цепочка, символы которой записаны в обратном порядке, обращение цепочки  $\alpha$  обозначается как  $\alpha^R$
- Например, если  $\alpha = abcdefgh$ , то  $\alpha^R = hgfedcba$
- Для пустой цепочки  $\varepsilon = \varepsilon^R$ ,  $\varepsilon^R = \varepsilon$
- Справедливо тождество:  $\forall \alpha, \beta: (\alpha\beta)^R = \beta^R \alpha^R$
- $n$ -ой степенью цепочки  $\alpha$  ( $\alpha^n$ ) называется конкатенация  $n$  цепочек  $\alpha$  (повторение этой цепочки  $n$  раз)
- Справедливо:  $\forall \alpha: \alpha^0 = \varepsilon$ ;  $\alpha^1 = \alpha$ ;  $\alpha^2 = \alpha\alpha$ ;  $\alpha^n = \alpha\alpha^{n-1} = \alpha^{n-1}\alpha$
- Для пустой цепочки:  $\forall n \geq 0: \varepsilon^n = \varepsilon$
- Число вхождений символа  $s$  в цепочку  $\alpha$  обозначается как  $|\alpha|_s$ , например:  $|babb|_a = 1$ ,  $|babb|_b = 3$ ,  $|babb|_c = 0$

# Основные определения

- Множество  $V^*$ , содержащее все цепочки в алфавите  $V$ , включая пустую цепочку  $\varepsilon$ , называется итерацией множества  $V$
- Если  $V=\{0,1\}$ , то  $V^* = \{\varepsilon, 0, 1, 00, 11, 01, 10, 000, 001, 011, \dots\}$
- Множество  $V^+$ , содержащее все цепочки в алфавите  $V$ , исключая пустую цепочку  $\varepsilon$ , называется усечённой итерацией множества  $V$ , следовательно,  $V^* = V^+ \cup \{\varepsilon\}$
- Язык в алфавите  $V$  – это счётное подмножество цепочек конечной длины из множества всех цепочек над алфавитом  $V$
- Цепочку символов, принадлежащую некоторому языку, называют предложением языка, а множество цепочек языка – множеством предложений этого языка

# Основные определения

- Каждый язык в алфавите  $V$  является подмножеством множества  $V^*$ , то есть выполняется отношение  $L(V) \subseteq V^*$
- Язык  $L(V)$  включает в себя язык  $L'(V)$ , если любая цепочка, входящая во второй язык ( $L'(V)$ ), одновременно входит и в первый ( $L(V)$ ):

$$L'(V) \subseteq L(V), \text{ если } \forall \alpha \in L'(V): \alpha \in L(V)$$

- Два языка  $L(V)$  и  $L'(V)$  совпадают (равны), если каждый из них включает в себя второй язык:

$$L'(V) = L(V), \text{ если } L'(V) \subseteq L(V) \text{ и } L(V) \subseteq L'(V)$$

- Два языка  $L(V)$  и  $L'(V)$  почти совпадают, если они различаются только на пустую цепочку символов:

$$L'(V) \cong L(V), \text{ если } L'(V) \cup \{\varepsilon\} = L(V) \cup \{\varepsilon\}$$

# Основные определения

- Язык можно определить
  1. Перечислением всех допустимых цепочек языка
  2. Выбором механизма порождения (генерации), то есть указанием способа формирования (порождения) цепочек (заданием грамматики языка)
  3. Выбором механизма распознавания, то есть определением метода распознавания цепочек языка
- Первый метод – иллюстративный:  
$$L(\{a,b\}) = \{aa, ab, ba, bb\}$$
- Иногда этот метод модифицируют описанием множеств входящих в язык цепочек с помощью формул:  
$$L = \{\alpha \in (a, b)^+, |\alpha| = 2\} \text{ (только что показанный язык)}$$
$$L(\{0,1\}) = \{01, 0011, 000111, \dots\} \equiv L(\{0,1\}) = \{0^n 1^n, n > 0\}_7$$

# Основные определения

- Второй способ связан с определением правил, с помощью которых можно строить правильные цепочки языка из символов алфавита языка, в этом случае используются порождающие грамматики (грамматиками Холмского), которые представляют собой основной способ реализации механизма порождения
- Третий способ требует наличия логического устройства (распознавателя) – автомата, который, получая входную цепочку символов алфавита, на выходе выдаёт ответ, принадлежит ли заданная цепочка данному языку или нет



# Основные определения

- Примеры распознавателей:
  1. Машина Тьюринга (МТ) для рекурсивно-перечислимых языков
  2. Линейно ограниченный автомат (ЛОА) для *контекстно-зависимых языков*
  3. Автомат с магазинной (внешней) памятью (МП-автомат) для *контекстно-свободных языков*
  4. Конечный автомат (КА – детерминированный ДКА или недетерминированный НКА) для *регулярных языков*

# Основные определения

- При изучении языков выделяют их лексику, синтаксис и семантику
- Лексика языка – это совокупность слов (словарный запас) языка
  - *Слово языка* или его *лексическая единица (лексема)* состоит из элементов алфавита языка и не содержит в себе никаких других конструкций
- Синтаксис определяет форму языка, задаёт набор цепочек символов, которые принадлежат этому языку, фиксирует набор правил, определяющих допустимые конструкции языка
- Семантика языка определяет значение (смысл) предложений языка

# Основные определения

- Грамматикой языка называется способ построения предложений этого языка
- Для языков программирования используется формальное описание грамматики, построенной на основе правил (продукций)
- *Правило* (продукция) есть *упорядоченная* пара цепочек символов ( $\alpha$ ,  $\beta$ )
- Правила записывают в виде  $\alpha \rightarrow \beta$   
Читается: “ $\alpha$  порождает  $\beta$ ”, “из  $\alpha$  следует  $\beta$ ”

# Основные определения

- Декартовым произведением  $A \times B$  множеств  $A$  и  $B$  называется множество пар  $\{(a, b) \mid a \in A, b \in B\}$
- Определение порождающей грамматики  
Порождающая грамматика  $G$  – это четвёрка  $(T, N, P, S)$ , где
  - $T$  – алфавит терминальных символов
  - $N$  – алфавит нетерминальных символовМножества  $N$  и  $T$  не пересекаются друг с другом:  $N \cap T = \emptyset$
- $P$  – конечное множество правил – подмножество множества  $(T \cup N)^+ \times (T \cup N)^*$  (первый элемент произведения не может быть пустой цепочкой)
- $S$  – начальный символ (цель) грамматики,  $S \in N$

# Основные определения

- Множество  $V = T \cup N$  называется полным алфавитом грамматики  $G$
- Элемент  $(\alpha, \beta)$  множества  $P$  называется правилом вывода и записывается в виде  $\alpha \rightarrow \beta$ , здесь  $\alpha \in (T \cup N)^+$  – левая часть правила, а  $\beta \in (T \cup N)^*$  – правая часть правила
- Левая часть любого правила из  $P$  обязана содержать хотя бы один нетерминальный символ
- Для записи совокупности правил вывода с одинаковыми левыми частями  
 $\alpha \rightarrow \beta_1 \quad \alpha \rightarrow \beta_2 \quad \alpha \rightarrow \beta_n$   
пользуются сокращённой записью  $\alpha \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$
- Каждое  $\beta_i, i = 1, 2, \dots, n$  называется альтернативой правила вывода из цепочки  $\alpha$

# Основные определения

- Цепочка  $\beta \in (T \cup N)^*$  непосредственно выводима из цепочки  $\alpha \in (T \cup N)^+$  в грамматике  $G = (T, N, P, S)$  ( $\alpha \rightarrow_G \beta$ ), если  $\alpha = \xi_1 \gamma \xi_2$ ,  $\beta = \xi_1 \delta \xi_2$ , где  $\xi_1, \xi_2, \delta \in (T \cup N)^*$ ,  $\gamma \in (T \cup N)^+$  и правило вывода  $\gamma \rightarrow \delta$  содержится в  $P$
- Цепочка  $\beta \in (T \cup N)^*$  выводима из цепочки  $\alpha \in (T \cup N)^+$  в грамматике  $G = (T, N, P, S)$  ( $\alpha \Rightarrow_G \beta$ ), если существуют цепочки  $\gamma_0, \gamma_1, \dots, \gamma_n$  ( $n \geq 0$ ):  $\alpha = \gamma_0 \rightarrow \gamma_1 \rightarrow \dots \rightarrow \gamma_n = \beta$
- Последовательность цепочек  $\gamma_0, \gamma_1, \dots, \gamma_n$  называется выводом цепочки  $\beta$  из цепочки  $\alpha$  длины  $n$
- Цепочка  $00A11$  непосредственно выводима из  $0A1$  в  $G_1$
- Цепочка  $000A111$  выводима из  $S$  ( $S \Rightarrow 000A111$ ) в  $G_1$ , так как имеется вывод  $S \rightarrow 0A1 \rightarrow 00A11 \rightarrow 000A111$  (длина вывода равна 3)

# Основные определения

- Определение языка, порождаемого грамматикой  
Языком, порождаемым грамматикой  $G = (T, N, P, S)$ , называется множество  $L(G) = \{\alpha \in T^* \mid S \Rightarrow \alpha\}$
- $L(G)$  – это все цепочки в алфавите  $T$ , которые выводимы из  $S$  с помощью  $P$ . Например,  $L(G_1) = \{0^n 1^n \mid n > 0\}$
- Задачи на поиск языка, порождаемого некоторой грамматикой, решаются путём последовательного применения подстановок в правилах грамматики:

$$S \rightarrow 0A1 \rightarrow (A \rightarrow \varepsilon) 01$$

$$S \rightarrow 0A1 \rightarrow (0A \rightarrow 00A1) 00A11 \rightarrow (A \rightarrow \varepsilon) 0011$$

$$S \rightarrow 0A1 \rightarrow (0A \rightarrow 00A1) 00A11 \rightarrow (0A \rightarrow 00A1) 000A111 \rightarrow \dots \Rightarrow (0A \rightarrow 00A1) 0^n A 1^n \rightarrow (A \rightarrow \varepsilon) 0^n 1^n$$

# Основные определения

- Цепочка  $\alpha \in (T \cup N)^*$ , для которой  $S \Rightarrow \alpha$ , называется сентенциальной формой в грамматике  $G = (T, N, P, S)$
- Цепочка  $\alpha \in T^*$ , полученная в результате законченного вывода, называется конечной сентенциальной формой
- Язык, порождаемый грамматикой  $G$ , можно определить как множество конечных (терминальных) сентенциальных форм грамматики  $G$
- Алфавитом языка  $L(G)$  является множество терминальных символов грамматики  $T$



# Основные определения

- Определение эквивалентности грамматик

Грамматики  $G_1$  и  $G_2$  называются эквивалентными, если их языки совпадают:  $L(G_1) = L(G_2)$

- $G_1$  и  $G_2$  эквивалентны, порождается язык  $L = \{0^n 1^n \mid n > 0\}$ :

$$\begin{array}{ll} G_1 = (\{0,1\}, \{A,S\}, P_1, S) & \text{и} & G_2 = (\{0,1\}, \{S\}, P_2, S) \\ P_1: S \rightarrow 0A1 & & P_2: S \rightarrow 0S1 \mid 01 \\ & & 0A \rightarrow 00A1 \\ & & A \rightarrow \varepsilon \end{array}$$

- Грамматики  $G_1$  и  $G_2$  почти эквивалентны, если совпадают языки  $L(G_1) \cup \{\varepsilon\} = L(G_2) \cup \{\varepsilon\}$

- $G_1$  и  $G_3$  почти эквивалентны:  $L(G_3) = \{0^n 1^n \mid n \geq 0\}$ :

$$G_3 = (\{0,1\}, \{S\}, P_3, S) \qquad P_3: S \rightarrow 0S1 \mid \varepsilon$$

# Основные определения

- Грамматики обладают свойством эквивалентности, если совпадают заданные ими языки, то есть, если  $L(G) = L(G')$
- Эквивалентные грамматики должны иметь, по крайней мере, пересекающиеся (чаще совпадающие) множества терминальных символов  $T \cap T' \neq \emptyset$
- Множества нетерминальных символов, правила и начальный символ эквивалентных грамматик могут существенно различаться
- *Следствие*: один язык может описываться разными грамматиками, что влияет на выбор методов анализа языков в компиляторах

# Типы грамматик и языков

- Выделяют четыре входящих друг в друга категории (*типа*) языков с соответствующими входящими друг в друга грамматиками
- Грамматики классифицируются по виду их правил вывода
- Чтобы отнести грамматику к какому-либо *типу*, все правила этой грамматики должны относиться к этому *типу*
- Каждому *типу* грамматик соответствует свой *класс* языков
- Если язык порождается грамматикой типа  $i$  (для  $i = 0, 1, 2, 3$ ), то он является языком типа  $i$

# Типы грамматик и языков

- **ТИП 0:** Грамматика  $G = (T, N, P, S)$  называется грамматикой типа 0, если на правила вывода не накладывается никаких ограничений (кроме тех, которые указаны в определении грамматики):

$$\alpha \rightarrow \beta \qquad \alpha \in (T \cup N)^+, \beta \in (T \cup N)^*$$

- В грамматиках *типа 0* можно встретить такие правила:
  - a)  $aAbCD \rightarrow aHD$  (то есть  $AbC \rightarrow H$  в контексте  $a \dots D$ )
  - b)  $PQ \rightarrow QR$
- *Класс языков типа 0 совпадает с классом рекурсивно-перечислимых языков*

# Типы грамматик и языков

- **ТИП 1:** Грамматика  $G = (T, N, P, S)$  называется неукорачивающей грамматикой, если левая часть каждого правила из  $P$  не длинее правой части, то есть каждое правило из  $P$  имеет вид:

$$\alpha \rightarrow \beta, \text{ где } \alpha \in V^+, \beta \in V^+ \text{ и } |\alpha| \leq |\beta|$$

- В неукорачивающей грамматике допускается наличие правила  $S \rightarrow \varepsilon$ , при условии, что  $S$  (начальный символ) не встречается в правых частях правил грамматики
- Грамматика с правилами  $\{ S \rightarrow Aaa \mid \varepsilon, Aa \rightarrow Sa \}$  не является неукорачивающей: Символ  $S$  встречается в правой части правила  $Aa \rightarrow Sa$ , и при выводе сентенциальная форма “укорачивается”:  $S \rightarrow Aaa \rightarrow Saa \rightarrow aa$

# Типы грамматик и языков

- **ТИП 1:** Грамматика  $G = (T, N, P, S)$  называется контекстно-зависимой, если каждое правило из  $P$  с непустой правой частью имеет вид:  
 $\alpha \rightarrow \beta$ , где  $\alpha = \xi_1 A \xi_2$ ,  $\beta = \xi_1 \gamma \xi_2$ ,  $A \in N$ ,  $\gamma \in V^+$ ,  $\xi_1, \xi_2 \in V^*$
- В контекстно-зависимой грамматике допускается наличие правила  $S \rightarrow \varepsilon$ , при условии, что  $S$  (начальный символ) не встречается в правых частях правил грамматики
- Цепочку  $\xi_1$  называют *левым контекстом*, а цепочку  $\xi_2$  называют *правым контекстом*
- *Язык, порождаемый контекстно-зависимой грамматикой, называется контекстно-зависимым*

# Типы грамматик и языков

- **ТИП 1:** Грамматика *типа 1* определяется как неукорачивающая грамматика
- **ТИП 1:** Грамматика *типа 1* определяется как контекстно-зависимая грамматика
- Из определений следует, что если язык, порождаемый контекстно-зависимой или неукорачивающей грамматикой  $G = (T, N, P, S)$ , содержит пустую цепочку, то эта цепочка выводится в  $G$  за один шаг с помощью правила  $S \rightarrow \varepsilon$
- Других выводов для цепочки  $\varepsilon$  в грамматике  $G$  не существует

# Типы грамматик и языков

- Если  $L$  — формальный язык, то такие утверждения эквивалентны:
- $\exists$  такая контекстно-зависимая грамматика  $G_1$ , что  $L = L(G_1)$
- $\exists$  такая неукорачивающая грамматика  $G_2$ , что  $L = L(G_2)$
- Контекстно-зависимая грамматика является неукорачивающей: в правилах контекстно-зависимых грамматик  $(\xi_1 A \xi_2 \rightarrow \xi_1 \gamma \xi_2)$  из сохранения контекста следует, что его длина  $(|\xi_1| + |\xi_2|)$  не меняется,  $|A| = 1$ ,  $|\gamma| > 0$ ,  $|A| \leq |\gamma|$
- Существует доказательство и обратной эквивалентности



# Типы грамматик и языков

- **ТИП 2:** Грамматика  $G = (T, N, P, S)$  называется контекстно-свободной, если каждое правило из  $P$  имеет вид:

$$A \rightarrow \beta, \quad \text{где } A \in N, \quad \beta \in (T \cup N)^*$$

- Цепочка  $\beta$  (правая часть правила) может быть пустой
- Грамматика *типа 2* – это контекстно-свободная грамматика
- Язык, порождаемый контекстно-свободной грамматикой, называется контекстно-свободным языком

# Типы грамматик и языков

- Среди контекстно-свободных грамматик выделяют неукорачивающие контекстно-свободные грамматики
- Такие грамматики имеют только правила с непустыми правыми частями (за одним исключением:  $S \rightarrow \varepsilon$ ), они являются частным случаем контекстно-зависимых грамматик с пустым контекстом,  $|\xi_1| + |\xi_2| = 0$
- Неукорачивающие контекстно-свободные грамматики отличаются тем, что правая часть их правил всегда не короче соответствующей левой части, то есть  $\beta \in (T \cup N)^+$  и  $|\beta| \geq 1$

# Типы грамматик и языков

- **ТИП 3:** Грамматика  $G = (T, N, P, S)$  называется праволинейной, если каждое правило из  $P$  имеет вид:  
 $A \rightarrow \gamma B$  либо  $A \rightarrow \gamma$  где  $A \in N, B \in N, \gamma \in T^*$
- **ТИП 3:** Грамматика  $G = (T, N, P, S)$  называется леволинейной, если каждое правило из  $P$  имеет вид:  
 $A \rightarrow B\gamma$  либо  $A \rightarrow \gamma$  где  $A \in N, B \in N, \gamma \in T^*$
- Грамматика *типа 3* – это *праволинейная грамматика*
- Множество языков, порождаемых праволинейными грамматиками, совпадает с множеством языков, порождаемых леволинейными грамматиками
- Если для некоторого формального языка существует право(лево-)линейная грамматика, то одновременно для него существует и лево(право-)линейная грамматика

# Типы грамматик и языков

- Языки, порождаемые грамматиками типа 3, называются регулярными
- Каждая грамматика типа 1 (а значит, и типа 2) может быть преобразована к грамматике типа 3, если можно избавиться от правил вида  $A \rightarrow \varphi A \psi$  (где  $\varphi$  и  $\psi$  не пусты), в противном случае они называются грамматиками с самовставлением
- К регулярным грамматикам не относятся грамматики, в которых смешаны праволинейные и леволинейные правила, такие грамматики следует относить к классу контекстно-свободных грамматик

# Типы грамматик и языков

- Регулярные грамматики могут быть неукорачивающими, если в них не встречаются правила с пустой правой частью
- Среди неукорачивающих регулярных грамматик выделяются автоматные грамматики
- Автоматные грамматики могут быть лево- и праволинейными
- Леволинейные автоматные грамматики имеют правила видов:  $A \rightarrow Bt$  или  $A \rightarrow t$ , где  $A, B \in N, t \in T$
- Праволинейные автоматные грамматики имеют правила видов:  $A \rightarrow tB$  или  $A \rightarrow t$ , где  $A, B \in N, t \in T$

# Типы грамматик и языков

- *Классы обычных и автоматных регулярных грамматик почти эквивалентны*
- Чтобы классы этих грамматик стали полностью эквивалентными, в автоматные грамматики вводят дополнительное правило вида  $S \rightarrow \varepsilon$ , где  $S$  – начальный символ грамматики
- Существует алгоритм преобразования произвольной регулярной грамматики к автоматному виду
- Для любой регулярной (автоматной) грамматики  $G$  существует неукорачивающая регулярная (автоматная) грамматика  $G'$ , такая что  $L(G) = L(G')$

# Отношения между грамматиками

- Регулярные грамматики являются контекстно-свободными
- Неукорачивающие контекстно-свободные грамматики являются контекстно-зависимыми
- Неукорачивающие грамматики относятся к *типу 0*
- Иерархия типов грамматик

*Неукорачивающие регулярные*

$\subset$  *Неукорачивающие контекстно-свободные*

$\subset$  *Контекстно-зависимые (либо неукорачивающие)*

$\subset$  *тип 0*

- Не все контекстно-свободные грамматики являются неукорачивающими: *тип 2* не вкладывается в *тип 1*

# Соотношения между языками

- Язык  $L(G)$  является языком типа  $k$ , если его можно описать грамматикой *типа  $k$*
- Языки классифицируются в соответствии с типами грамматик, с помощью которых они описываются
- Для классификации языка выбирается грамматика с максимальным классификационным типом

- Грамматика *типа 0*  
и грамматика *типа 2*

где  $P_1: S \rightarrow 0A1$

$0A \rightarrow 00A1$

$A \rightarrow \varepsilon$

свободный язык

$G_1 = (\{0,1\}, \{A,S\}, P_1, S)$

$G_2 = (\{0,1\}, \{S\}, P_2, S)$

$P_2: S \rightarrow 0S1 \mid 01$

описывают контекстно-

$L = L(G_1) = L(G_2) = \{0^n 1^n \mid n > 0\}$



# Соотношения между языками

- Языки с фразовой структурой (рекурсивно перечислимые множества) могут быть заданы только грамматикой *типа 0*
- Контекстно-зависимые языки применяются в анализе и переводе текстов на естественных языках, алгоритмы разбора контекстно-зависимых текстов имеют экспоненциальную сложность
- Современные языки программирования описываются с помощью контекстно-свободных грамматик
- Регулярные языки используются для лексического распознавания идентификаторов и констант

# Соотношения между языками

- Для языков существуют доказанные факты, которые можно использовать при их классификации:

языки  $L = \{ a^n / n \geq 1 \}$

регулярные

языки  $L = \{ a^n b^n / n \geq 1 \}$

контекстно-свободные

языки  $L = \{ a^n b^n c^n / n \geq 1 \}$

контекстно-зависимые

- В регулярных языках повторяется только одна последовательность символов –  $f(ab)^n c$
- В контекстно-свободных языках повторяются две цепочки –  $a^{3n+1} b^{n-2}$
- В контекстно-зависимых языках повторяются три цепочки

