



# Автоматы и формальные языки

---

Карпов Юрий Глебович  
профессор, д.т.н., зав.кафедрой  
“Распределенные вычисления и компьютерные сети”  
Санкт-Петербургского Политехнического университета

[karpov@dcn.infos.ru](mailto:karpov@dcn.infos.ru)



## Структура курса

---

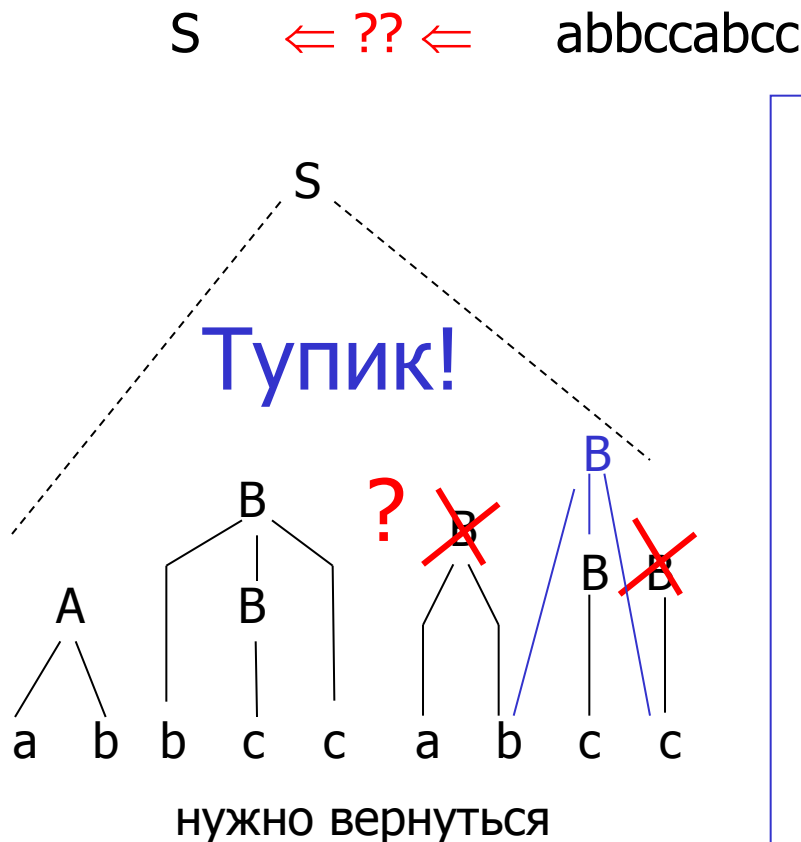
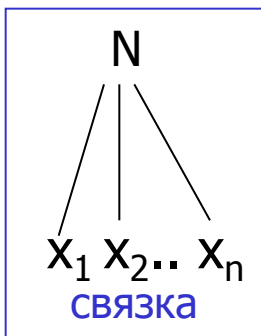
- Конечные автоматы-распознаватели – 4 л
- Порождающие грамматики Хомского – 3 л
- Атрибутные трансляции и двусмысленные КС-грамматики – 2 л
- Распознаватели КС-языков и трансляция – 6 л
  - Лекция 10. s-грамматики, LL(k)-грамматики, грамматики рекурсивного спуска
  - Лекция 11. Построение транслятора языка Милан методом рекурсивного спуска
  - Лекция 12. Грамматики предшествования, LR(k)-грамматики
  - Лекция 13. LR(k), SLR(k) и LALR(k)-грамматики
  - Лекция 14. Компиляторы компиляторов. Yacc и Bison
  - Лекция 15. Грамматики Кока-Янгера-Касами и Эрли
- Дополнительные лекции - 2 л

# Восходящие алгоритмы синтаксического анализа

$S \Rightarrow_1 abScB \Rightarrow_6 abScc \Rightarrow_2 abbAcc \Rightarrow_4 abbcBAcc \Rightarrow_3 abbcBabcc \Rightarrow_6 abbccabcc$

правый вывод

1.  $S \rightarrow abScB$
2.  $S \rightarrow bA$
3.  $A \rightarrow ab$
4.  $A \rightarrow cBA$
5.  $B \rightarrow bBc$
6.  $B \rightarrow c$



Восходящий синтаксический анализ работает так.

Начиная от терминальной строки листьев дерева пытаемся найти "связку" – правую часть продукции грамматики, которую нужно заменить нетерминалом (левой частью продукции), чтобы получить новый узел синтаксического дерева.

Поскольку ищем самую левую связку, восстанавливается правый вывод цепочки.

$S \leftarrow \dots \leftarrow_4 abb\textcolor{red}{c}BAcc \leftarrow_3 abbcB\textcolor{red}{a}bcc \leftarrow_6 abbcc\textcolor{red}{a}bcc$



## Восходящие методы СА

- Восходящие алгоритмы синтаксического анализа просматривая слева направо произвольную sentенциальную форму порождаемого грамматикой языка пытаются найти в ней подцепочку, являющуюся "**СВЯЗКОЙ**" или "**ОСНОВОЙ**" – правой частью продукции грамматики, которую нужно заменить нетерминалом - левой частью этой продукции - с получением предыдущей sentенциальной формы.
- Последовательное выполнение этой операции (возможно, с возвратами) позволяет восстановить правый канонический вывод данной цепочки из начального символа грамматики
- Простейший алгоритм "грубой силы", пытающийся решить эту задачу простым перебором, ищет очередную связку простой проверкой на совпадение правых частей всех продукций грамматики и подстрок sentенциальной формы.

# Как найти связку в очередной снтенциальной форме?

$S \Rightarrow_1 abScB \Rightarrow_6 abScc \Rightarrow_2 abbAcc \Rightarrow_4 abbcBAcc \Rightarrow_3 abbcBabcc \Rightarrow_6 abbccabcc$

$S \Leftarrow_1 abScB \Leftarrow_6 abScc \Leftarrow_2 abbAcc \Leftarrow_4 abbcBAcc \Leftarrow_3 abbcBabcc \Leftarrow_6 abbccabcc$

1.  $S \rightarrow abScB$

2.  $S \rightarrow bA$

3.  $A \rightarrow ab$

4.  $A \rightarrow cBA$

5.  $B \rightarrow bBc$

6.  $B \rightarrow c$

Почему при первой свертке нужно выбрать именно это c?

Почему при второй свертке нужно выбрать именно эту пару ab?

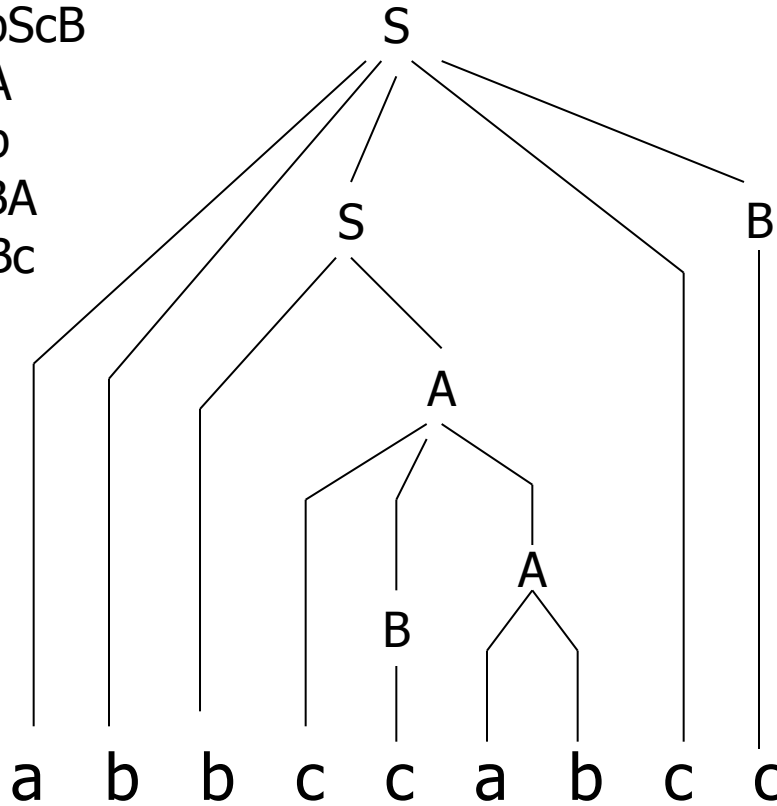
- Простой алгоритм “грубой силы” не использует никакой информации, кроме информации о простом вхождении символов в снтенциальную форму, и поэтому чрезвычайно неэффективен. Были разработаны методы, позволяющие выполнить восходящий синтаксический анализ с более полным учетом информации, содержащейся во входной цепочке и грамматике
- Мы рассматриваем два метода выделения самой левой связки в снтенциальной форме:
  - метод отношений предшествования
  - метод, основанный на использовании всей информации, содержащейся во входной цепочке до связки, и k символов после искомой связки, приводящий к выделению так называемого класса LR(k)-грамматик

Идем по цепочке слева направо, восстанавливаем ПРАВИЙ вывод

## Как найти связку? Святым духом?

$S \xleftarrow{1} abScB \xleftarrow{6} abScc \xleftarrow{4} abbAcc \xleftarrow{4} abbcBAcc \xleftarrow{3} abbcBabcc \xleftarrow{6} abbccabcc$

1.  $S \rightarrow abScB$
2.  $S \rightarrow bA$
3.  $A \rightarrow ab$
4.  $A \rightarrow cBA$
5.  $B \rightarrow bBc$
6.  $B \rightarrow c$



КАК найти в очередной сентенциальной форме на каждом шаге **связку**: ту подстроку, которая сворачивается к очередному нетерминалу (возможно, заглядывая немного вперед)??



6.

$\underbrace{a \ b \ S \ c \ B}_{S \leftarrow abScB}$

5.

$a \ b \ S \ c \ \underbrace{c}_{B \leftarrow c}$

4.

$a \ b \ \underbrace{b \ A}_{S \leftarrow bA} \ c \ c$

3.

$a \ b \ b \ \underbrace{c \ B \ A}_{A \leftarrow cBA} \ c \ c$

2.

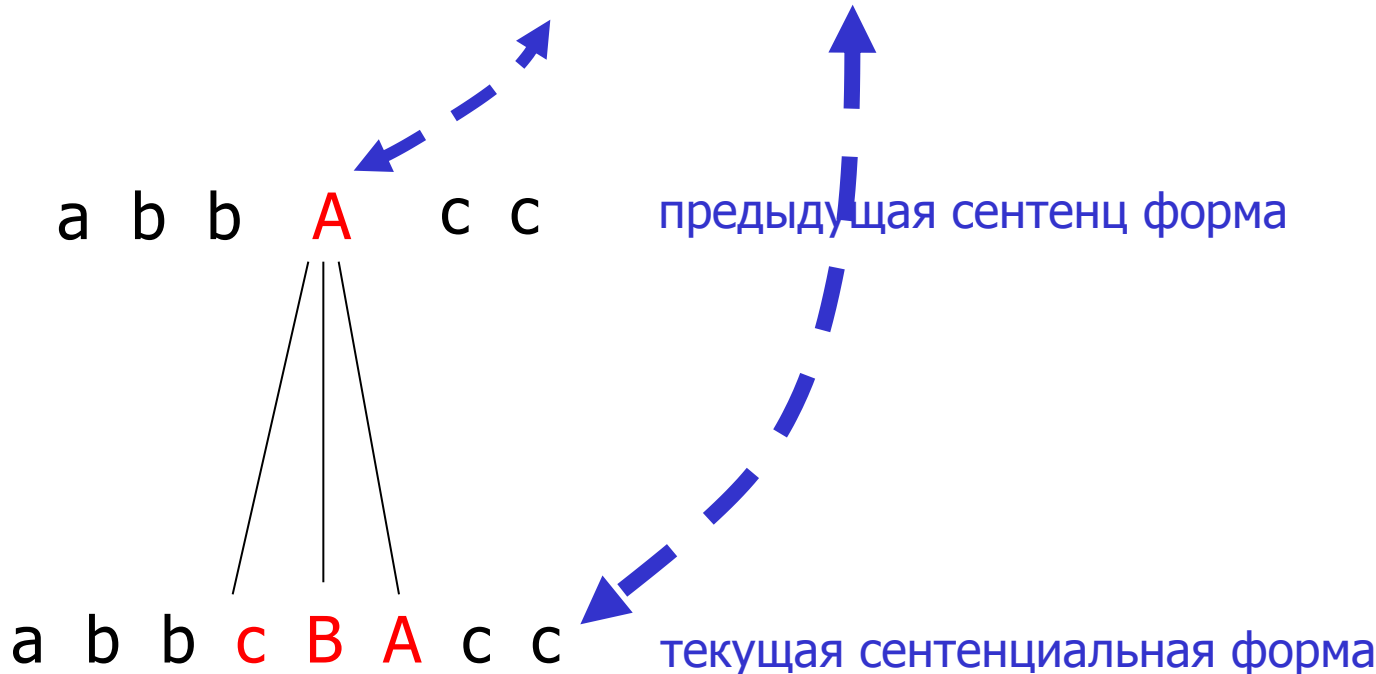
$a \ b \ b \ c \ B \ \underbrace{a \ b}_{A \leftarrow ab} \ c \ c$

1.

$a \ b \ b \ c \ \underbrace{c}_{B \leftarrow c} \ a \ b \ c \ c$

# Повторяющаяся задача: как перейти от очередной сентенциальной формы к предыдущей?

$S \xleftarrow{1} abScB \xleftarrow{2} abScc \xleftarrow{4} abbAcc \xleftarrow{6} abbcBAcc \xleftarrow{3} abbcBabcc \xleftarrow{6} abbccabcc$

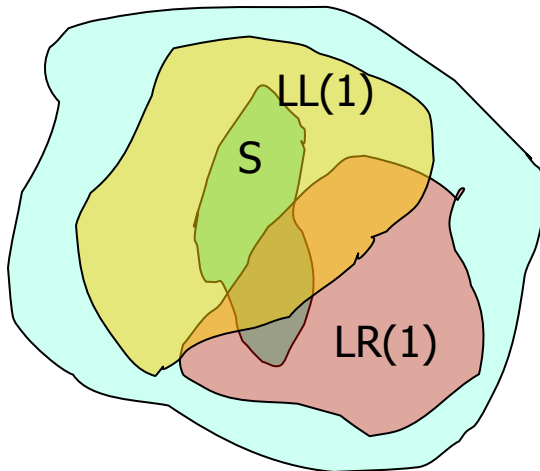


1.  $S \rightarrow abScB$
2.  $S \rightarrow bA$
3.  $A \rightarrow ab$
4.  $A \rightarrow cBA$
5.  $B \rightarrow bBc$
6.  $B \rightarrow c$

Почему не  $A \leftarrow ab$ , не  $B \leftarrow c$  а именно  $A \leftarrow cBA$  ???

Идя по очередной сентенциальной форме слева направо, хотим точно знать, где находится связка, заглянув не более, чем на  $k$  символов за нее. Свернув связку к нужному нетерминалу, получим предыдущую сентенц. форму

# LR(k)–грамматики



Подклассы КС-грамматик  
s-грамматики  
LL(k)-грамматики,  
Грамматики рекурсивного спуска  
Грамматики предшествования  
LR(k)-грамматики,  
LALR(k)-грамматики,  
...

Разработаны Д.Кнутом

LR(k) грамматики представляют наиболее широкий класс недвусмысленных КС-грамматик, для которых построен эффективный изящный метод восходящего синтаксического анализа

(т.е. метод, позволяющий по текущей сентенциальной форме найти связку и определить предыдущую сентенциальную форму)



# Множество ситуаций

**Смысл ситуации:** при просмотре очередной сентенциальной формы мы подошли к точке, в которой, если далее встретится цепочка  $\beta$ , а потом после  $\beta$  должна идти цепочка  $\lambda$  (*lookahead*), то мы можем выполнить свертку  $N \Leftarrow \alpha\beta$

$N \rightarrow \alpha \bullet \beta ; \lambda$  - ситуация

$\lambda$  - правый терминальный контекст длиной  $k$   
*lookahead*

$N \rightarrow \alpha \bullet \beta$  – продукция (правило грамматики), разделенное на две части

точка отделяет уже просмотренную часть продукции (правила грамматики)

- **С точки зрения возможных сверток** каждую позицию внутри сентенциальной формы в процессе ее просмотра можно описать *множеством ситуаций* – множеством продукций, соответствующих этой позиции, с меткой-указателем в каждой продукции, показывающей, в каком месте продукции находится анализатор на данном шаге просмотра сентенциальной формы.

