Лекція 9.3.

Чем удобны диаграммы Вирта

Диаграммы Вирта позволяют достаточно просто решать ряд практических проблем, связанных с построением распознавателей. Во многом это обуславливается тем, что решение переходит из разряда символических преобразований в область манипуляции с графами. В большинстве случаев работать с графами легче и проще, чем с формулами или таблицами. Вот только ряд задач, при решении которых удобно использовать ДВ.

1. Избавление от левых или правых рекурсий.
2. Преобразование исходных синтаксических правил к более простому виду.
3. Преобразование КС(n) грамматик к грамматикам с просмотром вперед на меньшее число символов, включая и преобразование к КС(1) грамматике.
4. Проверка на эквивалентность различных грамматик.
5. Освобождение от пустых правил (сквозных связей).
6. Непосредственное использование правил с левой рекурсией.
7. Использование синтаксиса для определения семантики.
8. Доказательство принадлежности заданной грамматики к КС(1) грамматике.

Наиболее эффективным применение диаграмм Вирта выглядит при комплексном решении представленных задач в ходе практической разработки синтаксиса языков программирования. При этом возможен произвольный порядок преобразования исходных синтаксических правил, что не всегда достижимо при использовании других, широко известных метаязыков.

Избавление от левых или правых рекурсий

Избавление от рекурсий уже было рассмотрено для лексических анализаторов. Оно с легкостью позволяло сводить различные описания лексем к виду, удобному для построения конечных автоматов. При разработке нисходящих распознавателей сведение левой или правой рекурсии к итерации позволяет уменьшить вложенность правил. Кроме того, наличие левой рекурсии не позволяет осуществлять нисходящий разбор. Поэтому, ее преобразование к итерации – это один из наиболее простых и удобных способов получения из грамматик, ориентированных на восходящий разбор, грамматик, используемых при нисходящем разборе. Далее преобразование рекурсий в итерации будет показано на различных примерах.

Преобразование исходных синтаксических правил к более простому виду

Диаграммы Вирта позволяют упростить исходную грамматику, что дает возможность прейти к более простым методам разбора. Еще раз рассмотрим S-грамматику $\mathbf{G_{7.3}}$. На рис. 9.1 представлена последовательность шагов, обеспечивающая переход от текстового представления к диаграммам Вирта (шаг 1). На следующем шаге проводится ***избавление от правых рекурсий*** в правилах, описывающих нетерминалы **S** и **R** (шаг 2). Далее осуществляется

преобразование к одному правилу (шаг 3) и проводится его упрощение (шаг 4). В результате, вместо автомата с магазинной памятью, можно построить более простой конечный автомат.

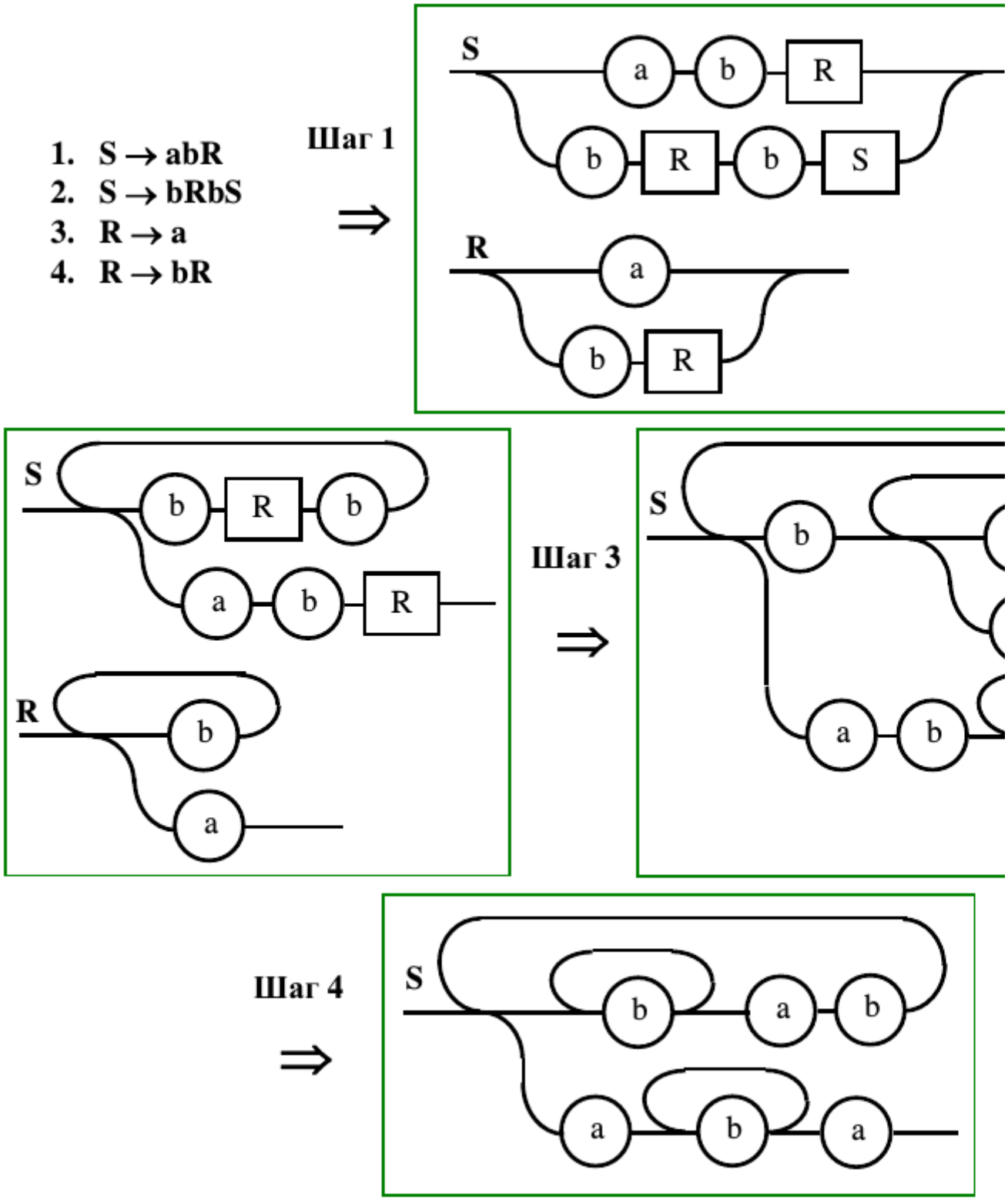


Рис. 9.1. Преобразование грамматики $G_{7.3}$ к более простому в

Преобразование КС(n) грамматик к грамматикам с

просмотром вперед на меньшее число символов

Подобные преобразования тоже используются для упрощения реализации. Кроме этого ускоряется разбор, так как уменьшается число анализируемых вариантов на каждом шаге. В ходе проводимых преобразований грамматику в дальнейшем можно свести к более простому виду. На рис. 9.2 рассматривается грамматика $\mathbf{G_{7.2}}$, относящаяся к КС(2) грамматике. После построения диаграмм Вирта (шаг 1) следует избавиться от правых рекурсий в обоих правилах и освободиться от недетерминированной альтернативы в правиле, описывающим **T**, за счет объединения общей подцепочки **"b"** двух альтернативных ветвей (шаг 2). На следующем шаге избавляемся от простого правила, описывающего нетерминал **T**, получая грамматику, реализуемую одним конечным автоматом.