# Соотношения между языками

- Каждый регулярный язык является контекстно-свободным языком, но существуют контекстно-свободные языки, которые не являются регулярными
  - язык  $L = \{ a^n b^n \mid n > 0 \}$  не регулярный
  - язык  $L = \{ a^n b^m \mid n, m > 0 \}$  регулярный
- Каждый контекстно-свободный язык является контекстно-зависимым, но существуют контекстно-зависимые языки, которые не являются контекстно-свободными ( $L = \{ a^n b^n c^n \mid n > 0 \}$ )
- Каждый контекстно-зависимый язык является языком типа 0, но существуют языки типа 0, которые не являются контекстно-зависимыми

# Соотношения между языками

- Иерархия классов языков:

*Тип 3 (регулярные)*

$\subset$  *Тип 2 (контекстно-свободные)*

$\subset$  *Тип 1 (контекстно-зависимые)*

$\subset$  *Тип 0*

- Диаграммы Венна



# Цепочки вывода

- Цепочка принадлежит языку, порождаемому грамматикой, если существует её вывод из начального символа этой грамматики
- Процесс построения такого вывода называется разбором (нисходящим или восходящим)
- Последовательность непосредственно выводимых цепочек языка называется выводом или цепочкой вывода
- Каждый переход от одной непосредственно выводимой цепочки к следующей называется шагом вывода

# Цепочки вывода

- Цепочка  $\beta$  может быть *выводима* из цепочки  $\alpha$  ( $\alpha \Rightarrow \beta$ ) за  $0$  или более шагов
- Если цепочка  $\beta$  *непосредственно выводима* из цепочки  $\alpha$  ( $\alpha \rightarrow \beta$ ), то имеется ровно один шаг вывода
- Если цепочка вывода из  $\alpha$  к  $\beta$  содержит одну или более промежуточных цепочек, цепочка  $\beta$  называется нетривиально выводимой из цепочки  $\alpha$
- Вывод цепочки  $\beta$  называется законченным, если на её основе нельзя сделать ни одного шага вывода
- Законченный вывод возможен, если конечная цепочка вывода  $\beta$  пуста, или содержит только терминальные символы

# Разбор по КС-грамматике

- Вывод цепочки  $\beta \in T^*$  из  $S \in N$  в грамматике  $G = (T, N, P, S)$  называется левым/левосторонним (правым/правосторонним), если в этом выводе каждая очередная сентенциальная форма получается из предыдущей формы заменой самого левого (правого) нетерминального символа
- Разные выводы для цепочки  $a+b+a$  в грамматике  
 $G = (\{a, b, +\}, \{S, T\}, \{S \rightarrow T \mid T+S; T \rightarrow a \mid b\}, S)$ 
  - (1)  $S \rightarrow T+S \rightarrow T+T+S \rightarrow T+T+T \rightarrow a+T+T \rightarrow a+b+T \rightarrow a+b+a$
  - (2)  $S \rightarrow T+S \rightarrow a+S \rightarrow a+T+S \rightarrow a+b+S \rightarrow a+b+T \rightarrow a+b+a$
  - (3)  $S \rightarrow T+S \rightarrow T+T+S \rightarrow T+T+T \rightarrow T+T+a \rightarrow T+b+a \rightarrow a+b+a$

# Дерево вывода

- Ориентированное упорядоченное помеченное дерево (граф) называется деревом вывода (или деревом разбора) в контекстно-свободной грамматике  $G = \{T, N, P, S\}$ , если выполнены следующие условия:
  - а) каждая вершина и каждый лист дерева помечены символом из множества терминальных и нетерминальных символов  $N \cup T \cup \{\varepsilon\}$ , при этом корень дерева помечен начальным (нетерминальным) символом  $S$ , промежуточные вершины помечены нетерминальными символами из  $N$ , а листья – символами из множества  $T \cup \{\varepsilon\}$



# Дерево вывода

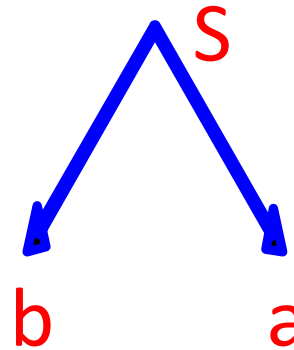
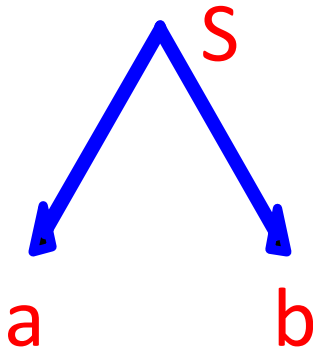
- Ориентированное упорядоченное помеченное дерево (граф) называется деревом вывода (или деревом разбора) в контекстно-свободной грамматике  $G = \{T, N, P, S\}$ , если выполнены следующие условия:
  - b) если вершина дерева помечена нетерминальным символом  $A \in N$ , а её непосредственные потомки помечены перечисленными слева направо терминальными или нетерминальными символами  $a_1, a_2, \dots, a_n$ , где каждое  $a_i \in (T \cup N)$ , то в грамматике  $G$  существует правило вывода  $A \rightarrow a_1 a_2 \dots a_n \in P$

# Дерево вывода

- Ориентированное упорядоченное помеченное дерево (граф) называется деревом вывода (или деревом разбора) в контекстно-свободной грамматике  $G = \{T, N, P, S\}$ , если выполнены следующие условия:
  - с) если вершина дерева помечена нетерминальным символом  $A \in N$ , а её непосредственный потомок помечен символом  $\varepsilon$ , то этот потомок является листом дерева, а в грамматике  $G$  существует правило вывода  $A \rightarrow \varepsilon$

# Дерево вывода

- Два разных упорядоченных дерева:

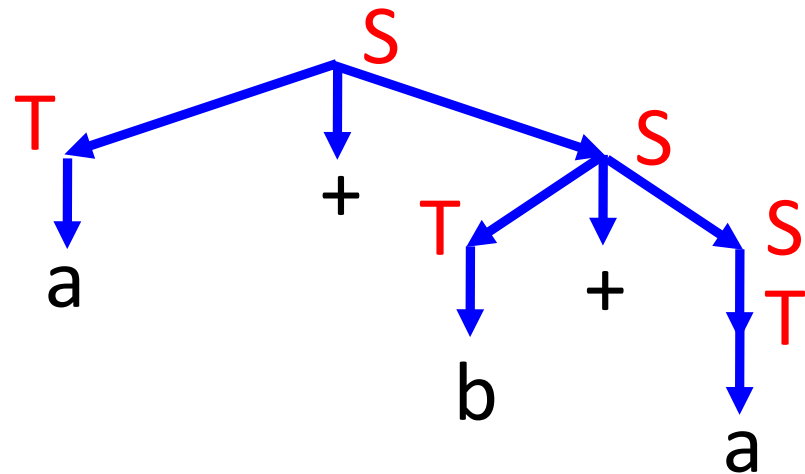


- Дерево вывода для цепочки  $a+b+a$  в грамматике

$$G = (\{a, b, +\}, \{S, T\}, P, S)$$

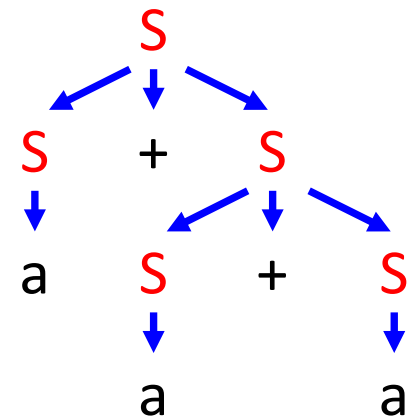
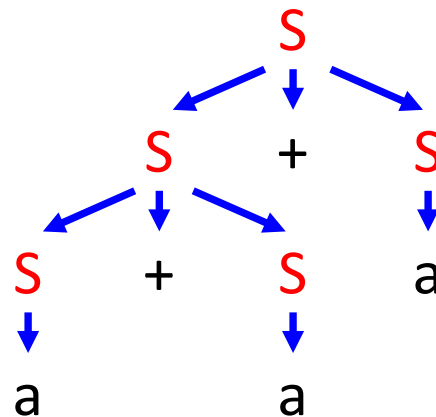
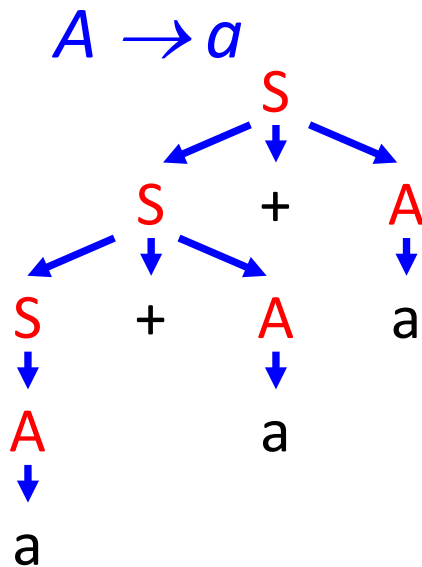
$$P: S \rightarrow T \mid T+S$$

$$T \rightarrow a \mid b$$



# Проблема однозначности

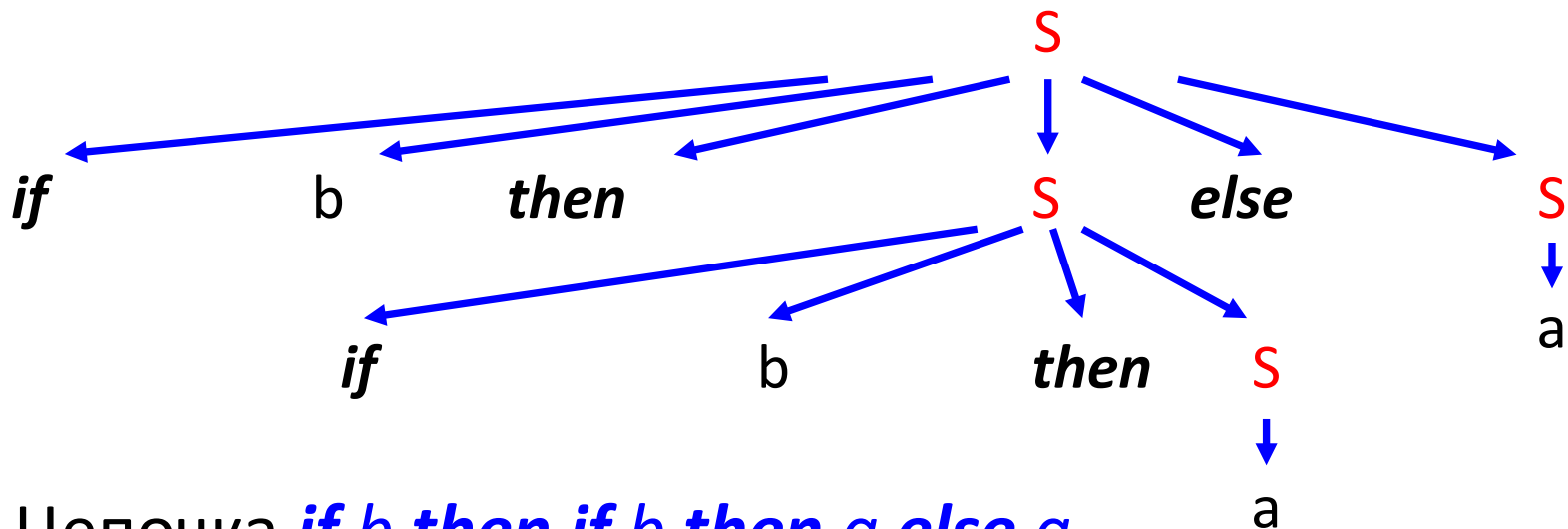
- Грамматика  $G$  называется неоднозначной, если существует (хотя бы одна) цепочка  $\alpha \in L(G)$ , для которой может быть построено два или более различных деревьев вывода
- $G_1 = (\{a, +\}, \{S, A\}, P_1, S)$   
 $P_1: S \rightarrow S + A \mid A$   
 $A \rightarrow a$   
 $G_2 = (\{a, +\}, \{S\}, P_2, S)$   
 $P_2: S \rightarrow S + S \mid a$



Цепочка  $a + a + a$

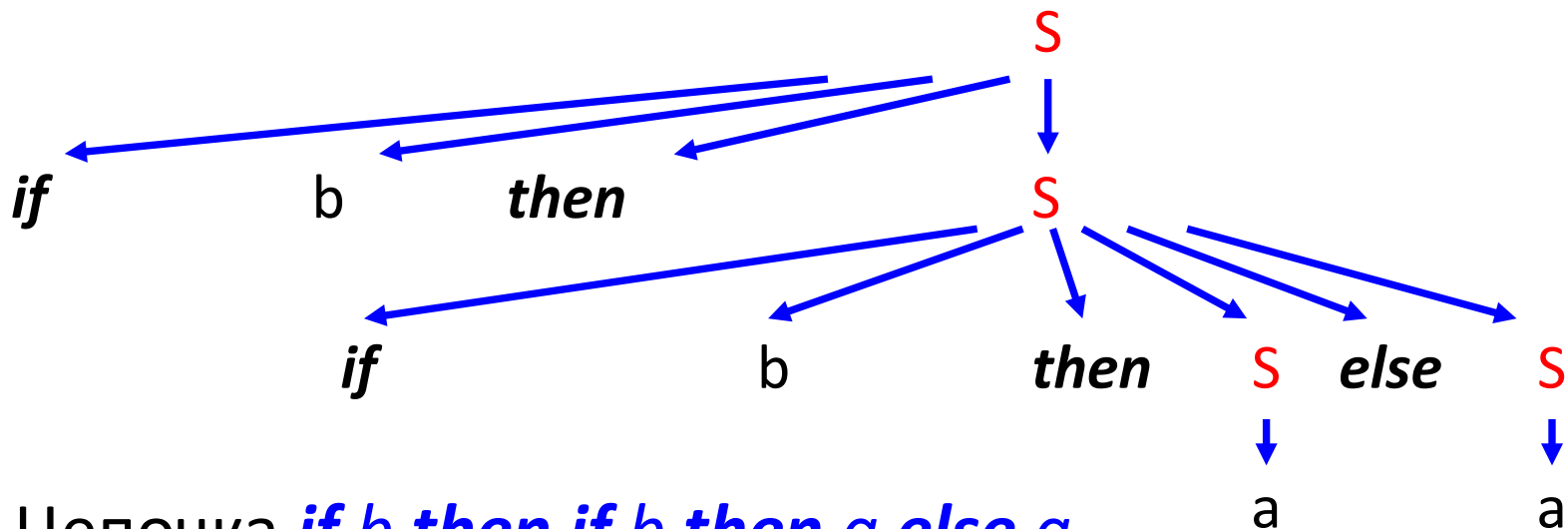
# Проблема однозначности

- Язык, порождаемый грамматикой, называется неоднозначным, если он *не может быть* порождён никакой однозначной грамматикой
- $G = (\{if, then, else, a, b\}, \{S\}, P, S)$   
 $P: S \rightarrow if\ b\ then\ S\ else\ S \mid if\ b\ then\ S \mid a$



# Проблема однозначности

- Язык, порождаемый грамматикой, называется неоднозначным, если он *не может быть* порождён никакой однозначной грамматикой
- $G = (\{if, then, else, a, b\}, \{S\}, P, S)$   
 $P: S \rightarrow if\ b\ then\ S\ else\ S \mid if\ b\ then\ S \mid a$



Цепочка *if b then if b then a else a*

# Проблема однозначности

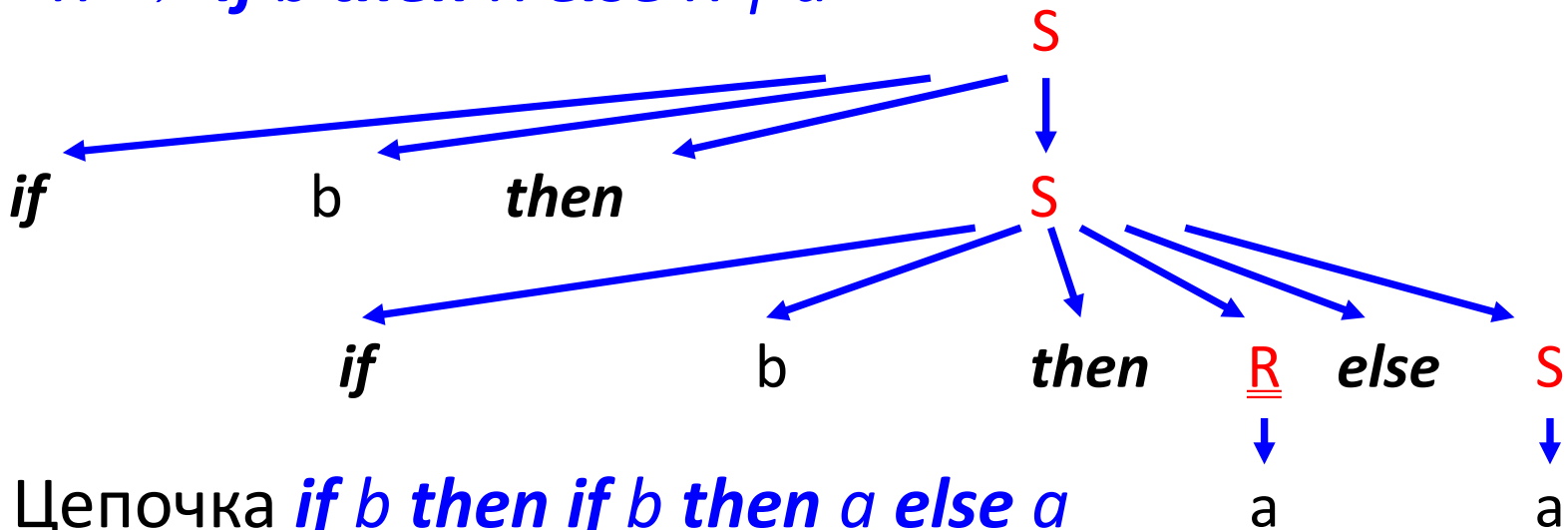
- Неоднозначность - это свойство грамматики, а не языка: для некоторых неоднозначных грамматик существуют эквивалентные им однозначные грамматики
- $G = (\{+, -, *, /, (, ), a, b\}, \{S\}, P, S)$

$$P_H: S \rightarrow S + S \mid S - S \mid S * S \mid S / S \mid (S) \mid a \mid b$$

$$\begin{aligned} P_O: S &\rightarrow S + T \mid S - T \mid T \\ T &\rightarrow T * E \mid T / E \mid E \\ E &\rightarrow (S) \mid a \mid b \end{aligned}$$

# Проблема однозначности

- Грамматика, используемая для определения языка программирования, должна быть однозначной
- $G = (\{\text{if, then, else, } a, b\}, \{S, R\}, P, S)$   
 $P: S \rightarrow \text{if } b \text{ then } R \text{ else } S \mid \text{if } b \text{ then } S \mid a$   
 $R \rightarrow \text{if } b \text{ then } R \text{ else } R \mid a$





# Проблема однозначности

- Для контекстно-свободных грамматик можно указать некоторые виды правил вывода, которые заведомо приводят к неоднозначности:

$$A \rightarrow AA \mid \alpha \qquad A \in N$$

$$A \rightarrow A\alpha A \mid \beta \qquad \alpha, \beta, \gamma \in (N \cup T)^*$$

$$A \rightarrow \alpha A \mid A\beta \mid \gamma$$

$$A \rightarrow \alpha A \mid \alpha A\beta A \mid \gamma$$

- Отсутствие перечисленных правил является необходимым, но не достаточным условием однозначности

# Бесплодные символы

- Символ  $A \in N$  называется бесплодным в грамматике  $G = (T, N, P, S)$ , если множество  $\{ \alpha \in T^* \mid A \Rightarrow \alpha \}$  пусто
- Алгоритм удаления бесплодных символов**  
**Вход:** Грамматика  $G = (T, N, P, S)$   
**Выход:** Грамматика  $G' = (T', N', P', S')$   
**Метод:** Рекурсивно строятся множества символов  $N_0, N_1, \dots$ 
  - $N_0 = \emptyset, i = 1$
  - $N_i = N_{i-1} \cup \{ A \mid (A \rightarrow \alpha) \in P, A \in N \text{ и } \alpha \in (N_{i-1} \cup T)^* \}$
  - Если  $N_i \neq N_{i-1}$ , то  $i = i + 1$  и переход к шагу 2
  - Иначе  $N' = N_i$                        $T' = T$                        $S' = S$
- $P'$  состоит из правил множества  $P$ , содержащих только символы из  $N' \cup T'$
- В грамматике  $G'$  нет бесплодных символов,  $L(G) = L(G')$

# Недостижимые символы

- Символ  $x \in (T \cup N)$  называется недостижимым в грамматике  $G = (T, N, P, S)$ , если он не появляется ни в одной сентенциальной форме этой грамматики
- Алгоритм удаления недостижимых символов**  
**Вход:** Грамматика  $G = (T, N, P, S)$   
**Выход:** Грамматика  $G' = (T', N', P', S')$   
**Метод:** Рекурсивно строятся множества символов  $V_0, V_1, \dots$ 
  - $V_0 = \{S\}, i = 1$
  - $V_i = V_{i-1} \cup \{x \mid x \in (T \cup N), (A \rightarrow \alpha x \beta) \in P, A \in V_{i-1}, \alpha, \beta \in (T \cup N)^*\}$
  - Если  $V_i \neq V_{i-1}$ , то  $i = i + 1$  и переход к шагу 2
  - Иначе  $N' = V_i \cap N$        $T' = V_i \cap T$        $S' = S$
- $P'$  состоит из правил множества  $P$ , содержащих только символы из  $V_i$
- В грамматике  $G'$  нет недостижимых символов,  $L(G) = L(G')$

# Правила с пустой правой частью

- $\varepsilon$ -правилами называются все правила вида  $A \rightarrow \varepsilon, A \in N$
- Контекстно-свободная грамматика называется *грамматикой без  $\varepsilon$ -правил*, если в ней не существует правил  $(A \rightarrow \varepsilon) \in P, A \neq S$  и существует только одно правило  $(S \rightarrow \varepsilon) \in P$  в том случае, когда  $\varepsilon \in L(G)$ , и при этом  $S$  не встречается в правой части ни одного правила
- Существует алгоритм преобразования произвольной контекстно-свободной грамматики к виду без  $\varepsilon$ -правил, то есть её преобразования к неукорачивающей контекстно-свободной грамматике