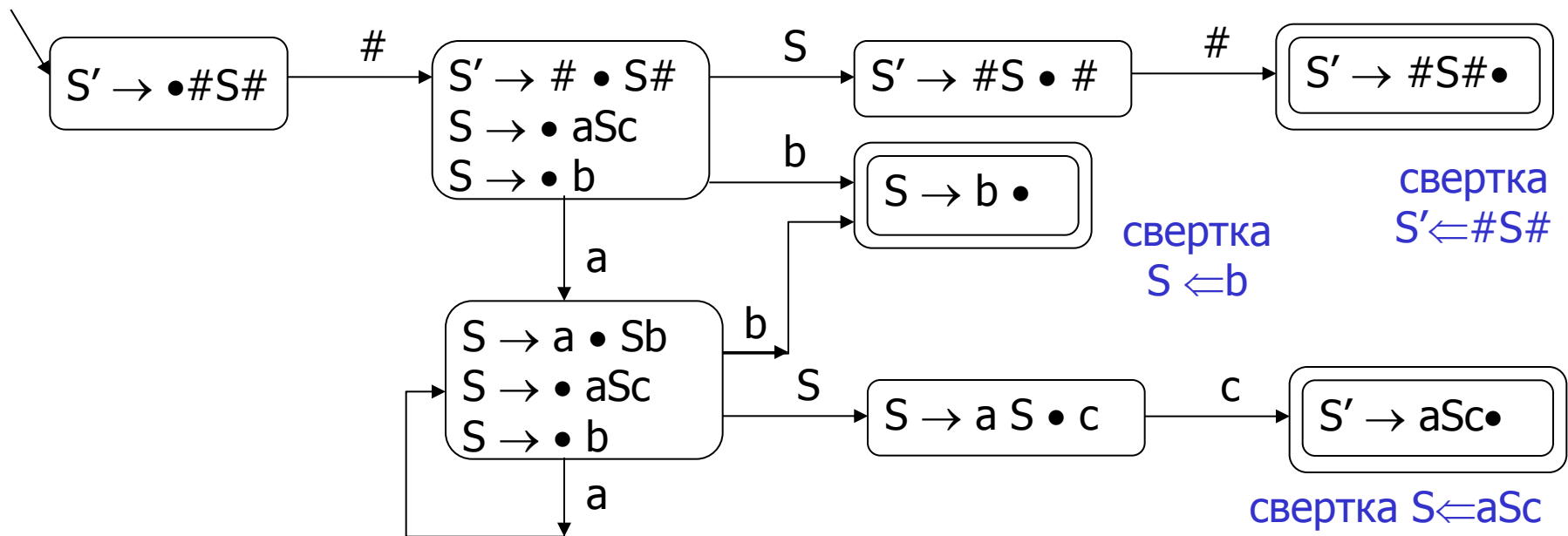
# LR(0) автомат разбора (без *lookahead*)

$$\begin{aligned} S' &\rightarrow \#S\# \\ S &\rightarrow aSc \\ S &\rightarrow b \end{aligned}$$

$\langle N \rightarrow \alpha \bullet \beta \rangle$  - ситуация для LR(0)

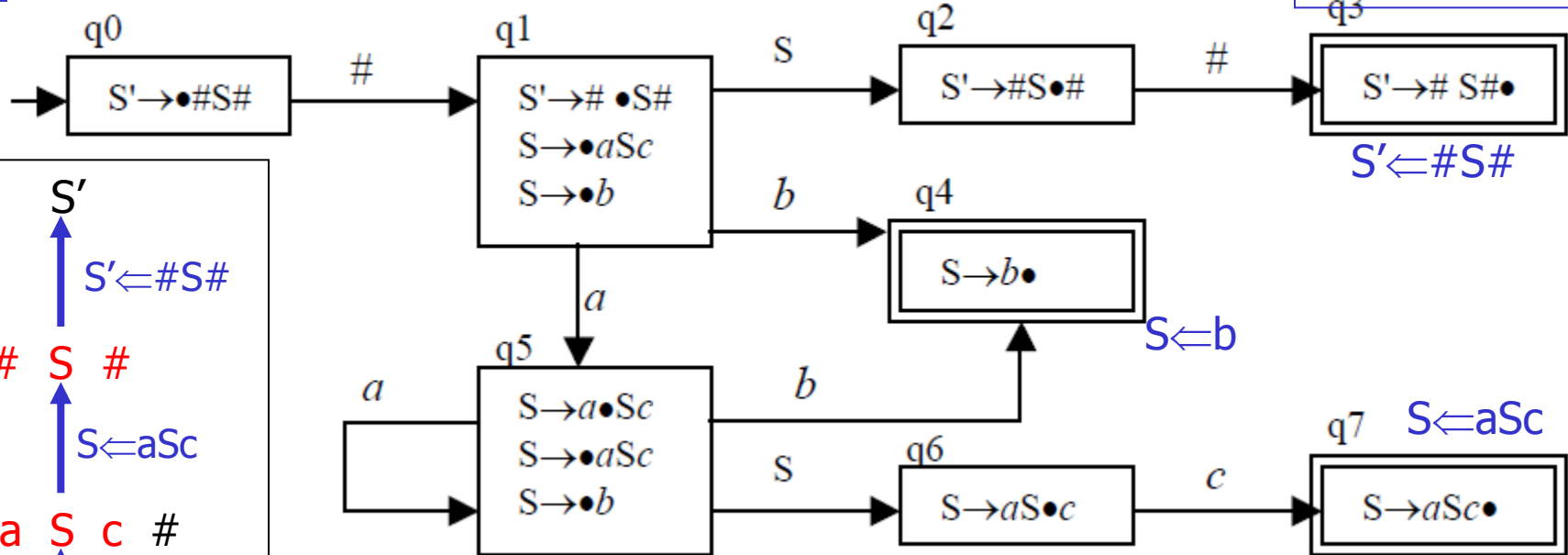
Состояние алгоритма разбора – множество ситуаций

Д.Кнут: число состояний в LR-алгоритме разбора для любой КС-грамматики конечно (почему??)



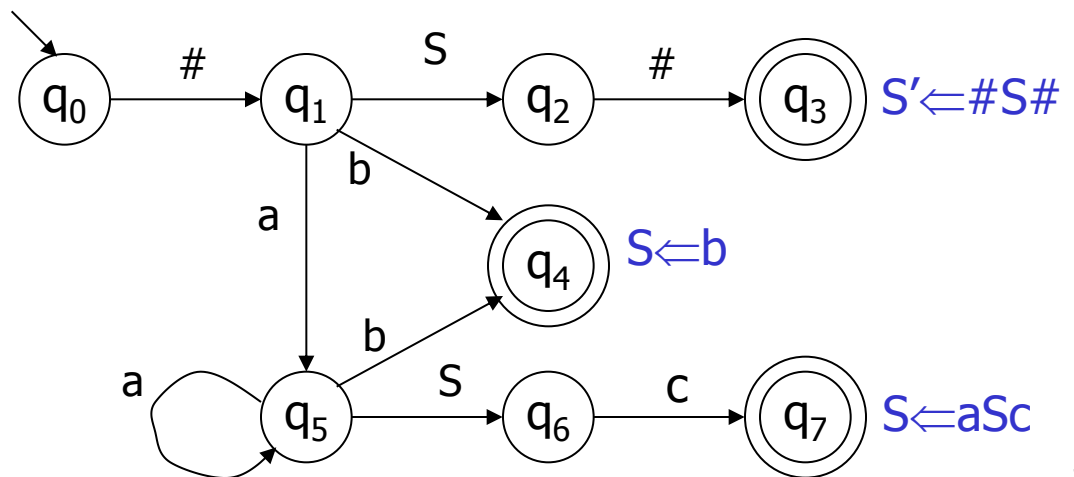
# LR(0) синтаксический анализ

$S' \rightarrow \#S\#$   
 $S \rightarrow aSc$   
 $S \rightarrow b$



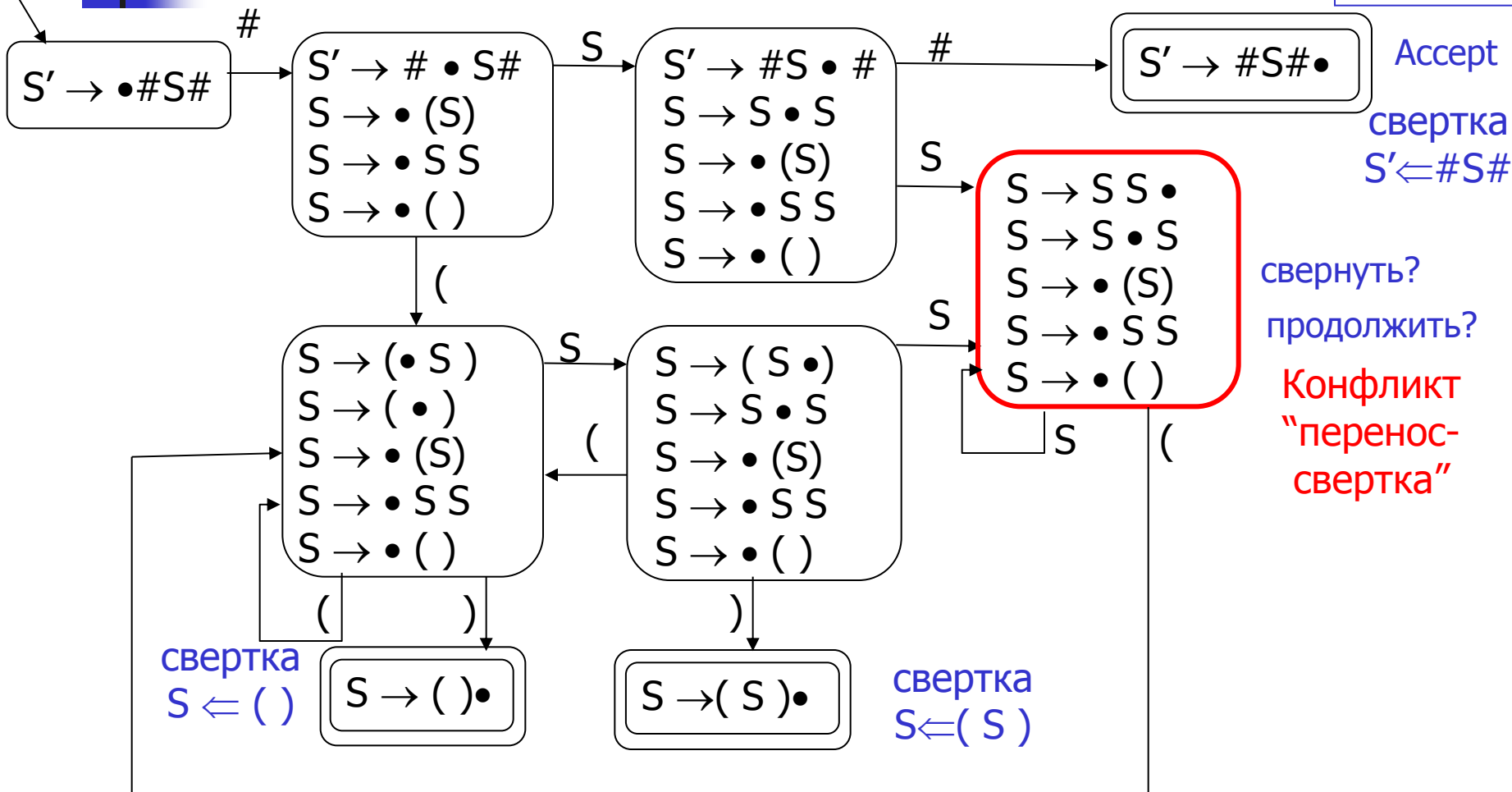
$S'$   
 $\# S \#$   
 $\# a S c \#$   
 $\# a a S c c \#$   
 $\# a a a S c c c \#$   
 $\# a a a b c c c \#$

$S' \leftarrow \#S\#$   
 $S \leftarrow aSc$   
 $S \leftarrow aSc$   
 $S \leftarrow aSc$   
 $S \leftarrow b$



# LR(0)-автомат разбора может иметь конфликты

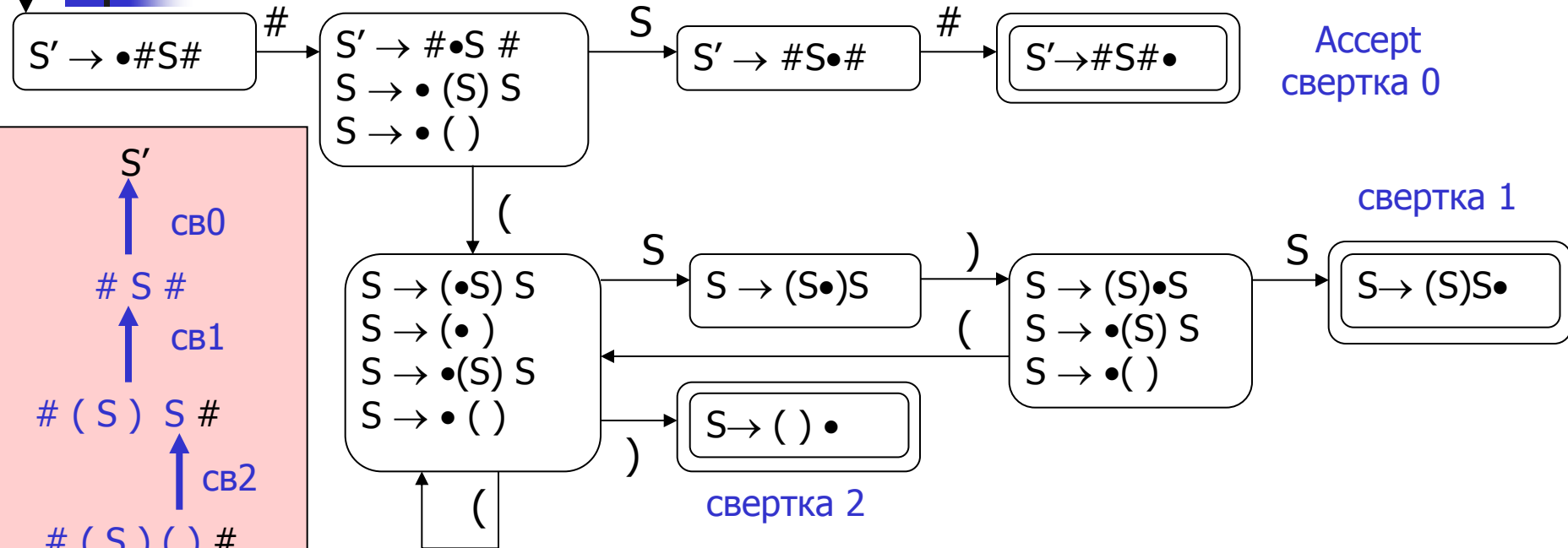
$S' \rightarrow \#S\#$   
 $S \rightarrow (S)$   
 $S \rightarrow S S$   
 $S \rightarrow ( )$



Наличие конфликта говорит о том, что грамматика не относится к классу LR(0)  
 А к какому классу? – мы не знаем, может быть, грамматика двусмысленна?

# Пример: LR(0) – анализатор

G: 0.  $S' \rightarrow \#S\#$   
 1.  $S \rightarrow (S)S$   
 2.  $S \rightarrow ( )$



$S' \Rightarrow_0 \#S\# \Rightarrow_1 \#(S)S\# \Rightarrow_2 \#(S)( )\# \Rightarrow_1 \#((S)S)( )\# \Rightarrow_2 \#((S)( ))( )\# \Rightarrow_2 \#((( ))( ))( )\#$  - **правый вывод**

Восстановление вывода цепочки:

$\# (( ( ) ) ( ) ) ( ) \#$

# LR(0) анализатор и синтаксические ошибки

Входная цепочка:

$\#a a [ [ a a ] a ] \#$

Accept

свертка1  
 $S' \leftarrow \#S\#$

свертка4  
 $S \leftarrow ab$

свертка2  
 $S \leftarrow aST$

свертка6  
 $T \leftarrow b$

свертка5  
 $T \leftarrow cT$

ошибка

$\#a a [ [ a a ] a ] \#$

$S' \rightarrow \bullet \#S\#$

$S' \rightarrow \# \bullet S\#$   
 $S \rightarrow \bullet aST$   
 $S \rightarrow \bullet [ S b ]$   
 $S \rightarrow \bullet a b$

$S' \rightarrow \#S \bullet \#$

$S' \rightarrow \#S\# \bullet$

1  $S' \rightarrow \#S\#$   
 2  $S \rightarrow aST$   
 3  $S \rightarrow [Sb]$   
 4  $S \rightarrow ab$   
 5  $T \rightarrow cT$   
 6  $T \rightarrow b$

$S \rightarrow [ \bullet S b ]$   
 $S \rightarrow \bullet aST$   
 $S \rightarrow \bullet [ S b ]$   
 $S \rightarrow \bullet a b$

$S \rightarrow a \bullet ST$   
 $S \rightarrow a \bullet b$   
 $S \rightarrow \bullet aST$   
 $S \rightarrow \bullet [ S b ]$   
 $S \rightarrow \bullet a b$

$S \rightarrow a b \bullet$

$S \rightarrow aS \bullet T$   
 $T \rightarrow \bullet cT$   
 $T \rightarrow \bullet b$

$S \rightarrow aST \bullet$

$T \rightarrow b \bullet$

$T \rightarrow c \bullet T$   
 $T \rightarrow \bullet cT$   
 $T \rightarrow \bullet b$

$T \rightarrow cT \bullet$

$S \rightarrow [ S \bullet b ]$

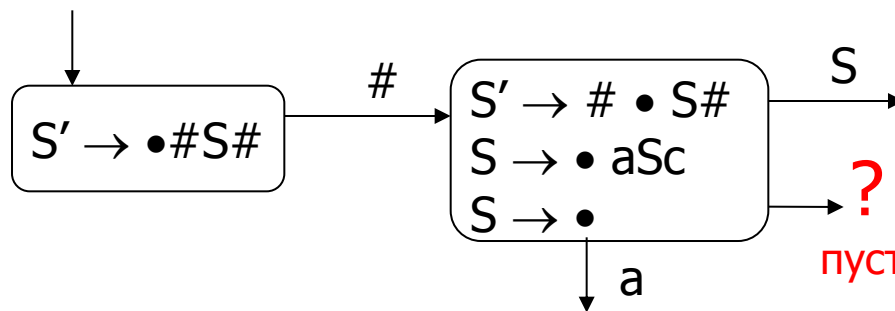
$S \rightarrow [ S b \bullet ]$

$S \rightarrow [ S b ] \bullet$

свертка3  
 $S \leftarrow [Sb]$

# LR(k)–грамматики: LR(0) недостаточно

0.  $S' \rightarrow \#S\#$
1.  $S \rightarrow aSc$
2.  $S \rightarrow \epsilon$



пустую цепочку к  $S$  свернуть?

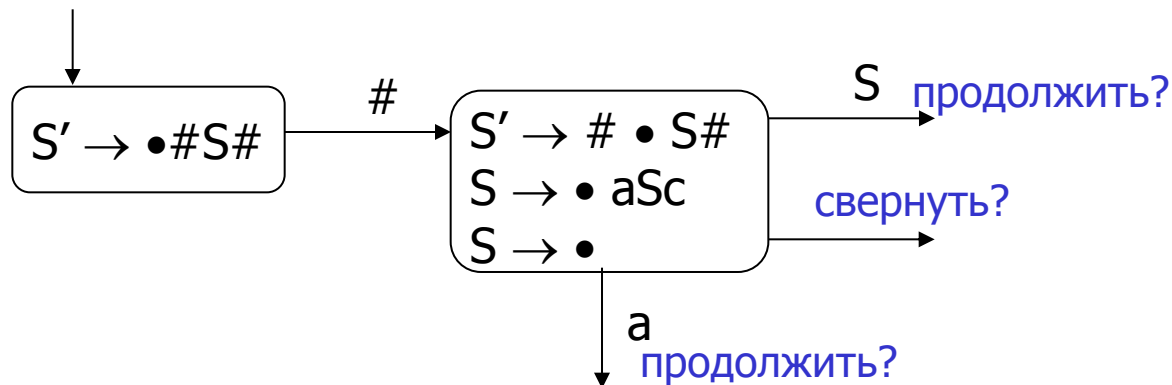
Сворачивать или продолжать?

- Эта грамматика отличается от предыдущей грамматики только одной продукцией: 2.  $S \rightarrow \epsilon$ . Начальное состояние LR(0)–анализатора - это характеристическое множество  $[S' \rightarrow \bullet \# S \#]$
- Следующее характеристическое множество  $[S' \rightarrow \# \bullet S \#, S \rightarrow \bullet a S c, S \rightarrow \bullet]$  отличается качественно: среди ситуаций есть терминальная ' $S \rightarrow \bullet$ ', но она не является единственной в множестве ситуаций
- Следовательно, здесь нельзя принять единственное решение о дальнейших действиях анализатора. Действительно, помеченная продукция  $S \rightarrow \bullet$  говорит о том, что мы достигли конца (пустой) связки, в то время как другие помеченные продукции говорят, что анализатор в данном состоянии анализа может находиться в середине других связок
- **НЕОДНОЗНАЧНОСТЬ, коллизия! Что делать??**

# Конфликт типа 'перенос/свертка'

0.  $S' \rightarrow \#S\#$
1.  $S \rightarrow aSc$
2.  $S \rightarrow \epsilon$

- После первого символа  $\#$  на входе распознавателя мы не можем однозначно принять решение о редукции - мы можем находиться не только в конце продукции  $S \rightarrow \epsilon$ , но и в начале продукции  $S \rightarrow aSc$  или же в середине продукции  $S' \rightarrow \#S\#$ . Именно это и говорит множество ситуаций  $[S' \rightarrow \# \bullet S\#, S \rightarrow \bullet aSc, S \rightarrow \bullet]$



Нет никакой информации, чтобы принять решение: свернуть ли "ничего", т.е. пустую строку, к  $S$

Конфликт: продолжать анализ (переносить очередной символ в стек), или сворачивать?