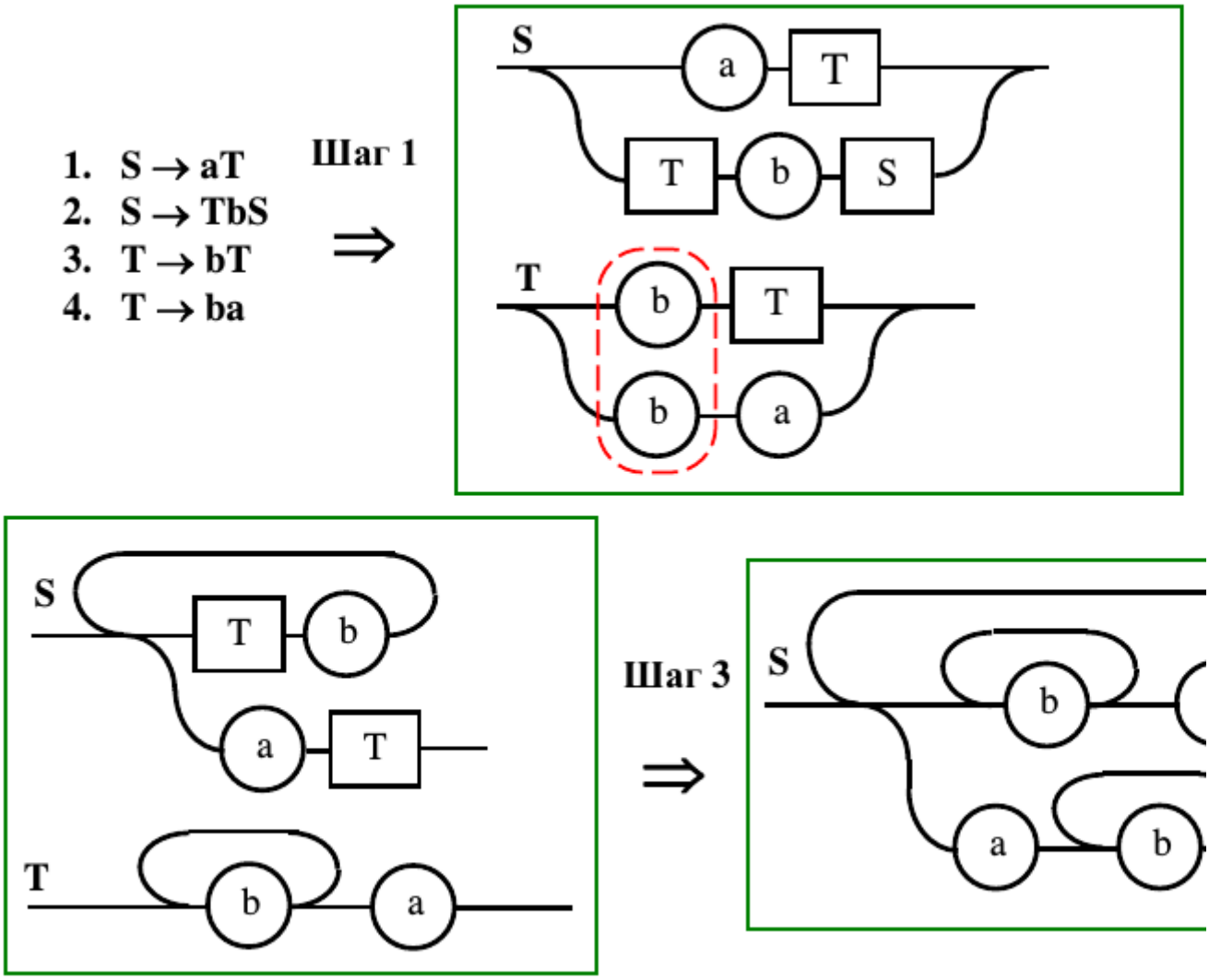


Рис. 9.2. Преобразование КС(2) грамматики $\mathbf{G_{7.2}}$ к КС(1) грамма

Проверка грамматик на эквивалентность

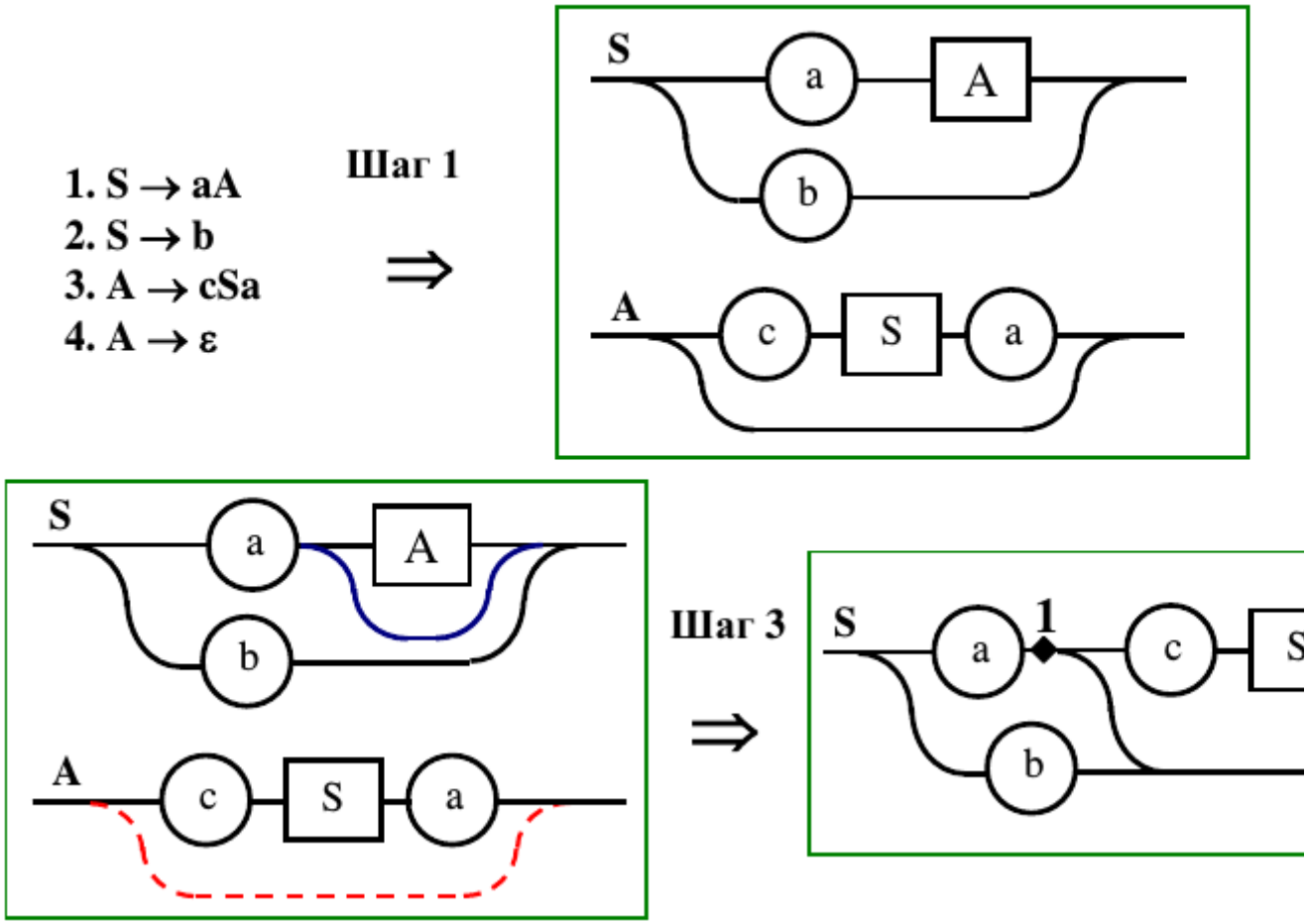
В предыдущих примерах попутно решается проблема доказательства эквивалентности грамматик $\mathbf{G_{7.2}}$ и $\mathbf{G_{7.3}}$. Она сводится к получению одинакового вида конечных правил. Это позволяет использовать в качестве исходного задания языка то представление, которое

удобнее для восприятия человеком, а не то, которое изначально ориентируется на определенную реализацию. Диаграммы Вирта обеспечивают удобное преобразование к нужному виду любых исходных правил.

Освобождение от пустых правил

Большинство грамматик, используемых при нисходящем разборе, имеют пустые правила, что, зачастую, затрудняет их понимание и не позволяет быстро определить, принадлежит ли данная грамматика требуемому классу. Диаграммы Вира позволяют избавиться от пустых правил и преобразовать при этом исходную грамматику к виду, более удобному для реализации. Пустым правилам в ДВ соответствуют «**сквозные связи**», напрямую соединяющие начало правила с его концом без захода в промежуточные вершины (терминальные или нетерминальные). Избавление от таких связей осуществляется удалением, что соответствует их выносу за пределы правила. При этом во всех правилах, содержащих измененный нетерминал, необходимо провести дополнительные связи, осуществляющие его обход (что и компенсирует удаленную сквозную связь). Вынос сквозной связи может привести к образованию «**пустой связи**», которая является линией, «накоротко» соединяющей две другие связи ДВ.

В качестве примера рассмотрим q-грамматику $\mathbf{G_{7.5}}$. На рис. 9.3 приведена последовательность шагов, в ходе которой осуществляется преобразование исходной грамматики к диаграммам Вирта (шаг 2), удаление сквозной связи из правила A, соответствующей пустому правилу, и построение связи-обхода нетерминала **A** там, где он располагается в диаграммах (шаг 2). При необходимости, можно упростить полученную грамматику (шаг 3). Следует отметить, что в результате преобразований появилась пустая связь, соединяющая точку 1 с концом диаграммы (показана на диаграмме, полученной после шага 3).



Освобождение от сквозных связей в одной диаграмме Вирта может привести к их появлению в других. В этом случае процесс повторяется до исчезновения сквозных связей или до тех пор, пока они не перейдут в диаграмму, задающую начальный нетерминал. Последняя ситуация указывает на то, что исходная грамматика допускает пустую цепочку.

Непосредственное использование правил с левой рекурсией

Если при изучении лексических анализаторов наиболее типичным примером лексемы является идентификатор, то классическим образцом, демонстрирующим особенности различных видов разбора, можно считать простое выражение. Наиболее наглядно язык, порождающий выражения задается с использованием правил, содержащих левую рекурсию. В качестве примера рассмотрим грамматику $\mathbf{G_{9.1}}$:

1. **E → T**
2. **E → E + T**
3. **T → P**
4. **T → T + P**
5. **P → (E)**
6. **P → i**

В связи с тем, что непосредственное использование этих правил для нисходящего разбора

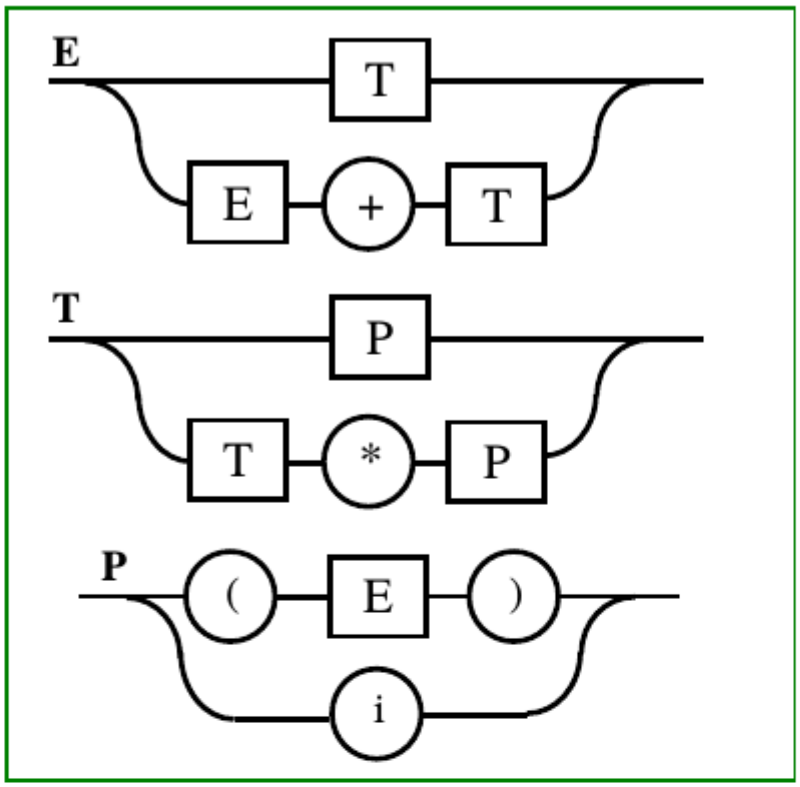
невозможно, они часто преобразуются в эквивалентные правила LL(1)-грамматики $\mathbf{G_{9.2}}$:

1. **E → T E_список**
2. **E_список → + T E_список**
3. **E-список →ε**
4. **T → P T_список**
5. **T_список → + P T_список**
6. **T_список → ε**
7. **P → (E)**
8. **P → i**

Подобные преобразования связаны с утомительной заменой левых рекурсий на правые. Вместе с тем, использование диаграмм Вирта позволяет легко осуществить непосредственную замену левых рекурсий грамматики $\mathbf{G_{9.1}}$ итерациями (рис 9.4).