# Удаление $\varepsilon$ -правил

- Алгоритм удаления  $\varepsilon$ -правил

**Вход:** Грамматика  $G = (T, N, P, S)$

**Выход:** Грамматика  $G' = (T', N', P', S')$

1.  $X_0 = \{A: (A \rightarrow \varepsilon) \in P\}; i = 1$
2.  $X_i = X_{i-1} \cup \{A: A \in N, (A \rightarrow \alpha) \in P, \alpha \in X_{i-1}^*\}$
3. Если  $X_i \neq X_{i-1}$ , то  $i = i + 1$  и переход к шагу 2, иначе к шагу 4  
В  $X_i$  входят символы из  $N$ , из которых выводятся  $\varepsilon$
4.  $N' = N$        $T' = T$       в  $P'$  входят все правила из  $P$ ,  
кроме правил вида  $A \rightarrow \varepsilon$
5. Если  $(A \rightarrow \alpha) \in P$  и в цепочку  $\alpha$  входят символы из множества  $X_i$ , на основе этой цепочки  $\alpha$  строится новое множество цепочек  $\{\alpha'\}$  путём поочерёдного исключения из  $\alpha$  всех возможных комбинаций вхождений символов  $X_i$ , все правила вида  $A \rightarrow \alpha'$  добавляются в  $P'$

# Удаление $\varepsilon$ -правил

- Изменять  $P'$  нужно следующим образом: для любого  $A \in X_i$  правило вида  $B \rightarrow \alpha_1 A \alpha_2 A \dots \alpha_n A \alpha_{n+1}$ , где  $\alpha_i \in ((N - \{A\}) \cup T)$ , заменить  $2^n$  правилами, соответствующими всем возможным комбинациям вхождений  $A$  между  $\alpha_i$ :

$$B \rightarrow \alpha_1 \alpha_2 \dots \alpha_n \alpha_{n+1}$$

$$B \rightarrow \alpha_1 \alpha_2 \dots \alpha_n A \alpha_{n+1}$$

...

$$B \rightarrow \alpha_1 \alpha_2 A \dots \alpha_n A \alpha_{n+1}$$

$$B \rightarrow \alpha_1 A \alpha_2 A \dots \alpha_n A \alpha_{n+1}$$

- Правила вида  $B \rightarrow \varepsilon$  в множество  $P'$  не включаются
- 6. Если  $S \in X_i$ , значит  $\varepsilon \in L(G)$ , в  $N'$  добавляется новый символ  $S'$ , который становится начальным символом грамматики  $G'$ , в  $P'$  добавляются два новых правила:  $S' \rightarrow \varepsilon \mid S$ ; иначе  $S' = S$
- 7.  $G' = (T', N', P', S')$

# Циклические правила

- Циклом (циклическим выводом) в грамматике  $G(T, N, P, S)$  называется вывод вида  $A \Rightarrow A$
- Циклы возможны только в том случае, если в грамматиках языков присутствуют цепные правила вида  $A \rightarrow B$ , где  $A, B \in N$
- В процессе работы алгоритма цепных правил могут вновь возникать бесплодные и недостижимые символы и правила, их содержащие

# Циклические правила

- Алгоритм удаления цепных правил

**Вход:** Грамматика  $G = (T, N, P, S)$

**Выход:** Грамматика  $G' = (T', N', P', S')$

1. Для всех символов  $X$  из  $N$  повторять шаги 2-4, затем переход к шагу 5
2.  $N^x_0 = \{X\}; i = 1$
3.  $N^x_i = N^x_{i-1} \cup \{B: (A \rightarrow B) \in P, B \in N^{x_{i-1}*}\}$
4. Если  $N^x_i \neq N^x_{i-1}$ , то  $i = i + 1$  и переход к шагу 3, иначе установить  $N^x_i = N^x_{i-1} - \{X\}$  и продолжить цикл по шагу 1
5.  $N' = N; T' = T; S' = S$   
в  $P'$  входят все правила из  $P$ , кроме правил вида  $A \rightarrow B$
6. Для всех правил  $(A \rightarrow \alpha) \in P'$ , если  $B \in N^A, B \neq A$   
в  $P'$  добавляются правила вида  $B \rightarrow \alpha$
7.  $G' = (T', N', P', S')$



# Приведение грамматик

- В процессе работы алгоритма удаления  $\epsilon$ -правил могут вновь возникать бесплодные и недостижимые символы и правила, их содержащие
- Если применить алгоритм удаления  $\epsilon$ -правил к регулярной (автоматной) грамматике, результатом будет неукорачивающая регулярная (автоматная) грамматика
- *Контекстно-свободная грамматика называется приведённой, если в ней нет бесполезных (бесплодных и недостижимых) символов*
- **Алгоритм приведения контекстно-свободной грамматики**
  1. Обнаруживаются и удаляются все бесплодные символы
  2. Обнаруживаются и удаляются все недостижимые символы
- Удаление символов сопровождается удалением правил вывода, содержащих эти символы

# Соглашения о грамматиках

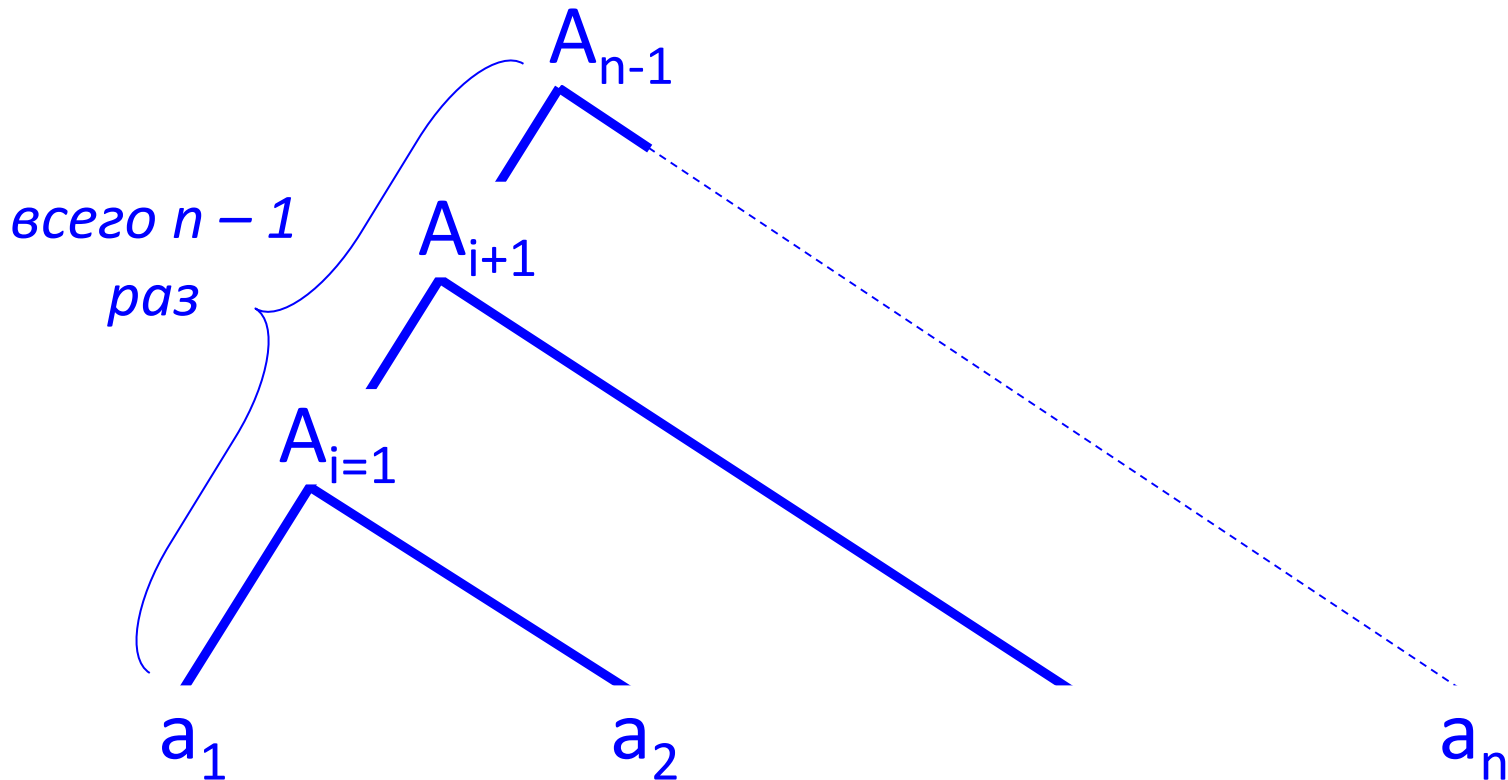
- Под регулярной грамматикой будем обычно понимать неукорачивающую леволинейную автоматную грамматику (в их правилах нет пустых правых частей):  $A \rightarrow Bt$  или  $A \rightarrow t$   
где  $A, B \in N, t \in T$
- Аналогичные праволинейные грамматики имеют правила видов:  $A \rightarrow tB$  или  $A \rightarrow t$   
где  $A, B \in N, t \in T$
- Все анализируемые цепочки заканчиваются специальным терминальным символом  $\perp$   
– признаком конца цепочки

# Алгоритм определения принадлежности цепочки языку

- Алгоритм определения принадлежности цепочки языку, порождаемому регулярной грамматикой:
  1. Первый символ исходной цепочки  $a_1a_2...a_n$  заменить нетерминалом  $A_1$ , для которого в грамматике есть правило вывода  $A_1 \rightarrow a_1$
  2. Многократно (до признака конца цепочки) выполнять шаги: полученный нетерминал  $A_{i-1}$  и очередной терминал  $a_i$  цепочки заменить нетерминалом  $A_i$ , для которого в грамматике есть правило вывода  $A_i \rightarrow A_{i-1}a_i$  ( $i = 2, 3, ..., n$ )

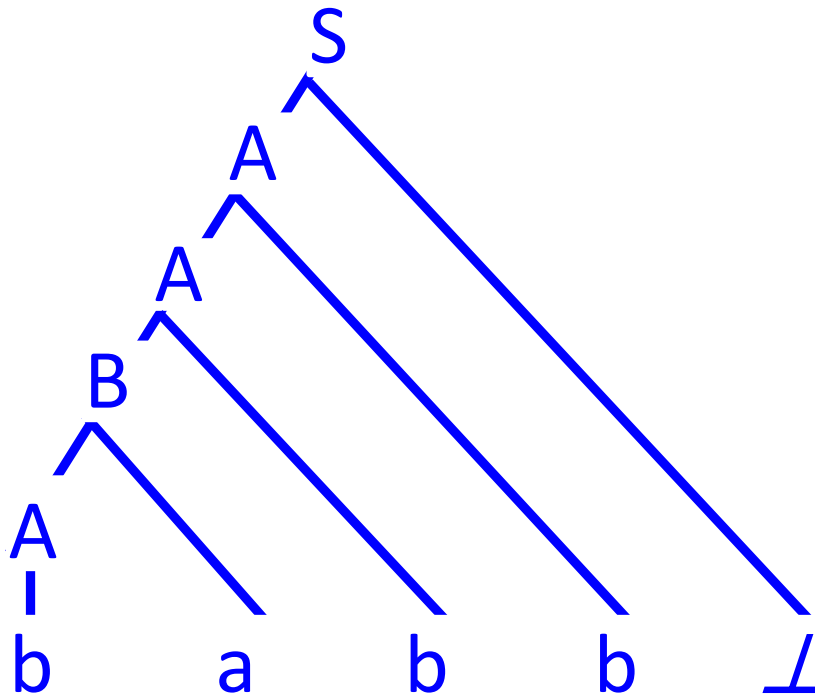
# Построение дерева разбора методом “снизу-вверх”

- Работа алгоритма разбора цепочки языка, порождаемого регулярной грамматикой:



# Работа алгоритма разбора по регулярной грамматике

- а) на последнем шаге свёртка произошла к символу  $S$ : цепочка принадлежит языку ( $a_1a_2...a_n\perp \in L(G)$ )



$G = (\{a, b\}, \{A, B, S\}, P, S)$

$P: A \rightarrow Ab \mid Bb \mid b$

$B \rightarrow Aa$

$S \rightarrow A\perp$

Разбор правильной  
цепочки  $babbb\perp$

применением правил

$A \rightarrow b$        $Aabbb\perp$

$B \rightarrow Aa$        $Bbbb\perp$

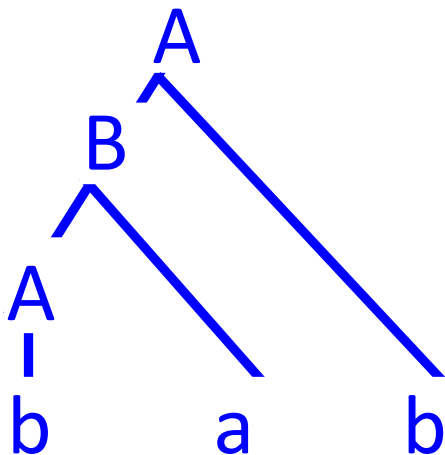
$A \rightarrow Bb$        $Ab\perp$

$A \rightarrow Ab$        $A\perp$

$S \rightarrow A\perp$        $S$

# Работа алгоритма разбора по регулярной грамматике

- b) на последнем шаге свёртка произошла к символу, отличному от  $S$ : цепочка не принадлежит языку ( $a_1a_2...a_n \not\in L(G)$ )



$G = (\{a, b\}, \{A, B, S\}, P, S)$

$P: A \rightarrow Ab \mid Bb \mid b$

$B \rightarrow Aa$

$S \rightarrow A\perp$

Разбор неправильной  
цепочки *bab*

применением правил

$A \rightarrow b$        $Aab$

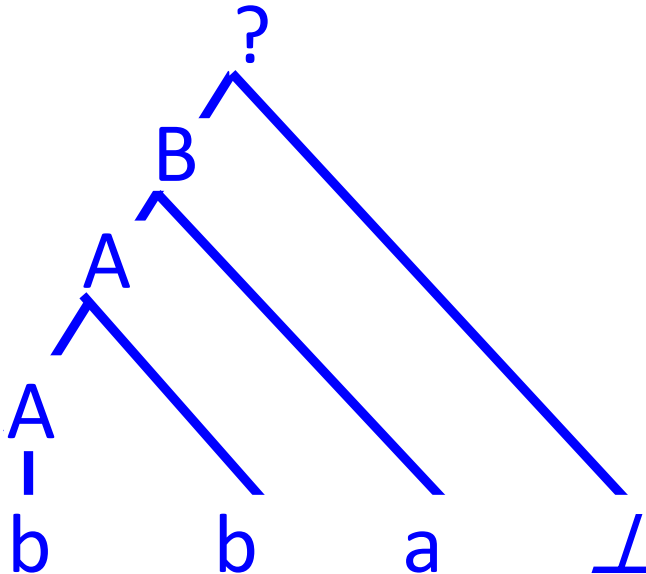
$B \rightarrow Aa$        $Bb$

$A \rightarrow Bb$        $A$

Символ  $A$  не является  
целью грамматики

# Работа алгоритма разбора по регулярной грамматике

с) для нетерминала  $A_{k-1}$  и очередного терминала  $a_k$  не нашлось нетерминала  $A_k$ , для которого есть правило  $A_k \rightarrow A_{k-1}a_k$ : цепочка не принадлежит языку ( $a_1a_2...a_n \notin L(G)$ )


$$G = (\{a, b\}, \{A, B, S\}, P, S)$$
$$P: A \rightarrow Ab \mid Bb \mid b$$
$$B \rightarrow Aa$$
$$S \rightarrow A \perp$$

## Разбор неправильной цепочки `bba_`

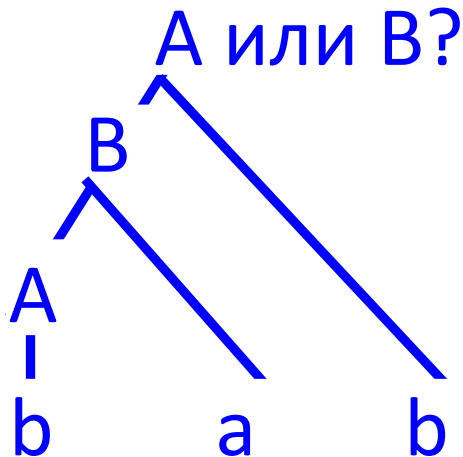
*применением правил*

$$A \rightarrow b \quad A b a \perp$$
$$A \rightarrow Ab \quad Aa \perp$$
$$B \rightarrow Aa \quad B \perp$$

Правила для  **$\perp$**  в грамматике нет

# Работа алгоритма разбора по регулярной грамматике

- d) в грамматике *разные нетерминалы имеют правила вывода с одинаковыми правыми частями*, к какому из них производить свёртку?



$G = (\{a, b\}, \{A, B, S\}, P, S)$

$P: A \rightarrow Ab \mid \mathbf{Bb} \mid b$

$B \rightarrow Aa$

$S \rightarrow A\underline{\phantom{a}}$

$B \rightarrow \mathbf{Bb}$

Попытка разбора  
цепочки  $\mathbf{bab}$  в

грамматике с  
дополнительным

правилом  $\mathbf{B} \rightarrow \mathbf{Bb}$ ,

неоднозначность:

$A \rightarrow Bb$

$B \rightarrow Bb$



# Детерминированный автомат

- Детерминированный конечный автомат (ДКА) – это пятёрка  $(K, T, \delta, H, S)$ , где
- $K$  – конечное множество состояний
- $T$  – алфавит автомата, конечное множество допустимых входных символов
- $\delta$  – функция переходов: отображение  $K \times T \rightarrow K$  (отображение декартова произведения множеств  $K$  и  $T$  на множество  $K$ )
- $H \in K$  – начальное состояние автомата
- $S \subseteq K$  – непустое ( $S \neq \emptyset$ ) конечное множество заключительных состояний

# Детерминированный автомат (ДКА)

- Запись  $\delta(A, t) = B$  означает, что из состояния  $A$  по входному символу  $t$  происходит переход автомата ДКА в состояние  $B$
- Автомат ДКА называют полностью определённым, если в каждом его состоянии существует функция перехода для всех возможных входных символов, то есть  $\forall a \in T, \forall q \in K \exists \delta(a, q) = R, \text{ где } R \in K$
- Функция переходов может быть определена лишь для подмножества  $K \times T$  (*частичная функция*)
- Если значение  $\delta(A, t)$  не определено, автомат не может продолжать работу и останавливается в состоянии “**Ошибка**”

# Детерминированный автомат (ДКА)

- Конечный автомат допускает (принимает) цепочку символов  $a_1a_2...a_n$ , если существуют переходы
$$\delta(H, a_1) = A_1 \quad \delta(A_1, a_2) = A_2 \quad \dots \quad \delta(A_{n-1}, a_n) = S$$
где  $a_i \in T, \quad i = 1, 2, \dots, n$  – алфавит автомата  
 $A_j \in K, \quad j = 1, 2, \dots, n-1$  – состояния автомата  
 $H$  – начальное состояние  
 $S$  – одно из заключительных состояний
- Множество цепочек, допускаемых конечным автоматом, составляет определяемый им язык
- Автоматы эквивалентны, если они задают один язык
- Все конечные автоматы являются распознавателями для регулярных языков

# ДКА и леволинейные грамматики

- **Построение ДКА по леволинейной грамматике**

**Вход:** Леволинейная грамматика  $G = (T, N, P, S)$

**Выход:** Конечный автомат  $ДКА = (K, V, \delta, H, C)$ ,  $L(G) = L(ДКА)$

- Привести грамматику  $G$  к автоматному виду
- Установить, что  $K = N \cup \{H\}$   $V = T$
- Для каждого правила  $A \rightarrow t \in P$ , где  $t \in T$  и  $A \in N$  в функцию перехода включается правило  $\delta(H, t) = A$
- Для каждого правила  $A \rightarrow Bt \in P$ , где  $t \in T$  и  $A, B \in N$  в функцию перехода включается правило  $\delta(B, t) = A$
- В множество конечных состояний автомата включается элемент, соответствующий цели грамматики  $G$ :  $C = \{S\}$

# ДКА и левولينейные грамматики

- **Построение левولينейной грамматики по ДКА**

**Вход:** Конечный автомат  $ДКА = (K, V, \delta, H, C)$

**Выход:** Левولينейная грамматика

$$G = (T, N, P, S), L(G) = L(ДКА)$$

- $N = K \setminus \{H\} \quad T = V$

- Для всех возможных состояний из множества  $K$  и всех возможных входных символов из множества  $V$ :

- Если  $\delta(A, t) = \emptyset$ , где  $A \in K, t \in V$ , никаких действий не выполняется
- Если  $\delta(A, t) = B$ , где  $A \in K, B \in K, t \in V$ :
  - если  $A = H$ , в  $P$  включается правило  $B \rightarrow t$
  - если  $A \neq H$ , в  $P$  включается правило  $B \rightarrow At$

# ДКА и леволinéйные грамматики

- **Построение леволinéйной грамматики по ДКА**
- Если множество конечных состояний  $C$  автомата ДКА содержит только одно состояние  $C = \{C_0\}$ , целевым символом  $S$  грамматики  $G$  становится символ множества  $N$ , соответствующий этому состоянию:  $S = C_0$
- Если множество конечных состояний  $C$  автомата ДКА содержит более одного состояния  $C = \{C_1, C_2, \dots, C_n\}$ ,  $n > 1$ , в множество нетерминальных символов  $N$  грамматики  $G$  добавляется новый нетерминальный символ  $S$ :  $N = N \cup \{S\}$
- В множество правил  $P$  грамматики  $G$  добавляются правила  $S \rightarrow C_1 \mid C_2 \mid \dots \mid C_n$
- $G = (T, N, P, S)$

# Диаграмма состояний

- Диаграмма состояний конечного автомата – это неупорядоченный ориентированный помеченный граф, который строится следующим образом:
  - вершины графа (называемые состояниями) помечаются нетерминалами грамматики
  - добавляется ещё одна вершина (называемая начальным состоянием), помечаемая символом, отличным от нетерминальных (например,  $H$ )
  - для каждого правила вида  $W \rightarrow t$  соединяются дугой состояния от  $H$  к  $W$ , дуга помечается символом  $t$
  - для каждого правила вида  $W \rightarrow Vt$  соединяются дугой состояния от  $V$  к  $W$ , дуга помечается символом  $t$