Без анализа следующего терминального символа ответить невозможно!

Нужно **ЗАГЛЯНУТЬ**, а не прочитать!

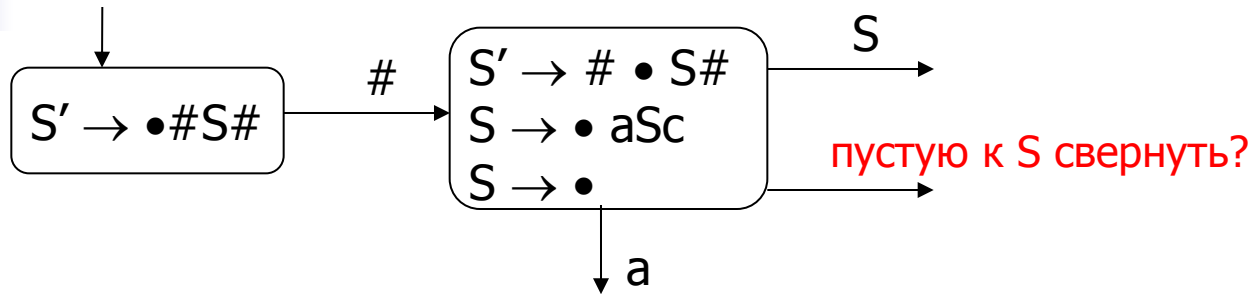
Грамматика  $G$  **не является**  $LR(0)$ -грамматикой.

*Очевидно, что любая грамматика с  $\epsilon$ -продукциями не относится к классу  $LR(0)$*



# LR(k)–анализатор для грамматики

0.  $S' \rightarrow \#S\#$
1.  $S \rightarrow aSc$
2.  $S \rightarrow \varepsilon$



**Конфликт! Необходимо учесть правый контекст** – следующие символы в анализируемой sentенциальной форме.

Именно это и делает LR(k)–анализатор при  $k > 0$ . Характеристические множества LR(k)–анализатора составляют ситуации вида ' $A \rightarrow \alpha \bullet \beta; \gamma$ ', где  $\gamma$  – цепочка длиной  $k$  из символов грамматики – правый контекст длиной  $k$ , в котором может находиться продукция  $A \rightarrow \alpha \beta$

В начальном состоянии LR(k)–анализатора предполагается, что любая цепочка порождаемого языка помещена в контекст из  $k$  последующих заключительных маркеров, например. вида ' $\perp$ '.

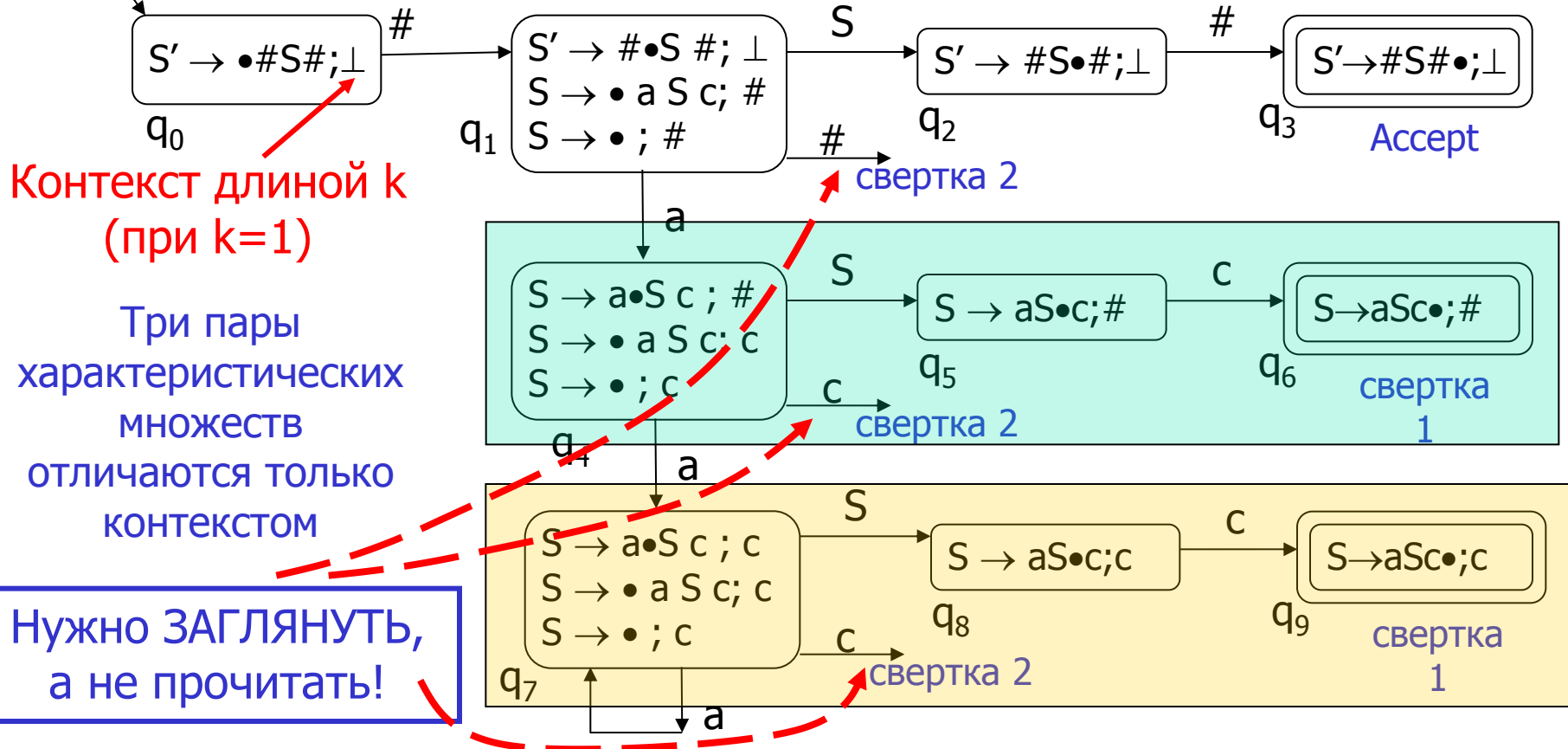
начальное  
состояние

$S' \rightarrow \bullet \# S \#; \perp \perp \perp \perp$

Начальный правый контекст длиной  $k$

# LR(1) анализатор для грамматики G

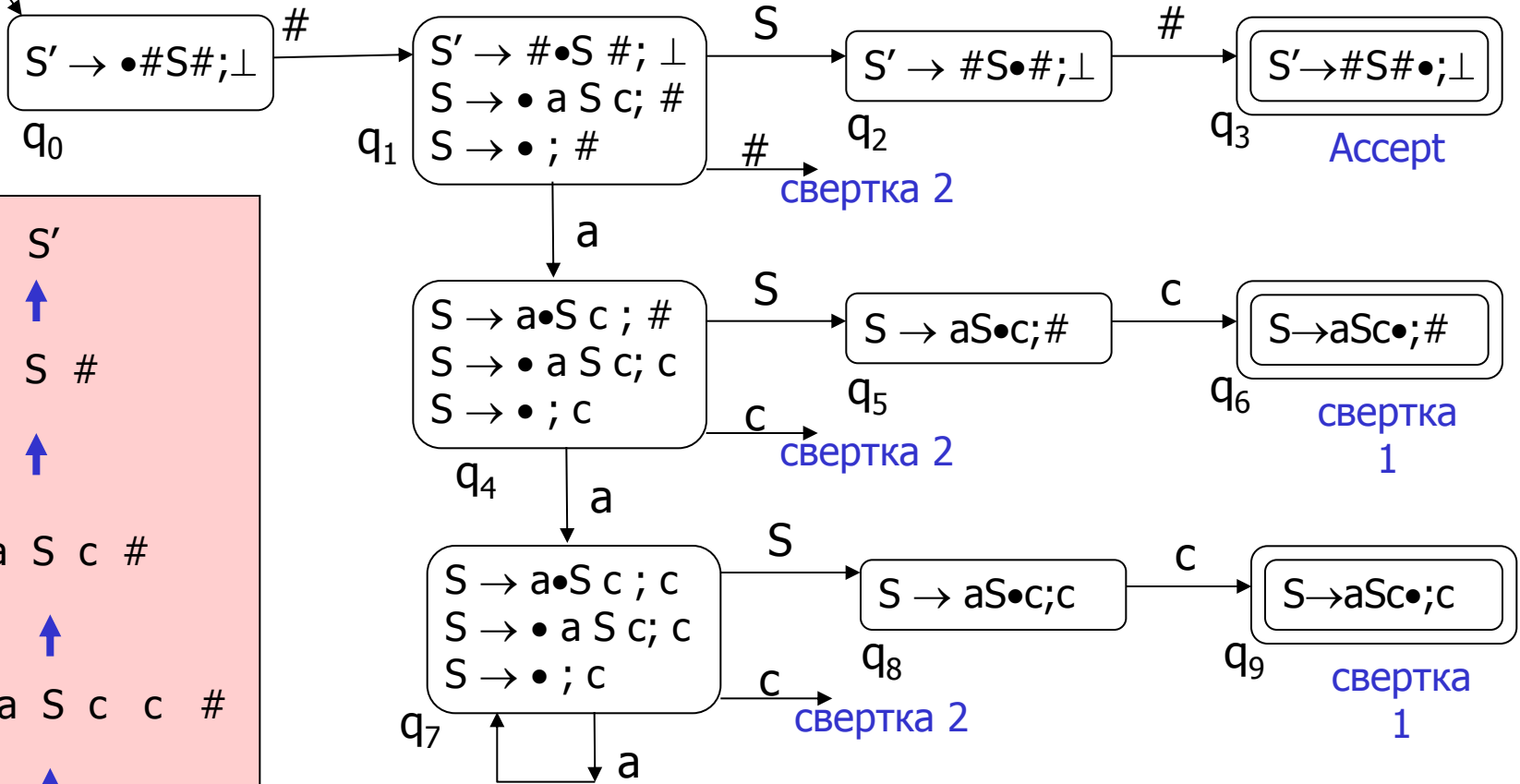
G: 0.  $S' \rightarrow \#S\#$   
 1.  $S \rightarrow aSc$   
 2.  $S \rightarrow \varepsilon$



Свертка 2 отличается от свертки 1: очередной символ только проверяется (lookahead). Свертка 1 – в состоянии  $q_3$  непосредственно сворачиваем, хотя можем и проверить контекст (в  $q_6$  это  $\#$ , в  $q_9$  – это  $c$ ) на ошибку!

# LR(1) анализатор для грамматики G

G: 0.  $S' \rightarrow \#S\#$   
 1.  $S \rightarrow aSc$   
 2.  $S \rightarrow \epsilon$

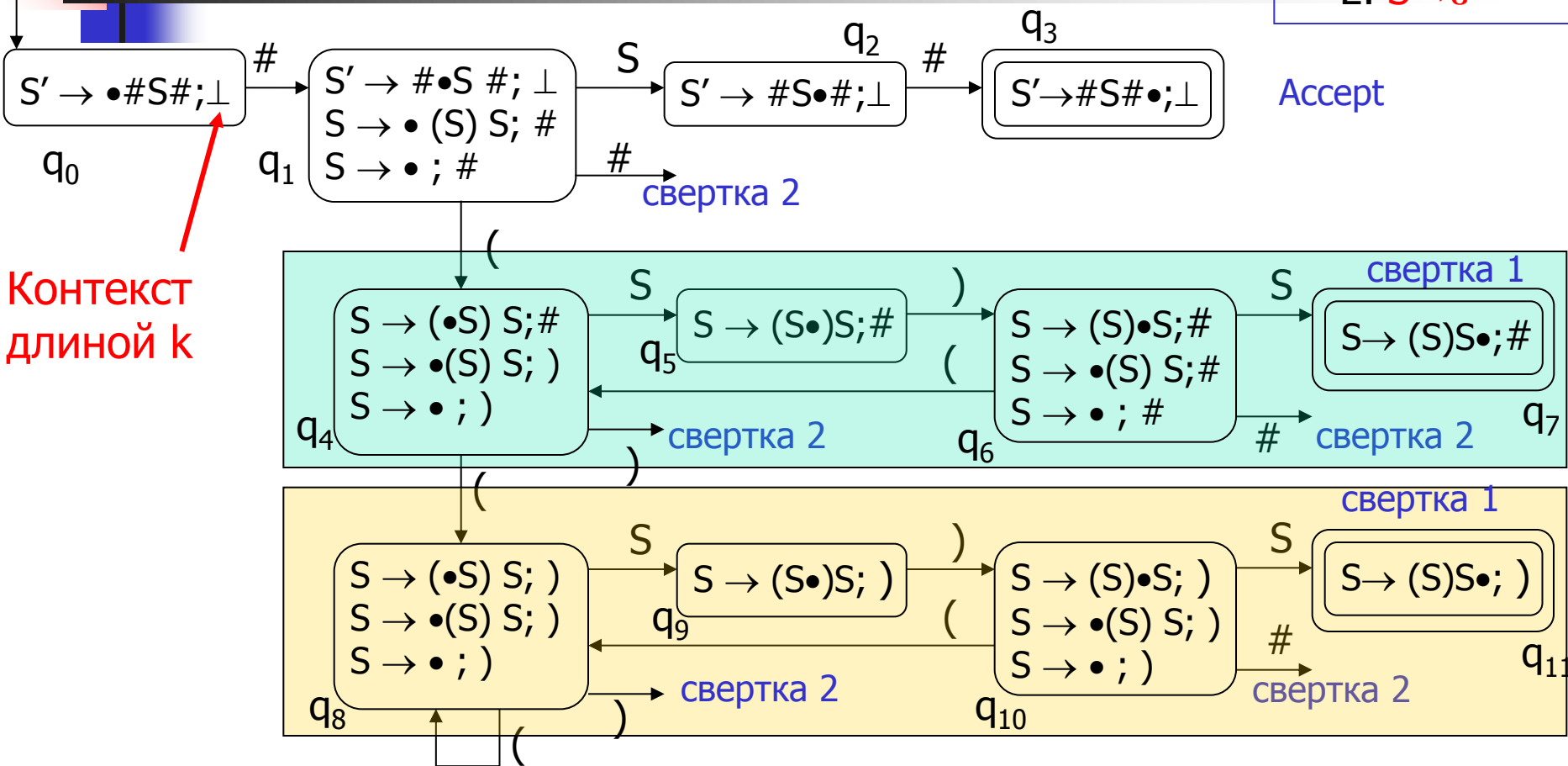


Разбор цепочки:

# a a a c c c #

# LR(1) – анализатор для грамматики скобочных выражений

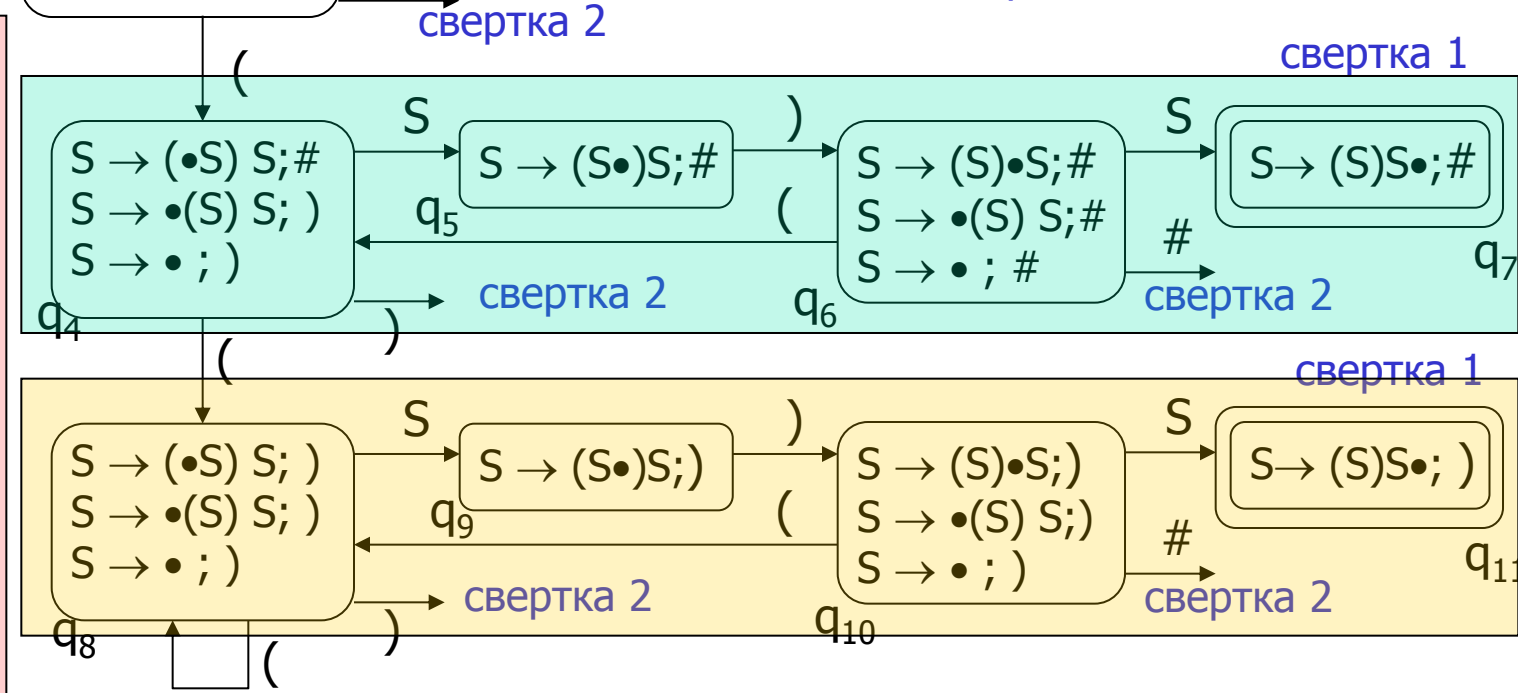
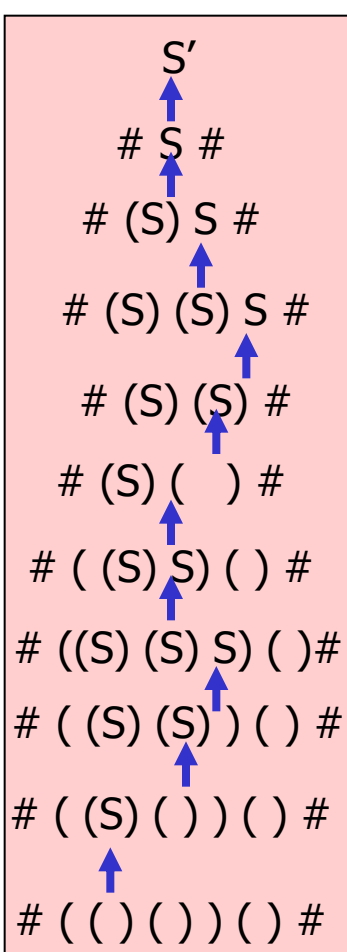
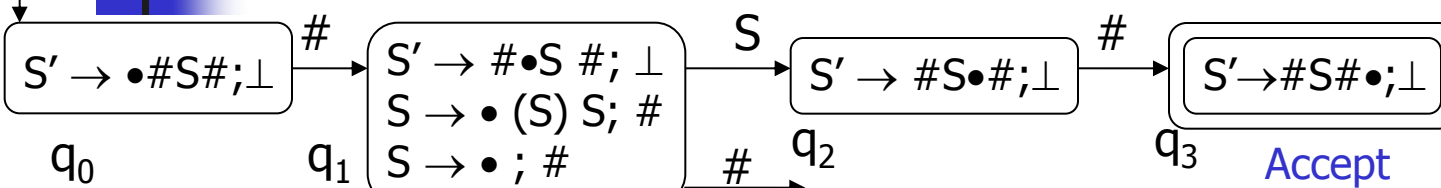
G: 0.  $S' \rightarrow \#S\#$   
1.  $S \rightarrow (S)S$   
2.  $S \rightarrow \varepsilon$