1. **E → T**
2. **E → E + T**
3. **T → P**
4. **T → T + P**
5. **P → (E)**
6. **P → i**

Шаг 1 ⇒



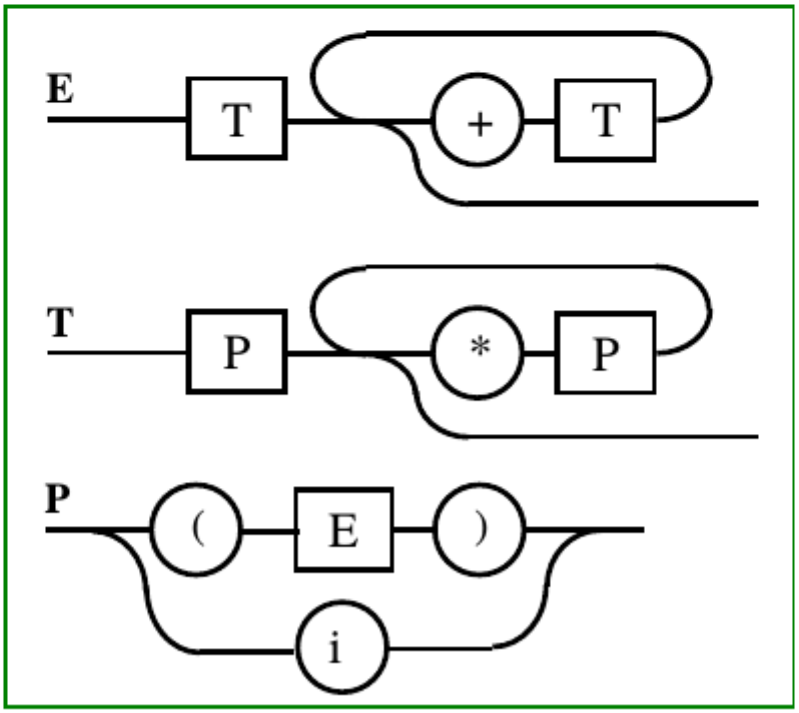


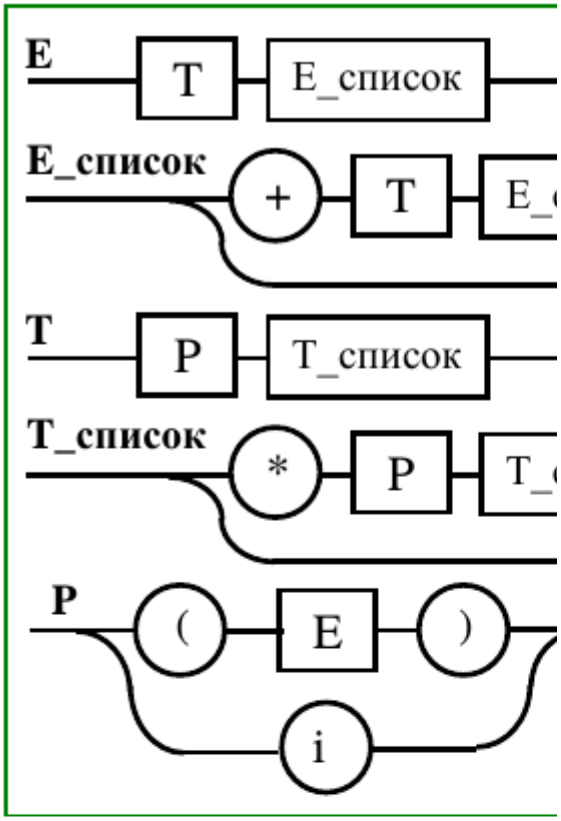
Рис. 9.4. Преобразование грамматики с левой рекурсией $G_{9.1}$ к итеративным диаграммам Вирта

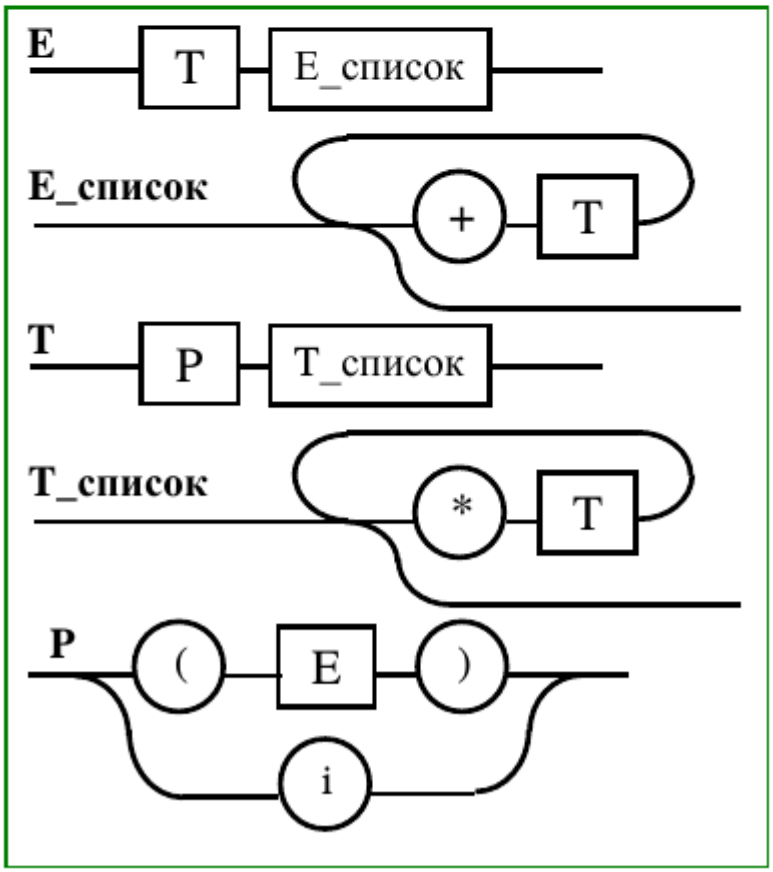
То, что полученные правила эквивалентны приведенной выше LL(1) грамматике $G_{9.2}$, можно доказать, проведя преобразования последней к эквивалентному виду. Эти преобразования представлены на рис. 9.5.

1. **E → T E_список**
2. **E_список → + T E_список**
3. **E-список → ε**
4. **T → P T_список**
5. **T_список → + P T_список**
6. **T_список = ε**
7. **P = (E)**
8. **P = i**

Шаг 1

⇒





Шаг 3

⇒

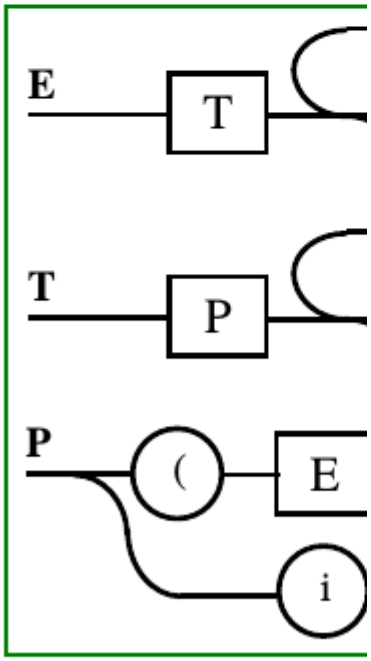


Рис. 9.5. Преобразование грамматики с правой рекурси
к итеративным диаграммам Вирта

Использование синтаксиса для определения семантики

Синтаксические правила, построенные с использованием диаграмм Вирта, позволяют определять элементы семантики языка программирования. Следует отметить, что и другие метаязыки, в том числе и текстовые, могут использоваться для задания семантики. Однако ДВ обеспечивают большую гибкость.

Управление приоритетом инфиксных операций

Использование синтаксиса для определения приоритета инфиксных операций возможно с использованием любого метаязыка. Для этого выражения определяются иерархически, как это показано на рис. 9.4, 9.5. Главное – не переборщить с преобразованиями диаграмм. Получив эквивалентные диаграммы для рассмотренных выше синтаксических описаний выражений, построенных на основе грамматик $\mathbf{G_{9.1}}$ и $\mathbf{G_{9.2}}$, мы можем сделать еще несколько общих шагов по их дальнейшему преобразованию (рис. 9.6). В начале можно избавиться от избыточных нетерминалов, сделав правила более компактными (шаг 1). Этот шаг не изменяет приоритетов операций. Но он и не вносит дополнительной оптимизации в разрабатываемый далее код, так как не уменьшает число состояний автомата. Эти состояния отмечены пронумерованными жирными точками на первой и второй версиях диаграмм правила, описывающего **E**. До этого нам ничего не давала аналогичная оптимизация идентификатора (см. тему 3). Поэтому попытаемся сократить число правил, подставив в правило, описывающее **E**, диаграмму, описывающую **T** (шаг 2). Но избавление от нетерминала **T** привело к тому, что операции сложения и умножения стали равноправными по семантике. Их приоритет по ДВ уже не определить. Конечно, существуют и такие языки программирования, в которых приоритет этих операций равный. И там подобный прием может оказаться полезен. Однако мы считали и считаем, что имеем язык, в котором приоритет операции умножения выше, чем у операции сложения. Теперь же, чтобы учесть наше мнение, необходимо вводить в программу дополнительную семантику, что может оказаться посложнее, чем использование для этих целей синтаксиса.