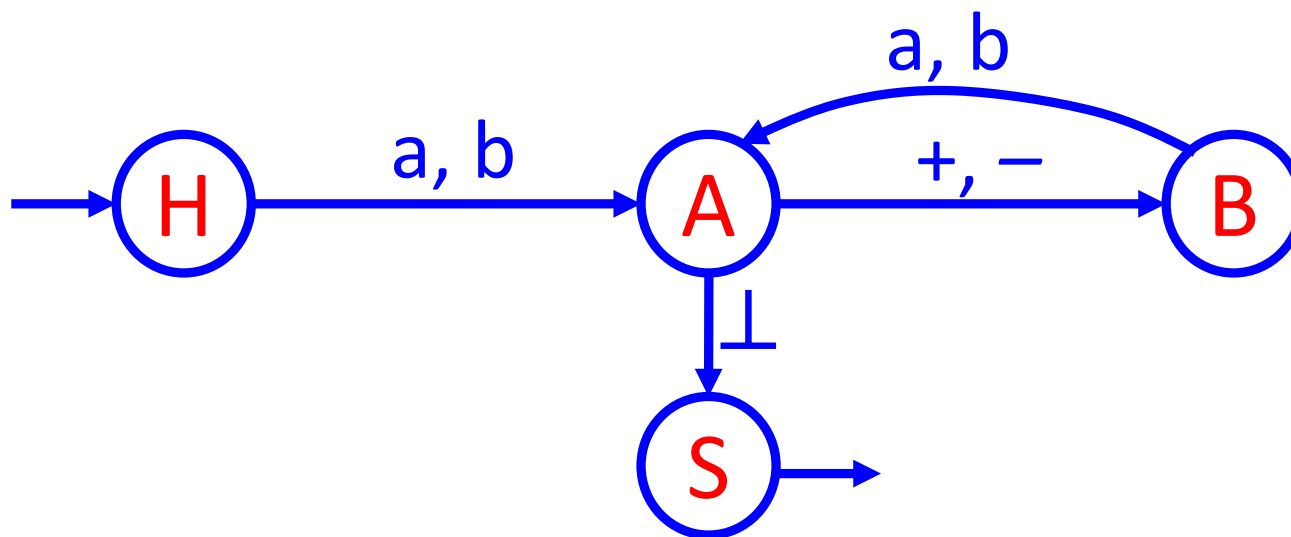
# ДС и леволинейные грамматики

- **Построение леволинейной грамматики по ДС**
- Каждой дуге из начального состояния  $H$  в состояние  $W$ , помеченной символом  $t$ , соответствует правило  $W \rightarrow t$
- Каждой дуге из состояния  $V$  в состояние  $W$ , помеченной символом  $t$ , соответствует правило  $W \rightarrow Vt$
- Заключительное состояние  $S$  объявляется начальным символом грамматики



# Пример построения грамматики

- Задан ДКА  $M = (\{H, A, B, S\}, \{a, b, +, -, \perp\}, \delta, H, S)$ , где
$$\begin{array}{llll} \delta(H, a) = A & \delta(A, +) = B & \delta(A, \perp) = S & \delta(B, a) = A \\ \delta(H, b) = A & \delta(A, -) = B & \delta(B, b) = A & \end{array}$$
- Если переход из некоторого состояния по некоторому символу невозможен, то есть  $\delta(C, c) = \emptyset$ , правило для этого перехода просто не выписывается



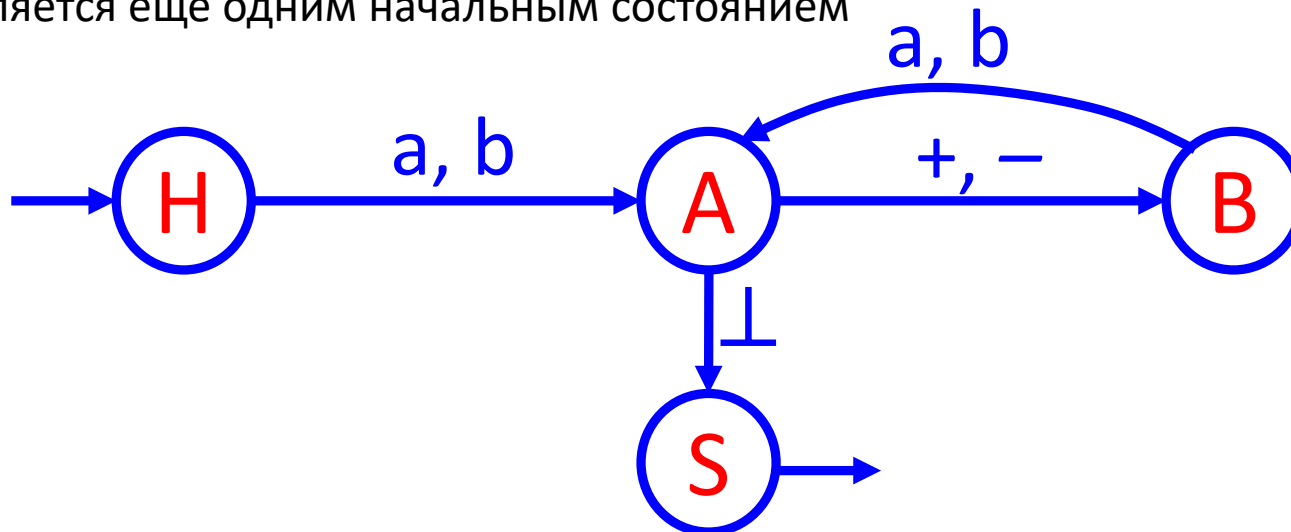
# Пример построения грамматики

- Стрелки в правила леволинейных грамматик вставляются с противоходом по отношению к стрелкам на диаграмме
- Правила начинают выписывать с заключительного состояния автомата:  
 $S \rightarrow A\perp$
- Если для некоторого нетерминального символа имеется правило с пустой правой частью (например,  $B \rightarrow \varepsilon$ ), соответствующее ему состояние объявляется ещё одним начальным состоянием

$$S \rightarrow A\perp$$

$$A \rightarrow Ba \mid Bb \mid a \mid b$$

$$B \rightarrow A+ \mid A-$$



# Пример диаграммы состояний

- Диаграмма состояний для леволинейной грамматики

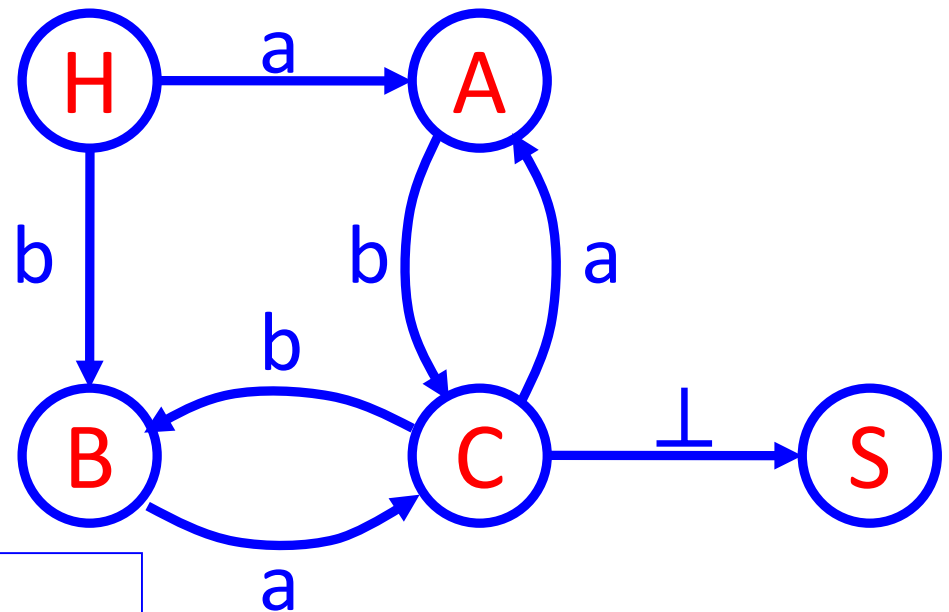
$$G_{left} = (\{a, b, \perp\}, \{S, A, B, C\}, P, S)$$

$$P: S \rightarrow C\perp$$

$$C \rightarrow Ab \mid Ba$$

$$A \rightarrow a \mid Ca$$

$$B \rightarrow b \mid Cb$$



Функция

переходов:

$$\delta(H, a) = A$$

$$\delta(H, b) = B$$

$$\delta(A, b) = C$$

$$\delta(B, a) = C$$

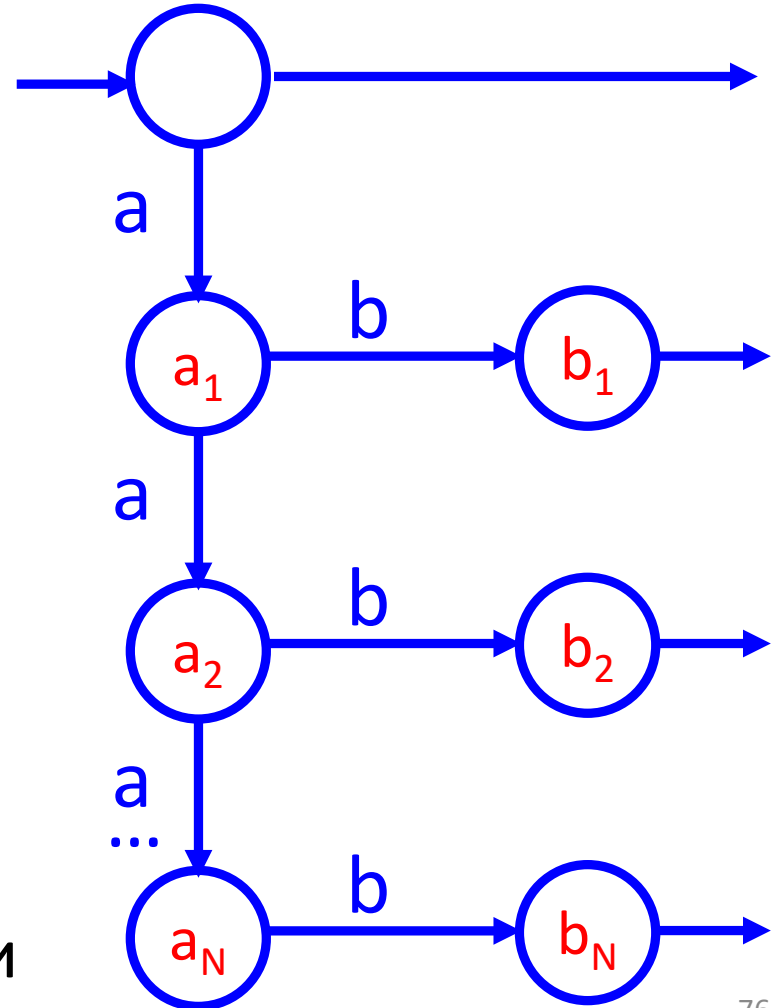
$$\delta(C, a) = A$$

$$\delta(C, b) = B$$

$$\delta(C, \perp) = S$$

# Пример диаграммы состояний

- Диаграмма состояний автомата, допускающего язык  $L = \{ a^n b^n \mid 0 \leq n \leq N \}$
- При любом конечном значении  $N$  такой язык надо рассматривать как язык регулярный
- Моделью памяти в таком автомате служат состояния этого автомата
- На диаграмме показаны начальное состояние, несколько промежуточных и несколько заключительных состояний



# ДС и левولينейные грамматики

- **Построение левولينейной грамматики по ДС**
- Каждой дуге из начального состояния  $H$  в состояние  $W$ , помеченной символом  $t$ , соответствует правило  $W \rightarrow t$
- Каждой дуге из состояния  $V$  в состояние  $W$ , помеченной символом  $t$ , соответствует правило  $W \rightarrow Vt$
- Заключительное состояние  $S$  объявляется начальным символом грамматики

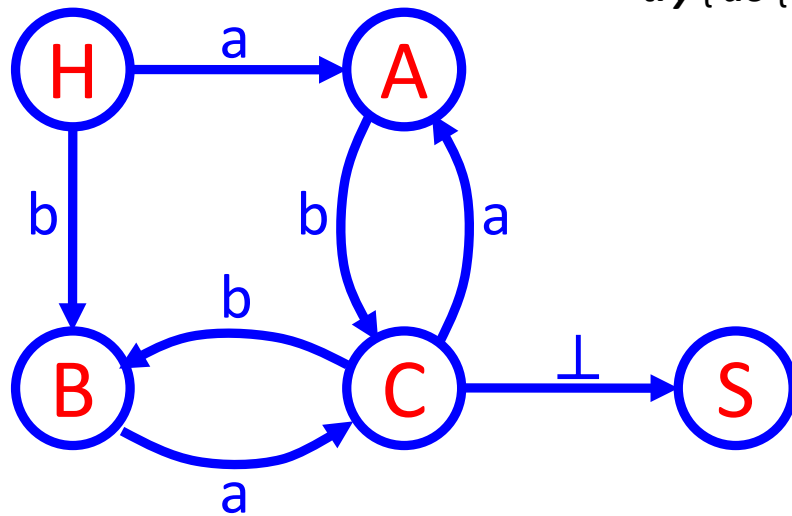
# Алгоритм разбора цепочки по диаграмме состояний

- Текущим объявляется состояние  $N$
- Многократно (до тех пор, пока не прочитан признак конца цепочки) выполняются следующие шаги: читается очередной символ исходной цепочки и осуществляется переход из текущего состояния в новое состояние по дуге, помеченной прочитанным символом; новое состояние становится текущим

# Алгоритм разбора цепочки по диаграмме состояний

- Прочитана вся цепочка; на каждом шаге находилась единственная дуга, помеченная очередным символом анализируемой цепочки; в результате последнего перехода достигнуто состояние  $S$ , *цепочка принадлежит языку  $L(G)$*
- Прочитана вся цепочка; на каждом шаге находилась единственная дуга; в результате последнего шага достигнуто состояние, отличное от  $S$ , *цепочка не принадлежит языку  $L(G)$*
- На некотором шаге не нашлось дуги, выходящей из текущего состояния и помеченной очередным анализируемым символом, *цепочка не принадлежит языку  $L(G)$*
- На некотором шаге оказалось, что из текущего состояния есть несколько дуг, помеченных очередным символом, ведущих в разные состояния, *разбор недетерминирован*

# Программа анализатора



Грамматика:

$S \rightarrow C\perp$

$C \rightarrow Ab \mid Ba$

$A \rightarrow a \mid Ca$

$B \rightarrow b \mid Cb$

```

Analyze () { FA_State= H;
    try { do { switch (FA_State)
        { case H:
            if (c == 'a') { GetS (); FA_State = A; }
            else if (c == 'b') { GetS (); FA_State = B; }
            else
                break;
        case A:
            if (c == 'b') { GetS (); FA_State = C; }
            else
                break;
        case B:
            if (c == 'a') { GetS (); FA_State = C; }
            else
                break;
        case C:
            if (c == 'a') { GetS (); FA_State = A; }
            else if (c == 'b') { GetS (); FA_State = B; }
            else if (c == '⊥')
                FA_State = S;
            else
                break;
        case Error: throw c;
        }
    } while (FA_State != S);
}
catch (char c) { cout << "Плохой символ" << c << endl; }
}
    
```



# Разбор цепочки анализатором

- При анализе цепочки  $abba\underline{L}$  возникает такая последовательность переходов:

$$H \xrightarrow{a} A \xrightarrow{b} C \xrightarrow{b} B \xrightarrow{a} C \xrightarrow{\underline{L}} S$$

- Каждая смена состояния означает “свёртку” sentential form путём замены в ней пары “нетерминал-терминал”  $Nt$  на нетерминал  $L$ , где  $L \rightarrow Nt$  есть правило вывода в грамматике
- Возникает такая последовательность свёрток, соответствующая сменам состояний:

$$\underline{abba}\underline{L} \leftarrow \underline{A}bba\underline{L} \leftarrow \underline{C}ba\underline{L} \leftarrow \underline{B}a\underline{L} \leftarrow \underline{C}\underline{L} \leftarrow S$$

Грамматика:

$$S \rightarrow C\underline{L}$$

$$C \rightarrow Ab \mid Ba$$

$$A \rightarrow a \mid Ca$$

$$B \rightarrow b \mid Cb$$

# Недетерминированность разбора

- Дана грамматика  $G = (\{a, b, \perp\}, \{S, A, B\}, P, S)$   
где  $P$ :  
 $S \rightarrow A\perp$   
 $A \rightarrow a \mid Bb$   
 $B \rightarrow b \mid Bb$
- Для этой грамматики разбор будет недетерминированным, так как у нетерминальных символов  $A$  и  $B$  есть одинаковые правые части –  $Bb$
- Такой грамматике будет соответствовать недетерминированный конечный автомат

# Недетерминированный автомат

- Недетерминированный конечный автомат (НКА) – это пятёрка  $(K, T, \delta, H, S)$ , где
- $K$  – конечное множество состояний
- $T$  – конечное множество допустимых входных символов (алфавит автомата)
- $\delta$  – функция переходов: отображение  $K \times T \rightarrow \rho(K)$  (отображение декартова произведения множеств  $K$  и  $T$  в множество подмножеств  $K$ )
- $H \subset K$  – конечное множество начальных состояний
- $S \subset K$  – конечное множество заключительных состояний

# Недетерминированный автомат

- Запись  $\delta(A, t) = \{B_1, B_2, \dots, B_n\}$  означает, что из состояния  $A$  по входному символу  $t$  происходит переход автомата ДКА в любое из состояний  $B_i$  ( $i = 1, 2, \dots, n$ )

Класс языков, определяемых НКА, совпадает с классом языков, определяемых ДКА

- Для языка  $L$  эквивалентны утверждения:
  1.  $L$  порождается регулярной грамматикой
  2.  $L$  допускается ДКА
  3.  $L$  допускается НКА

# КА и левولينейные грамматики

- **Построение КА по левولينейной грамматике**

**Вход:** Левولينейная грамматика  $G = (T, N, P, S)$

**Выход:** Конечный автомат  $KA = (K, V, \delta, H, C), L(G) = L(KA)$

- Установить, что  $K = N$   $V = T$
- Для каждого правила  $A \rightarrow t \in P$ , где  $t \in T$  и  $A \in N$  в функцию перехода включается правило  $\delta(H, t) = A$   
В таком случае установить, что  $K = N \cup \{H\}$
- Для каждого правила  $A \rightarrow Bt \in P$ , где  $t \in T$  и  $A, B \in N$  в функцию перехода включается правило  $\delta(B, t) = A$
- В множество начальных состояний включаются элемент  $H$ , а также все элементы  $A$ , для которых есть правила  $A \rightarrow \varepsilon$
- В множество конечных состояний автомата включается элемент, соответствующий цели грамматики  $G$ :  $C = \{S\}$

# КА и левولينейные грамматики

- **Построение левولينейной грамматики по КА**

**Вход:** Конечный автомат  $KA = (K, V, \delta, H, C)$ ,  $L(G) = L(KA)$

**Выход:** Левولينейная грамматика  $G = (T, N, P, S)$

- Установить, что  $N = K$   $T = V$
- Для каждого перехода  $\delta(A, t) = B$  в грамматику включается правило  $B \rightarrow At \in P$ , где  $t \in T$  и  $A, B \in N$
- Для каждого начального состояния автомата в грамматику включается правило  $B \rightarrow \varepsilon \in P$ , где  $B \in N$
- Начальным символом грамматики, будет нетерминал, соответствующий заключительному состоянию  $G: S = \{H\}$
- К грамматике применяется алгоритм устранения  $\varepsilon$ -правил

# НКА по левوليной грамматике

- Диаграмма состояний для левوليной грамматики

$$G_{left} = (\{a, b\}, \{S, A, B, C\}, P, S)$$

$$P: S \rightarrow Sa \quad (1)$$

$$S \rightarrow Aa \quad (2)$$

$$A \rightarrow Ab \quad (3)$$

$$A \rightarrow a \quad (4)$$

$$A \rightarrow \varepsilon \quad (5)$$

Свёртки в грамматике для  
цепочки **aba**

$$aba \leftarrow_4 Aba \leftarrow_3 Aa \leftarrow_2 S$$

Путь в автомате для  
цепочки **aba**

$$\rightarrow H^a \rightarrow_4 A^b \rightarrow_3 A^a \rightarrow_2 S \rightarrow$$

Функция

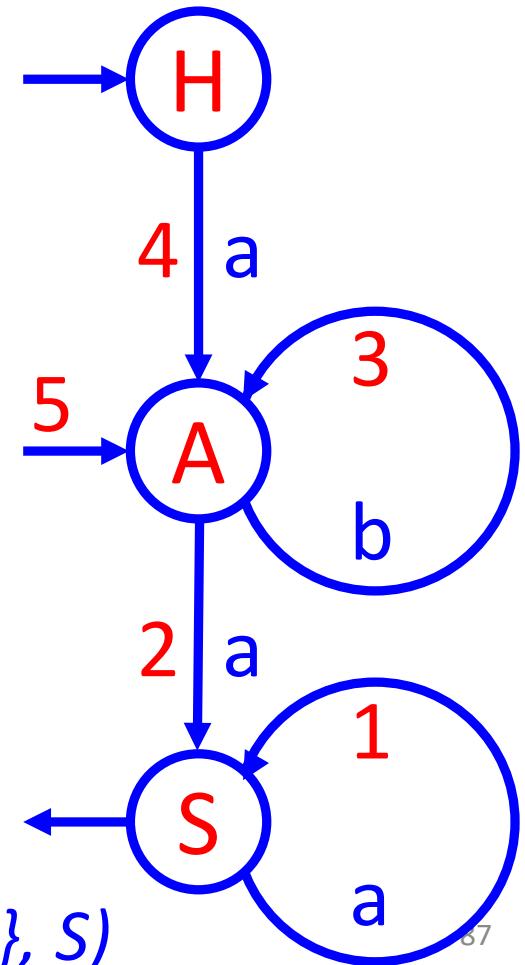
переходов:

$$\delta(A, a) = S$$

$$\delta(A, b) = A$$

$$\delta(S, a) = S$$

$$\delta(H, a) = A$$



$$\text{Автомат НКА} = (\{H, A, S\}, \{a, b\}, \delta, \{H, A\}, S)$$

# Преобразование НКА в ДКА

- Алгоритм преобразования НКА в ДКА

Вход: Конечный автомат  $НКА = (K, V, \delta, H, S)$

Выход: Конечный автомат  $ДКА = (K', V', \delta', H', S')$   
 $L(ДКА) = L(НКА)$

1. Множество состояний  $K'$  нового автомата  $M'$  строится из комбинаций всех состояний  $K$  старого автомата  $M$  (как множество подмножеств множества  $K$ )

Каждое состояние из  $K'$  обозначается как  $[A_1 A_2 \dots A_n]$ ,  
где  $A_i \in K, 1 \leq i \leq n$

Всего имеется не более  $2^n - 1$  состояний нового автомата  $M'$



# Преобразование НКА в ДКА

- **Алгоритм преобразования НКА в ДКА**

**Вход:** Конечный автомат  $НКА = (K, V, \delta, H, S)$

**Выход:** Конечный автомат  $ДКА = (K', V', \delta', H', S')$   
 $L(ДКА) = L(НКА)$

2. Пусть  $H$  есть множество начальных состояний старого автомата:  $\{H_1, H_2, \dots, H_p\}$ , в качестве исходного состояния ДКА  $M'$  при построении отображения  $\delta'$  (функции переходов) берётся состояние  $[A_1 A_2 \dots A_p]$ , где  $A_i \in H, 1 \leq i \leq p$  или  $[H_1 H_2 \dots H_p]$ , то есть объединение всех начальных состояний НКА  $M$   
Это состояние включается в множество состояний  $K'$  нового автомата (сначала оно единственное)