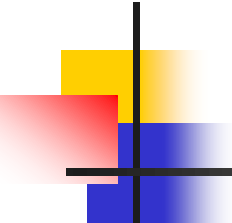
Состояния  $q_4, q_5, q_6$  и  $q_7$  отличаются от состояний  $q_8, q_9, q_{10}$  и  $q_{11}$  только контекстом в одной ситуации

# LR(1) – анализатор языка скобочных выражений

G: 0.  $S' \rightarrow \#S\#$   
 1.  $S \rightarrow (S)S$   
 2.  $S \rightarrow \varepsilon$

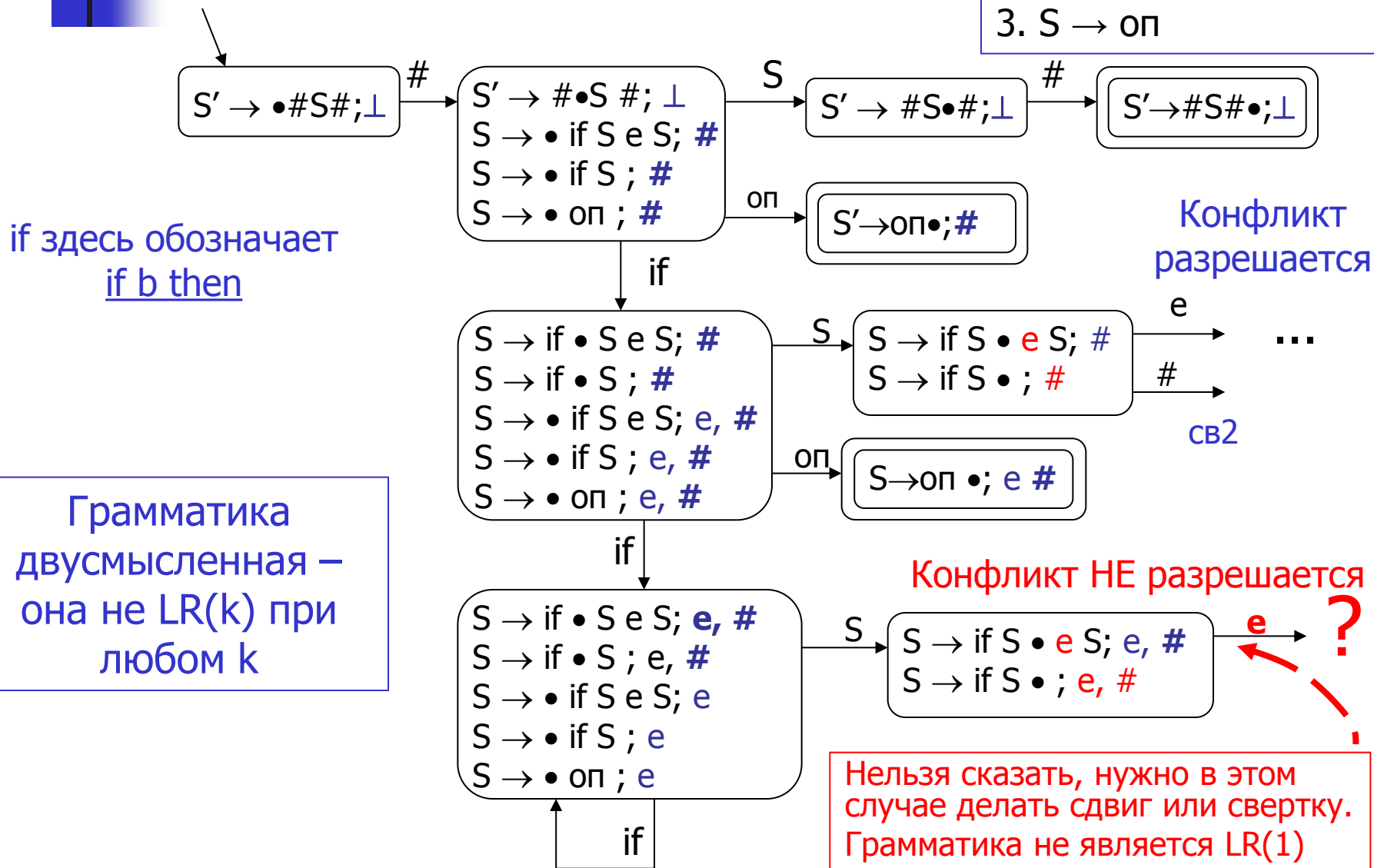


$S' \Rightarrow_0 \#S\# \Rightarrow_1 \#(S)S\# \Rightarrow_1 \#(S)(S)S\# \Rightarrow_2 \#(S)(S)\# \Rightarrow_2 \#(S) ( ) \#$   
 $\Rightarrow_1 \# ( (S) S ) ( ) \# \Rightarrow_1 \# ( (S) (S) S ) ( ) \# \Rightarrow_2 \# ((S) ( ) ) ( ) \# \Rightarrow_2$   
 $\#(( ) ( ) ) ( ) \#$  - **правый вывод**

- 
- Для двусмысленных грамматик никаким контекстом разрешить коллизии при свертках нельзя: некоторые цепочки имеют несколько различных синтаксических деревьев

# Грамматика условных операторов - двусмысленная

0.  $S' \rightarrow \#S\#$
1.  $S \rightarrow \text{if } b \text{ then } S \text{ else } S$
2.  $S \rightarrow \text{if } b \text{ then } S$
3.  $S \rightarrow \text{оп}$





## Соотношение LR(1) и LL(1) грамматик

---

- Чтобы грамматика была LR(1), необходимо распознавать вхождение правой части правила вывода, просмотрев все, что выведено из этой правой части и текущий символ входной цепочки. Это требование существенно менее строгое, чем требование для LL(1)-грамматики, когда необходимо определить применимое правило, видя только первый символ, выводимый из его правой части.
- Таким образом, класс LL(1)-грамматик является собственным подклассом класса LR(1)-грамматик





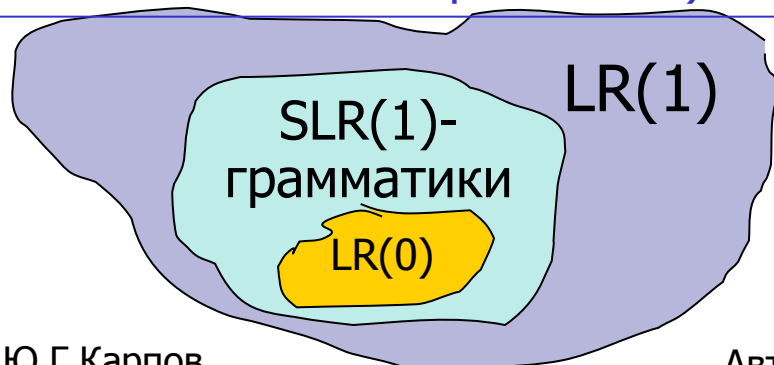
## Привлекательность LR(k)-анализа

- LR(k)-анализ - наиболее мощный из известных эффективный метод анализа без возвратов
- LR(1)-анализаторы могут быть построены для практически всех конструкций языков программирования
- Класс грамматик, которые могут быть проанализированы LR(1)-методом, строго включает класс грамматик, которые могут быть проанализированы нисходящими предсказывающими анализаторами типа LL(1)
- Существуют недвусмысленные грамматики, которые не являются LR(k)-грамматиками при любом k. Пример:  $S \rightarrow aSa | \varepsilon$
- Т.е. LR(k)-грамматики НЕ покрывают все недвусмысленные КС-грамматики

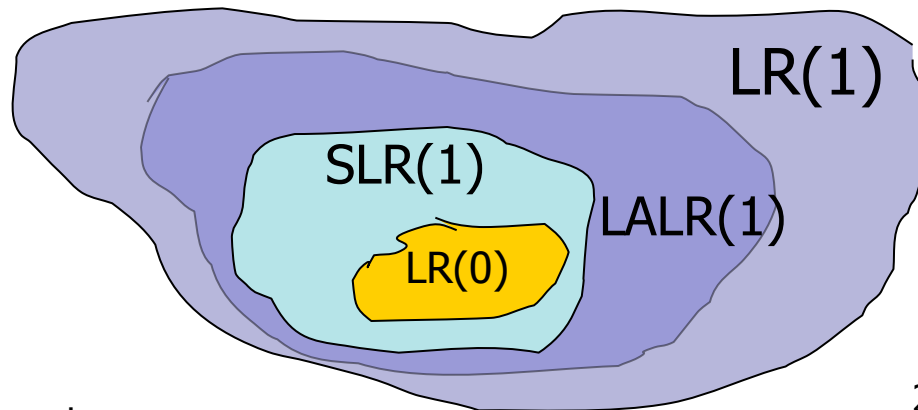
## SLR(1)-анализатор – это LR(0)-анализатор, дополненный простым правилом разрешения коллизий

- LR(k)-грамматики очень мощные, но хотя автомат LR(1) разбора прекрасно строится по ясному и понятному алгоритму, этот автомат чрезвычайно громоздкий из-за того, что даже при  $k=1$  ситуаций, отличающихся своими контекстами, очень много, а их подмножеств огромное число даже для простых грамматик
- Наибольшее распространение получили чуть упрощенные анализаторы, которые основаны на следующей идее:
  - для заданной грамматики строится LR(0)-анализатор (который имеет обычно небольшое число состояний)
  - **конфликты в этом LR(0)-анализаторе пытаемся разрешить локально**, в тех конкретных состояниях, в которых они возникли

**SLR(k)** – Simple LR(k)-анализатор  
(конфликт в LR(0) со сверткой  $N \leftarrow \alpha$   
пытаемся разрешить простейшим  
образом – с чего начинаются  
цепочки **после** нетерминала N)



**LALR(k)** – LookAhead LR(k)-анализатор  
(конфликты разрешаются несколько  
более сложным образом)



## SLR(k) и LALR(k) - анализаторы

- LR(k)–анализаторы весьма мощны: любой практически интересный язык может быть порожден LR(k) – грамматикой при некотором k
- Однако, при  $k > 0$  для любых практических грамматик LR(k)–анализаторы слишком громоздки. Даже LR(1)–анализатор для грамматики  $G_{6.3}$  с тремя продукциями содержит 10 состояний!
  - Причиной является то, что добавленный контекст сильно увеличивает число различных характеристических множеств. Однако этот контекст не всегда нужен. Фактически, он нужен только для разрешения конфликтов в тех характеристических множествах, которые содержат кроме терминальной помеченной продукции вида  $A \rightarrow a \bullet$  какие-либо другие помеченные продукции, что не позволяет принять однозначно решение о возможности редукции в соответствии с этой продукцией
- Можно ли учитывать контекст в LR(0)–анализаторе только непосредственно при возникновении подобной неоднозначности? Оказывается, это сделать можно, хотя и более грубо, чем в LR(k)–анализаторе. Но для многих случаев даже такой грубой информации оказывается достаточно для разрешения конфликта
  - Именно на этой идее построен SLR(k)-анализатор. Другой подход к разрешению неоднозначности использован в LALR(k)-анализаторах. Это наиболее оптимальные по качеству анализаторы: число состояний в них такое же, как в LR(0)–анализаторах, а учет контекста для разрешения конфликтов поводится значительно тоньше, чем в LR(0)–анализаторах (в которых контекст вообще не анализируется)

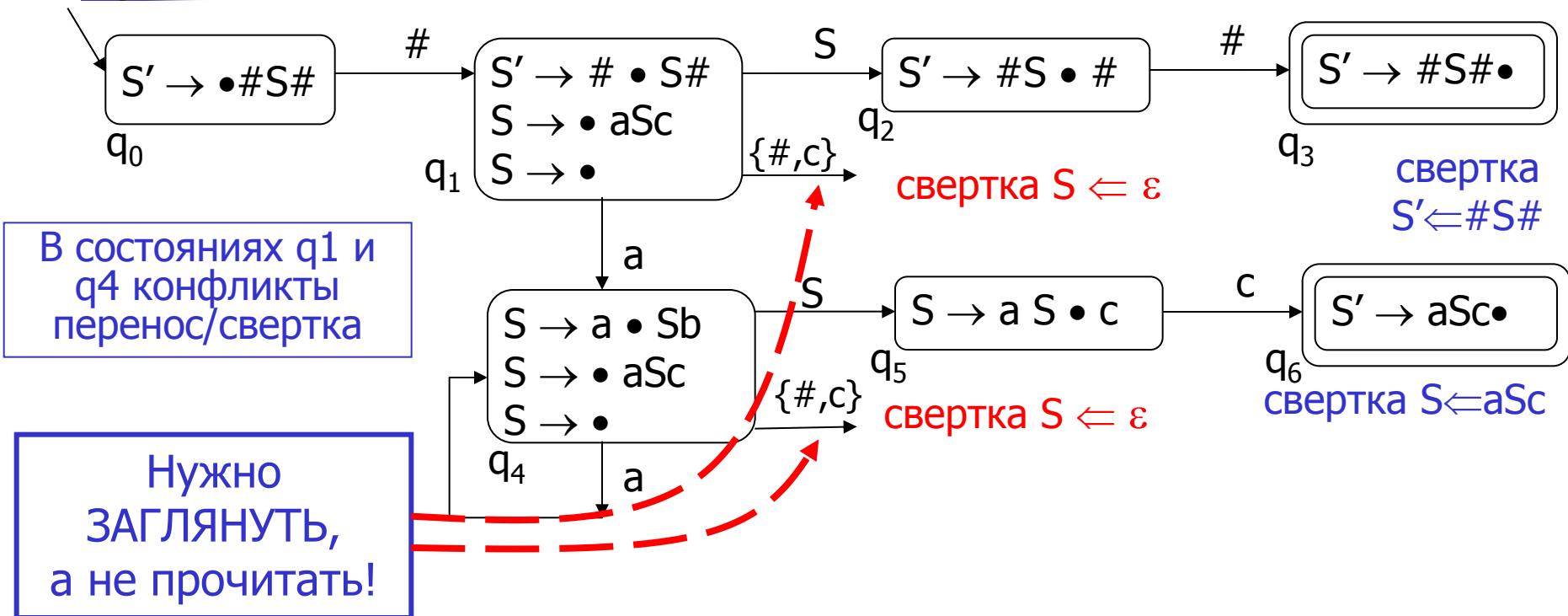


---

# SLR(1)–грамматики

# SLR(1) автомат разбора: функция Follow

$S' \rightarrow \#S\#$   
 $S \rightarrow aSc$   
 $S \rightarrow \varepsilon$

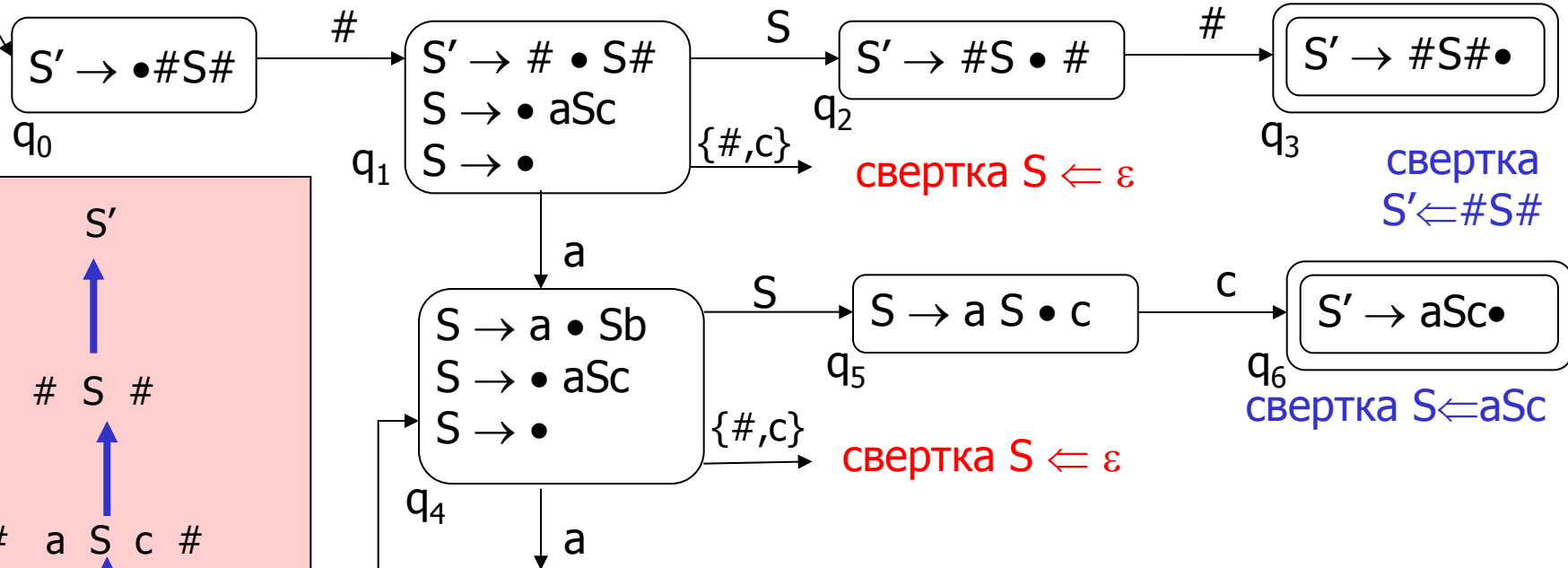


- Строим LR(0) – автомат разбора. Коллизии разрешаем, проверяя Follow(N)
- Для того, чтобы решить, нужно ли пустую цепочку свернуть к S, проверим, какие терминальные символы могут стоять в сенценциальных формах ПОСЛЕ символа S?, т.е. определяем множество символов Follow(S)

Функцию Follow(S) легко построить по грамматике:  $\text{Follow}(S) = \{\#, c\}$   
 Поскольку  $\{\#, c\} \cap \{a, S\} = \emptyset$ , то можем принять решение однозначно!