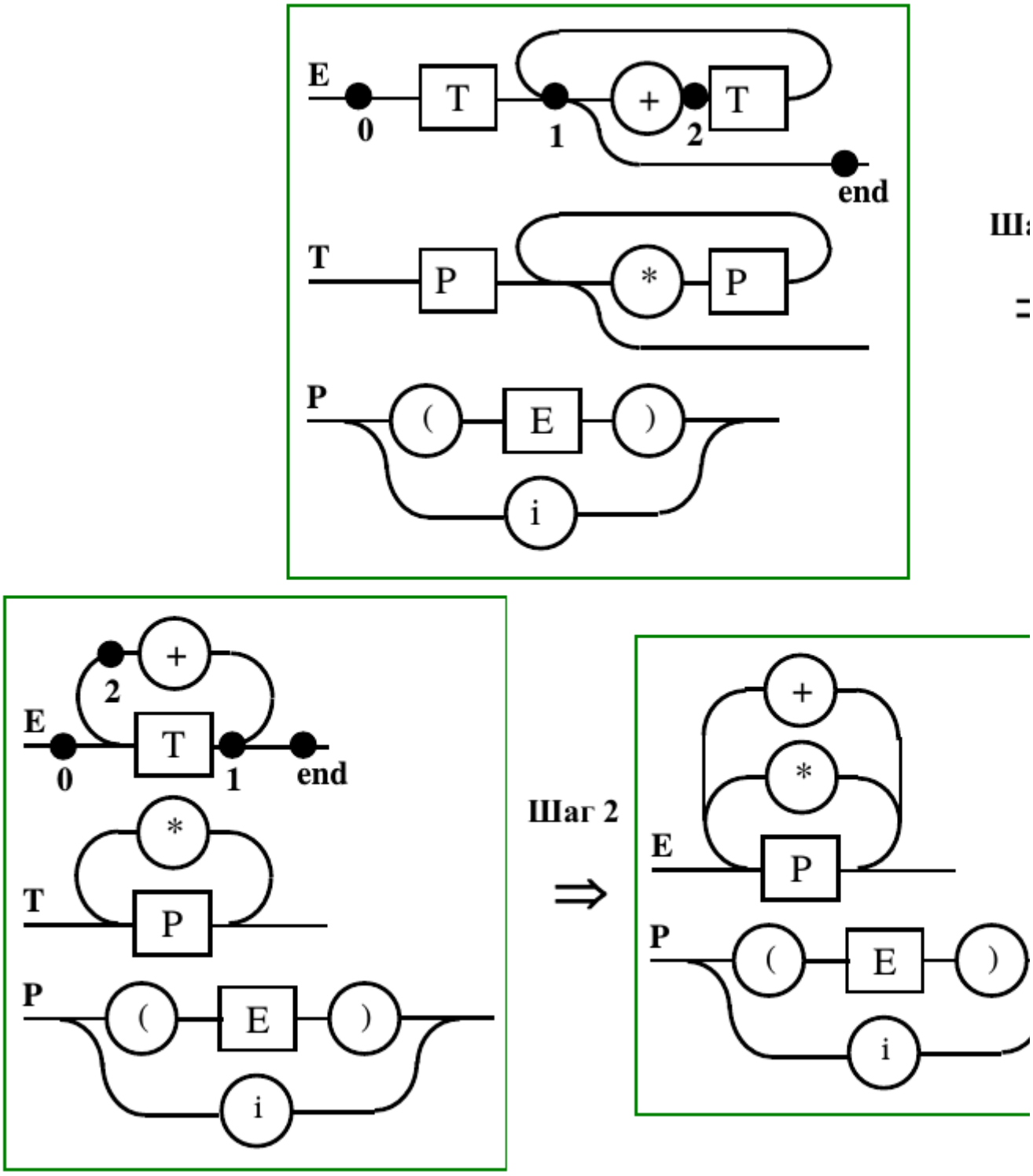


Рис. 9.6. Исчезновение приоритетов операций в ходе преобразо
диаграммам Вирта, описывающих выражения

Когда рекурсия полезна для задания приоритетов

Использование правил с правой рекурсией может оказаться полезным, если выполнение некоторой двуместной операции производится справа налево. Здесь и проявляются преимущества метаязыков, поддерживающих описание синтаксиса с использованием

итераций: они легко позволяют задавать любой порядок выполнения операций. Предположим, что инфиксной операцией с самым высоким приоритетом будет возведение в степень (обозначим значком “^”), обычно выполняемое справа налево. Правила, описывающие синтаксис выражения для этого случая приведены на рис. 9.7. Использование правой рекурсии позволяет не задумываться над заданием семантики для вновь введенной операции. Использование же итерации для операций сложения и умножения позволяет их обрабатывать (и выполнить при интерпретации) слева направо.

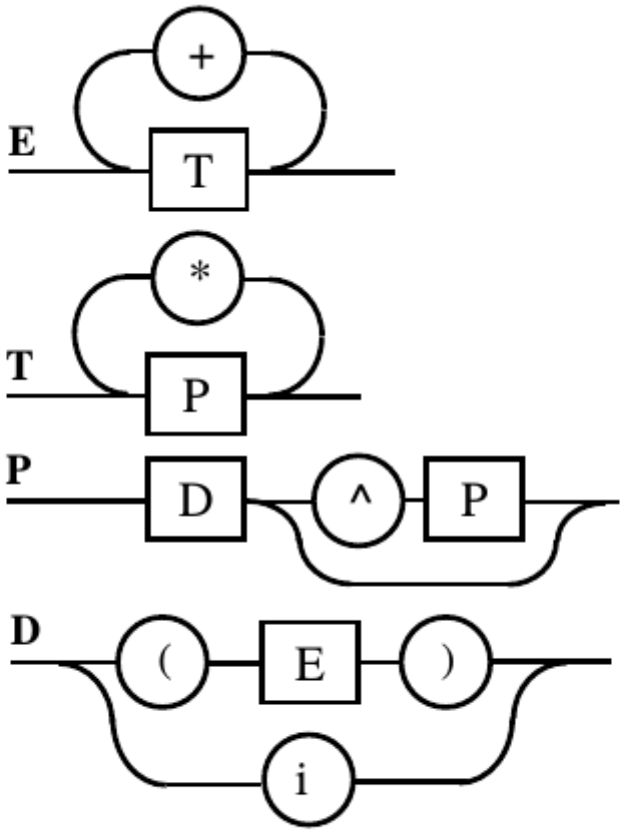


Рис. 9.7. Использование правой рекурсии для задания приоритета справа необходимого для операции возведения в степень

Доказательство принадлежности к КС(1) грамматике

Использование диаграмм Вирта позволяет достаточно просто определять принадлежность к КС(1) грамматике, пригодной для построения простого нисходящего распознавателя. Это класс грамматик включает и LL(1) грамматики, доказательство принадлежности к которому мы так и не построили. Теперь это делать и не надо, раз мы можем проанализировать принадлежность к более широкому классу.

Альтернативные точки

Простота доказательства во многом обуславливается графическим видом правил. В начале строятся ДВ. После этого, отдельно для каждого правила осуществляется анализ альтернатив, определяемых точками ветвления связей, входящих в различные терминалы и нетерминалы (рис. 9.8). Принадлежность к КС(1) грамматике доказывается в том случае, если для всех точек

ветвления не будут найдены два или более одинаковых терминала, проверяемых на первом шаге. То есть, доказывается отсутствие в конечном автомате недетерминированных переходов из некоторого состояния. Конечно, существует большая вероятность того, что проверяемые связи входят в нетерминалы. Поэтому необходимо (возможно, рекуррентно) исследовать точки связи на входе в правила, описывающие эти нетерминалы. В результате такого анализа с каждой точкой ветвления связей (назовем ее альтернативной точкой) можно сопоставить множество доступных терминалов.

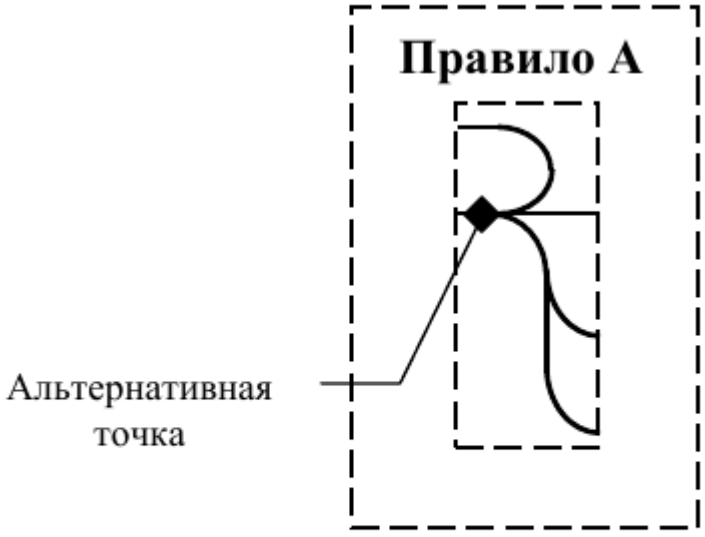


Рис. 9.8. Альтернативные точки диаграммам Вирта, используемые для анализа на принадлежность к КС(1) грамматике.

Пример подобной проверки представлен на рис. 9.9. Для определения всех возможных терминалов в точке **0** диаграммы **S** необходимо выяснить все терминалы в точке **0** правила **A**. Однако, **A** в этой точке имеет в качестве альтернатив нетерминалы **B** и **C**, для которых дальнейшую проверку делать не нужно, так как их альтернативные точки содержат терминалы. Остается возвратиться к исходной диаграмме, собирая на обратном пути терминалы. Таким образом, **B** в точке **0** начинается с **{c}**, **C** в точке **0** начинается с **{a, e}**. Правило **A** разветвляется на объединенное множество **{a, c, e}**, а на долю **S** приходится **{a, b, c, d, e}**.