# Преобразование НКА в ДКА

3. Начиная с исходного состояния  $[H_1H_2...H_p]$ , для каждого нового состояния  $[A_1A_2...A_n]$  из  $K'$  и каждого входного символа  $t \in T$  строятся переходы

$\delta'([A_1A_2...A_n], t) = [B_1B_2...B_m]$ , где для  $\forall k: 1 \leq k \leq m$   
 $\exists i: 1 \leq i \leq n$  такое, что  $\delta(A_i, t) = B_k$

$\{B_1, B_2, ..., B_m\}$  – это все состояния НКА, в которые есть переходы из состояний  $\{A_1, A_2, ..., A_n\}$  по символу  $t$

В ДКА  $M'$  формируется детерминированный переход по символу  $t$  из состояния  $[A_1A_2...A_n]$  в состояние  $[B_1B_2...B_m]$  (если  $m = 0$ ,  $\delta'([A_1A_2...A_n], t) = \emptyset$ )

Все новые состояния  $[B_1B_2...B_m]$  ( $m \neq 0$ ) включаются в  $K'$

# Преобразование НКА в ДКА

4. Пусть конечное состояние старого автомата  $S$  есть множество состояний  $\{S_1, S_2, \dots, S_q\}$ , тогда  $S'$  – все состояния из  $K'$  вида  $[... S_r ...]$ , где  $S_r \in S$  для  $1 \leq r \leq q$
- Если  $S'$  состоит более, чем из одного элемента, изменяют входной язык, добавляя маркер ' $\perp$ ' в конец каждой цепочки

Вводится новое состояние  $S$ , и для каждого состояния  $Q$  из множества  $S'$  добавляется переход  $\delta'(Q, \perp) = S$

$S$  объявляется единственным заключительным состоянием

# ДС и праволинейные грамматики

- Если все правила праволинейной грамматики с односимвольной правой частью имеют вид  $V \rightarrow \perp$ :
  - Состояниями ДС будут нетерминалы грамматики и одно специальное заключительное состояние  $S$ , в которое для каждого правила вида  $V \rightarrow \perp$  проводится дуга из  $V$ , помеченная признаком конца  $\perp$
  - Для каждого правила вида  $V \rightarrow tW$  проводится дуга из  $V$  в  $W$ , помеченная символом  $t$
  - Начальным состоянием будет начальный символ  $H$

# Разбор цепочки

- При анализе цепочки  $abba\perp$  имеем ту же последовательность переходов:

$$H \xrightarrow{a} A \xrightarrow{b} C \xrightarrow{b} B \xrightarrow{a} C \xrightarrow{\perp} S$$

- Каждая смена состояния теперь означает “свёртку” сентенциальной формы путём замены в ней нетерминала  $L$  на пару “нетерминал-терминал”  $tN$ , где  $L \rightarrow tN$  – правило грамматики

- Возникает такая последовательность замен нетерминальных символов, соответствующая сменам состояний при построении дерева сверху вниз:

$$H \rightarrow aA \rightarrow abC \rightarrow abbB \rightarrow abbaC \rightarrow abba\perp$$

Грамматика:

$$H \rightarrow aA \mid bB$$

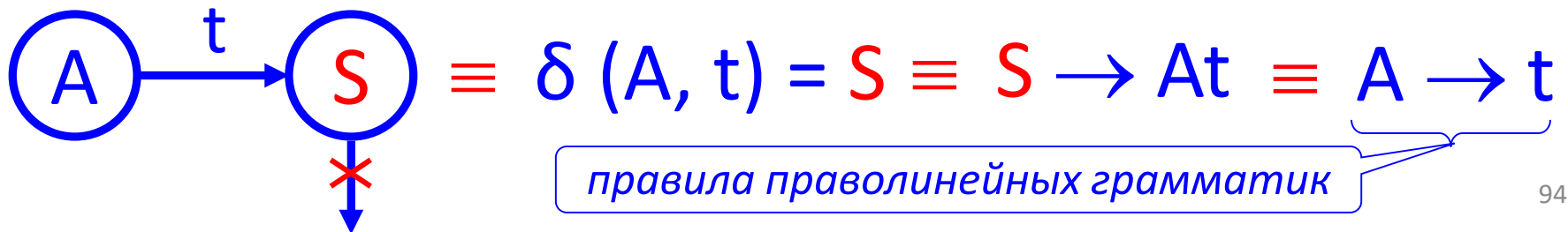
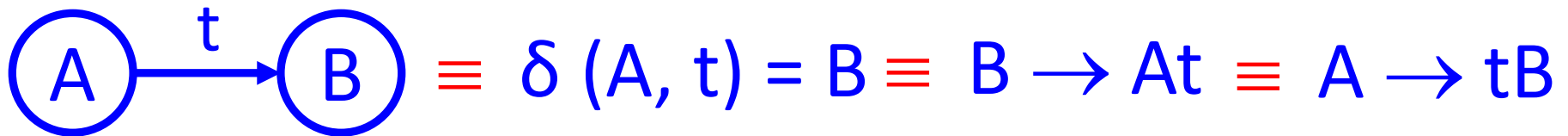
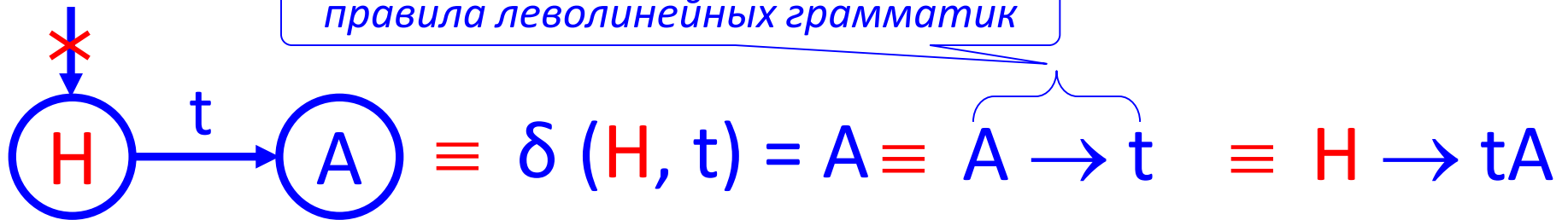
$$A \rightarrow bC$$

$$B \rightarrow aC$$

$$C \rightarrow aA \mid bB \mid \perp$$

# Диаграммы состояний, функции переходов, правила регулярных грамматик

*правила левolineйных грамматик*



*правила праволинейных грамматик*

# КА и праволинейные грамматики

- **Построение КА по праволинейной грамматике**

**Вход:** Праволинейная грамматика  $G = (T, N, P, S)$

**Выход:** Конечный автомат  $KA = (K, V, \delta, H, C)$ ,  $L(G) = L(KA)$

- Установить, что  $K = N$   $V = T$   $H = S$

В множество  $K$  может также войти состояние  $F$ , которое в таком случае объявляется заключительным

- Для каждого правила  $A \rightarrow t \in P$ , где  $t \in T$  и  $A \in N$  в функцию перехода включается правило  $\delta(A, t) = F$

В таком случае установить, что  $K = N \cup \{F\}$

- Для каждого правила  $A \rightarrow tB \in P$ , где  $t \in T$  и  $A, B \in N$  в функцию перехода включается правило  $\delta(A, t) = B$

- В множество конечных состояний включаются элемент  $F$ , а также все элементы  $A$ , для которых есть правила  $A \rightarrow \varepsilon$

# КА и праволинейные грамматики

- **Построение праволинейной грамматики по КА**

**Вход:** Конечный автомат  $KA = (K, V, \delta, H, C)$ ,  $L(G) = L(KA)$

**Выход:** Праволинейная грамматика  $G = (T, N, P, S)$

- Установить, что  $N = K$   $T = V$   $S = H$
- Для каждого перехода  $\delta(A, t) = B$  в грамматику включается правило  $B \rightarrow At \in P$ , где  $t \in T$  и  $A, B \in N$
- Для каждого заключительного состояния автомата в грамматику включается правило вида  $B \rightarrow \varepsilon \in P$ , где  $B \in C$
- Начальным символом грамматики, будет нетерминал, соответствующий начальному состоянию  $G$ :  $S = \{H\}$
- К грамматике применяется алгоритм устранения  $\varepsilon$ -правил



# КА по праволинейной грамматике

- Диаграмма состояний для праволинейной грамматики  $G_{left} = (\{a, b\}, \{S, A, B, C\}, P, S)$

$P: S \rightarrow aS$  (1)

$S \rightarrow bA$  (2)

$S \rightarrow d$  (3)

$A \rightarrow bA$  (4)

$A \rightarrow \varepsilon$  (5)

Вывод в грамматике для цепочки **abb**

$\rightarrow S \xrightarrow{1} aS \xrightarrow{2} abA \xrightarrow{4} abbA \xrightarrow{5} abb$

Путь в автомате для цепочки **abb**

$\rightarrow S \xrightarrow{1} S \xrightarrow{2} A \xrightarrow{4} A \xrightarrow{5}$

Функция

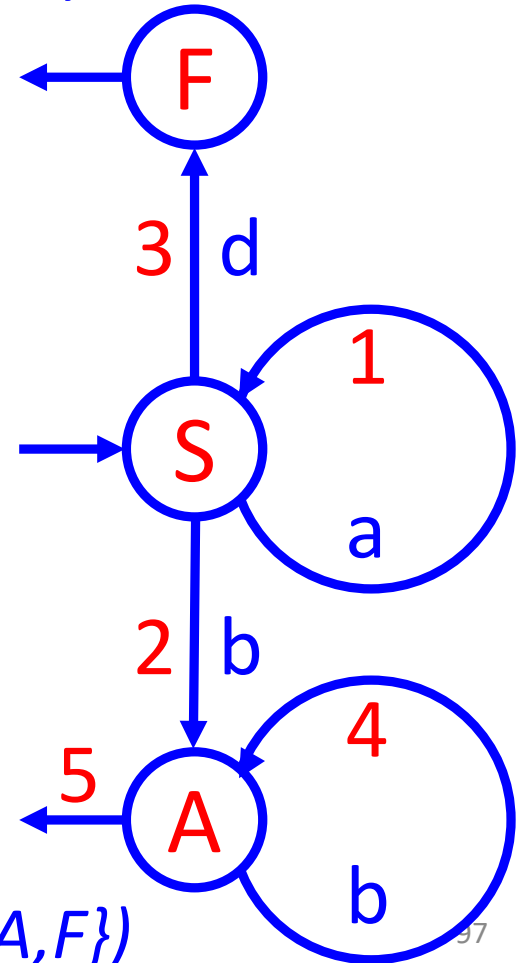
переходов:

$\delta(A, b) = A$

$\delta(S, a) = S$

$\delta(S, b) = A$

$\delta(S, d) = F$



Автомат НКА =  $(\{A, F, S\}, \{a, b, d\}, \delta, S, \{A, F\})$

# Выявление недетерминированности разбора

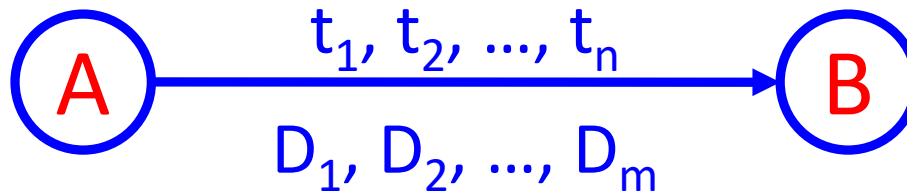
- Левостроенная грамматика: в разных правилах имеются одинаковые правые части
- Правостроенная грамматика: в правилах для одного символа имеются альтернативы, начинающиеся с одинаковых терминальных символов
- Диаграмма состояний: из одной вершины выходят несколько дуг с одинаковыми надписями, либо несколько вершин являются входными
- Функция переходов: разные значения для одного и того же набора параметров (переход из некоторого состояния в разные состояния по одному символу), либо несколько состояний являются начальными

# Диаграмма с действиями

- В конечном автомате лексического анализатора действия отображаются с помощью диаграммы состояний: с каждым переходом из одного состояния в другое на диаграмме состояний связывается выполнение функции действия  $D(k, t)$ , где  $k$  – текущее состояние, а  $t$  – текущий входной символ автомата
- Функция  $D(k, t)$  может выполнять действия:
  - размещение новой лексемы в таблице лексем
  - проверка наличия лексемы-имени в таблице имён
  - внесение новой записи в таблицу имён
  - выдача сообщений об ошибках, обнаруженных в процессе лексического анализа
  - остановка процесса компиляции

# Диаграмма с действиями

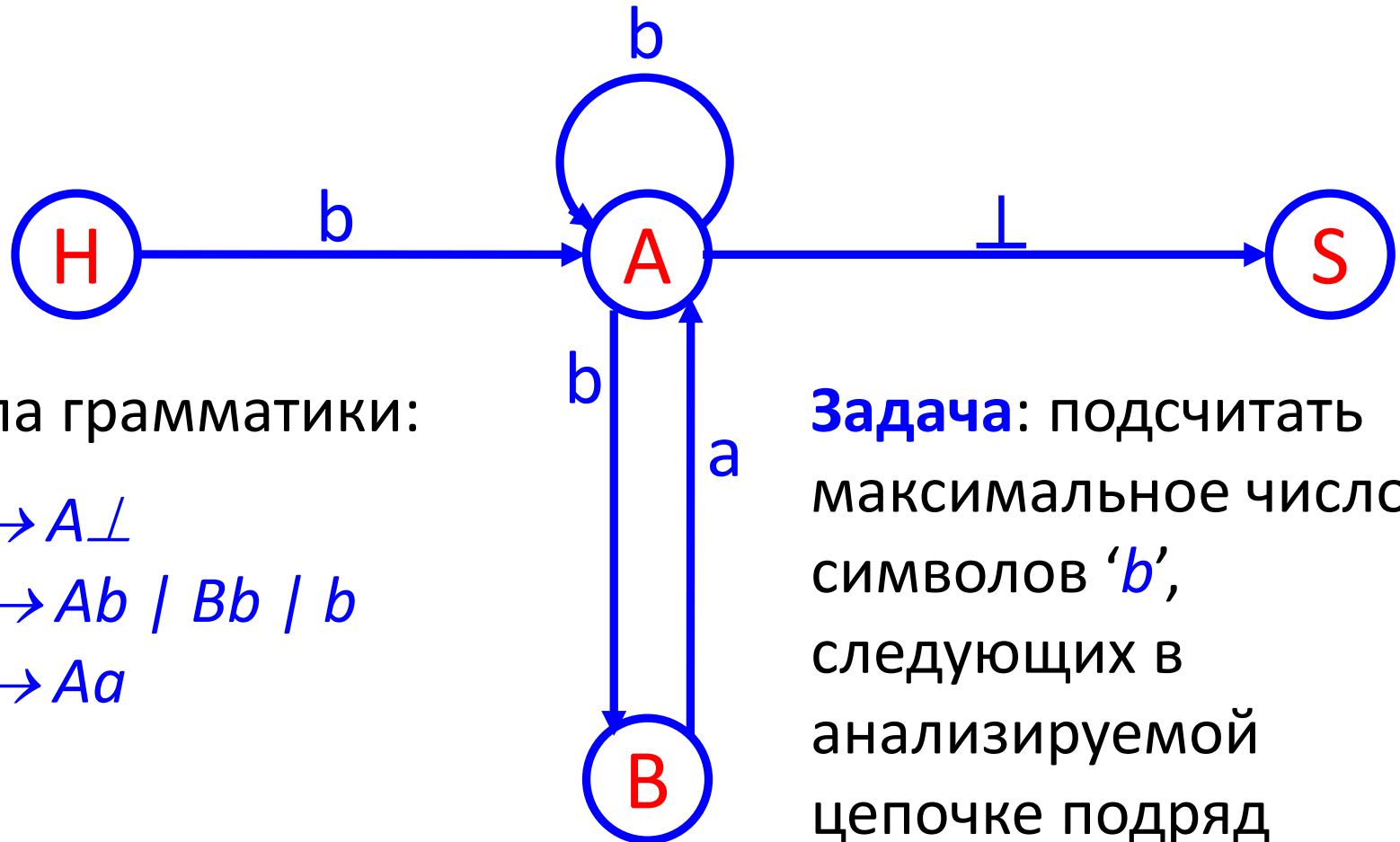
- Функция действия  $D(k, t)$  записывается на диаграмме состояний с помощью дополнительных пометок под дугами, соединяющими состояния автомата
- Каждая дуга может выглядеть так:



- $t_i$  – символы анализируемого языка: если в состоянии  $A$  очередной анализируемый символ языка совпадает с  $t_i$  для какого-либо  $i = 1, 2, \dots, n$ , осуществляется переход в состояние  $B$ , при этом выполняются действия  $D_1, D_2, \dots, D_m$

# Диаграмма с действиями

**Дано:** грамматика  $G = (\{a, b, \perp\}, \{S, A, B\}, P, S)$



Правила грамматики:

$P: S \rightarrow A\perp$

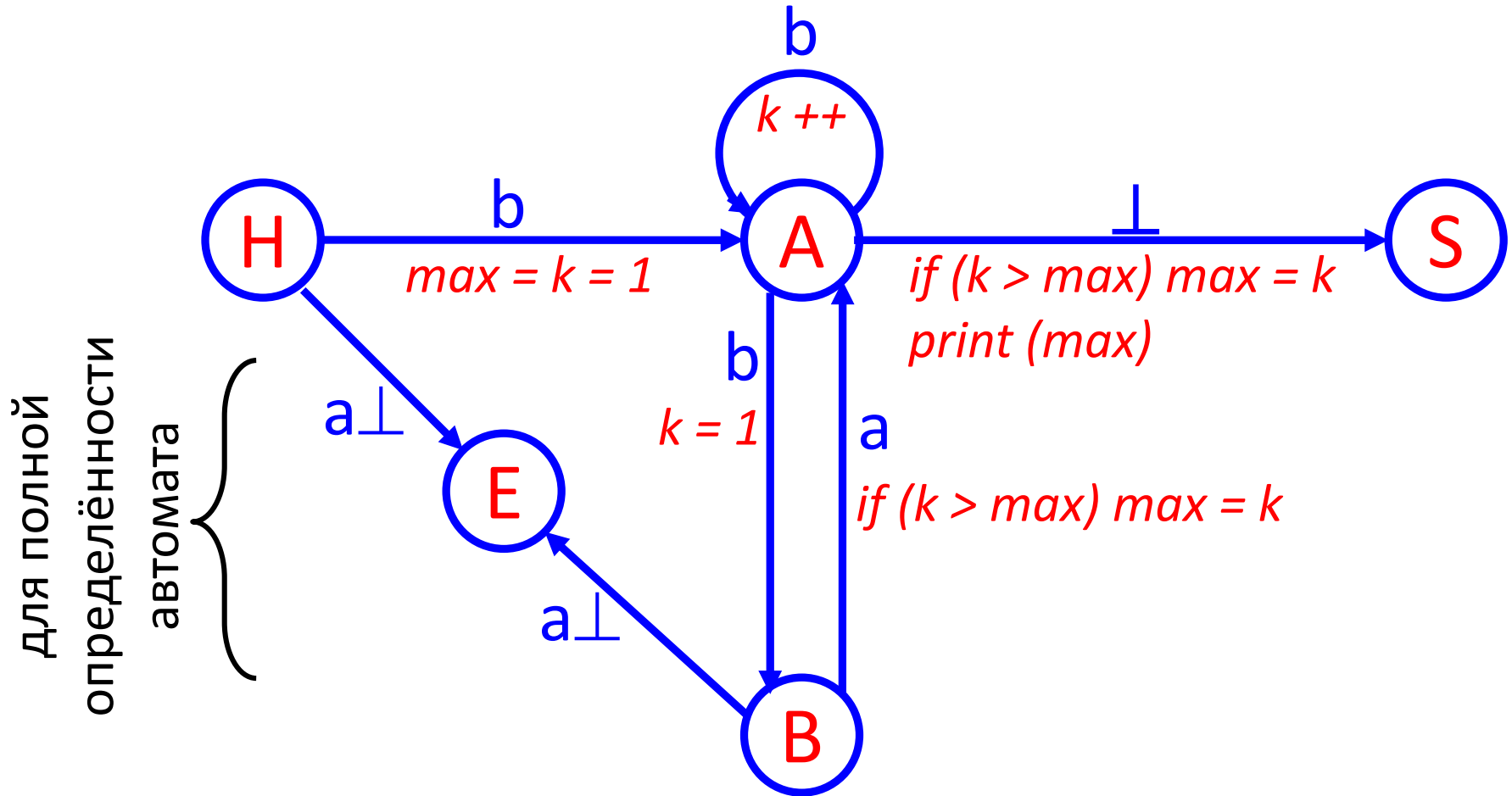
$A \rightarrow Ab \mid Bb \mid b$

$B \rightarrow Aa$

**Задача:** подсчитать максимальное число символов ' $b$ ', следующих в анализируемой цепочке подряд

# Диаграмма с действиями

**Решение:** диаграмма состояний с действиями



# Метод рекурсивного спуска

- Для каждого нетерминального символа  $A \in N$  грамматики  $G(N, T, P, S)$  строится процедура, которая получает на вход цепочку символов  $\alpha$  и текущее состояние указателя ввода символов из этой цепочки
- Если для символа  $A$  больше одного правила, то ищется правило вида  $A \rightarrow a\gamma$ , где  $\gamma \in (T \cup N)^*$  и  $a \in T$  совпадает с текущим символом входной цепочки
- Если такое правило ( $A \rightarrow a\gamma$ ) найдено (либо правило  $A \rightarrow \gamma$  – единственное для  $A$ ), то для каждого нетерминального символа из цепочки  $\gamma$  рекурсивно вызывается процедура разбора этого символа