# SLR(1) автомат разбора

$S' \rightarrow \#S\#$   
 $S \rightarrow aSc$   
 $S \rightarrow \varepsilon$



$S'$   
 $\uparrow$   
 $\# S \#$   
 $\uparrow$   
 $\# a S c \#$   
 $\uparrow$   
 $\# a a S c c \#$   
 $\uparrow$   
 $\# a a a S c c c \#$   
 $\uparrow$   
 $\# a a a a c c c \#$

Свертка по lookAhead отличается от обычной свертки!!  
 Мы просто заглядываем вперед, символ не читаем

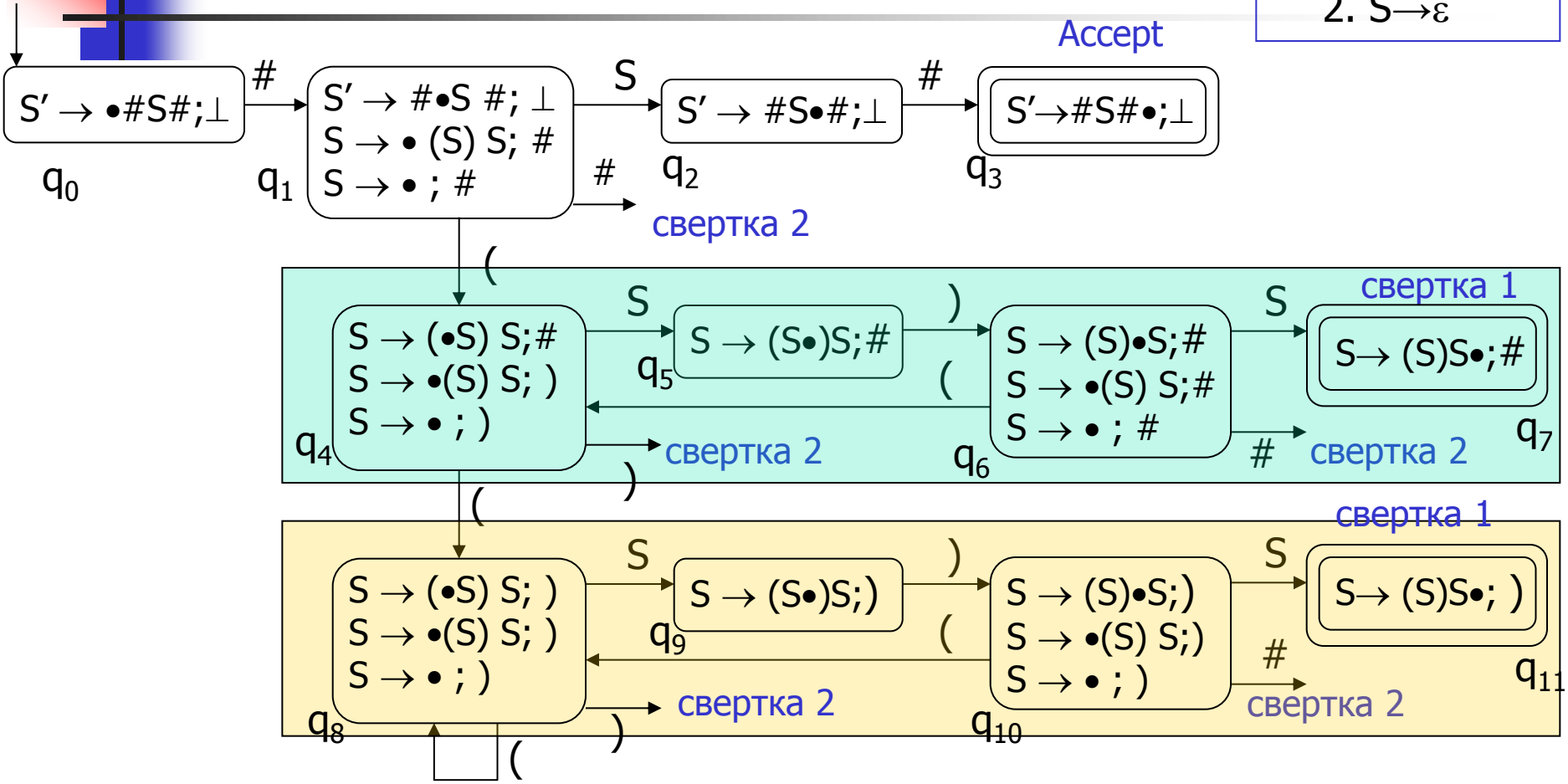
Разбор цепочки:

$\# a a a c c c \#$

Нужно ЗАГЛЯНУТЬ,  
 а не прочитать!

# LR(1) – анализатор языка вложенных скобок

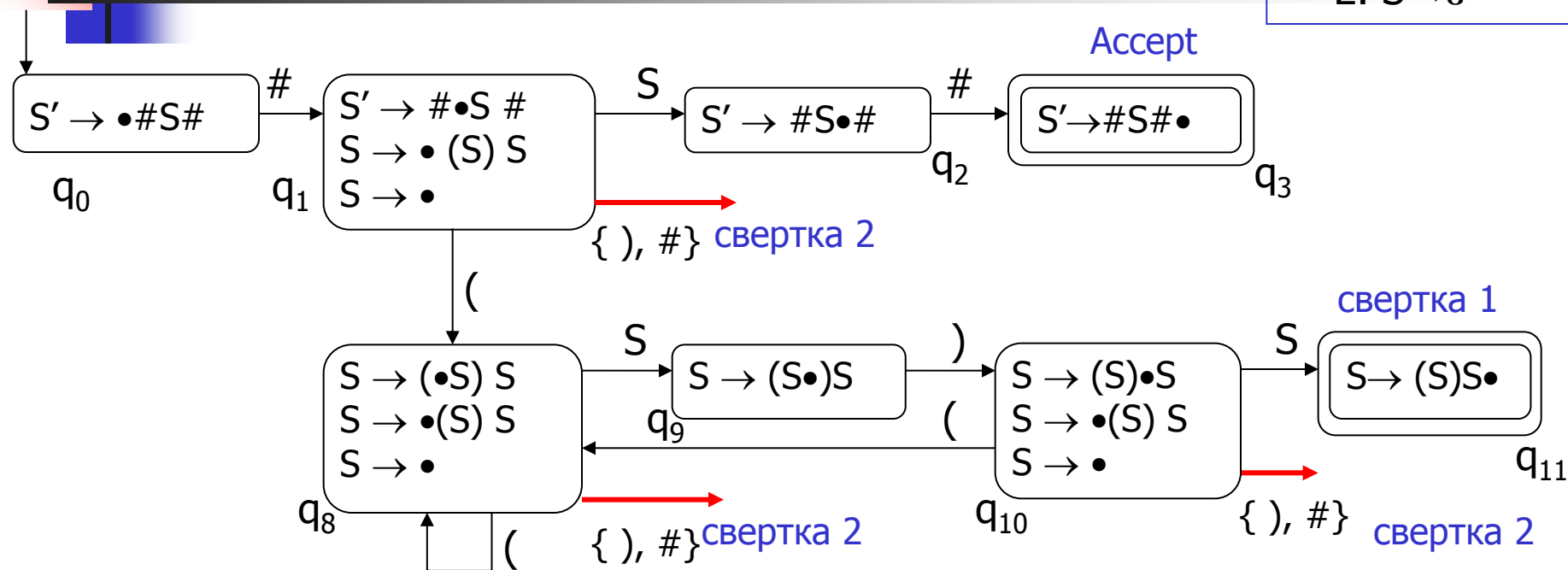
G: 0.  $S' \rightarrow \#S\#$   
 1.  $S \rightarrow (S)S$   
 2.  $S \rightarrow \varepsilon$



Две группы состояний отличаются только контекстами в ситуациях

# SLR(1) – анализатор языка вложенных скобок значительно проще, чем LR(1)

G: 0.  $S' \rightarrow \#S\#$   
 1.  $S \rightarrow (S)S$   
 2.  $S \rightarrow \varepsilon$



Строим LR(0)-анализатор

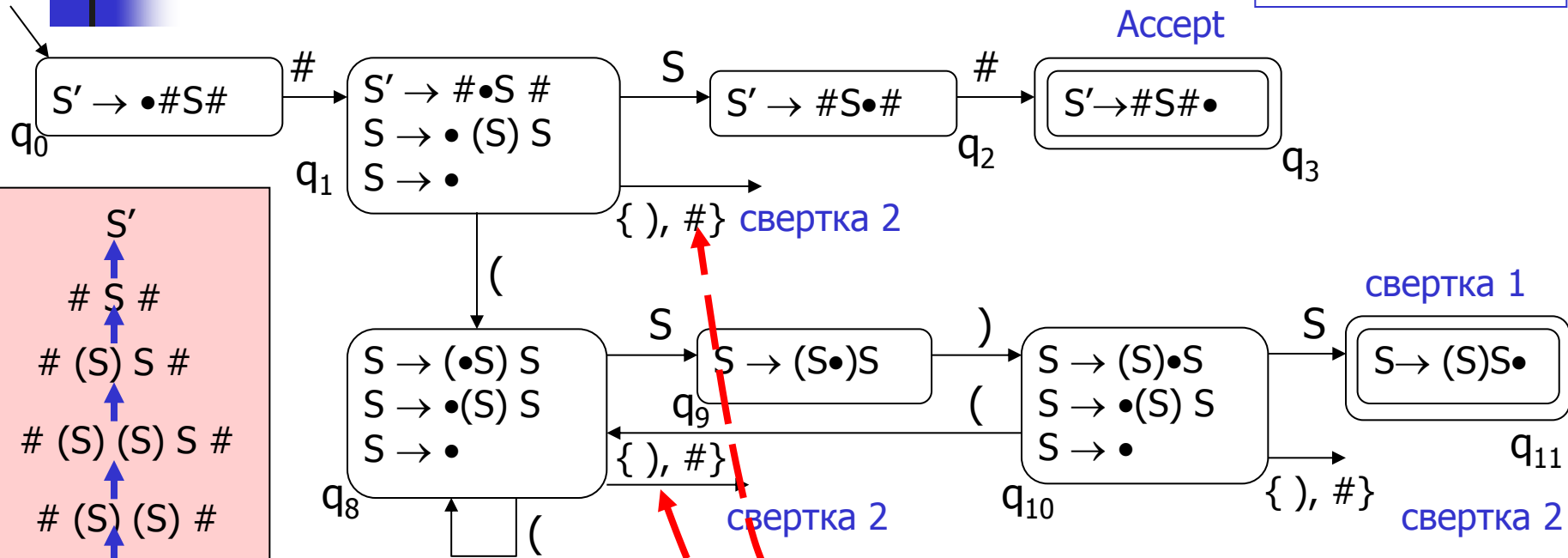
Какие символы могут стоять в сенсациональных формах ПОСЛЕ символа S?

$\text{Follow}(S) = \{ \#, ) \}$

Поскольку  $\{ \#, ) \} \cap \{ (, S \} = \emptyset$ , то можем принять решение однозначно!  
 Это SLR(1)-грамматика



G: 0.  $S' \rightarrow \#S\#$   
 1.  $S \rightarrow (S)S$   
 2.  $S \rightarrow \varepsilon$



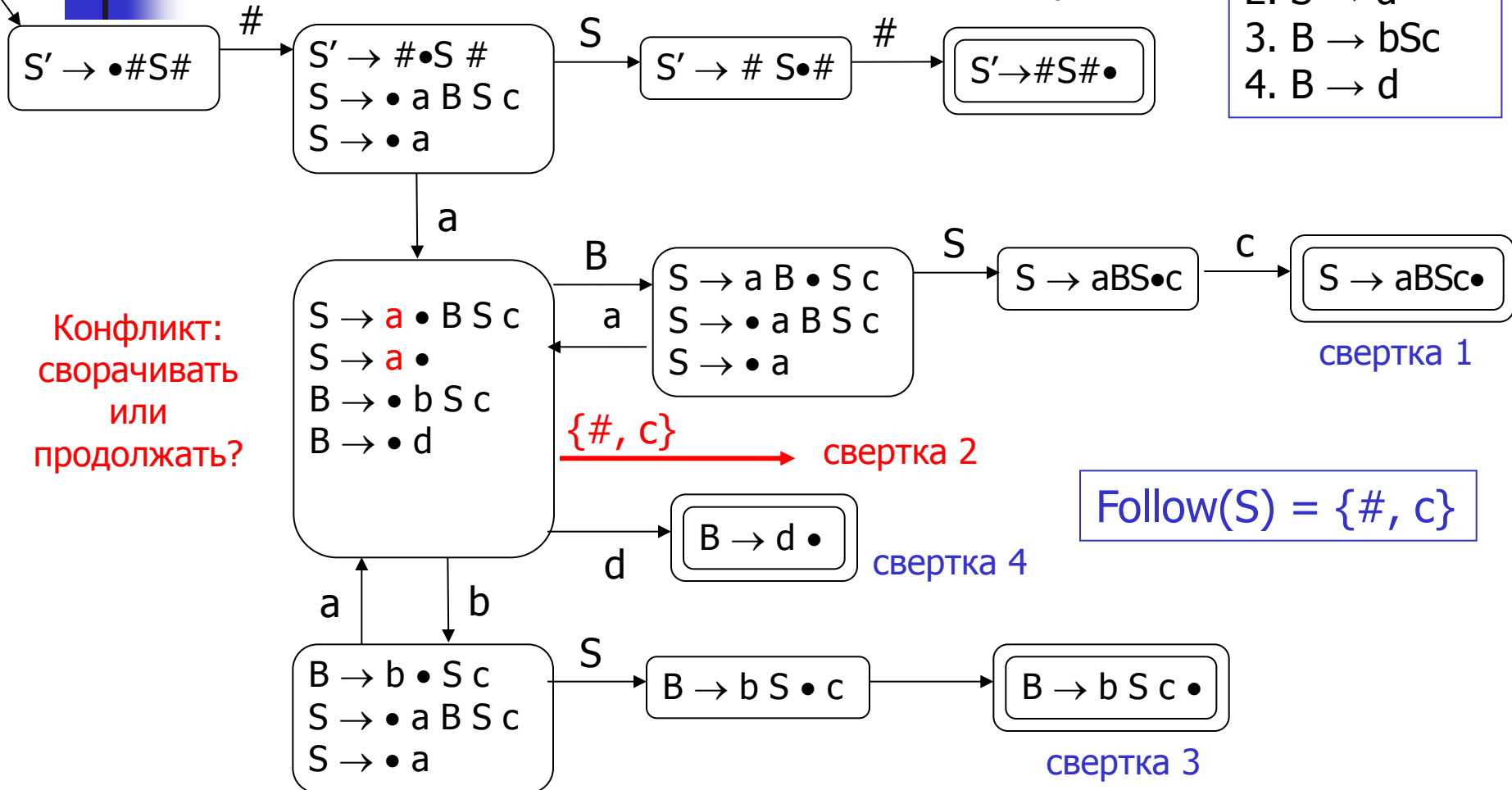
Нужно ЗАГЛЯНУТЬ,  
а не прочитать!

## Разбор цепочки:

# ( ( ) ( ) ) ( ) #

# Пример построения SLR(1)-анализатора

Конфликт:  
сворачивать  
или  
продолжать?



- Поскольку  $\text{Follow}(S) = \{\#, c\}$  не пересекается с другими альтернативными переходами, эта грамматика – SLR(1)



---

# LALR(1)–грамматики

## Разрешение конфликтов в SLR(k)-грамматиках

- В SLR(k)-грамматиках конфликты разрешаются довольно грубо: действительно, множество  $\text{Follow}_k(A)$  учитывает все возможные контексты, в которых может встретиться нетерминал в любых возможных выводах. В то же время конфликтующие продукции могут иметь значительно более узкий контекст, который можно найти, применив чуть более тонкий анализ.

Рассмотрим состояние  $s_1$  в LR(0)-анализаторе грамматики  $G$ , порождающей язык  $a^n c^n$ :

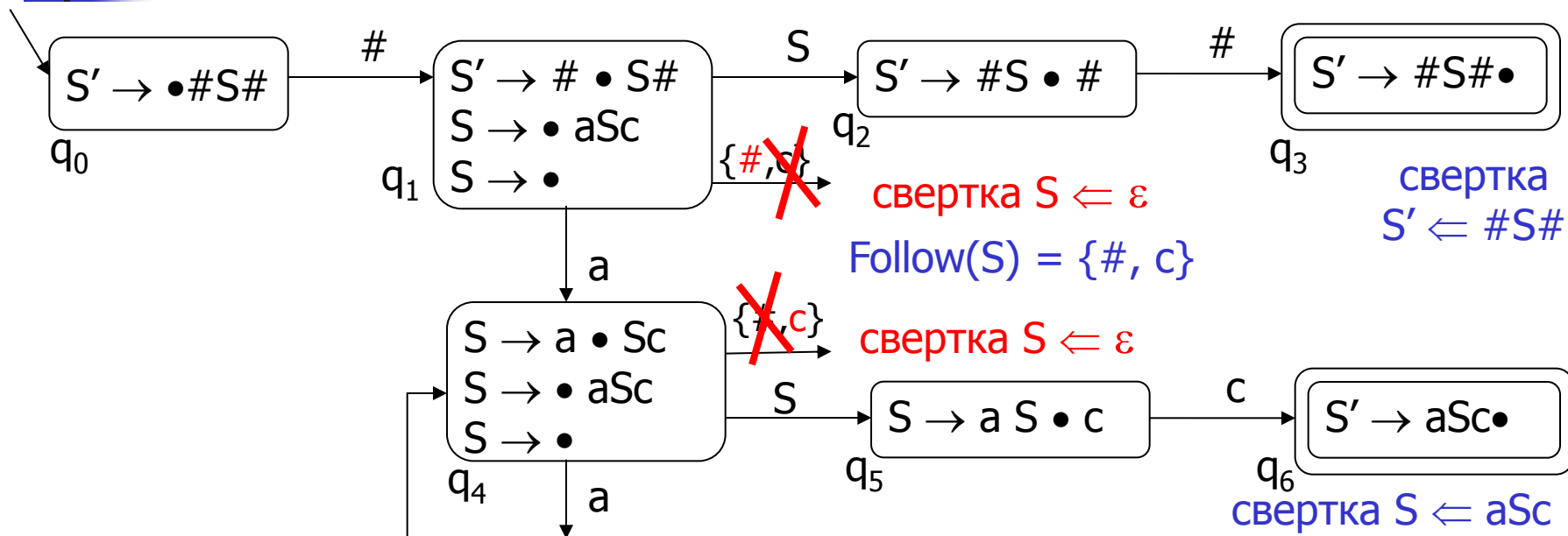
(q1):  $\left[ \begin{array}{ll} S' \rightarrow \# \cdot S \# & S \Rightarrow q_2; \\ S \rightarrow \cdot a S c & a \Rightarrow q_3 \\ S \rightarrow \cdot \varepsilon & \end{array} \right] \quad \{?\} - \text{свертка?}$

$G:: S' \rightarrow \# S \#$
$S \rightarrow a S c$
$S \rightarrow \varepsilon$

- Продукция  $S \rightarrow \cdot \varepsilon$  появилась в характеристическом множестве  $q_1$  потому, что мы могли ожидать начала связки для продукций, порожденных из  $S$
- Но сам этот нетерминал  $S$  появился из-за того, что в характеристическом множестве была помеченная продукция  $S' \rightarrow \# \cdot S \#$  с меткой, стоящей **перед** этим нетерминалом. Но эта же продукция определяет более узкий контекст, в котором мы можем ожидать появление  $S$  – это, конечно, только символ  $\#$ . Аналогично можно сузить контекст в состоянии  $q_3$