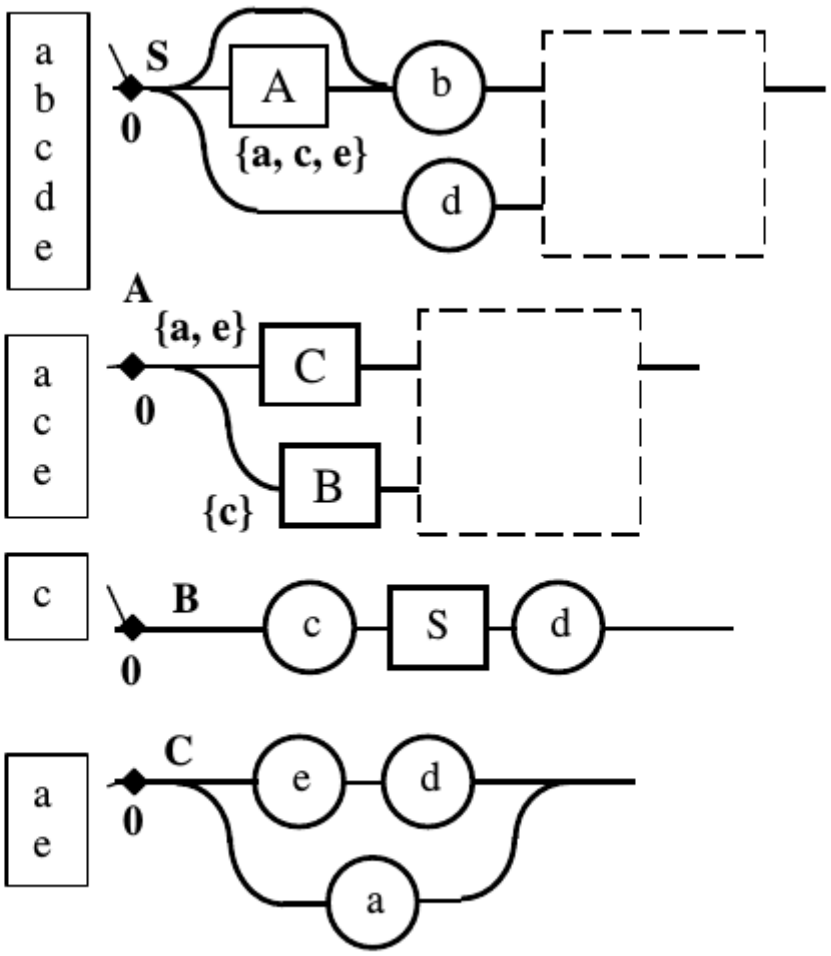
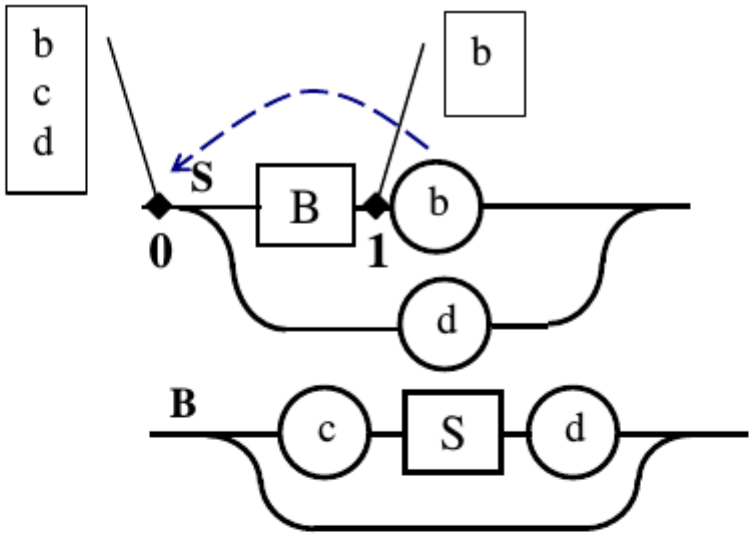


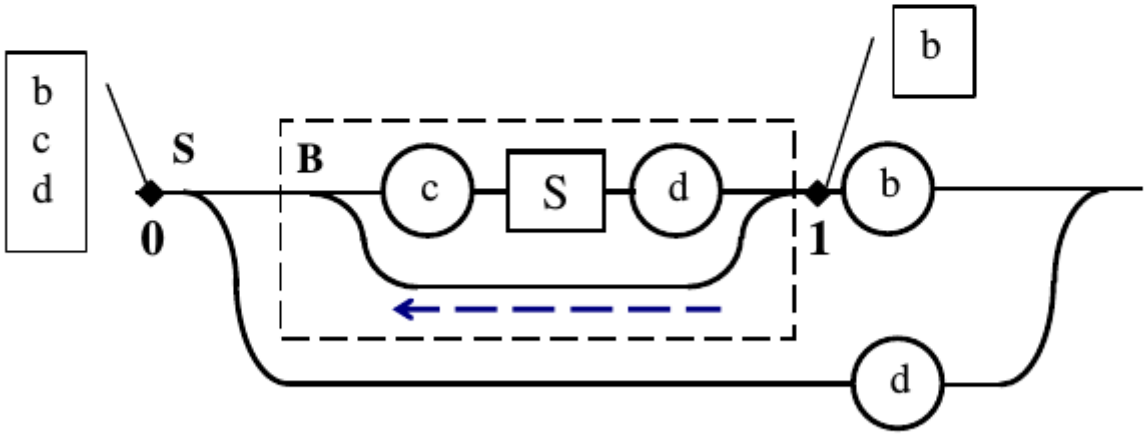
Рис. 9.9. Рекуррентное выявление терминалов в альтернативных точках

Принцип перетекания

При наличии в диаграмме Вирта сквозной связи анализ несколько изменяется. Он усложняется из-за того, что возможные варианты терминалов на входе такого правила могут совпадать с аналогичными терминалами диаграмм, в которые данное правило входит в качестве нетерминала. Преобразования, представленные на рис. 9.10, показывают, каким образом сквозная связь, наряду с альтернативами в исходной точке, приводит к рассмотрению альтернатив, следующих за нетерминалом, представленным этим правилом (в данном случае, за **B**). Происходит, как бы, «перетекание» значений терминалов по сквозной связи в точку анализа альтернативы. Учитывая ориентацию связей ДВ, обычно не указываемую явно стрелками, можно говорить о «перетекании против течения». В результате в альтернативных точках, имеющих входы в правила со сквозными связями, «собираются» терминалы, следующие за этими правилами.



а) сквозная связь в одном из нетерминалов, обеспечивает «перетекание» терминала, расположенного за этим правилом.



б) влияние сквозной связи на альтернативу, следующую за **B**, явно просматривается после подстановки правила.

Рис. 9.10. Иллюстрация принципа «перетекания», связанного с наличием сквозной связи в диаграмме, используемой в другом правиле

Следует отметить, что при наличии в альтернативной точке нескольких нетерминалов со сквозными связями приходится учитывать их общую «прозрачность» путем «перетекания против течения» всех следующих за ними терминалов. Возможна также ситуация, когда такие нетерминалы образуют последовательную «прозрачную» цепочку. В этом случае терминалы, «двигаясь против течения» оставляют свой образ в каждой из пройденной альтернативной точке. Если нетерминал со сквозной связью располагается между входом и выходом некоторой диаграммы, то такая диаграмма тоже может рассматриваться как имеющая сквозную связь. Подобная ситуация возникает в демонстрационном примере, расположенном ниже.

Чтобы не запутаться в сложных структурах диаграмм Вирта, целесообразно провести их предварительные эквивалентные преобразования, связанные с избавлением от пустых правил (сквозных связей). На рис. 9.11 представлены ДВ, полученные из уже рассмотренных (на рис.

9.10) диаграмм.

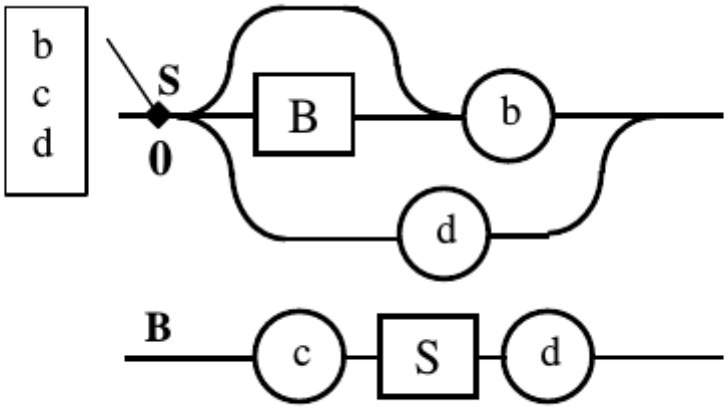


Рис. 9.11. Избавление от пустых связей позволяет упростить анализ

Концентрация альтернатив в конце правил

Рассмотрим еще одну «неприятную» ситуацию, возможную при анализе альтернатив: размещение нетерминала, содержащего сквозную связь, в конце диаграммы. В этом случае возникает необходимость в анализе альтернатив тех правил, в которых используется «неудобное» правило используется как нетерминал. Для облегчения анализа предлагается следующая процедура. В начале необходимо перенести в конец рассматриваемого правила множество терминалов из всех альтернативных точек, доступных через сквозные связи при «перетекании против течения». «концентрировать» альтернативные терминалы на конце правила, чтобы использовать их затем при анализе в других правилах. На рис. 9.12 Нетерминал **S** содержит **B**. В диаграмме **B** имеется сквозная связь, обуславливающая «перетекание» терминалов с конца правила **S** в альтернативу, расположенную перед **B**. Однако мы не знаем, какие терминалы следуют за **S** в каждом конкретном случае. Поэтому записываем все терминалы альтернативы в конце **S** и добавляем их к анализу альтернатив в точках, следующих непосредственно за нетерминалом **S** (выписывая их в сноске на нетерминал). Принадлежность к КС(1) грамматике доказывается в том случае, если множества терминалов, принадлежащих различным правилам, не пересекаются.