# Метод рекурсивного спуска

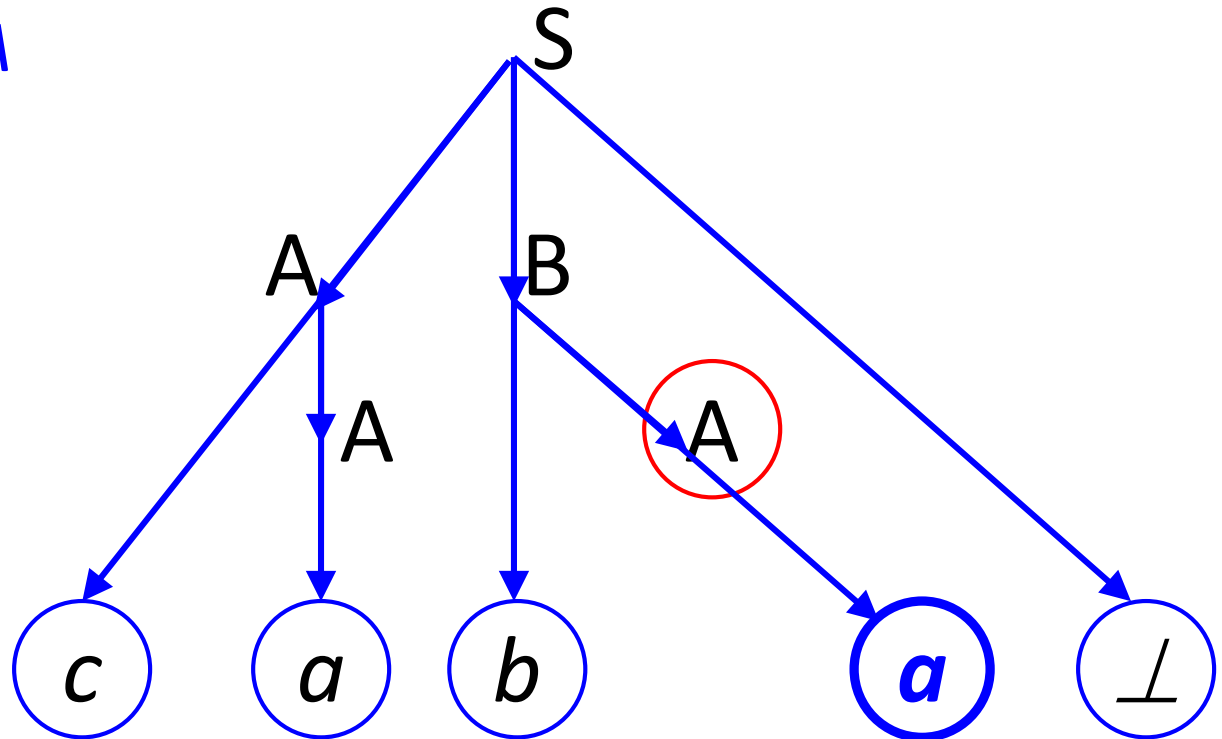
- Грамматика  $G = (\{a, b, c, \perp\}, \{S, A, B\}, P, S)$  и цепочка  $caba\perp$

$P: S \rightarrow AB\perp$

$A \rightarrow a \mid cA$

$B \rightarrow bA$

$A \rightarrow a$



- $S \rightarrow AB\perp \rightarrow cAB\perp \rightarrow caB\perp \rightarrow cabA\perp \rightarrow caba\perp$



# Метод рекурсивного спуска

- Процедура каждого нетерминального символа грамматики, *начиная с указанного места исходной цепочки, ищет подцепочку, которая начинается с текущего символа и выводится из этого нетерминального символа*
- Пример распознавания рекурсивным спуском:
  - Процедура *GetL ()* вводит очередной символ языка
  - Процедура *S ()* начинает работу, когда первый символ уже прочитан

$S \rightarrow AB\perp$

$A \rightarrow a \mid cA$

$B \rightarrow bA$

```
void S () { A (); B (); if (c != '⊥') ERROR (); }
void A () { if (c == 'a') GetL ();
            else if (c == 'c') { GetL (); A (); }
            else ERROR (); }
void B () { if (c == 'b') { GetL (); A (); }
            else ERROR (); }
```

# Метод рекурсивного спуска

- Метод рекурсивного спуска работоспособен, если на каждом шаге вывода выбор правила для замены левого нетерминала безошибочно принимается по первому символу из неп прочитанной входной цепочки

## Достаточные условия применимости метода рекурсивного спуска

Метод применим, если каждое правило грамматики имеет вид:

- либо для символа  $A$  имеется единственное правило вывода  $A \rightarrow \alpha$ , где  $\alpha \in (T \cup N)^*$
- либо (если для символа  $A$  правил вывода несколько) все правила начинаются с различных терминальных символов:

$$A \rightarrow a_1\alpha_1 \mid a_2\alpha_2 \mid \dots \mid a_n\alpha_n \quad \begin{array}{l} a_i \in T \text{ для всех } i = 1, 2, \dots, n \\ \alpha_i \in (T \cup N)^* \\ a_i \neq a_j \text{ для } i \neq j \end{array}$$

# Метод рекурсивного спуска

- Рекурсивный спуск применим для грамматики

$G = (\{a, b, c, \perp\}, \{S, A, B\}, P, S)$ , где

$P: \quad S \rightarrow AB\perp \quad A \rightarrow a \mid cA \quad B \rightarrow bA$

- Неоднозначная грамматика (метод не применим):

$P_n: \quad S \rightarrow aA \mid B \mid c \quad A \rightarrow aA \mid c \quad B \rightarrow aA \mid a$

- Однозначная грамматика с неоднозначными прогнозами:

$P_o: \quad S \rightarrow A \mid B \quad A \rightarrow aA \mid c \quad B \rightarrow aB \mid b$

- Наличие в грамматике правил вида  $X \rightarrow \alpha$  и  $X \rightarrow \beta$ , из правых частей которых выводятся цепочки, начинающиеся одним и тем же терминалом  $a$ , то есть  $\alpha \Rightarrow a\alpha'$  и  $\beta \Rightarrow a\beta'$ , делает неоднозначным прогноз по символу  $a$ , в таких случаях метод рекурсивного спуска неприменим

# Применение рекурсивного спуска

- Множество  $first(A)$  – это множество терминальных символов, которыми начинаются цепочки, выводимые из  $A$  в грамматике  $G = (T, N, P, S)$ :
  - $first(A) = \{a \in T \mid A \Rightarrow a\alpha, A \in (T \cup N)^+, \alpha \in (T \cup N)^*\}$
  - $first(\varepsilon) = \emptyset$
  - Для альтернатив правила  $S \rightarrow A \mid B$  в грамматике  $G_o$ :  
 $first(A) = \{a, c\}, first(B) = \{a, b\}$   
пересечение  $first(A) \cap first(B) = \{a\} \neq \emptyset$   
метод рекурсивного спуска к  $G_o$  неприменим
- Наличие в грамматике двух разных правил  $X \rightarrow \alpha \mid \beta$ , таких что  $first(\alpha) \cap first(\beta) \neq \emptyset$ , делает метод рекурсивного спуска неприменимым

$S \rightarrow A \mid B$
$A \rightarrow aA \mid c$
$B \rightarrow aB \mid b$

# Применение рекурсивного спуска

- Если в грамматике для правил  $X \rightarrow \alpha \mid \beta$  выполняются соотношения  $\alpha \Rightarrow \varepsilon$  и  $\beta \Rightarrow \varepsilon$ , то метод рекурсивного спуска **заведомо неприменим**
- Для грамматики  $G_{pc}$  наличие  $\varepsilon$ -правила не приводит к невозможности использования метода рекурсивного спуска, для второй грамматики  $G_{nrc}$  препятствие есть:

$$G_{pc}: \begin{array}{l} S \rightarrow cAd \mid d \\ A \rightarrow aA \mid \varepsilon \end{array}$$

$$G_{nrc}: \begin{array}{l} S \rightarrow Bd \\ A \rightarrow aA \mid \varepsilon \\ B \rightarrow cAa \mid a \end{array}$$

- В грамматике  $G_{nrc}$  любой вывод, содержащий  $A$ , имеет вид:  $S \rightarrow Bd \rightarrow cAad \rightarrow \dots \rightarrow ca\dots aAad$ , и сделать выбор по текущему символу невозможно

# Рекурсивный спуск для итераций

- Общий вид правил для описания синтаксиса последовательностей однотипных конструкций:

$L \rightarrow a / a, L$  (в сокращённой форме:  $L \rightarrow a \{,a\}$ )

- Условия применимости метода рекурсивного спуска для грамматик с правилами для списков не выполнены: в цепочке  $a, a, a, a, a$  из  $L$  могут выводиться
  - $a$
  - $a, a$
  - $a, a, a, a, a$
- Разбор детерминирован, если всегда выбирается самая длинная подцепочка

# Рекурсивный спуск для итераций

- При анализе цепочек грамматики

$G = (\{a\}, \{L\}, P, L)$ , где  $P = \{L \rightarrow a \mid a, L\}$

методом рекурсивного спуска процедура  $L()$  будет содержать оператор цикла:

```
void L () { if (c != 'a')                ERROR ();  
            while ((GetL (), c) == ',')  
                if ((GetL (), c) != 'a') ERROR ();  
            }
```

# Преобразование грамматик

- Для произвольной контекстно-свободной грамматики нельзя сказать, анализируется заданный ею язык методом рекурсивного спуска или нет
- *Проблема поиска эквивалентной контекстно-свободной грамматики, для которой метод рекурсивного спуска применим, есть **алгоритмически неразрешимая проблема***
- Для некоторых частных видов грамматик, не удовлетворяющих требованиям применимости метода рекурсивного спуска, удаются *преобразования*, позволяющие получить эквивалентные грамматики, пригодные для анализа этим методом



# Применение рекурсивного спуска

1. Устранение левой рекурсии
2. Левая факторизация
3. Подстановка нетерминалов  
(терминализация)
4. Преобразования,  
учитывающие наличие  
 $\epsilon$ -альтернатив

# Устранение левой рекурсии

- Если в грамматике для цепочек есть нетерминальные символы, правила вывода которых леворекурсивны :

$$A \rightarrow A\alpha_1 \mid \dots \mid A\alpha_n \mid \beta_1 \mid \dots \mid \beta_m \quad \begin{array}{l} \alpha_i \in (T \cup N)^+ \quad i = 1, 2, \dots, n \\ \beta_j \in (T \cup N)^* \quad j = 1, 2, \dots, m \end{array}$$

применять метод рекурсивного спуска нельзя

- Непосредственную левую рекурсию можно заменить правой (цепочки  $\beta_j \{\alpha_i\}$ ):

$$\begin{array}{l} A \rightarrow \beta_1 A' \mid \dots \mid \beta_m A' \\ A' \rightarrow \alpha_1 A' \mid \dots \mid \alpha_n A' \mid \varepsilon \end{array}$$

- Если для символа есть одни лишь леворекурсивные правила (альтернативы  $\beta_j$  отсутствуют), то символ  $A'$  не вводится, а правила для символа  $A$  становятся такими:

$$A \rightarrow \alpha_1 A \mid \dots \mid \alpha_n A \mid \varepsilon$$

# Устранение левой рекурсии

- Левая рекурсия может не быть непосредственной:

$$\begin{aligned} S &\rightarrow Aa \mid b \\ A &\rightarrow Ac \mid Sd \mid \varepsilon \end{aligned}$$

из  $S$  имеется вывод  $S \rightarrow Aa \rightarrow Sda$ , но эта рекурсия не является непосредственной, поэтому удаление левой рекурсии необходимо повторить

- Сначала выполняется подстановка правила для  $S$ :

$$A \rightarrow Ac \mid Aad \mid bd \mid \varepsilon$$

- Далее:  $S \rightarrow Aa \mid b$

$$A \rightarrow bdA' \mid A'$$

$$A' \rightarrow cA' \mid adA' \mid \varepsilon$$

$$\begin{array}{ll} \alpha_1 = c & \alpha_2 = ad \\ \beta_1 = bd & \beta_2 = \varepsilon \end{array}$$

# Левая факторизация

- В грамматике есть правила, начинающиеся одинаковыми символами:

$$A \rightarrow a\alpha_1 \mid a\alpha_2 \mid \dots \mid a\alpha_n \mid \beta_1 \mid \dots \mid \beta_m$$

где  $a \in (T \cup N)^+$ ;  $\alpha_i, \beta_j \in (T \cup N)^*$ ,  $\beta_j$  не начинается с 'a'

- Можно объединить правила с общими началами в одно правило, введя новый символ  $A'$ :

$$A \rightarrow aA' \mid \beta_1 \mid \dots \mid \beta_m$$

$$A' \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$$

- Грамматика
 
$$S \rightarrow \underline{\text{if } E \text{ then } S} \mid \underline{\text{if } E \text{ then } S \text{ else } S} \mid a$$

$$E \rightarrow b$$

преобразуется в

$$S \rightarrow \underline{\text{if } E \text{ then } S} S' \mid a$$

$$S' \rightarrow \text{else } S \mid \varepsilon$$

$$E \rightarrow b$$

# Подстановка нетерминалов

- В грамматике есть нетерминальный символ, у которого несколько правил вывода, и среди них есть правила, начинающиеся нетерминальными символами:

$$A \rightarrow B_1\alpha_1 \mid \dots \mid B_n\alpha_n \mid a_1\beta_1 \mid \dots \mid a_m\beta_m$$

$$B_1 \rightarrow \gamma_{11} \mid \dots \mid \gamma_{1k}$$

...

$$B_n \rightarrow \gamma_{n1} \mid \dots \mid \gamma_{np}$$

где  $B_i \in N$   $a_j \in T$   $\alpha_i, \beta_j \in (T \cup N)^*$   $\gamma_{ij} \in (T \cup N)^+$

- Можно заменить символы  $B_i$  их правилами:

$$A \rightarrow \gamma_{11}\alpha_1 \mid \dots \mid \gamma_{1k}\alpha_1 \mid \dots \mid \gamma_{n1}\alpha_n \mid \dots \mid \gamma_{np}\alpha_n \mid a_1\beta_1 \mid \dots \mid a_m\beta_m$$

# $\varepsilon$ -правила

- Наличие  $\varepsilon$ -альтернатив ( $A \rightarrow a_1\alpha_1 \mid \dots \mid a_n\alpha_n \mid \varepsilon$ ) не всегда препятствует применению рекурсивного спуска
- Анализатор для грамматики  $G_1 = (\{a, b, \perp\}, \{S, A\}, P, S)$

$P_1: \quad S \rightarrow bAa\perp \quad A \rightarrow aA \mid \varepsilon$

```
void S () {   if (c != 'b')      ERROR ();
              GetL ();          A ();
              if (c != 'a')      ERROR ();
              GetL ();
              if (c != '\perp')   ERROR ();
            }
```

```
void A () {   if (c == 'a') { GetL (); A (); } // нет 'else'
            } // здесь есть проблема с анализом цепочки 'baaa\perp'
```

# ε-правила

- Проблемы возникают, если подцепочка, следующая за цепочкой, выводимой из  $A$ , начинается таким же символом, как и цепочка, выводимая из  $A$
- В противном случае проблем нет:  
для грамматики  $G_2 = (\{a, b, c, \perp\}, \{S, A\}, P, S)$ , где

$$P_2: \quad S \rightarrow bA\textcolor{red}{c}\perp \\ A \rightarrow aA \mid \varepsilon$$

удастся построить анализатор, работающий методом рекурсивного спуска

# ε-правила

- Проблемы возникают, если подцепочка, следующая за цепочкой, выводимой из  $A$ , начинается таким же символом, как и цепочка, выводимая из  $A$
- В противном случае проблем нет:

```
void S () {   if (c != 'b')      ERROR ();
              GetL ();          A ();
              if (c != 'c')      ERROR ();
              GetL ();
              if (c != '⊥')      ERROR ();
            }
void A () {   if (c == 'a') { GetL (); A (); } // нет 'else'
            }
```



## Достаточные условия применимости рекурсивного спуска для грамматик с $\varepsilon$ -правилами

- $first(A)$  – множество терминальных символов, которыми начинаются цепочки, выводимые из  $A$  в грамматике  $G = (T, N, P, S)$ :  
$$first(A) = \{a \in T \mid A \Rightarrow a\alpha, A \in (T \cup N)^+, \alpha \in (T \cup N)^*\}$$
- $follow(A)$  – множество терминальных символов, которые следуют за цепочками, выводимыми из  $A$ :  
$$follow(A) = \{a \in T \mid S \Rightarrow \alpha A \beta, \beta \Rightarrow a\gamma, A \in N, \alpha, \gamma \in (T \cup N)^*, \beta \in (T \cup N)^+\}$$
- Если  $first(A) \cap follow(A) = \emptyset$ , метод рекурсивного спуска **применим** к данной грамматике
- Если  $first(A) \cap follow(A) \neq \emptyset$ , метод рекурсивного спуска **неприменим** к данной грамматике

# Канонический вид грамматики и достаточные условия для метода рекурсивного спуска

1. либо  $X \rightarrow \alpha$  и это единственное правило вывода для  $X$

2. либо  $X \rightarrow a_1\alpha_1 / a_2\alpha_2 / \dots / a_n\alpha_n$

3. либо  $X \rightarrow a_1\alpha_1 / a_2\alpha_2 / \dots / a_n\alpha_n / \varepsilon$   
и  $\text{first}(X) \cap \text{follow}(X) = \emptyset$

$a_i \in T$  для всех  $i = 1, 2, \dots, n$        $a_i \neq a_j$  для  $i \neq j$   
 $\alpha, \alpha_i \in (T \cup N)^*$

# ε-правила

- Метод неприменим для грамматики

$G_1 = (\{a, b, \perp\}, \{S, A\}, P, S)$ , где

$$P_1: \quad S \rightarrow bAa\perp \\ A \rightarrow aA \mid \varepsilon$$

$first(A) = \{a\}, follow(A) = \{a\} \quad first(A) \cap follow(A) \neq \emptyset$

- Метод применим для грамматики

$G_2 = (\{a, b, c, \perp\}, \{S, A\}, P, S)$ , где

$$P_2: \quad S \rightarrow bAc\perp \\ A \rightarrow aA \mid \varepsilon$$

$first(A) = \{a\}, follow(A) = \{c\} \quad first(A) \cap follow(A) = \emptyset$

# $\varepsilon$ -правила

- Грамматику с правилом, в котором для некоторого символа  $A$  имеется  $\varepsilon$ -альтернатива, можно преобразовать, введя символ  $A'$  ( $A' \equiv A\beta$ ):

$$\begin{aligned} A &\rightarrow \alpha_1 A \mid \dots \mid \alpha_n A \mid \beta_1 \mid \dots \mid \beta_m \mid \varepsilon & A, B \in N \\ B &\rightarrow \alpha A \beta & \alpha, \beta, \alpha_i, \beta_j \in (T \cup N)^* \\ \text{first}(A) \cap \text{follow}(A) &\neq \emptyset \end{aligned}$$

$$\begin{aligned} A &\rightarrow \alpha_1 A' \mid \dots \mid \alpha_n A' \mid \beta_1 \beta \mid \dots \mid \beta_m \beta \mid \beta \\ B &\rightarrow \alpha A' \\ A' &\rightarrow \alpha_1 A' \mid \dots \mid \alpha_n A' \mid \beta_1 \beta \mid \dots \mid \beta_m \beta \mid \beta \end{aligned}$$

- Из  $B$  выводятся цепочки вида  $\alpha \{\alpha_i\} \beta_j \beta$ , либо  $\alpha \{\alpha_i\} \beta$

# $\varepsilon$ -правила

- Преобразовать грамматику  $G_1 = (\{a, b, \perp\}, \{S, A\}, P, S)$  для применения метода рекурсивного спуска:

$$P_1: \quad S \rightarrow bAa\perp \\ A \rightarrow aA \mid \varepsilon$$

- Для удаления  $\varepsilon$ -правила вводится правило  $A' \rightarrow Aa\perp$ :

$$S \rightarrow bA' \quad A' \rightarrow aA' \mid a\perp \quad A \rightarrow aA \mid \varepsilon$$

- Правило для  $A$  удаляется (символ  $A$  бесполезен):

$$S \rightarrow bA' \quad A' \rightarrow aA' \mid a\perp$$

- Объединяются общие начала альтернатив:

$$S \rightarrow bA' \quad A' \rightarrow aA'' \quad A'' \rightarrow A' \mid \perp$$

- Проводится терминализация правила для символа  $A''$ :

$$S \rightarrow bA' \quad A' \rightarrow aA'' \quad A'' \rightarrow aA'' \mid \perp$$

# $\varepsilon$ -правила

- Грамматика содержит пустые правые части в двух правилах:  
 $S \rightarrow aA$   
 $A \rightarrow BC \mid B$   
 $C \rightarrow b \mid \varepsilon$   
 $B \rightarrow \varepsilon$
- Для нетерминала  $A$  из обеих альтернатив выводится пустая цепочка:  
 $BC \Rightarrow \varepsilon$  и  $B \rightarrow \varepsilon$
- Для цепочки “ $a$ ” строятся два различных дерева вывода:  
 $S \rightarrow aA \rightarrow aB \rightarrow a\varepsilon \equiv a$   
 $S \rightarrow aA \rightarrow aBC \rightarrow a\varepsilon C \rightarrow a\varepsilon\varepsilon \equiv a$

# $\varepsilon$ -правила

- Грамматика содержит пустую правую часть в одном правиле:  $S \rightarrow Bd$

$$B \rightarrow a \mid cAa$$

$$A \rightarrow aA \mid \varepsilon$$

$$\text{first}(a) = \{a\}$$

$$\text{first}(cAa) = \{c\}$$

$$\text{first}(a) \cap \text{first}(cAa) = \emptyset$$

- Любой вывод, содержащий  $A$ , имеет вид:

$$S \rightarrow Bd \rightarrow cAad \rightarrow \dots \rightarrow ca\dots aAad$$

# Критерий применимости метода рекурсивного спуска

- Метод рекурсивного спуска применим к контекстно-свободной грамматике  $G$ , если и только если для любых двух её правил  $X \rightarrow \alpha \mid \beta$  выполняются условия:
  1.  $first(\alpha) \cap first(\beta) = \emptyset$
  2. Справедливо не более, чем одно из двух соотношений:  $\alpha \Rightarrow \varepsilon$ ,  $\beta \Rightarrow \varepsilon$
  3. Если  $\beta \Rightarrow \varepsilon$ , то  $first(X) \cap follow(X) = \emptyset$



# Преобразование итераций

- Грамматика со списком элементов, ограниченных символом, совпадающим с внутренним разделителем элементов списка, как пример итерации в правилах:

$$S \rightarrow LB$$

$$L \rightarrow a \{, a\}$$

$$B \rightarrow , b$$

$$S \rightarrow LB$$

$$L \rightarrow a \mid a , L$$

$$B \rightarrow , b$$

- Вводится дополнительный нетерминальный символ  $L'$ :

$$S \rightarrow LB$$

$$L' \rightarrow , L \mid \varepsilon$$

$$L \rightarrow aL'$$

$$B \rightarrow , b$$

$$\textit{first}(L') = \{, \}$$

$$\textit{follow}(L') = \textit{follow}(L) = \textit{first}(B) = \{, \}$$

$$\textit{first}(L') \cap \textit{follow}(L') \neq \emptyset$$

# Преобразование итераций

- Подправляется правило для  $L'$  ( $L' \rightarrow ,aL' \mid \varepsilon$ ):

$$S \rightarrow LB$$

$$L'' \rightarrow ,aL'' \mid \varepsilon$$

$$L \rightarrow aL''$$

$$B \rightarrow , b$$

$$\cancel{L' \rightarrow ,aL' \mid \varepsilon}$$

*из исходных правил:*

$$L(B): \alpha = a \quad \beta = \varepsilon$$

$$L'(A): \alpha_1 = ,a \quad \beta_i = \varepsilon$$

*недостижимое правило*

- Очередное поколение правил в точности повторяет предыдущее, поэтому преобразования по показанным методикам не могут привести к получению грамматики, которые методом рекурсивного спуска смогут обрабатывать подобные списки