# SLR(1) автомат разбора: функция Follow

$S' \rightarrow \#S\#$   
 $S \rightarrow aSc$   
 $S \rightarrow \varepsilon$



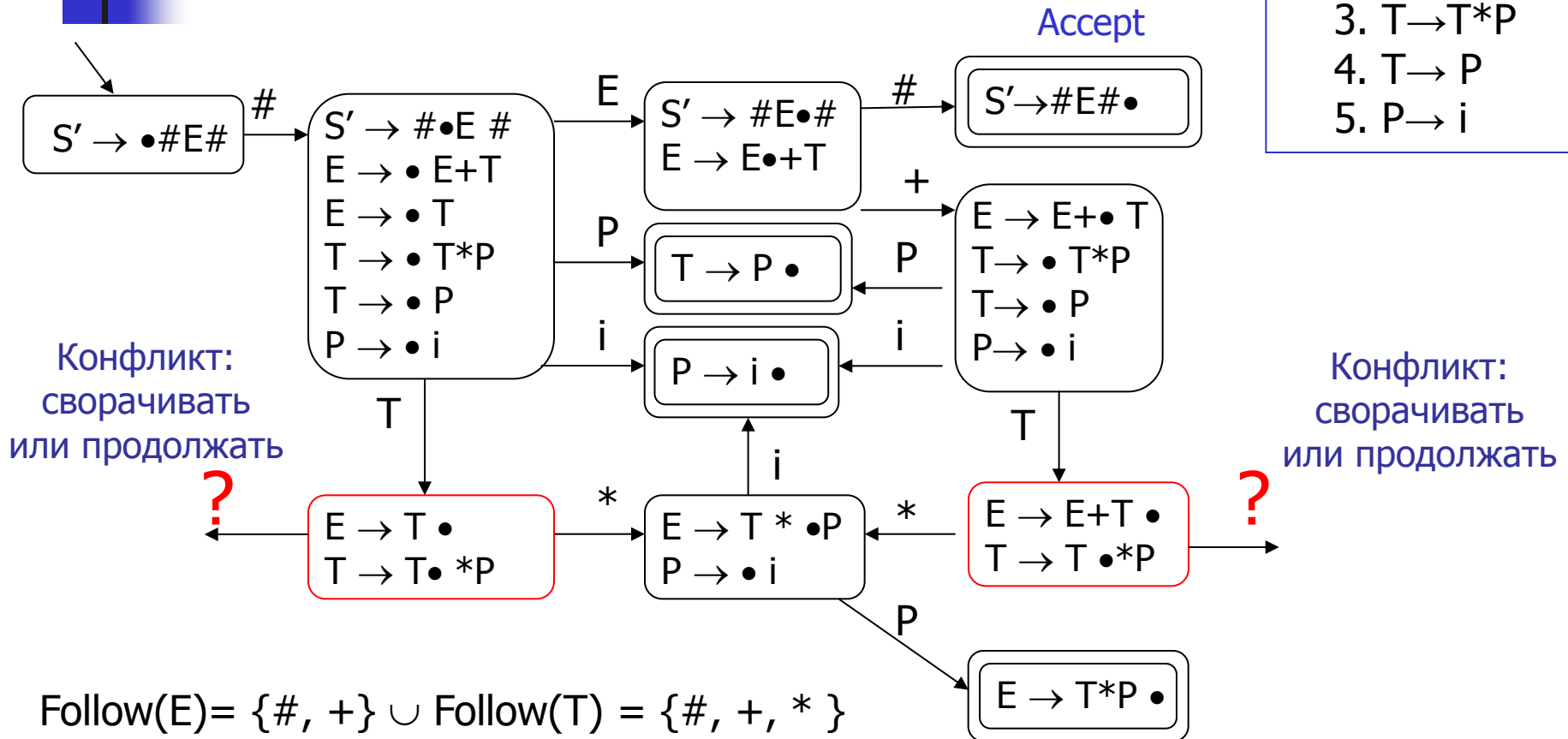
- Контекст в SLR(1) определяем функцией Follow(S), т.е. учитываем BCE, что может стоять за некоторым нетерминалом

**Реальные контексты более узки!**

В этой грамматике нам такое сужение возможных контекстов не нужно, все разрешается и с помощью Follow, но в общем случае этот подход расширяет возможности анализатора

# SLR(1) – анализатор недостаточен

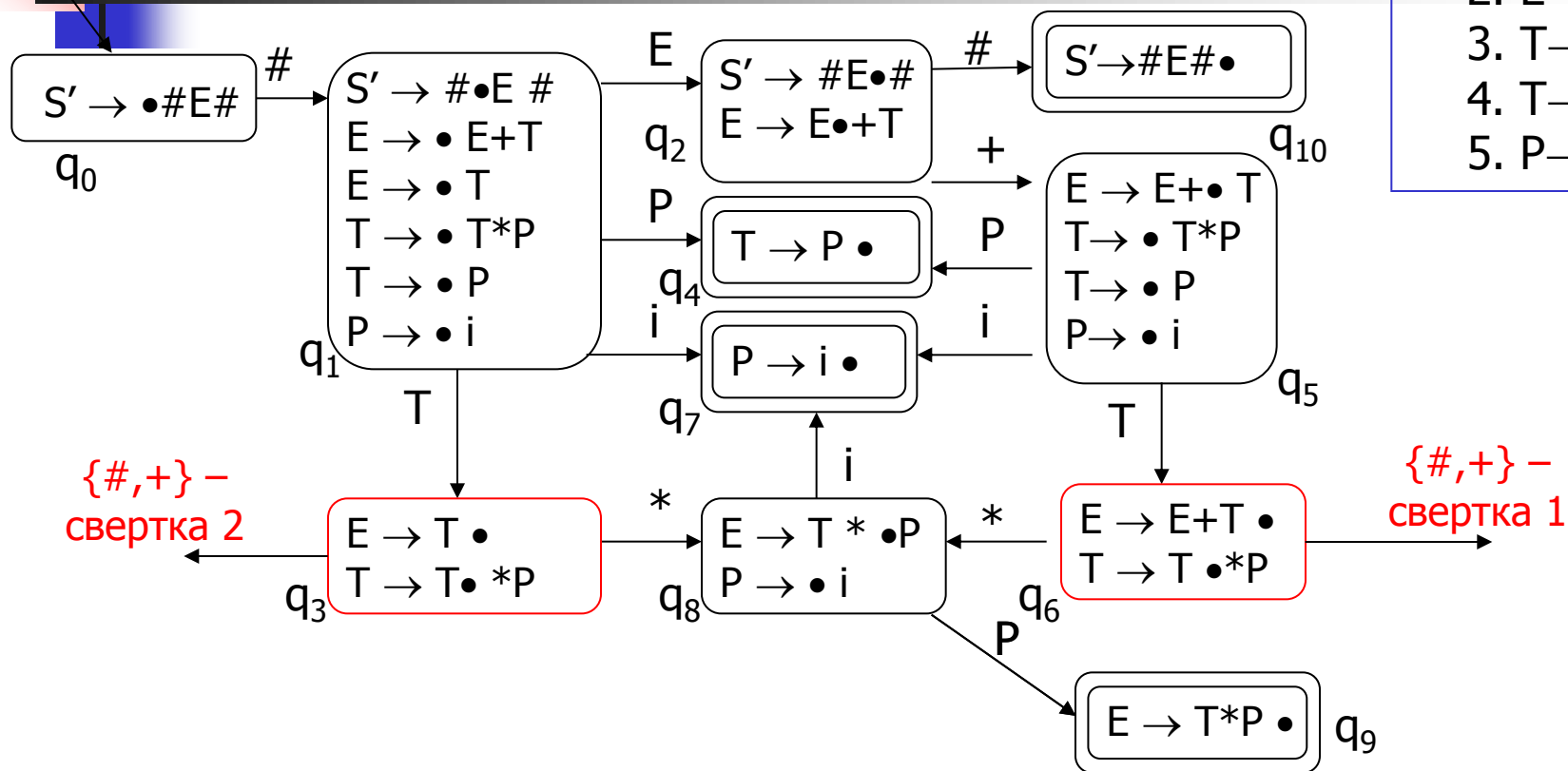
G: 0.  $S' \rightarrow \#E\#$   
 1.  $E \rightarrow E+T$   
 2.  $E \rightarrow T$   
 3.  $T \rightarrow T*P$   
 4.  $T \rightarrow P$   
 5.  $P \rightarrow i$



Конфликт не разрешается, поскольку контексты пересекаются – символ  $*$  входит в контекст и очередной символ в строке!

Можно провести более тонкий анализ контекста

# LALR(1) - анализатор



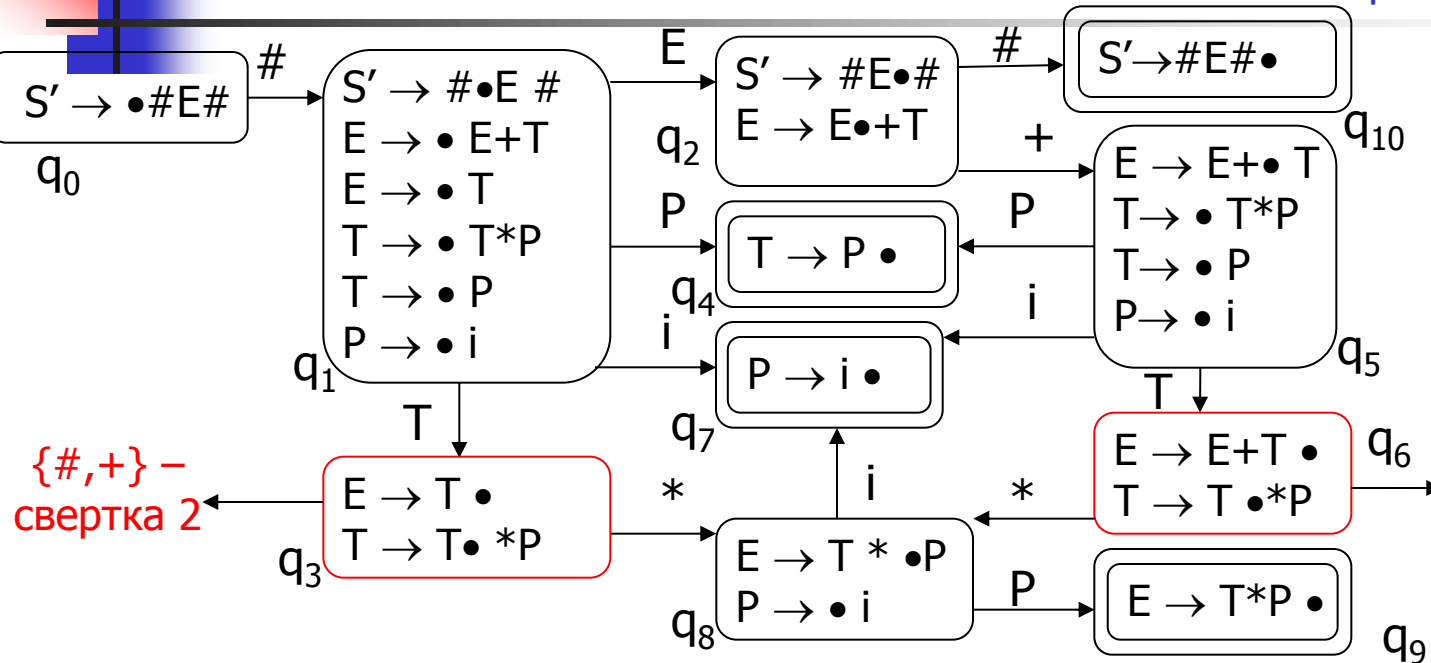
В  $q_3$  ситуация  $E \rightarrow T \bullet$  появилась из ситуации  $E \rightarrow \bullet T$  в состоянии  $q_1$ , а она – из ситуаций  $E \rightarrow \bullet E+T$  и  $S' \rightarrow \# \bullet E \#$ . В этих ситуациях после  $E$  стоят  $\#$  и  $+$ . Поэтому контекст, по которому в  $q_8$  нужно принимать решение о свертке по правилу  $E \rightarrow T \bullet$  состоит из пары  $\{\#, +\}$  – те символы, которые стоят после  $E$  в этих правилах.

Этот контекст не пересекается с  $\{*\}$ , поэтому решение: при следующих символах  $\{\#, +\}$  – свертка, при  $\{*\}$  – продолжение анализа, перенос в стек. Такой же анализ и для  $q_6$ .

# Синтаксический анализ со стеком

Accept

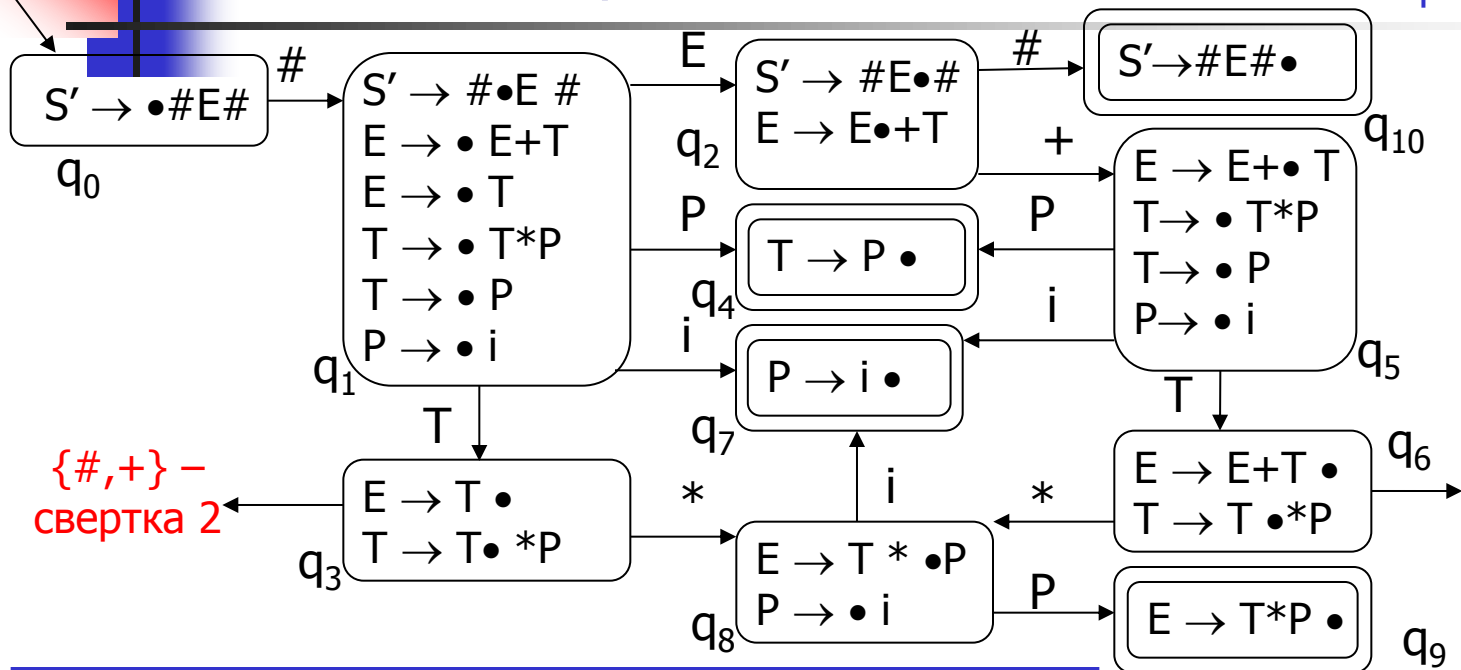
- G: 0.  $S' \rightarrow \#E\#$   
 1.  $E \rightarrow E+T$   
 2.  $E \rightarrow T$   
 3.  $T \rightarrow T*P$   
 4.  $T \rightarrow P$   
 5.  $P \rightarrow i$



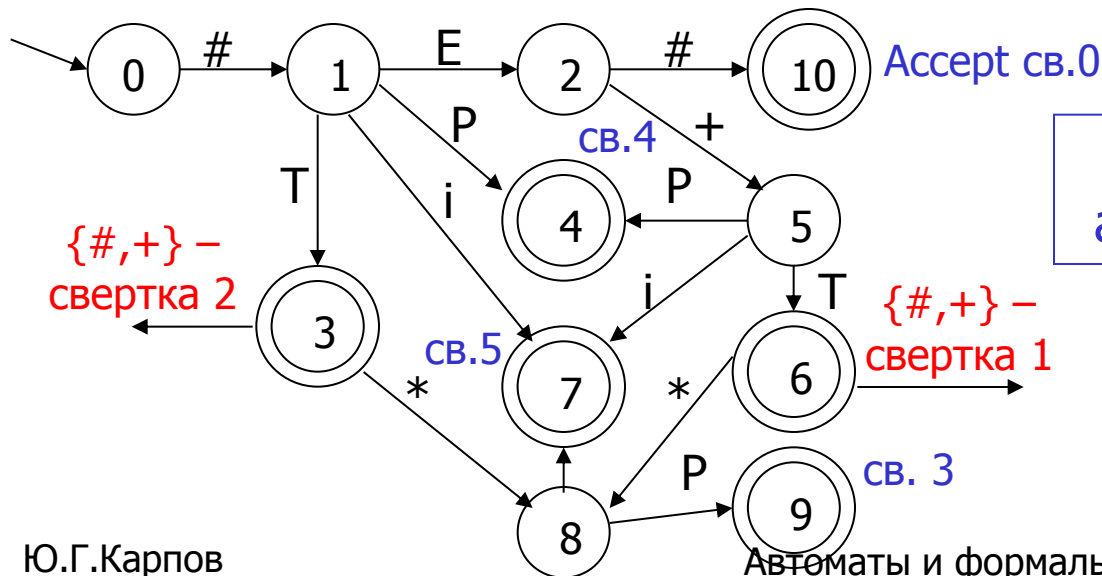
Магазин СИМВОЛ-СОСТОЯНИЕ	Что нужно свернуть	Остаток входа	Действие
$\perp q_0$		$\# i + i * i + i * i \#$	сдвиг
$\perp q_0 \# q_1$		$i + i * i + i * i \#$	сдвиг
$\perp q_0 \# q_1 i q_7$	$i$	$+ i * i + i * i \#$	$P \leftarrow i$ по правилу 5
$\perp q_0 \# q_1 P q_4$	$P$	$+ i * i + i * i \#$	$T \leftarrow P$ по правилу 4
$\perp q_0 \# q_1 T q_3$	$T$	$+ i * i + i * i \#$	$E \leftarrow T$ по правилу 2



# Таблица "переход/действия"



- G: 0.  $S' \rightarrow \# E \#$   
 1.  $E \rightarrow E + T$   
 2.  $E \rightarrow T$   
 3.  $T \rightarrow T * P$   
 4.  $T \rightarrow P$   
 5.  $P \rightarrow i$



Классическая грамматика арифметических выражений



## Идеи, используемые при построении SLR(k) и LALR(k) - анализаторов

- Общая идея при построении обоих этих классов анализаторов проста: для грамматики строится LR(0)– анализатор, после чего при наличии конфликтов их пытаются разрешить непосредственно в месте их возникновения учетом контекста длиной k символов
- SLR(k) и LALR(k) синтаксические анализаторы различаются только методом выявления возможности применимости редукции для терминальной ситуации в характеристических множествах
- SLR(k)- и LALR(k)- грамматиками называются такие грамматики, для которых соответствующие синтаксические анализаторы дают однозначное решение вопроса о выполнении редукции для каждой терминальной помеченной продукции.