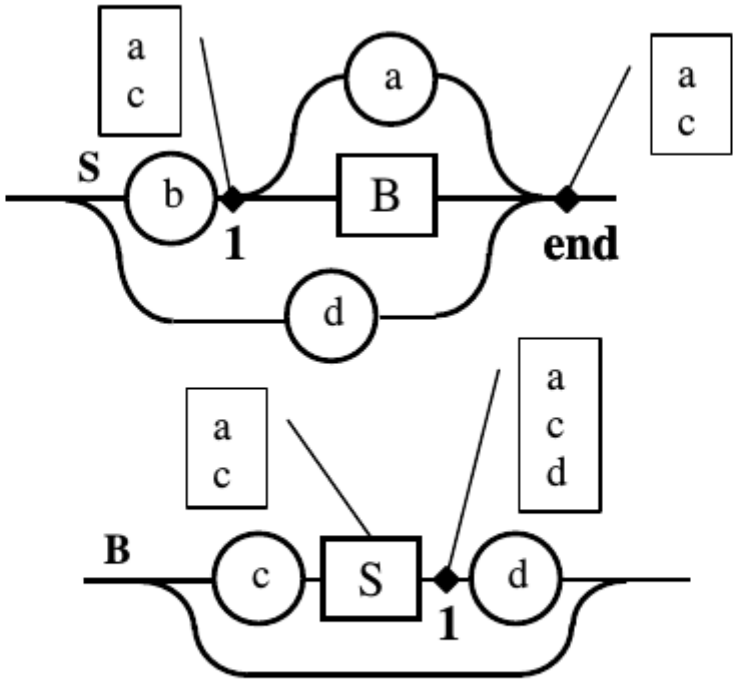


Рис. 9.12. Нетерминал B со сквозной связью, размещенный на конце пра
ведет к «перетеканию» терминалов в конец правила S

Можно поступить также, как и в предыдущем случае: избавиться от сквозных связей. Результат представлен на рис. 9.13. Появление пустой связи на конце правила, может быть и увеличила наглядность, но не изменила сам принцип анализа

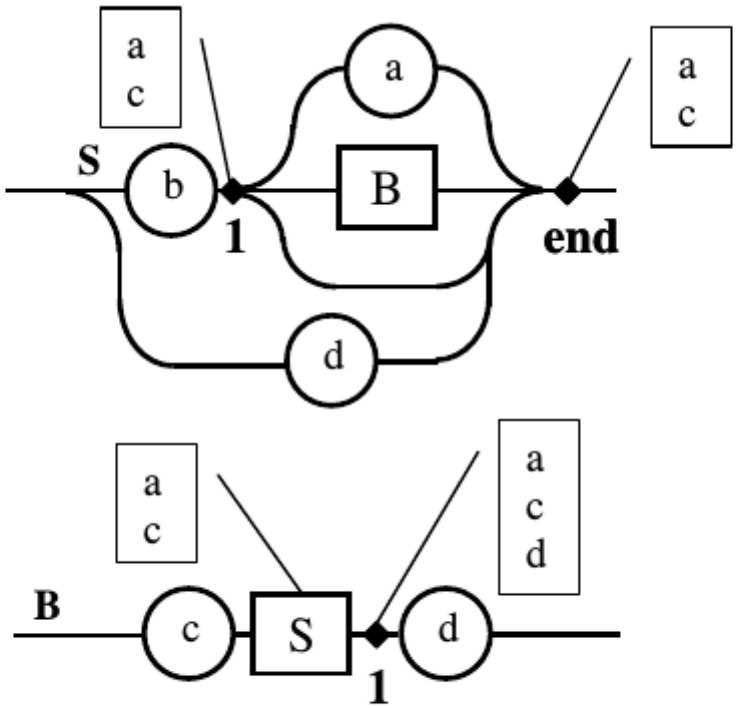
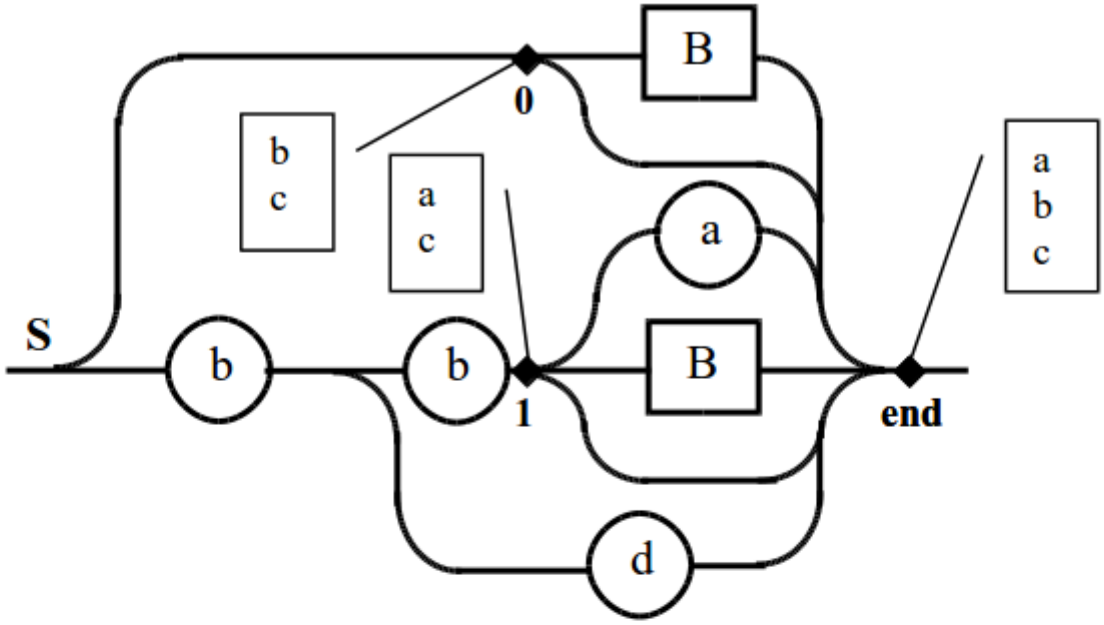


Рис. 9.13. Избавление от сквозной связи в нетерминале B ведет к появле
пустой связи на конце S, и практически не изменяет анализ правил

Возможны и более сложные комбинации, например иерархическое вложение правил с пустыми связями на конце. Их число которых можно сократить предварительным преобразованием диаграмм. Однако следует остановиться на примере, демонстрирующем наличие нескольких пустых связей, выходящих из разных точек и соединяющихся на конце диаграммы Вирта (рис. 9.14). Особенностью таких диаграмм является то, что внутри них множества терминалов, сформированные в различных альтернативных точках, могут пересекаться при их концентрации в конце диаграммы. Но это не влияет на разбор, так как различные альтернативы не связаны друг с другом. Результирующее множество, передаваемое в этом случае во внешние альтернативные точки, определяется путем объединения всех терминальных символов, сконцентрированных на конце правила.



$$\{b, c\} \cup \{a, c\} = \{a, b, c\}$$

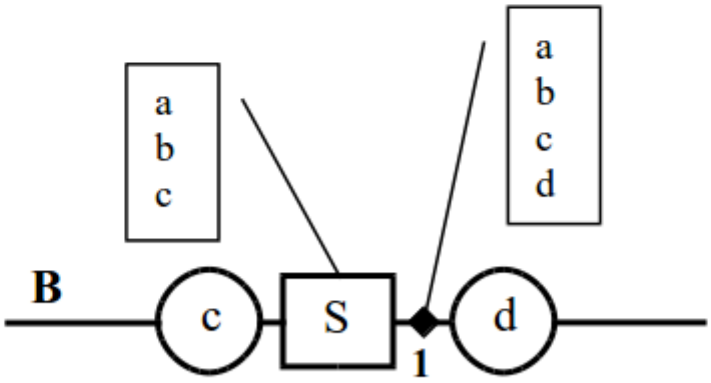


Рис. 9.14. Наличие нескольких пустых связей концентрирует на конце прави
объединенное множество терминалов, используемое для анализа альтернати
других правилах

Пример непосредственного доказательства

Докажем принадлежность $\mathbf{G}_{7.7}$ к КС(1)-грамматике. Для этого, на шаге 1, построим соответствующую диаграмму Вирта (9.15). Последующие действия заключаются в разметке альтернатив на построенных диаграммах. Не вызывает проблем разметка правила **C**. В нем анализ альтернатив присутствует только на входе (точка 0). При этом возможные значения входных символов {e, a} не совпадают. В правиле **B** существует сквозная связь, что необходимо учесть при разметке других правил. Кроме этого, сразу видно, что оно начинается с терминала “c” (точка 0). Следует так же отметить, что правило **B** располагается в **A** таким образом, что соединяет его вход и выход. Это позволяет считать, что и правило **A** содержит сквозную связь.

1. $S \rightarrow AbB$
2. $S \rightarrow d$
3. $A \rightarrow CAb$
4. $A \rightarrow B$
5. $B \rightarrow cSd$
6. $B \rightarrow \varepsilon$
7. $C \rightarrow a$
8. $C \rightarrow ed$