# Грамматика с действиями

- В тела процедур вставляются вызовы дополнительных “семантических” процедур:

$$\underline{A \rightarrow aB \mid bC}$$

$$A \rightarrow a\langle D_1 \rangle B\langle D_1; D_2 \rangle \mid bC\langle D_3 \rangle$$
$$A, B, C \in N; a, b \in T$$

$\langle D_i \rangle$  – вызов семантической процедуры  $D_i()$ ,  $i=1,2,3$

- Процедура рекурсивного спуска, выполняющая синтаксический анализ и дополнительные действия:

```
void A () { if (c == 'a') { GetS (); D1 (); B (); D1 (); D2 (); }  
            else if (c == 'b') { GetS ();           C ();           D3 (); }  
            else ERROR ();  
}
```

# Примеры грамматик

- Грамматика, которая позволяет распознавать цепочки языка

$L = \{\alpha \in (0,1)^+ \mid \alpha \text{ содержит равное количество } 0 \text{ и } 1\}$ :

$$G_1 = (\{0, 1\}, \{A, S\}, P_1, S)$$

$$G_2 = (\{0, 1\}, \{S\}, P_2, S)$$

$$P_1: S \rightarrow 0A1 \mid 1A0$$

$$P_2: S \rightarrow 0S1 \mid 1S0 \mid 01 \mid 10$$

$$0A \rightarrow 00A1$$

$$1A \rightarrow 11A0$$

$$A \rightarrow \varepsilon$$

- Грамматика с действиями, дающая тот же результат:

$$S \rightarrow \langle k0 = 0; k1 = 0 \rangle A \mid$$

$$A \rightarrow 0 \langle k0 ++ \rangle A \mid 1 \langle k1 ++ \rangle A \mid$$

$$0 \langle k0 ++; \text{check} () \rangle \mid 1 \langle k1 ++; \text{check} () \rangle$$

void check ()

```
{ if (k0 == k1) { cout << "ХОРОШО!!!" << endl; return 0; }  
  else          { cout << "ПЛОХО!!!"   << endl; return 1; }  
}
```

# Контекстные условия в выражениях

- Разбор выражения:

x \* y + 5 > x  
int \* int  
int + int  
int > int  
bool

- Изменённая цепочка:

x > 5 \* y  
int > int  
bool \* int  
**ошибка**

# Контекстные условия в выражениях

- Разбор выражения:

x \* y + 5 > x  
int \* int  
int + int  
int > int  
bool

- Изменённая цепочка:

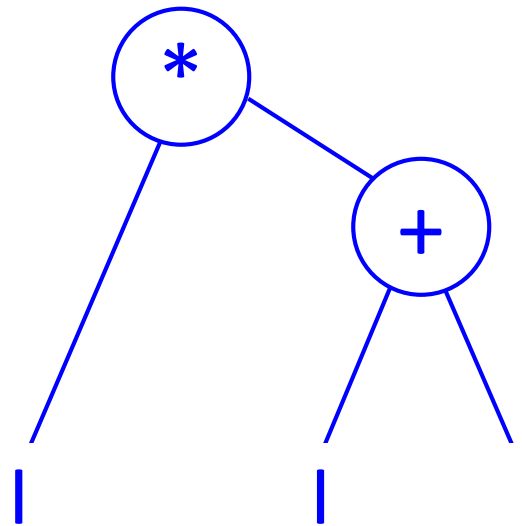
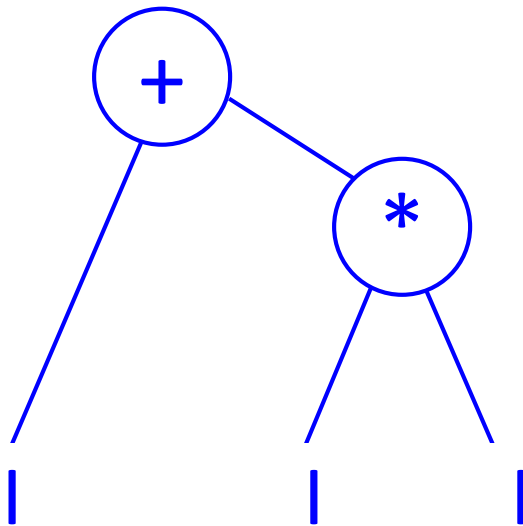
x > 5 \* y  
int \* int  
int > int  
bool

# Контекстные условия в выражениях

- Простейшая грамматика для выражений:

$$G_1: E \rightarrow E + E \mid E * E \mid (E) \mid I$$

- Разные деревья вывода для выражения  $I + I * I$ :



# Контекстные условия в выражениях

- Однозначная правоассоциативная грамматика  $G_2$ , не отражающая старшинство операций:

$$G_2: \begin{array}{l} E \rightarrow T + E \mid T * E \mid T \\ T \rightarrow I \mid (E) \end{array} \quad \xleftarrow{I + I * I}$$

- Симметричная грамматике  $G_2$  однозначная левоассоциативная грамматика  $G_3$ :

$$G_3: \begin{array}{l} E \rightarrow E + T \mid E * T \mid T \\ T \rightarrow I \mid (E) \end{array} \quad \xrightarrow{I + I * I}$$



# Контекстные условия в выражениях

- Правоассоциативная, учитывающая старшинство операций грамматика  $G_4$ :

$$G_4: E \rightarrow T \mid T + E$$

$$T \rightarrow F \mid F * T$$

$$F \rightarrow I \mid (E)$$

$$10 * 3 / 5 = 0$$

- Левассоциативная грамматика  $G_5$ :

$$G_5: E \rightarrow T \mid E + T$$

$$T \rightarrow F \mid T * F$$

$$F \rightarrow I \mid (E)$$

рекурсивный спуск  
неприменим

# Контекстные условия в выражениях

- Грамматика  $G_6$  (рекурсия заменена итерацией):

$$G_6: E \rightarrow T \{ + T \}$$

$$T \rightarrow F \{ * F \}$$

$$F \rightarrow I \mid (E)$$

- Семантические проверки:

$$E \rightarrow T \{ + T \langle D_T \rangle \}$$

$$T \rightarrow F \{ * F \langle D_F \rangle \}$$

$$F \rightarrow I \langle D_I \rangle \mid (E)$$

- $D_I$  – проверка одиночного операнда
- $D_T$  – проверка совместимости слагаемых
- $D_F$  – проверка совместимости множителей

# Основные свойства языков внутреннего представления

- Способность фиксации синтаксической структуры исходной программы
- Возможность автоматической генерации текста на языках внутреннего представления во время синтаксического анализа
- Относительная простота трансляции в объектный код, либо достаточная эффективность интерпретации конструкций языков внутреннего представления

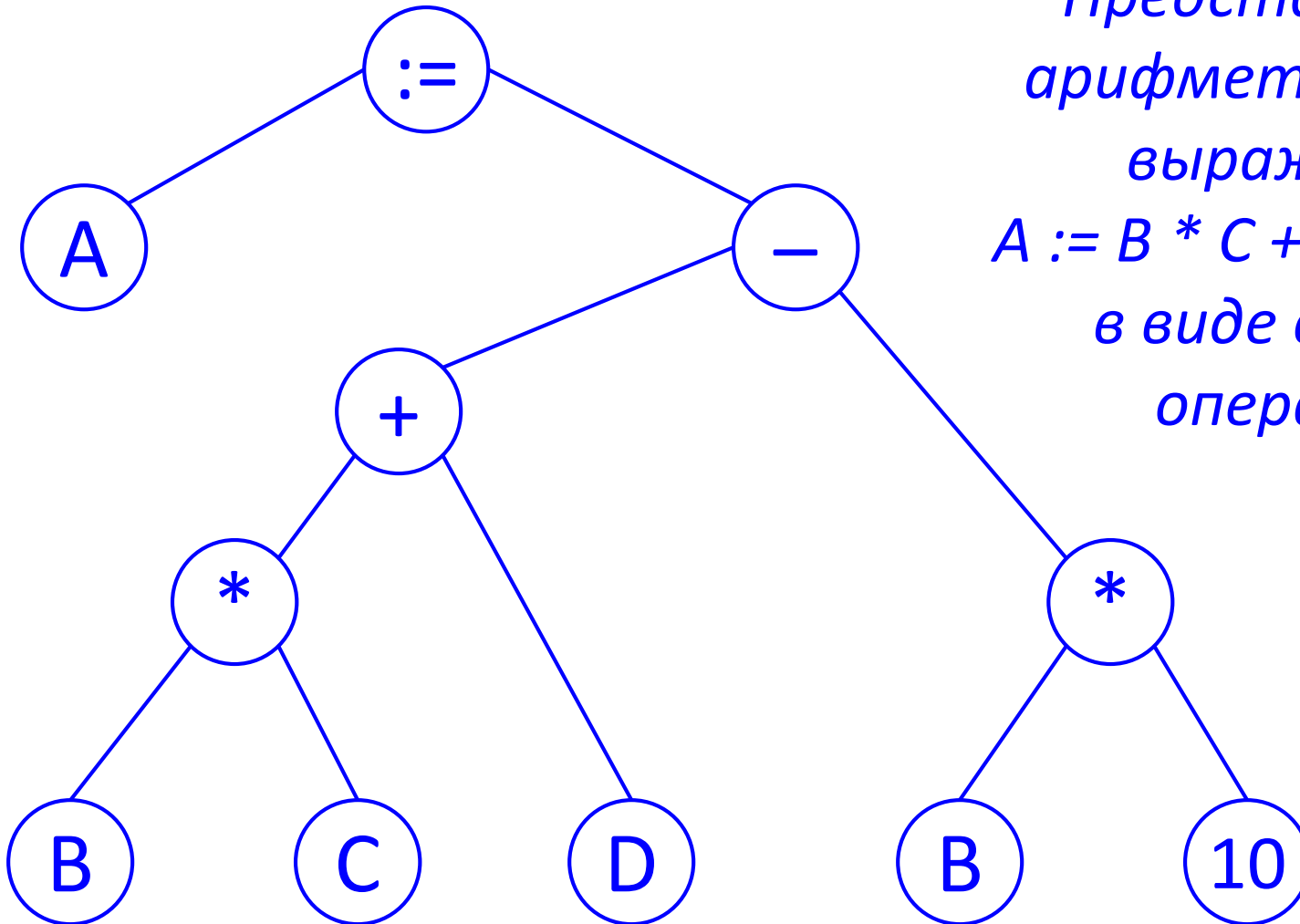
# Способы внутреннего представления программ

- a) связанные списочные структуры, представляющие синтаксическое дерево
- b) многоадресный код с явно именуемыми результатами (тетрады)
- c) многоадресный код с неявно именуемыми результатами (триады)
- d) инфиксная запись
- e) префиксная запись
- f) постфиксная запись
- g) язык ассемблера целевой машины

# Связные списочные структуры

*Представление  
арифметического  
выражения*

*$A := B * C + D - B * 10$   
в виде дерева  
операций*



# Многоадресный код с явно именуемым результатом

- Тетрады представляют собой запись операций в форме четырёх составляющих: операции, двух операндов и результата операции:

<операция> (<операнд1>, <операнд2>, <результат>)

- Представление выражения  $A := B * C + D - B * 10$  в виде тетрад, составляющих линейную последовательность

команд:

1	*	B	C	T1
2	+	T1	D	T2
3	*	B	10	T3
4	−	T2	T3	T4
5	:=	T4	∅	A

# Многоадресный код с неявно именуемым результатом

- Триады представляют собой запись операций в форме трёх составляющих: операции и двух операндов, один из которых является и результатом операции:

$\langle \text{операция} \rangle (\langle \text{операнд1} \rangle, \langle \text{операнд2} \rangle)$

- Представление выражения  $A := B * C + D - B * 10$  в виде триад, составляющих линейную последовательность

команд:

1	*	B	C
2	+	^1	D
3	*	B	10
4	−	^2	^3
5	:=	A	^4

# Другие виды внутреннего представления

- Инфиксная запись:  $A := B * C + D - B * 10$
- Префиксная (прямая польская) запись:  
 $:= A - + * B C D * B 10$
- Постфиксная (обратная или инверсная польская) запись (ПОЛИЗ):  $A B C * D + B 10 * - :=$
- Свойства ПОЛИЗ:
  - Операнды следуют в том же порядке, в каком они следуют в инфиксной записи
  - Операции следуют в том порядке, в каком они должны вычисляться (слева направо)
  - Операции следуют непосредственно за своими операндами



# Определение ПОЛИЗ

1. Если  $E$  является единственным операндом, то ПОЛИЗ выражения  $E$  есть этот операнд
2. ПОЛИЗ выражения  $E_1 \theta E_2$ , где  $\theta$  – знак бинарной операции, а  $E_1$  и  $E_2$  операнды для  $\theta$ , есть запись  $\underline{\underline{E_1 E_2}} \theta$ , где  $\underline{\underline{E_1}}$  и  $\underline{\underline{E_2}}$  – ПОЛИЗ выражений  $E_1$  и  $E_2$  соответственно
3. ПОЛИЗ выражения  $\theta E$ , где  $\theta$  – знак унарной операции, а  $E$  – операнд операции  $\theta$ , есть запись  $\underline{\underline{E}} \theta$ , где  $\underline{\underline{E}}$  – ПОЛИЗ выражения  $E$
4. ПОЛИЗ выражения  $(E)$  есть ПОЛИЗ выражения  $E$

# ПОЛИЗ выражений

- *Простым* называется выражение, состоящее из одной константы или имени переменной
- Границы подвыражений *сложных* выражений могут явно ограничиваться скобками: в выражении ' $(a - b) + c$ ' левым операндом операции '+' является подвыражение ' $a - b$ ', а правым — простое выражение ' $c$ '
- Когда скобки явно не расставлены, учитывается приоритет операций, а также ассоциативность операций одинакового приоритета

# ПОЛИЗ выражений

- В выражении ' $a + b * c$ ' операнд ' $b$ ' относится к операции умножения, и эквивалентное выражение со скобками будет таким: ' $a + (b * c)$ '
- В выражении ' $a - b + c$ ' операнд ' $b$ ' относится к левой операции (к “минусу”, а не к “плюсу”) и эквивалентное выражение со скобками будет таким: ' $(a - b) + c$ '
- Левоассоциативные операции группируются с помощью скобок слева направо: ' $a - b + c - d$ ' эквивалентно ' $((a - b) + c) - d$ '

# Алгоритм Дейкстры

- Пока есть ещё символы для чтения:
  - Читается очередной символ
  - Если символ – операнд, он добавляется к выходной строке
  - Если символ – имя функции, он помещается в стек
  - Если символ есть разделитель параметров функции:
    - До тех пор, пока верхним элементом стека не станет открывающая скобка, выталкиваются элементы из стека в выходную строку. Если открывающей скобки нет, значит, в выражении не согласованы скобки, либо неверно поставлен разделитель. Сам разделитель помещается в стек в виде открывающей скобки

# Алгоритм Дейкстры

- Если символ есть операция (**o1**), то:
  1. пока...
    - ...(если операция **o1** правоассоциативна)  
приоритет **o1** меньше ( $<$ ) приоритета  
операции, находящейся на вершине стека...
    - ...(иначе) приоритет **o1** не больше ( $\leq$ )  
приоритета операции, находящей на вершине  
стека...

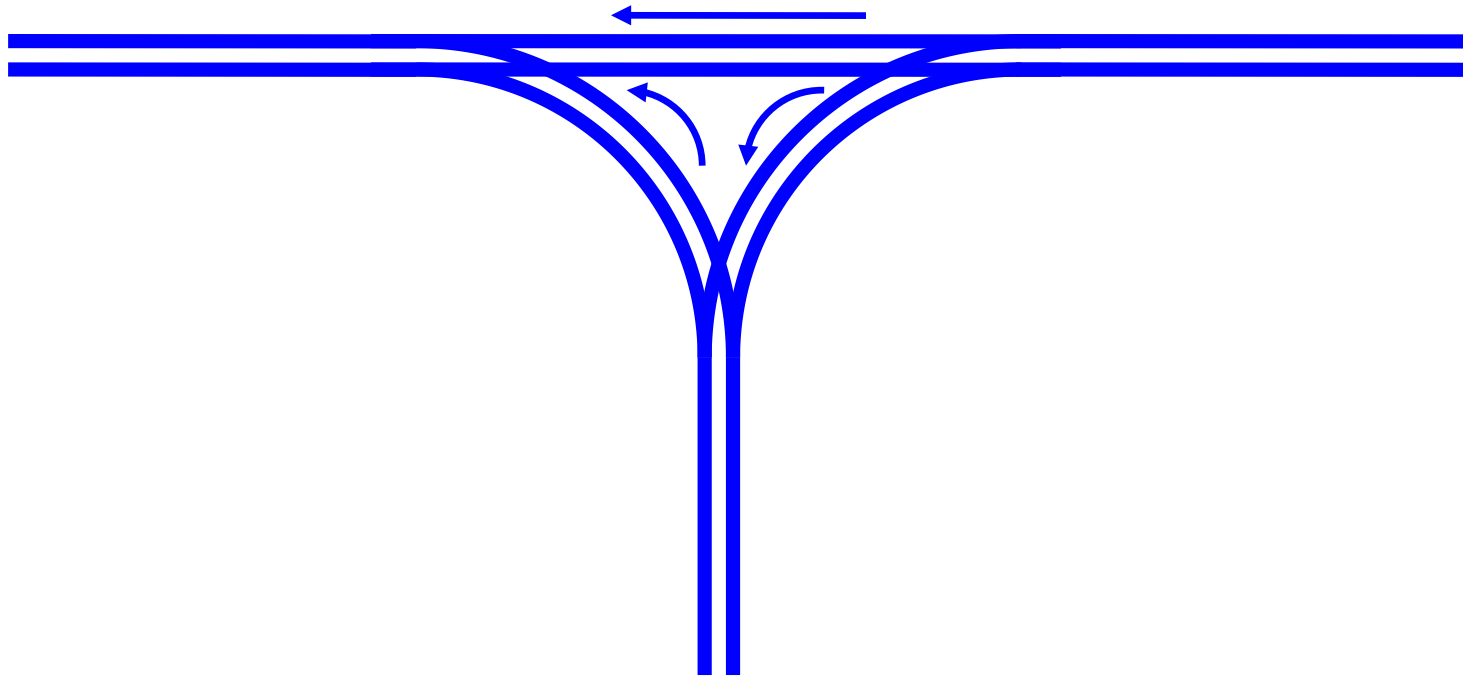
...верхние элементы стека  
выталкиваются в выходную строку
  2. операция **o1** помещается в стек

# Алгоритм Дейкстры

- Если символ есть открывающая скобка, он помещается в стек
- Если символ есть закрывающая скобка, элементы из стека выталкиваются в выходную строку до тех пор, пока на вершине стека не окажется открывающая скобка. Открывающая скобка удаляется из стека, но в выходную строку не добавляется. Если после этого шага на вершине стека оказывается имя функции, оно добавляется к выходной строке
- Когда входная строка заканчивается, все символы из стека выталкиваются в выходную строку

# Алгоритм Дейкстры (сортировочная станция)

$A \ B \ C * D + B \ 10 * - := \quad A := B * C + D - B * 10$



# Ассоциативность в ПОЛИЗ

- Сложение есть левоассоциативная операция, следующие друг за другом операции с таким же приоритетом выполняются слева направо:

$$a + b + c + E \quad ((a + b) + c) + E \quad a b + c + E +$$

- Операция присваивания Си++ есть операция правоассоциативная, в операциях множественного присваивания операции выполняются справа налево:

$$a = b = c = E \quad a = (b = (c = E)) \quad a b c E = = =$$



# Алгоритм интерпретации ПОЛИЗ

- Выражение в ПОЛИЗ при вычислении просматривается **слева направо**, при этом если очередной элемент ПОЛИЗ это:
  - операнд – его значение заносится в стек
  - знак *n*-местной операции – из стека извлекаются нужные операнды, выполняется операция, результат заносится в стек
- В конце вычислений в стеке остаётся один элемент – значение всего выражения
- Для интерпретации необходима информация, хранящаяся в таблицах

# Вызов функции в ПОЛИЗ

- ПОЛИЗ вызова функции представляет собой последовательность фактических параметров в ПОЛИЗ, за которой следует имя функции

$$f(p, s) \Rightarrow \underline{p} \ \underline{s} \ f()$$

- Для функций с переменным числом параметров перед именем функции в ПОЛИЗ вставляется дополнительный параметр — количество настоящих фактических параметров

$$\textit{printf}(\textit{fmt}, ...) \Rightarrow \underline{\textit{fmt}} \ \underline{...} \ \textit{num\_par} \ \textit{printf}()$$

# Оператор-выражение в ПОЛИЗ

- В некоторых языках программирования присваивание является операцией, а не оператором, поэтому при интерпретации ПОЛИЗ присвоенное значение сохраняется в стеке (как результат операции)
- Для удаления ненужного значения с вершины стека для корректной интерпретации ПОЛИЗ этих языков используется специальная операция ‘;’ с единственным побочным эффектом удаления текущего элемента из стека

# Оператор присваивания

- **Оператор присваивания** является двухместным:  $I := E$
- Оператор присваивания в ПОЛИЗ:  
или  $\&I \underline{\underline{E}} := ;$   
 $\underline{\underline{I}} \underline{\underline{E}} := ;$
- Операнды двухместной операции присваивания  $':=$ 
  - адрес переменной  $I$   
(обозначается как  $\&I$  или  $\underline{\underline{I}}$ ) и
  - ПОЛИЗ выражения  $E$  (обозначается как  $\underline{\underline{E}}$ )
- Операция  $';'$  удаляет ненужный результат

# Унарный минус в ПОЛИЗ

- Способы устранения неоднозначности с унарным и бинарным минусом:
  - замена унарной операции на бинарную, считая, что ' $-a$ ' означает ' $(0-a)$ ':

$$a*(-x+10*y-123)+b \Rightarrow a \text{ 0 } x - 10 y * + 123 - * b +$$

- введение специального знака для обозначения унарной операции (замена ' $-a$ ' на ' $@a$ ':

$$a*(-x+10*y-123)+b \Rightarrow a \text{ x } @ 10 y * + 123 - * b +$$

# Унарные операции увеличения и уменьшения Си++

- Запись  $++x$  эквивалентна  $x = x + 1$   
правильный перевод:  $\&x \ x \ 1 \ + \ :=$
- Рекомендуемый вариант:  $\&x \ ++$      $\&x \ --$
- Запись  $x++$  эквивалентна  $y = x, x = x + 1, y$   
правильный перевод:  $\&y \ x \ := \ ; \ \&x \ x \ 1 \ + \ := \ ; \ y$
- Рекомендуемый вариант:  $\&x \ \#+$      $\&x \ \#-$

# Безусловные переходы в ПОЛИЗ

- Технически ПОЛИЗ представляется контейнером с итератором произвольного доступа
- В предположении, что ПОЛИЗ оператора, помеченного меткой  $L_0$ , начинается с номера  $p_0$ , оператор перехода ***goto***  $L_0$  записывается как

$p_0 !$

где  $p_0 !$  есть одноместная операция выбора элемента ПОЛИЗ, номер которого равен  $p_0$ , и предшествующий ей её операнд

# Условные переходы в ПОЛИЗ

- Для перевода в ПОЛИЗ условных операторов и операторов цикла используется бинарная операция условного перехода “по лжи” с семантикой

***if (! (B<sub>1</sub>)) goto L<sub>1</sub>***

- ПОЛИЗ операции условного перехода:

***B<sub>1</sub> p<sub>1</sub> !F*** где

***B<sub>1</sub>*** – ПОЛИЗ условного выражения ***B<sub>1</sub>***

***p<sub>1</sub>*** – номер элемента, с которого начинается  
ПОЛИЗ оператора, помеченного меткой ***L<sub>1</sub>***

***!F*** – знак операции



# Условный оператор

- Семантика условного оператора

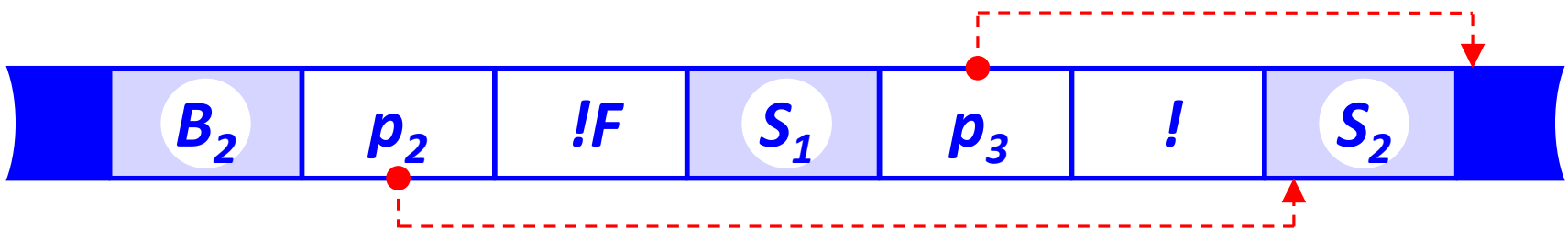
*if*  $B_2$  *then*  $S_1$  *else*  $S_2$

*if*  $(!(B_2))$  *goto*  $L_2$ ;  $S_1$ ; *goto*  $L_3$ ;  $L_2$ :  $S_2$ ;  $L_3$ : ...