## Представление автомата - распознавателя

В алгоритмах синтаксического анализа для LR-грамматик автомат-распознаватель представляется в виде таблицы переходов и выходов в следующей форме:

<i>Состояния</i>	<i>Терминалы</i>	<i>Нетерминалы</i>
0 1 2 ...	<i>Действия</i>	<i>Переходы</i>

Таблица состоит из строк, помеченных состояниями автомата, и столбцов, помеченных терминалами (включая символ конца строки  $\perp$ ) и нетерминалами грамматики.



## Правила заполнения таблицы

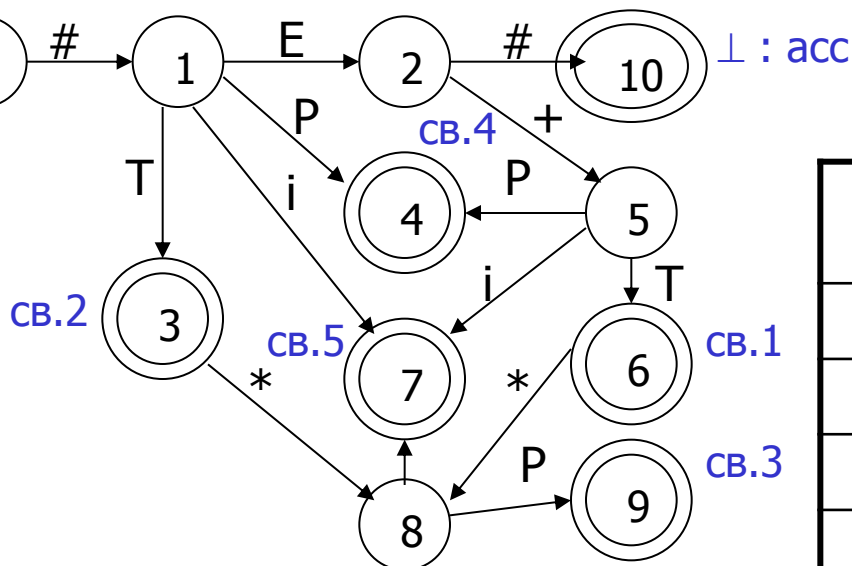
Таблица заполняется следующим образом:

- Если характеристическое множество, соответствующее состоянию  $q_i$ , включает помеченную продукцию  $A \rightarrow a \bullet a\beta$ , где  $a$  – терминальный символ, и в распознающем автомате из  $q_i$  под воздействием  $a$  есть переход в  $q_j$ , то определить *Действия*  $[i, a] = \text{"сдвиг } j\text{"}$ .
- Если характеристическое множество, соответствующее состоянию  $q_i$ , включает помеченную продукцию  $j: A \rightarrow a \bullet$ , то для всех  $a \in \text{Next}(A \rightarrow a)$  определить *Действия*  $[i, a] = \text{"редукция } j: A \leftarrow a\text{"}$ . Здесь  $\text{Next}(A)$  обозначает правый контекст этой продукции в данном состоянии.
- Если характеристическое множество, соответствующее состоянию  $q_i$ , включает помеченную продукцию  $j: S' \rightarrow S \bullet \perp$ , то определить *Действия*  $[i, \perp] = \text{"Принять цепочку (Accept)"}\text{"}$ .
- Если из состояния  $q_i$  под воздействием нетерминала  $A$  есть переход в состояние  $q_j$ , то определить *Переход*  $[i, A] = \text{"goto } j\text{"}$ .

# LALR(1)-анализатор для грамматики арифметических выражений

G: 0.  $S' \rightarrow \#E\#$   
 1.  $E \rightarrow E+T$   
 2.  $E \rightarrow T$   
 3.  $T \rightarrow T*P$   
 4.  $T \rightarrow P$   
 5.  $P \rightarrow i$

si = shift i  
 rj = reduce j



- Таблица строится по графу переходов
- si означает shift (сдвиг) по строке с переходом в состояние qi
- rj означает reduce (свертка) по правилу j
- i означает переход в состояние qi

N сост	Actions					Go to		
	i	+	*	#	⊥	E	T	P
0				s1				
1	s7					2	3	4
2		s5		s10				
3		r2	s8	r2				
4		r4	r4	r4				
5	s7						6	4
6		r1	s7	r1				
7		r5	r5	r5				
8	s7							9
9		r3	r3	r3				
10					acc			



## Анализ таблицы

- Таблица содержит для каждого состояния распознавателя его реакцию на очередной входной символ сентенциальной формы. Эти символы могут быть как терминальными, так и нетерминальными.
- Символы  $s_j$  на пересечении строки  $i$  и столбца  $a$  означают "shift  $j$ ", т.е. сдвиг по входной цепочке с переходом в состояние  $j$ . В стек записывается терминал  $a$  и номер очередного состояния  $j$ .
- Символы  $r_j$  на пересечении строки  $i$  и столбца  $a$  означают "reduce  $j$ ", т.е. необходимость редукции продукции  $j$  грамматики.
- Символы  $j$  на пересечении строки  $i$  и столбца, помеченного нетерминалом  $A$ , означают Переход  $[i, A] = \text{"goto } j\text{"}$ , где  $j$  - номер состояния, в которое LR-автомат переходит из состояния  $i$  под воздействием  $A$ .
- При необходимости редукции  $A \leftarrow a$  на очередном шаге разбора из стека выталкивается  $2*|a|$  символов (т.е. путь в распознающем автомате, соответствующий правой части продукции). Пусть  $k$  – номер состояния, появившийся в верхушке стека после выталкивания этого пути. В стек заносится символ  $A$  и номер состояния *Переход*  $[k, A]$ .



---

## Обработка ошибок в LR-анализаторе

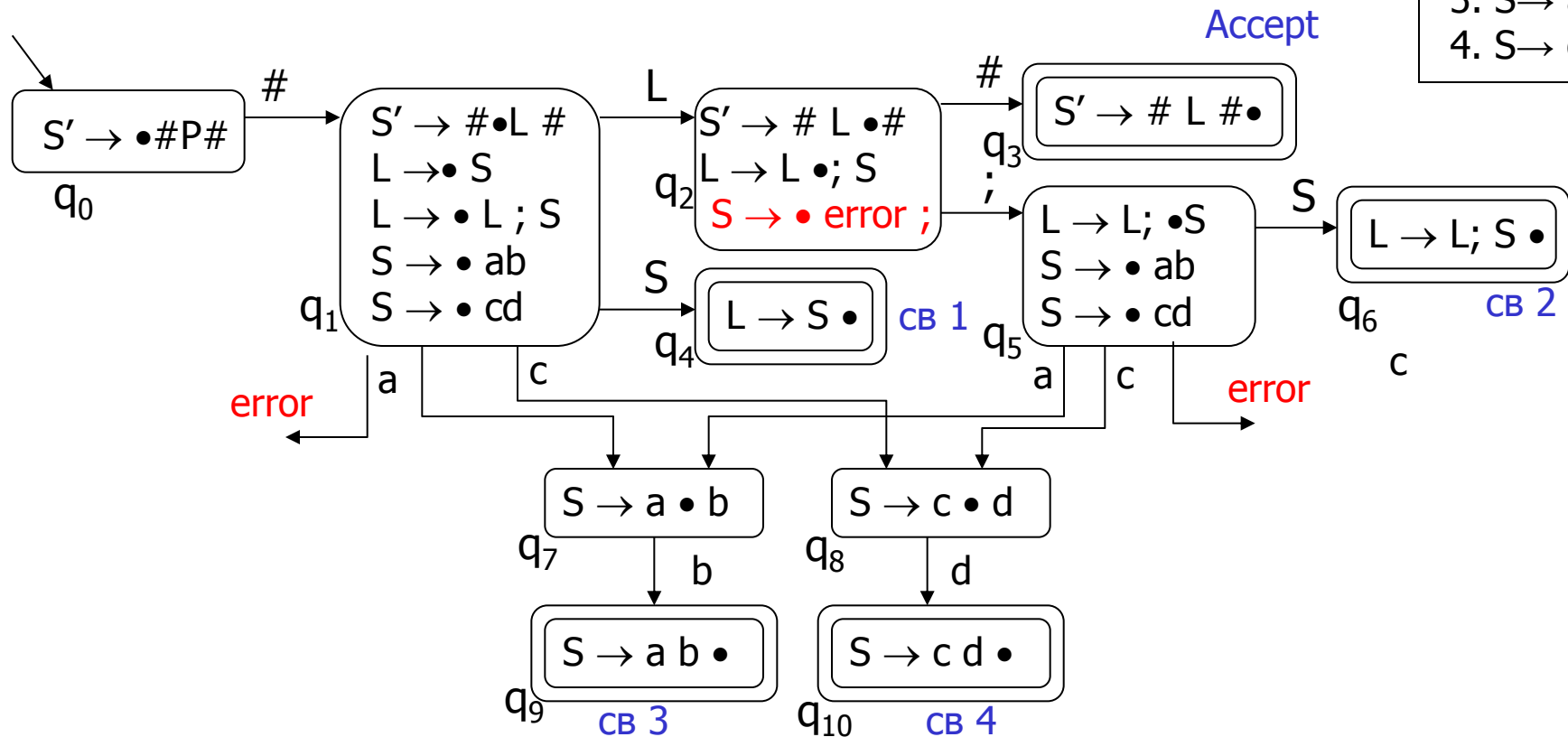
## Восстановление после ошибок в Yacc

- Редко бывает достаточно остановить всю обработку при обнаружении ошибки; более полезно продолжить сканирование входных данных для нахождения дальнейших синтаксических ошибок. Это приводит к проблеме рестарта парсера после ошибки. Обычно для этого отбрасывается некоторое количество лексем из входной строки и попытки привести в нужное состояние парсер, так что ввод может продолжаться.
- Восстановление после ошибок управляется пользователем с помощью введения в грамматику "правил ошибки" вида
- **$A \rightarrow \text{error } w$ :**
- Здесь error - ключевое слово YACC. Когда встречается синтаксическая ошибка, анализатор трактует состояние, набор ситуаций для которого содержит правило для error, некоторым специальным образом: символы из стека выталкиваются до тех пор, пока на верхушке стека не будет обнаружено состояние, для которого набор ситуаций содержит ситуацию вида  $[A \rightarrow \bullet \text{error } w]$ . После чего в стек фиктивно помещается символ error, как если бы он встретился во входной строке.
- Если w пусто, делается свертка. После этого анализатор пропускает входные символы, пока не найдет такой, с которым можно продолжить нормальный разбор.
- Если w не пусто, просматривается входная строка и делается попытка свернуть w. Если w состоит только из терминалов, эта строка ищется во входном потоке.

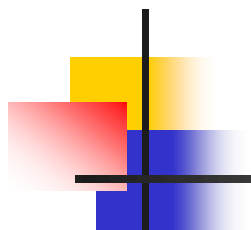


# LALR(1) – анализатор с обработкой ошибок

0.  $S' \rightarrow \#L\#$
1.  $L \rightarrow S$
2.  $L \rightarrow L ; S$
3.  $S \rightarrow ab$
4.  $S \rightarrow cd$



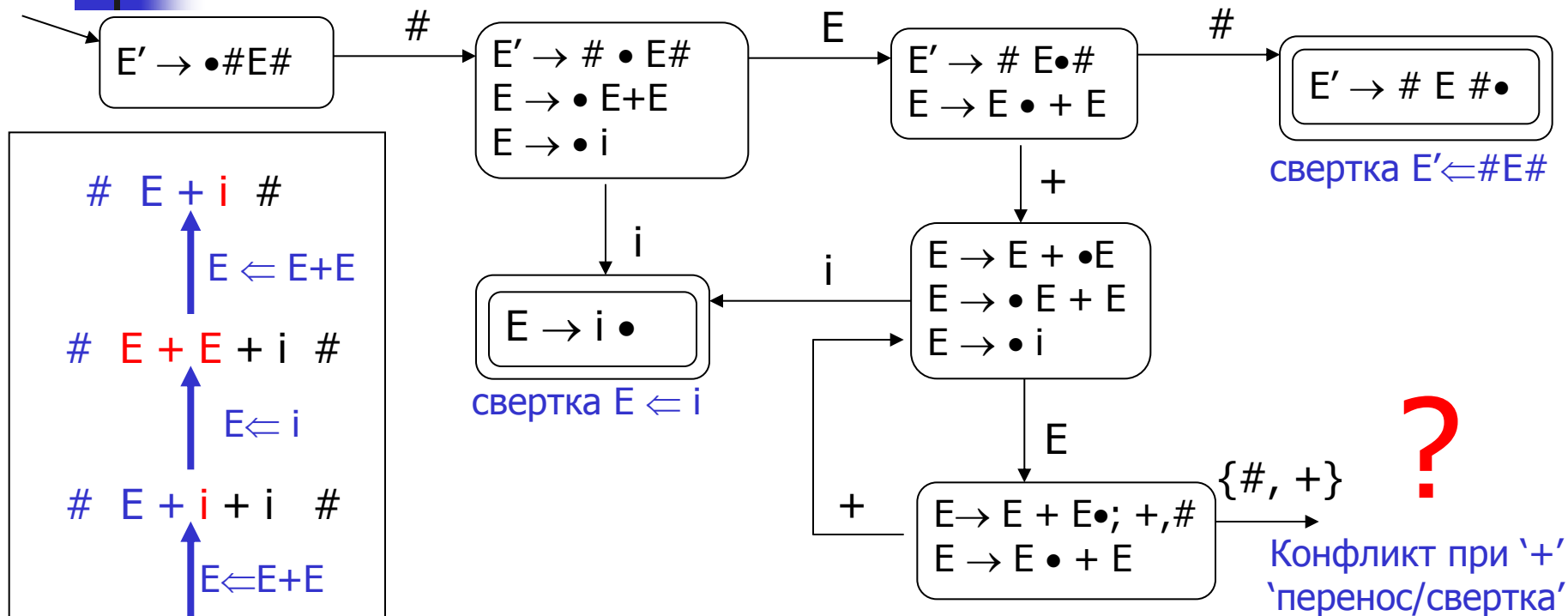
В ошибочной цепочке  $\#ab; aa; cd; cf; ab\#$  будут найдены обе ошибки



## Разрешение конфликтов в анализаторах двусмысленных грамматик

# LR анализ при конфликтах: перенос/свертка решается в пользу свертки

0.  $E' \rightarrow \#E\#$
1.  $E \rightarrow E+E$
2.  $E \rightarrow i$

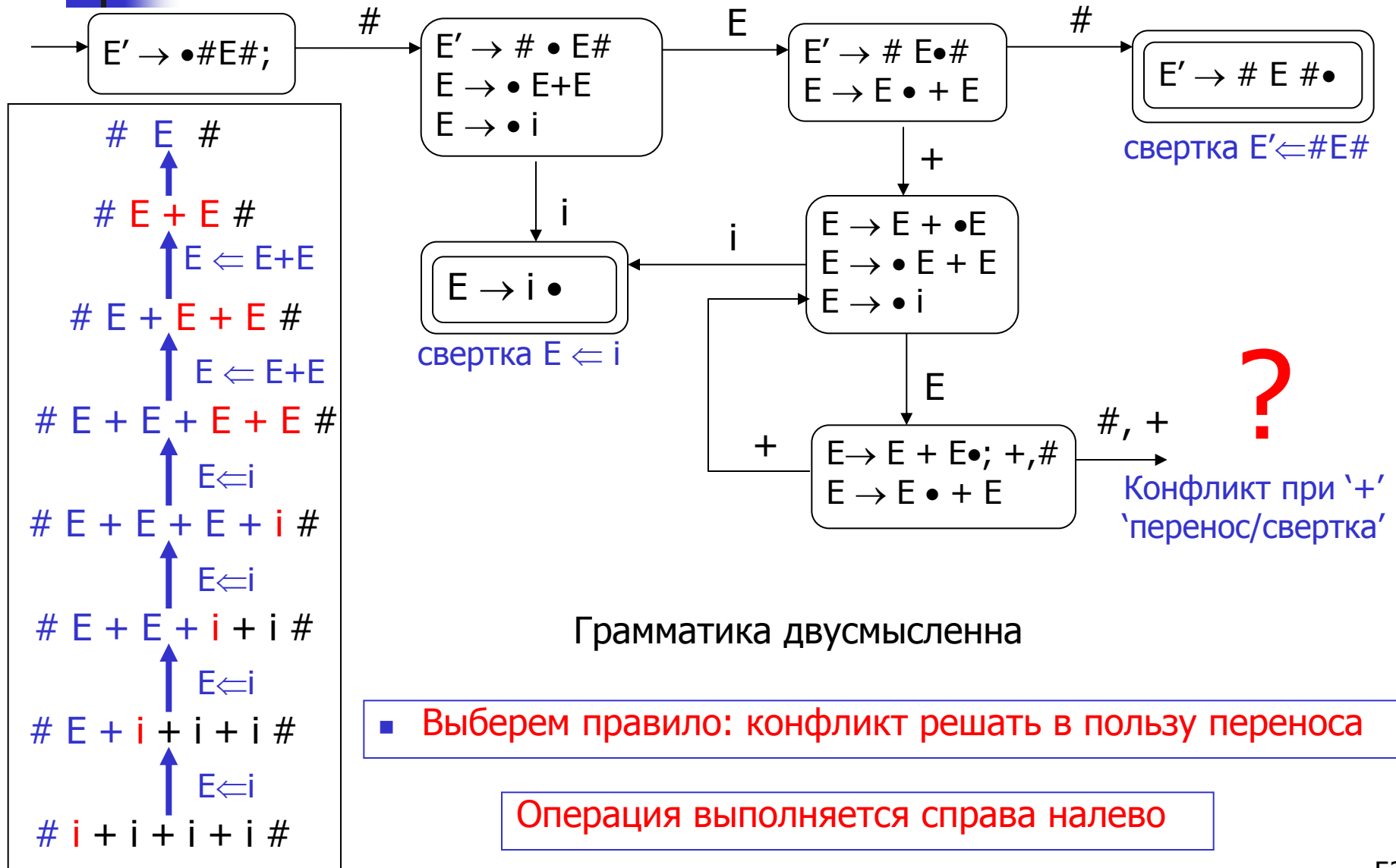


- Грамматика двусмысленна. Конфликт перенос/свертка показывает, что нет однозначного решения

- В некоторых случаях (при тщательном подборе правил) насильственное управление разрешением конфликтов может дать правильное и экономное решение. Выберем здесь правило: конфликт решать в пользу **свертки**. Операция выполняется слева направо