Шаг 1 $\Rightarrow$

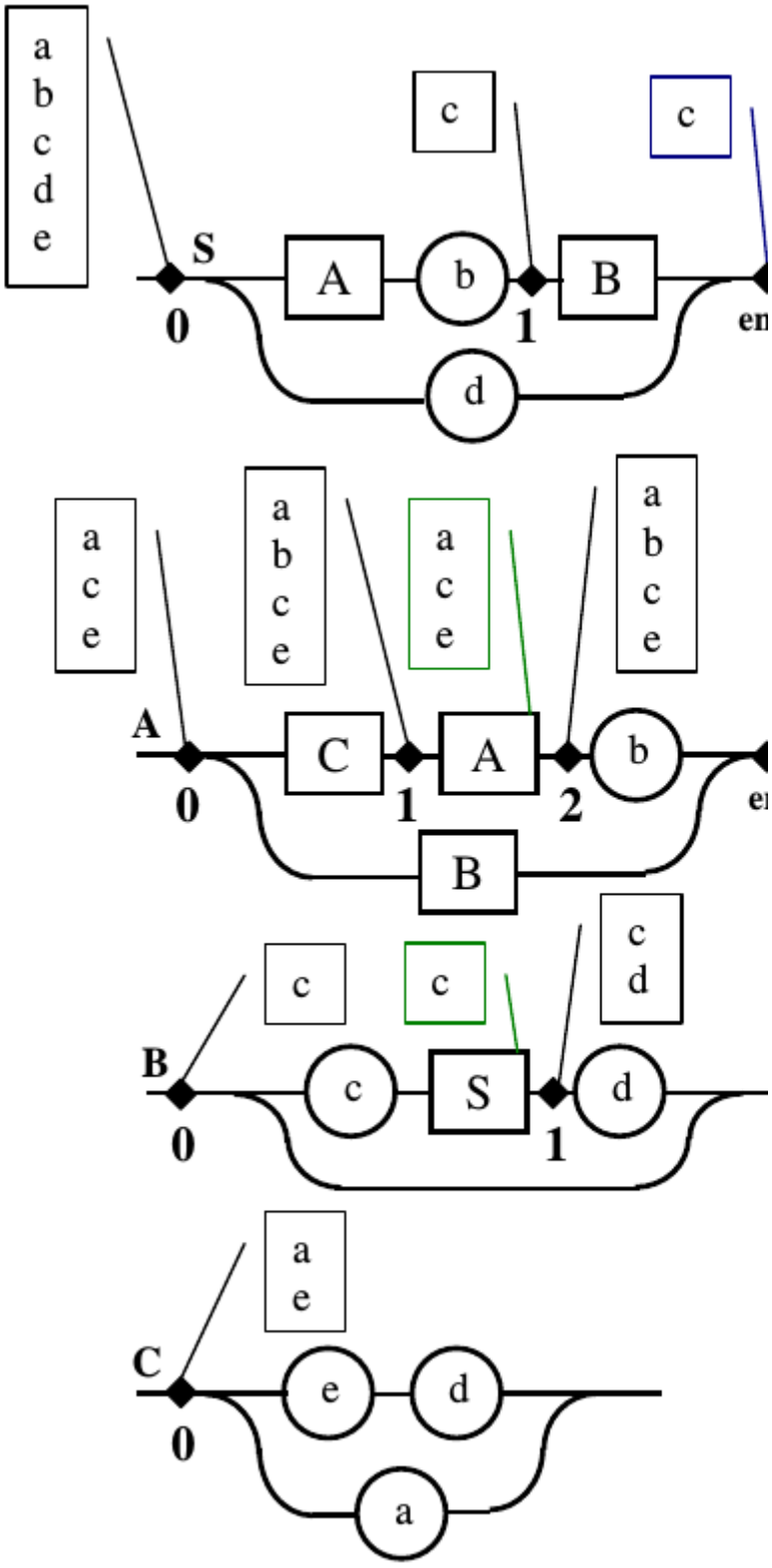


Рис. 9.15. Непосредственное доказательство принадлежности $G_{7.7}$ к КС(1

В точке 0 правила **A** альтернативные терминалы определяются нетерминалами **B** и **C**, образуя множество $A_0 = \{a, c, e\}$. Учитывая наличие сквозной связи, это множество можно сконцентрировать на конце правила, чтобы использовать его при анализе альтернатив, следующих за **A** в других диаграммах. В точке 1 правила A будет образовано множество альтернативных терминалов $A_1 = \{a, b, c, e\}$, что является перетеканием терминала b по

сквозной связи, существующей в **A**. Точно такое же множество будет и в точке 2 этого правила, так как альтернатива с терминалом b будет конкурировать с терминалами, сконцентрировавшимися на конце правила **A**.

В точке 0 правила **S** множество терминалов S_0 = {a, b, c, d, e}. Оно определяется множеством терминалов, возможных в точке 0 правила **A**, альтернативой d и терминалом b, «перетекшим» через сквозную связь, существующую в **A**, но реально определенную в **B**.

Интерес также представляет точка 1 правила **S**, так как следующий после нее нетерминал **B** имеет сквозную связь и располагается на конце **S**. Множество альтернатив в ней определяется только терминалом "**c**", который и концентрируется на конце **S**. Это учитывается в точке 1 правила **B**, которая следует после **S**.

Рассмотрев множество всевозможных альтернативных точек, мы не нашил ни одной, в которой оказались бы совпадающие терминалы. Следовательно, представленные правила определяют КС(1)-грамматику.

Доказательство с использованием предварительного преобразования диаграмм Вирта

Вместо того, чтобы сразу доказывать принадлежность диаграмм к КС(1) грамматикам, можно попытаться осуществить их преобразование к более простому виду. Пример подобного подхода, для уже рассмотренной грамматики $\mathbf{G_{7.5}}$, приведен на рис. 9.16. Первый шаг, связанный с построением исходных диаграмм представлен на рис 9.15. Шаги 2 и 3 посвящаем последовательному избавлению от пустых связей. После этого проанализируем альтернативные точки ветвления полученных правил. В точке 0 правила **C** , как и ранее, возможными альтернативами являются "a" и "e" (C_0 = {a, e}). В точке 0 правила **A** возможны альтернативные нетерминалы **C** и **B**, причем первый из них начинается с символов "a" и "e", а второй – с "c". Следовательно, A_0 = {a, c, e}. Сразу видно, что в точке 1 правила **A** возможны терминалы "a", "e", "c", "b" (A_1 = {a, b, c, e}). Таким образом, правило **A** удовлетворяет условиям принадлежности к КС(1)-грамматике. В точке 0 правила **S** для альтернативных связей первыми являются непересекающиеся символы "a", "e", "c", "b", "d" (S_0 = {a, b, c, d, e}). Также в точке 1 правила **S** существует пустая связь, которая ведет в конец диаграммы. То есть, необходимо сконцентрировать на конце **S** возможные терминалы и учесть их при анализе альтернативных точек, следующих за **S** в других правилах. К множеству концентрируемых терминалов относится толь "**c**", расположенный в начале правила **B**. Нетерминал **S** встречается лишь внутри **B**, а вслед за ним (точка 1) идет терминал "**d**". Следовательно, все корректно. Так как пустые правила отсутствуют и альтернатив на конце других правил тоже нет, дальнейший разбор полетов по этой ветви можно прекратить.

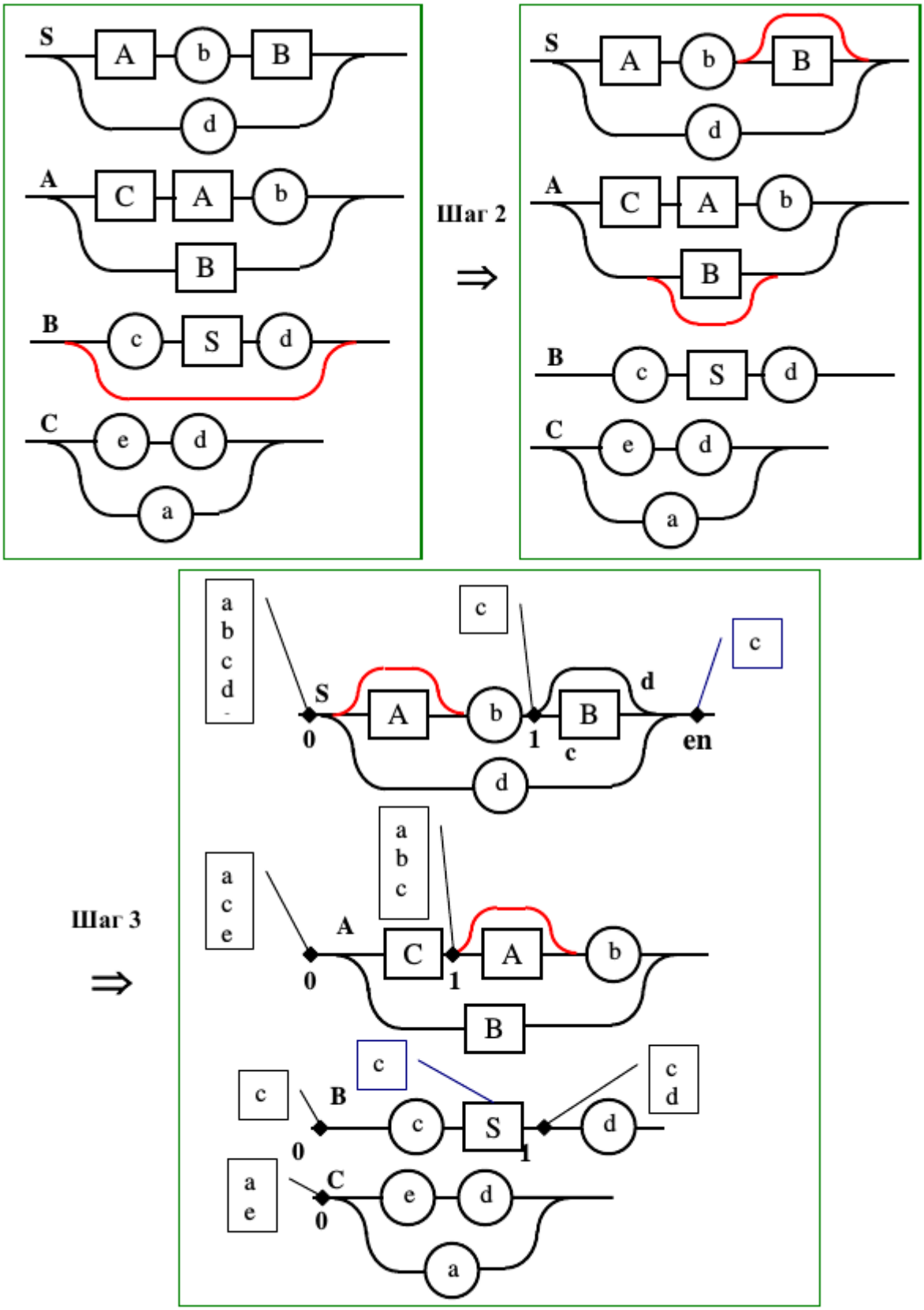
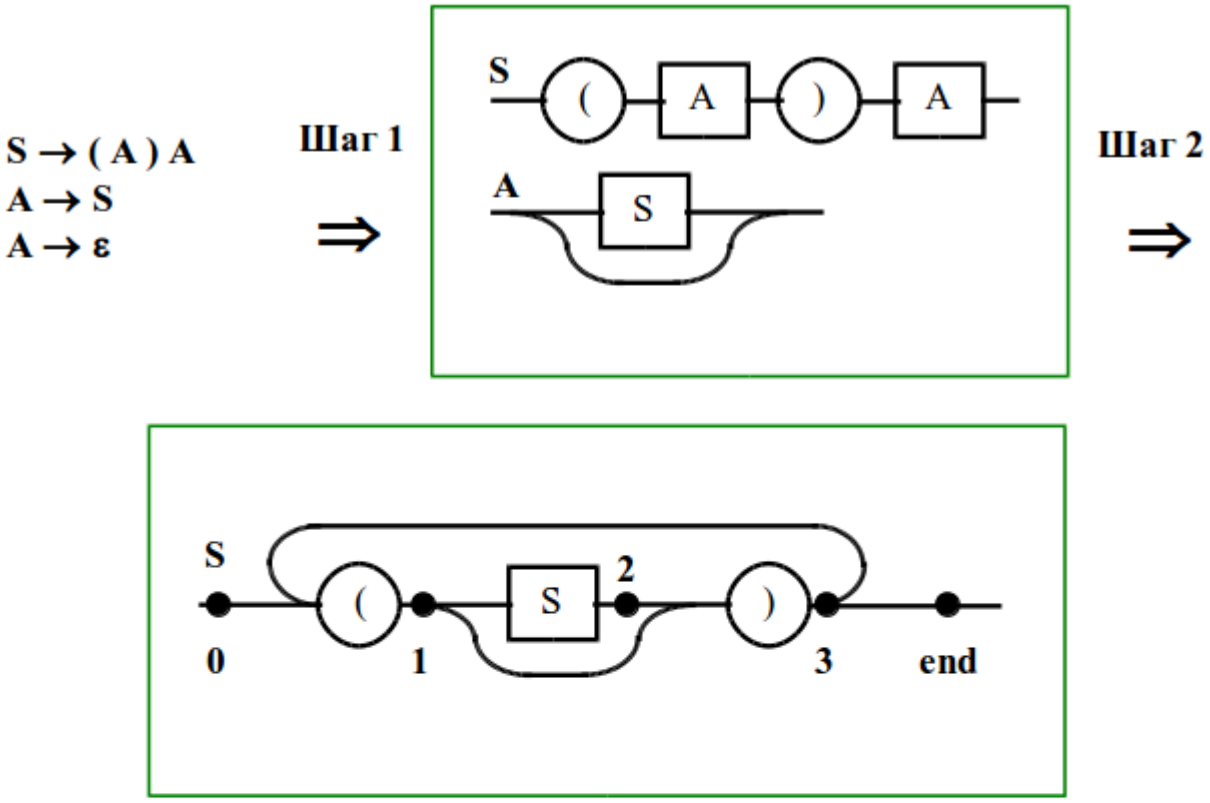