- ПОЛИЗ условного оператора:

$\underline{\underline{B_2}}$   $p_2$   $!F$   $\underline{\underline{S_1}}$   $p_3$   $!$   $\underline{\underline{S_2}}$  ... где

$p_i$  – номер элемента, с которого начинается ПОЛИЗ оператора с меткой  $L_i$ ,  $i = 2, 3$



# Условный оператор языка Си++

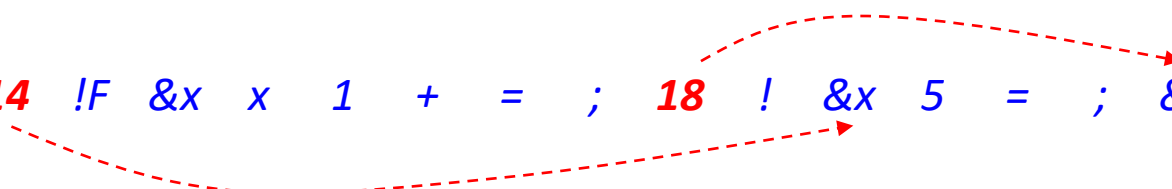
- Условный оператор

*if* (*x* > 0) *x* = *x* + 1; *else* *x* = 5; *y* = *x*;

- ПОЛИЗ условного оператора:

*x* 0 > 14 !F &*x* *x* 1 + = ; 18 ! &*x* 5 = ; &*y* *x* = ;

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21



- Для реализации других операторов языков программирования могут потребоваться другие дополнения к операциям ПОЛИЗ. Например, для реализации переключателя языка Си++ (*switch-case-...-case*) требуется операция дублирования верхнего элемента стека.

# Оператор цикла с предусловием

- Семантика оператора цикла с предусловием

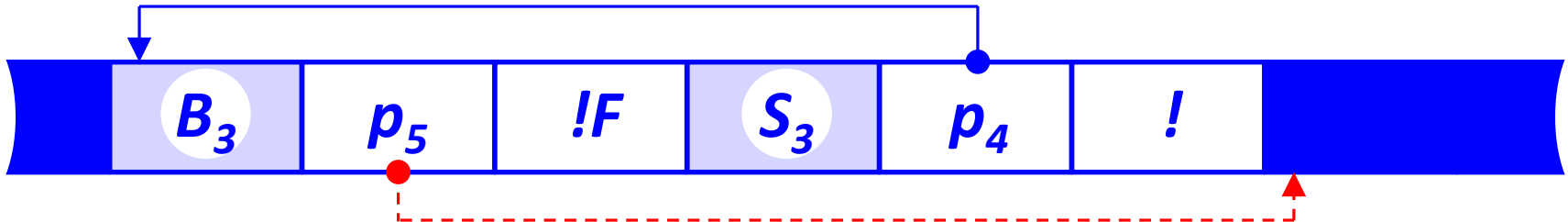
*while*  $B_3$  *do*  $S_3$

$L_4$ : *if*  $(!(B_3))$  *goto*  $L_5$ ;  $S_3$ ; *goto*  $L_4$ ;  $L_5$ : ...

- ПОЛИЗ оператора цикла с предусловием:

$\underline{\underline{B_3}}$   $p_5$   $!F$   $\underline{\underline{S_3}}$   $p_4$   $!$  ... где

$p_i$  – номер элемента, с которого начинается  
ПОЛИЗ оператора с меткой  $L_i$ ,  $i = 4, 5$



# Оператор цикла с постусловием

- Семантика оператора цикла с постусловием

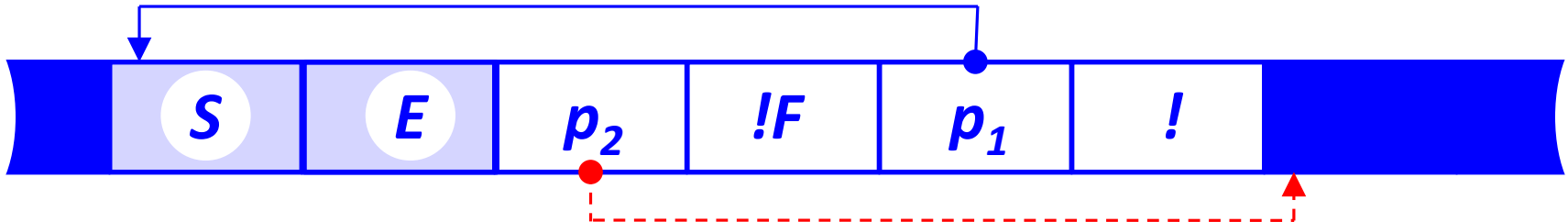
*do S while (E)*

*L<sub>1</sub>: S; if (! (E)) goto L<sub>2</sub>; goto L<sub>1</sub>; L<sub>2</sub>: ...*

- ПОЛИЗ оператора цикла с постусловием:

*S E p<sub>2</sub> !F p<sub>1</sub> ! ...*      где

*p<sub>i</sub>* – номер элемента, с которого начинается  
ПОЛИЗ оператора с меткой *L<sub>i</sub>*, *i* = 1, 2

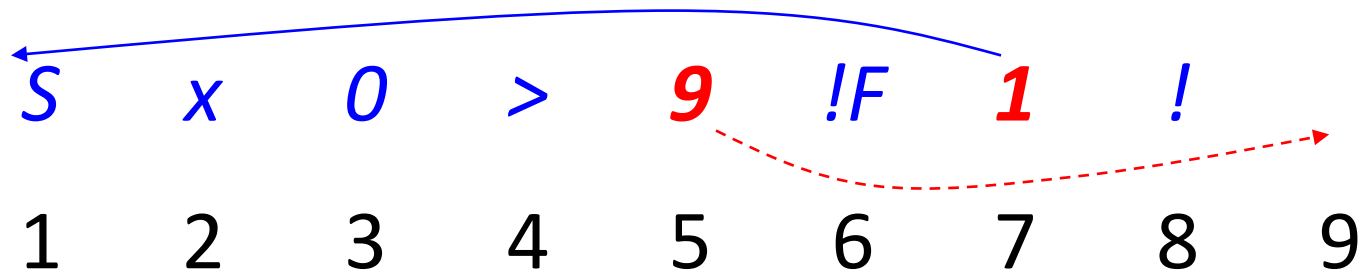


# Пример оператора цикла

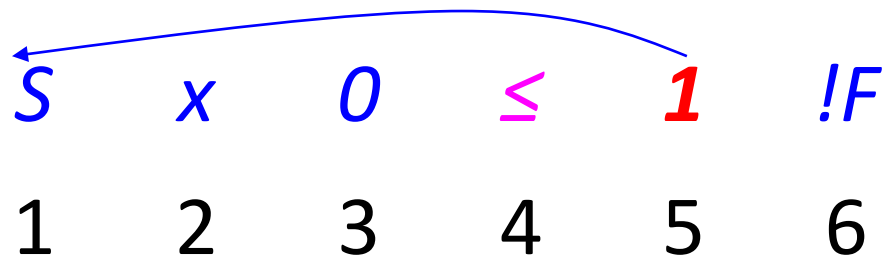
- Оператор цикла с постусловием:

*do S while (x > 0);*

- ПОЛИЗ оператора цикла с постусловием:



- Ошибочный перевод (*repeat S until x ≤ 0*):



# Оператор цикла языка Си++

- Семантика оператора цикла с пересчётом

*for* ( $E_1; E_2; E_3$ )  $S$

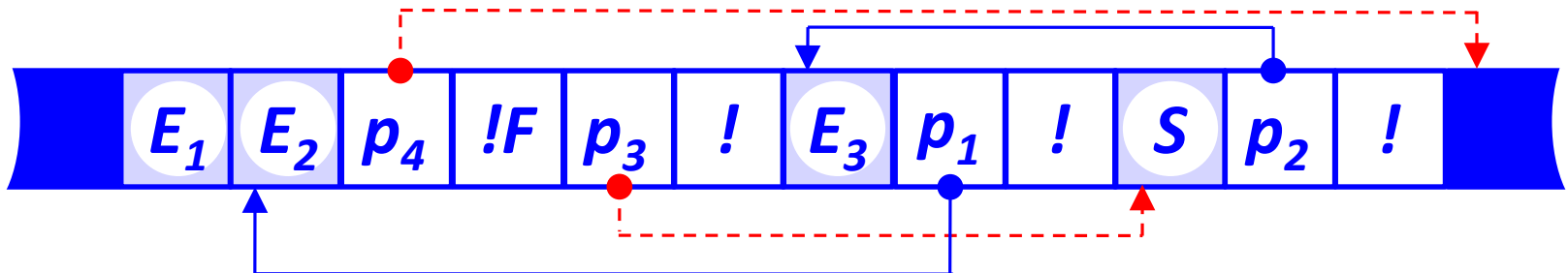
$E_1; L_1: \text{if } (! (E_2)) \text{ goto } L_4; \quad \text{goto } L_3;$

$L_2: \quad E_3; \text{ goto } L_1; L_3: S; \text{ goto } L_2; L_4: \dots$

- ПОЛИЗ оператора цикла с пересчётом:

$\underline{\underline{E_1}} \quad \underline{\underline{E_2}} \quad p_4 \quad !F \quad p_3 \quad ! \quad \underline{\underline{E_3}} \quad p_1 \quad ! \quad \underline{\underline{S}} \quad p_2 \quad ! \dots$  где

$p_i$  – номер элемента с меткой  $L_i, i = 2, 3$

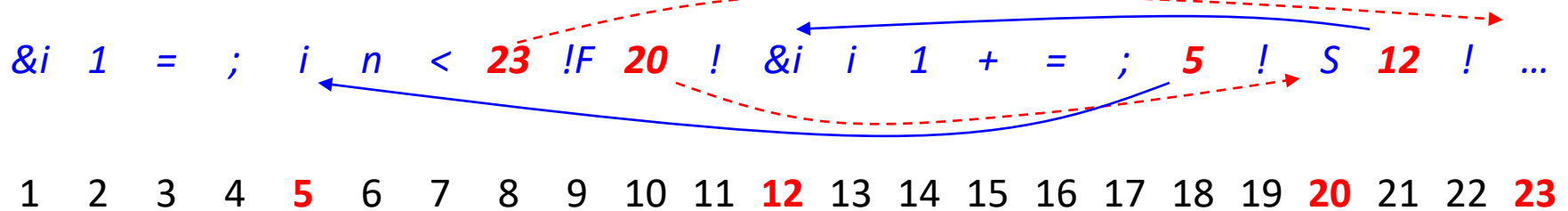


# Пример оператора цикла

- Семантика оператора цикла с пересчётом:

*for* ( $i = 1; i < n; i = i + 1$ )  $S$ ;

- ПОЛИЗ оператора цикла с пересчётом:



- Ошибочная трактовка:

$E_1; L_1: \text{if } (! (E_2)) \text{ goto } L_2; S; E_3; \text{goto } L_1; L_2: \dots$

# Условное выражение

- Семантика условного выражения:

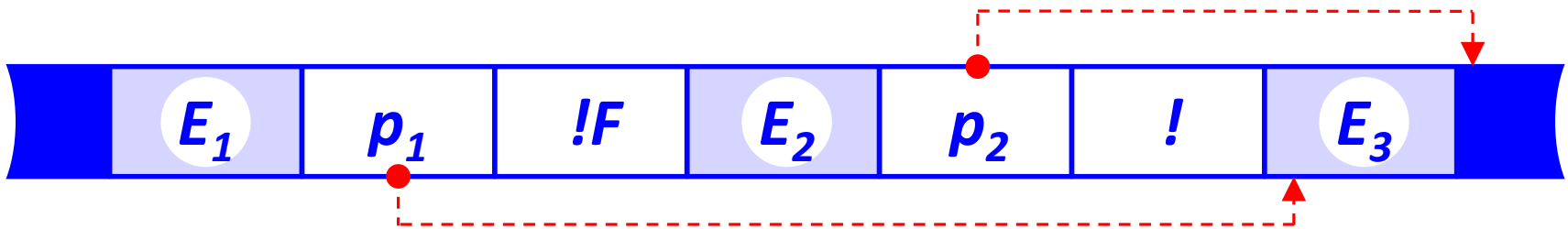
$$E_1 ? E_2 : E_3$$

*if* (! ( $E_1$ )) *goto*  $L_1$ ;  $E_2$ ; *goto*  $L_2$ ;  $L_1$ :  $E_3$ ;  $L_2$ : ...

- ПОЛИЗ условного выражения:

$\underline{E}_1 \ p_1 \ !F \ \underline{E}_2 \ p_2 \ ! \ \underline{E}_3 \ \dots$  где


$p_i$  – номер элемента с меткой  $L_i$ ,  $i = 1, 2$



- Неверный перевод:  $\underline{E}_1 \ \underline{E}_2 \ \underline{E}_3 \ ?$



# Ошибочные трактовки

- Оператор:  $\text{if } (a > b) a = a - b$   
неверно переводится  
 $a > b \ \&a \ a = a - b \ ; \ \text{if}$
- Объясняющие примеры:  $x < y ? x++ : x--$   
 $\text{if } (x < y) x++; \text{else } x--;$
- Не допускается размножение операций:  
 $z = x < y ? x - a : x + a$
- Условное выражение иногда трактуется неверно  
 $x < y ? z = x - a : z = x + a$
- Правильно так:  $\&z \ x \ y < p_1 ! F \ x \ a - p_2 ! x \ a + =$ 

# Пустой оператор

- Оператор:

```
while(c++ < y); // while(c++ < y){ }  
x=x+1;
```

переводится так:

```
&c #+ y < 9 !F 1 ! &x x 1 + = ;  
1   2   3   4   5   6   7   8   9  10  11  12  13  14
```

- Так же выглядит постфиксная запись оператора

```
do; while(c++ < y);  
x=x+1;
```

# Логические операции

- Разные подходы, принятые в языках Паскаль и Си к реализации логических операций  $a \ \&\& \ b$  и  $a \ || \ b$ , приводят к разному представлению этих операций в ПОЛИЗ:

$\underline{a} \ \underline{b} \ \&\&$

$\underline{a} \ \underline{b} \ ||$

ПОЛИЗ операций  $\&\&$  и  $||$  в языке Паскаль

$\underline{a} \ ? \ \underline{b} \ ? \ 1 : 0 : 0$

Семантика операции  $\&\&$  в языках Си, Си++

$\underline{a} \ ? \ 1 : \underline{b} \ ? \ 1 : 0$

Семантика операции  $||$  в языках Си, Си++

- ПОЛИЗ операции  $\&\&$  в языках Си, Си++

$a \ 13 \ !F \ b \ 10 \ !F \ 1 \ 14 \ ! \ 0 \ 14 \ ! \ 0$

1    2    3    4    5    6    7    8    9    10    11    12    13    14

- ПОЛИЗ операции  $||$  в языках Си, Си++

$a \ 7 \ !F \ 1 \ 14 \ ! \ b \ 13 \ !F \ 1 \ 14 \ ! \ 0$

1    2    3    4    5    6    7    8    9    10    11    12    13    14

# Синтаксически управляемый перевод

- Синтаксис и семантика языков взаимосвязаны
- При синтаксически управляемом переводе в соответствии с семантикой входных и выходных правил каждому правилу входного языка сопоставляются правила выходного языка
- С каждой вершиной дерева синтаксического разбора  $N$  связывается цепочка  $C(N)$
- Образ вершины  $N$  строится путём сцепления в определённом порядке последовательности  $C(N)$  и последовательностей цепочек, связанных со всеми вершинами, являющимися прямыми потомками вершины  $N$
- Процесс перевода продолжается снизу вверх в порядке, управляемом структурой дерева
- Перевод программы состоит в поиске образа корня дерева

# Формальный перевод

- Формальный перевод  $\tau$  – это подмножество множества всевозможных пар цепочек  $(\alpha, \beta)$  в алфавитах  $T_1$  и  $T_2$ :  $\tau \subseteq (T_1^* \times T_2^*)$   
где  $\alpha \in T_1^*, \beta \in T_2^*$
- Входной язык перевода  $\tau$ :
$$L_{\text{вх}} = \{\alpha \mid \exists \beta: (\alpha, \beta) \in \tau\}$$
- Целевой (выходной) язык перевода  $\tau$ :
$$L_{\text{ц}} = \{\beta \mid \exists \alpha: (\alpha, \beta) \in \tau\}$$
- Перевод  $\tau$  *неоднозначен*, если для некоторых  $\alpha \in T_1^*, \beta, \gamma \in T_2^*, \beta \neq \gamma$   
 $(\alpha, \beta) \in \tau$  и  $(\alpha, \gamma) \in \tau$

# Синтаксически управляемый перевод

- С помощью грамматики с действиями выполнить перевод цепочек языка  $L_1 = \{0^n 1^m \mid n \geq 0, m > 0\}$  в цепочки языка  $L_2 = \{a^m b^n \mid n \geq 0, m > 0\}$
- Определение перевода  $\tau$* : для любых  $n \geq 0, m > 0$  цепочке  $0^n 1^m \in L_1$  соответствует цепочка  $a^m b^n \in L_2$ ,  
 $\tau = \{ (0^n 1^m, a^m b^n) \mid n \geq 0, m > 0 \}$ ,  $L_{\text{вх}}(\tau) = L_1$ ,  $L_{\text{ц}}(\tau) = L_2$
- Грамматика языка  $L_1$ :  
$$S \rightarrow 0S \mid 1A$$
$$A \rightarrow 1A \mid \varepsilon$$
- Действия по переводу цепочек " $0^n 1^m$ " в цепочки " $a^m b^n$ ":  
$$S \rightarrow 0S \text{ <Put ('b')> } \mid 1 \text{ <Put ('a')> } A$$
$$A \rightarrow 1 \text{ <Put ('a')> } A \mid \varepsilon$$

# Синтаксически управляемый перевод

- Перевести цепочки языка  $L_1 = \{a^n c^m b^n \mid n, m \geq 0\}$  в цепочки языка  $L_2 = \{0^m 1^{n+m} \mid n, m \geq 0\}$
- Цепочку “ $aaccsbb$ ” ( $n = 2, m = 3$ ) перевести в цепочку “ $00011111$ ” ( $m = 3, n + m = 5$ )
- Грамматика языка  $L_1$ :  $S \rightarrow aSb \mid A$   
 $A \rightarrow cA \mid \varepsilon$
- Вычисление множеств  $first(A)$  и  $follow(A)$ :  
 $first(A) = \{c\}$        $follow(A) = \{b\}$
- Действия по переводу цепочек “ $a^n c^m b^n$ ” в цепочки “ $0^m 1^{n+m}$ ”:  
 $S \rightarrow aSb <cout << '1'> \mid A$   
 $A \rightarrow c <cout << '0'> A <cout << '1'> \mid \varepsilon$

# Преобразование в ПОЛИЗ

- Простые выражения:
$$E \rightarrow T \{+ T\}$$
$$T \rightarrow F \{* F\}$$
$$F \rightarrow a \mid b \mid (E)$$
- Грамматика с действиями:
$$E \rightarrow T \{+ T <Put ('+')>\}$$
$$T \rightarrow F \{* F <Put ('*')>\}$$
$$F \rightarrow a <Put ('a')> \mid b <Put ('b')> \mid (E)$$
- Генератор, совмещённый с синтаксическим анализатором:

```
void E () { T (); while (c == '+') { GetS (); T (); Put ('+'); } }
void T () { F (); while (c == '*') { GetS (); F (); Put ('*'); } }
void F () {     if (c == 'a')      { GetS ();      Put ('a'); }
               else if (c == 'b') { GetS ();      Put ('b'); }
               else if (c == '(') { GetS (); E ();
                                   if (c == ')') GetS (); else Error (); }
               else Error ();
}
```