# LR анализ при конфликтах: перенос/свертка решается в пользу переноса

0.  $E' \rightarrow \#E\#$
1.  $E \rightarrow E+E$
2.  $E \rightarrow i$





## LR анализ двусмысленных грамматик: более сложные правила при конфликтах

- Можно ли в LALR(1)-анализаторе для двусмысленной грамматики:

$$S' \rightarrow \#E\#$$

$$E \rightarrow E+E \mid E * E \mid /$$

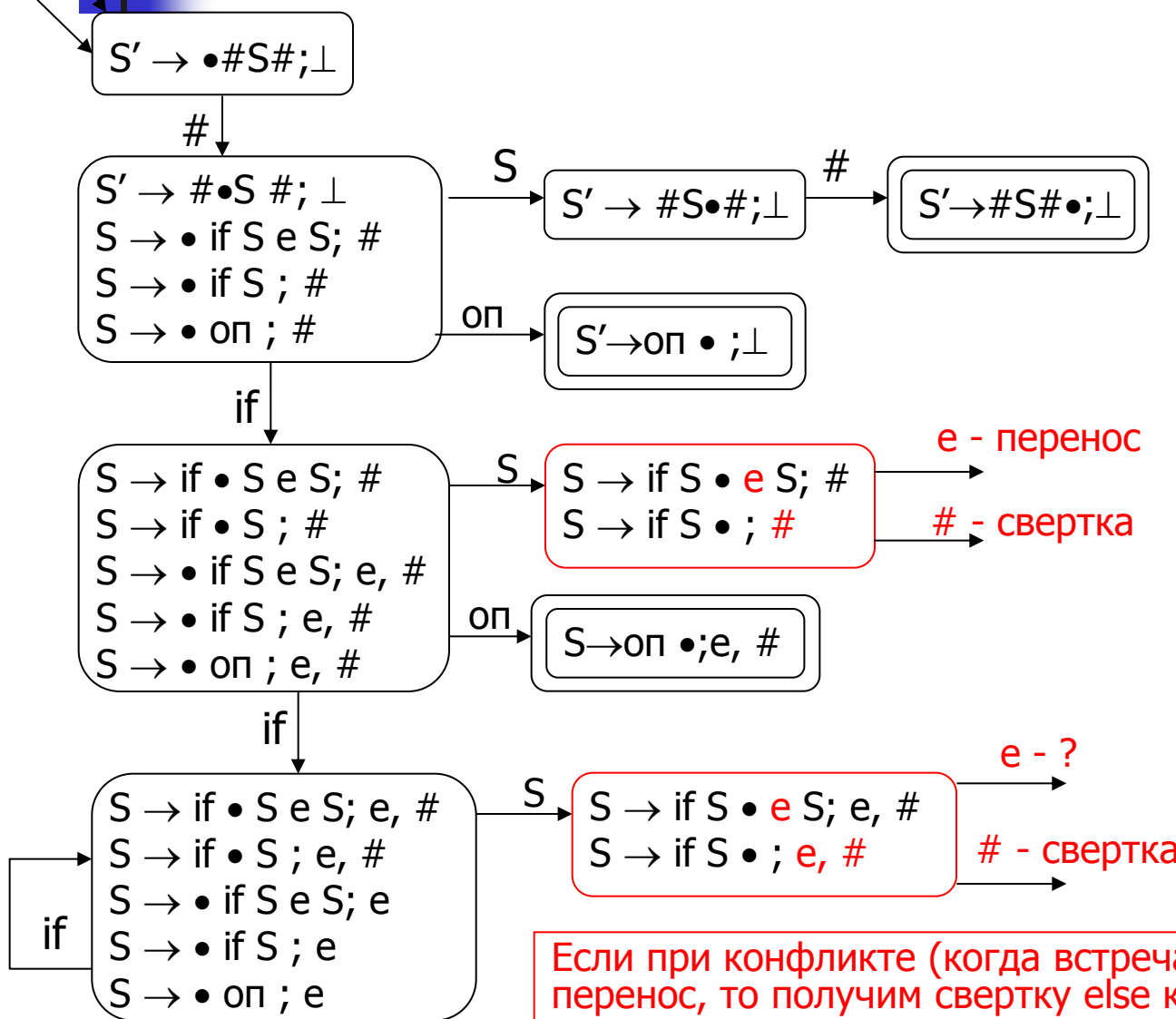
правила разрешения конфликтов перенос/свертка и свертка/свертка  
определить так, чтобы:

- а) соблюдался бы приоритет операций;
- б) операции одного приоритета выполнялись бы слева направо (справа налево)

Для самостоятельной проработки

# LR(1)-анализатор двусмысленной грамматики условных операторов

0.  $S' \rightarrow \#S\#$
1.  $S \rightarrow \text{if } b \text{ then } S \text{ else } S$
2.  $S \rightarrow \text{if } b \text{ then } S$
3.  $S \rightarrow \text{оп}$



Конфликт разрешается анализом контекста

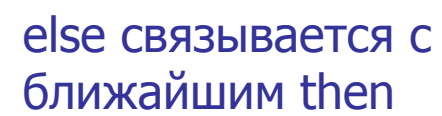
Что делать при e?

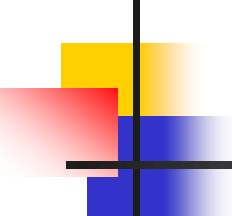
Конфликт не разрешится при любом k

Если при конфликте (когда встречается e) всегда выбирать перенос, то получим свертку else к ближайшему then

0.  $S' \rightarrow \#S\#$
1.  $S \rightarrow \text{if } b \text{ then } S \text{ else } S$
2.  $S \rightarrow \text{if } b \text{ then } S$
3.  $S \rightarrow \text{op}$

0.  $S' \rightarrow \#S\#$
1.  $S \rightarrow M$
2.  $S \rightarrow U$
3.  $M \rightarrow \text{if } b \text{ then } M \text{ else } M$
4.  $M \rightarrow \text{on}$
5.  $U \rightarrow \text{if } b \text{ then } S$
6.  $U \rightarrow \text{if } b \text{ then } M \text{ else } U$





---

# Семантические вычисления при восходящем синтаксическом анализе

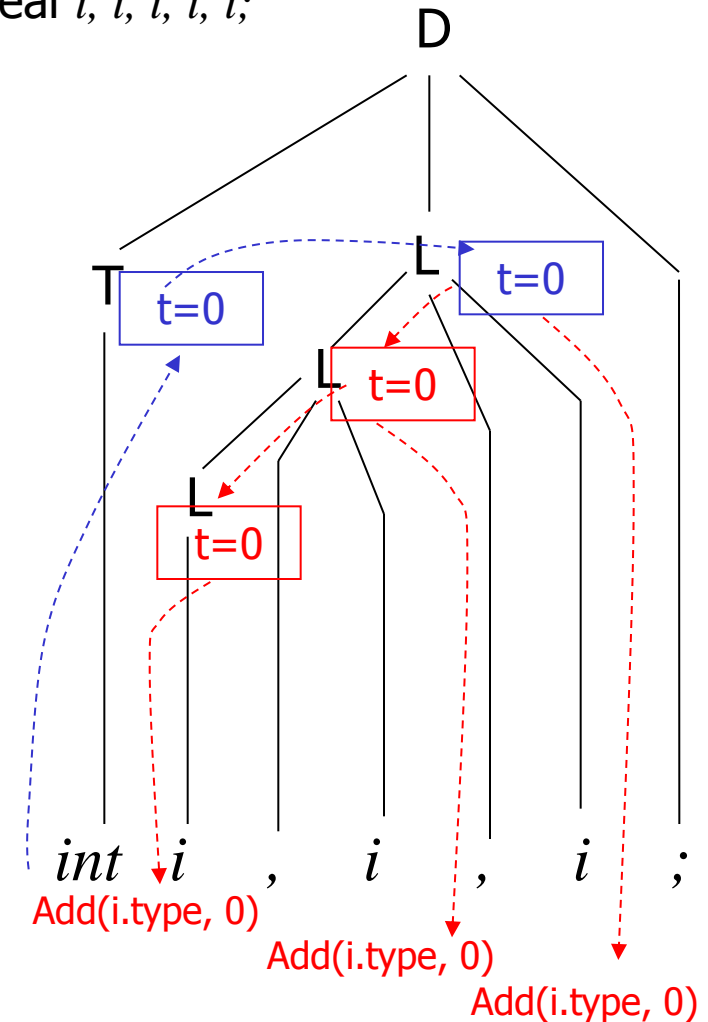


# Пример: атрибутивная грамматика описаний типов

Примеры порождаемых цепочек: `int i, i, i, i; real i, i, i, i, i;`

N	Синтаксис	Семантика
1	$D \rightarrow T : L ;$	$L.type := T.type$
2	$T \rightarrow \text{int}$	$T.type := 0$
3	$T \rightarrow \text{real}$	$T.type := 1$
4	$L \rightarrow L_1, i$	$L_1.type := L.type$ $Add(i.type, L.type)$
5	$L \rightarrow i$	$Add(i.type, L.type)$

Эту семантику нельзя выполнить при свертке по правилам 4 и 5 грамматики: мы еще не знаем атрибуты левой части правил



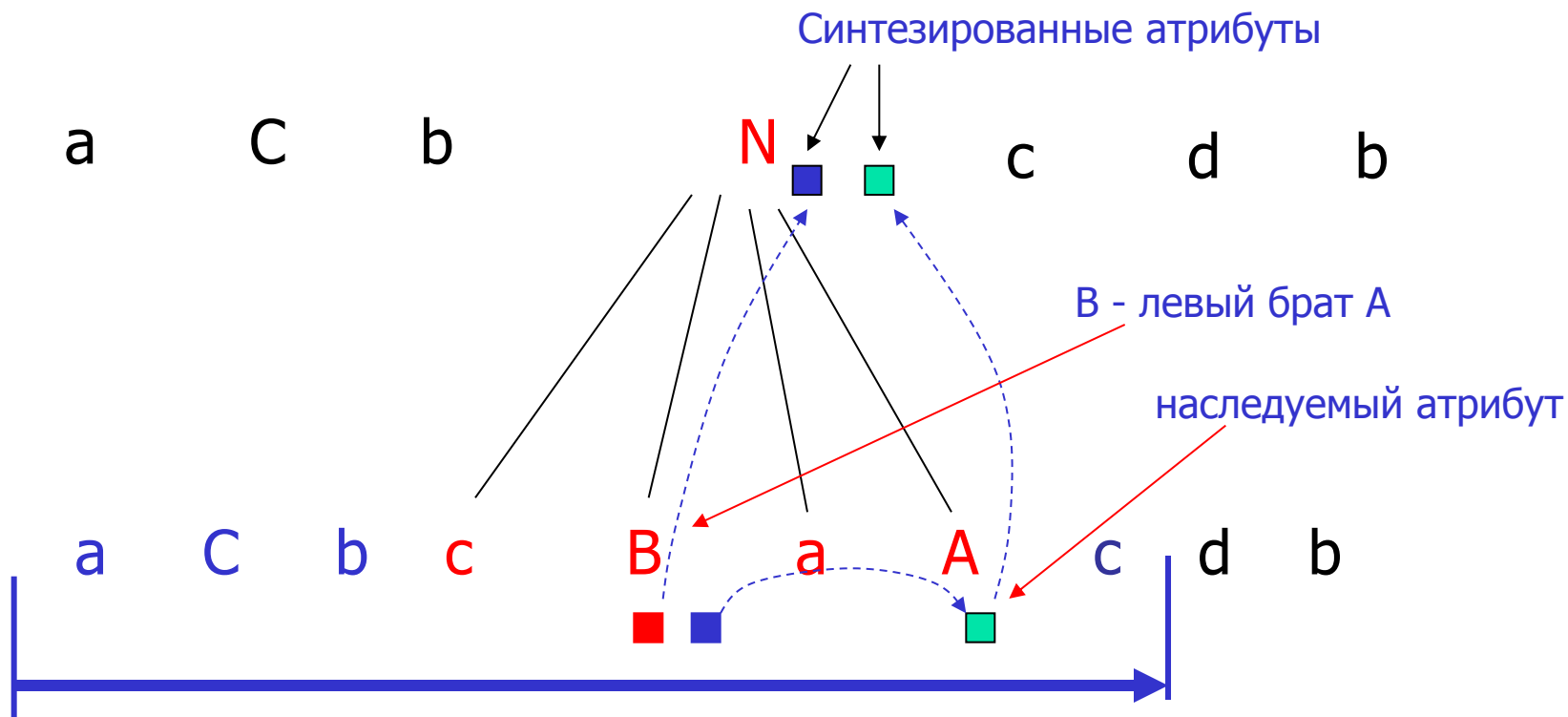
$Add(i.type, t)$  помещает  $t$  в поле  $type$  переменной с именем  $i$  таблицы имен



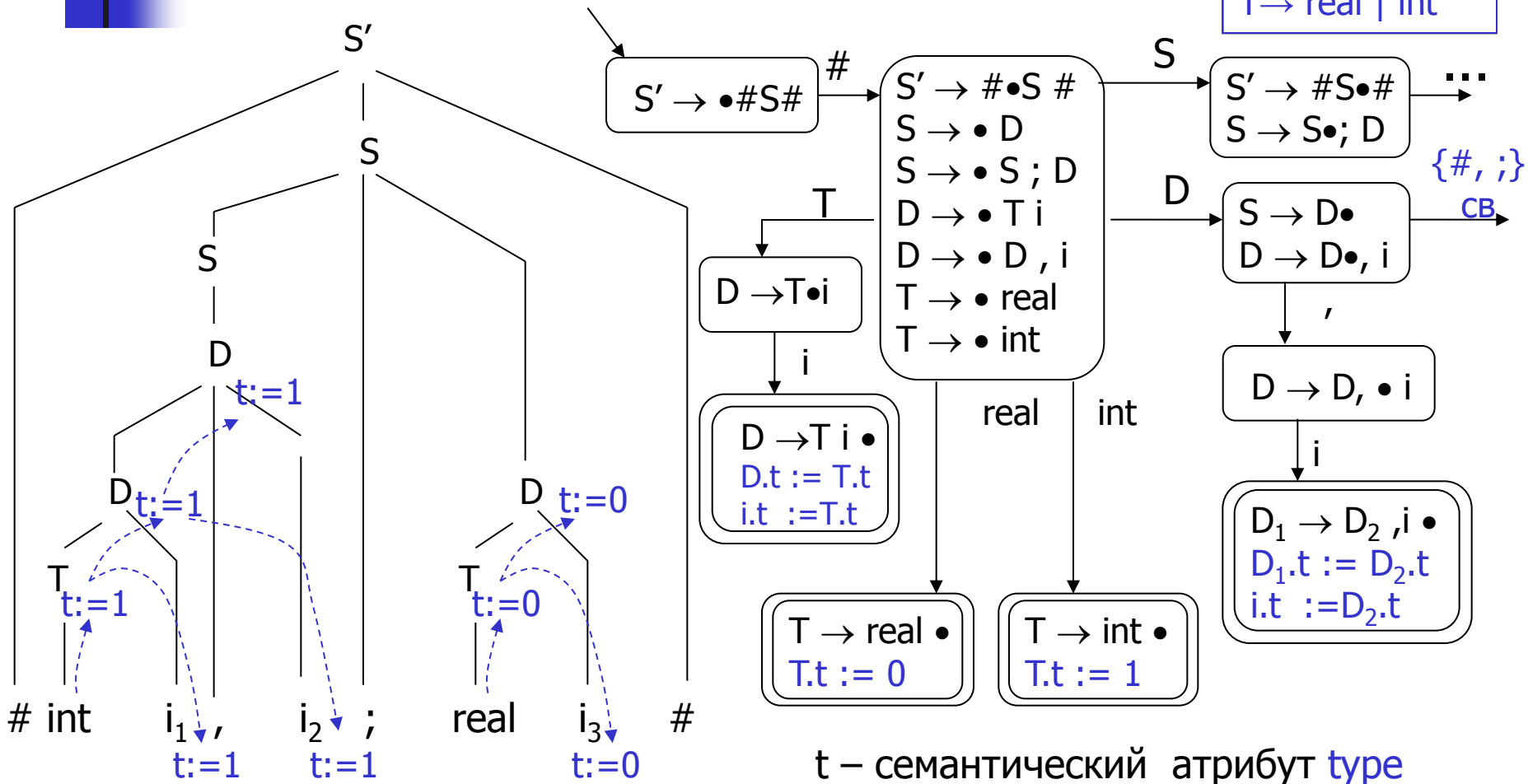
## S-атрибутные и L-атрибутные грамматики

- После осуществления шага свертки генерируется выход LR(1)-анализатора, т.е. исполняются семантические действия, связанные с правилом, по которому делается свертка, например, печатаются правила, по которым делается свертка
- Восходящий синтаксический анализ очень удобен для реализации семантики **синтезированных** атрибутов, когда семантические атрибуты нетерминала левой части правила грамматики являются функциями атрибутов правых частей правила (S-атрибутная семантика)
- Наследуемые атрибуты могут вычисляться двумя способами:
  - можно строить дерево вывода до тех пор, пока атрибуты нетерминалов можно будет подсчитать
  - использовать только L-атрибутную семантику: здесь атрибуты могут быть как наследуемыми так и синтезируемыми. Наследуемые атрибуты нетерминала в каждом правиле грамматики могут зависеть только от наследуемых атрибутов левой части правила и от любых атрибутов символов, находящихся **слева** от данного нетерминала. В этом случае семантические вычисления выполняются одновременно с построением синтаксического дерева

# Атрибутная семантика при восходящем анализе



Если атрибуты нетерминала А синтезированные, или наследуемые, но зависят только от атрибутов его левых братьев, то их можно вычислить прямо при свертке (редукции) при восходящем синтаксическом анализе

$$\begin{aligned} S' &\rightarrow \#S\# \\ S &\rightarrow D \mid S; D \\ D &\rightarrow T \mid i \mid D, i \\ T &\rightarrow \text{real} \mid \text{int} \end{aligned}$$


При восстановлении дерева снизу вверх при свертке семантические атрибуты вычисляются через ранее определенные значения



## Заключение

- LR(k) алгоритмы очень мощные: они мощнее всех рассматривавшихся ранее алгоритмов для подклассов КС-грамматик. Однако, существуют недвусмысленные грамматики, которые не являются LR(k)-грамматиками при любом k
- Восходящие алгоритмы синтаксического анализа удобны для атрибутивных семантических вычислений: вместе со сверткой (редукцией) по синтаксическому правилу выполняются семантики. Особенно это удобно для синтезированных атрибутов
- LR(k) анализаторы очень удобны для обнаружения ошибок: обычно ошибки обнаруживаются непосредственно в том месте, в котором нарушается синтаксис – правило построения конструкций
- LR(k) анализаторы очень громоздки: они имеют огромное число состояний из-за необходимости сохранения контекста. Даже LR(1) анализаторы, фактически, не применяются на практике. Вместо них используются несколько упрощенные LR-анализаторы,  
    SLR(k) – Simple LR(k) анализаторы и  
    LALR(k) – LookAhead LR(k) анализаторы
- В некоторых случаях управляя правилами разрешения конфликтов перенос/свертка и свертка/свертка можно использовать LR-анализаторы для правильного



---

Спасибо за внимание