Рис. 9.16. Доказательство принадлежности $\mathbf{G}_{7.7}$ к КС(1) грамматике с использованием предварительного преобразования правил

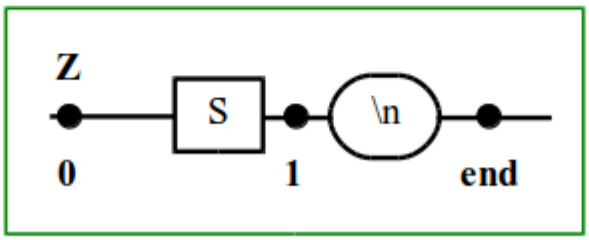
Проведенный анализ показывает принадлежность рассматриваемой грамматики к КС(1) типу. На мой взгляд, преобразование диаграмм позволило упростить доказательство. Нисходящий распознаватель на основе иерархически порождаемых конечных автоматов можно построить по любому варианту полученных ДВ (я обычно предпочитаю последний вариант, так как проведенные преобразования приводят к более простым для восприятия правилам).

В последний раз о распознавателе вложенности круглых скобок

Теперь у нас есть вся необходимая информация для построения распознавателя вложенности круглых скобок еще одним способом. За основу возьмем грамматику $\mathbf{G}_{7.1}$. Представим исходные правила диаграммами Вирта и проведем эквивалентные преобразования (рис. 9.17). Правила, полученные после третьего шага, будем использовать для программной реализации автомата методом рекурсивного спуска.



а) диаграмма Вирта для исходной грамматики



б) дополнительная диаграмма, учитывающая концевой маркер

Рис. 9.17. Построение диаграмм Вирта для программной реализации распознавания вложенности круглы по грамматике $\mathbf{G}_{7.1}$.

Добавим в полученные правила концевой маркер, еще ранее обозначенный символом перевода строки (\n), В предыдущих реализациях он добавлялся на этапе построения автомата с магазинной памятью. Так как мы строим иерархически порождаемый конечный автомат сразу же по ДВ, то этот символ необходимо включить в правила. В противном случае к концу правильной входной цепочки можно будет приписать все, что угодно, в том числе и круглые скобки.

Программная реализация автомата **S** строится непосредственно по диаграмме Вирта с использованием техники, используемой ранее в модуле лексического анализа. То есть, я продолжаю использовать **goto**, чтобы сохранить однозначное соответствие между

диаграммами и программой. Все служебные функции этой программы полностью взяты из предшествующих версий.

Однако одного правила **S** не хватает для корректной реализации разбора скобочных выражений. Дело в том, что используемые ранее автоматы с магазинной памятью автоматически добавляли концевой маркер, отсутствующий в исходных правилах. А без него, вслед за правильной цепочкой могут идти любые символы, нарушающие целостное восприятие анализируемой строки. Особенно, если этими символами вновь являются скобки. Поэтому необходимо построить дополнительную диаграмму **Z**, учитывающую концевой маркер. К сожалению, его нельзя непосредственно подставить в конце правила S, так как оно рекурсивно обращается к себе, и поэтому будет требовать маркер в середине скобочного выражения. Чтобы не нарушать целостного восприятия, текст программы вновь приводится полностью.

```
//
// BrackRecVirth.cpp - рекурсивный распознаватель вложенности круглых скобок.
// Построен на основе диаграмм Вирта, эквивалентных следующей РБНФ
//
// $S = {/ "(" [ S ] ")" /}.
//

#include
#include
#include
#include

using namespace std;

string str; // Строка с входной цепочкой, имитирующая входную ленту
int i;  // Текущее положение входной головки

int erFlag; // Флаг, фиксирующий наличие ошибок в середине правила

// Функция, реализующая чтение символов в буфер из входного потока.
// Используется для ввода с клавиатуры распознаваемой строки.
// Ввод осуществляется до нажатия на клавишу Enter.
// Символ '\n' является концевым маркером входной строки.
void GetOneLine(istream &is, string &str) {
  char ch;
  str = "";
  for(;;) {
    is.get(ch);
    if(is.fail() || ch == '\n') break;
    str += ch;
  }
  str += '\n'; // Добавляется концевой маркер
}

// Функция, реализующая распознавание нетерминала S.
bool S() {
//_0: // Начало диаграммы
  if(str[i] == '(') {
    i++;
    goto _1;
  }
  return false;// Первый символ цепочки некорректен
```

```
      // что это ошибка, лучше определить снаружи
  _1: // Точка 1 диаграммы
    if(str[i] == ')') {
      i++;
      goto _3;
    }
    if(S()) {
      goto _2;
    }
    erFlag++;
    cout  << "Position " << i << ", "
          << "Error 1: I want closed bracket or next opened bracket!\n";
    return false;
  _2: // Точка 2 диаграммы
    if(str[i] == ')') {
      i++;
      goto _3;
    }
    erFlag++;
    cout  << "Position " << i << ", "
          << "Error 2: I want closed bracket!\n";
    return false;
  _3: // Точка 3 диаграммы
    if(str[i] == '(') {
      i++;
      goto _1;
    }
    goto _end;
  _end: // Точка end диаграммы
  return true;
}

// Функция, реализующая распознавание нетерминала Z.
bool Z() {
//_0: // Начало диаграммы
  if(S()) goto _1;
  return false;// Первый символ цепочки некорректен
      // что это ошибка, лучше определить снаружи
  _1: // Точка 1 диаграммы
    // За последней скобкой должен быть "конец строки"
  if(str[i] == '\n')
  {
    goto _end; // Все прошло нормально
  }
  erFlag++;
  cout << "Position " << i << ", "
       << "Error 3: I want end of line!\n";
  return false;
  _end:
  return true;
}

// Функция запускающая разбор и определяющая корректность его завершения,
// если первый символ не принадлежит цепочки
bool Parser() {
  // Начальная инициализация.
  erFlag = 0;
  i = 0;

  // Процесс пошел!
  if(Z())
```

```
  {
    return true; // Все прошло нормально
  }
  else {
    if(erFlag)
      cout  << "Position " << i << ", "
        << "Error 4: Internal Error!\n";
    else
      cout  << "Position " << i << ", "
        << "Error 5: Incorrect first symbol of S!\n";
    return false; // Есть ошибки
  }
}

// Главная функция используется для тестирования до тех пор,
// пока не будет прочитана пустая строка
int main () {
  string strCursor;
  str = "";
  // Цикл распознавания различных входных цепочек
  do {
    // Чтение очередной входной цепочки в буфер
    cout << "Input bracket\'s expression!: ";

    // Формируем очередную строку скобок для распознавания.
    GetOneLine(cin, str);

    // Здесь начинается разбор принятой строки.
    if(Parser())
      cout << "+++++ OK! +++++\n";
    else
      cout << "----- Fatal error (look upper error message)! -----\n";

    // Вывод разобранной строки и значения позиции входой головки.
    cout << "Line: " << str;
    strCursor = " Pos: " + string(i, '-');
    strCursor +='^';
    cout << strCursor << "  i = " << i << "\n\n";

  } while(str != "\n");
  cout << "Goodbye!\n";
  return 1;
}
```

Исходные тексты данного примера размещены в архиве.

Если у Вас есть желание посмотреть, к чему может привести отсутствие концевого маркера, замените в функции **Parser** вызов **Z** на **S**.

Контрольные вопросы и задания

1. В чем проявляются основные преимущества диаграмм Вирта при разработке распознавателей?
2. Какие задачи диаграммы Вирта позволяют решать проще?
3. Что дает избавление от рекурсий?
4. Каким образом можно осуществлять упрощение правил?
5. Каким образом можно перейти к КС(n-1) грамматике от КС(n) грамматики.

6. Для чего можно использовать освобождение от пустых правил?
7. Какие проблемы возможны при освобождении от пустых правил?
8. Как освобождаться от пустых правил при иерархическом возникновении сквозных связей?
9. Можно ли использовать исходные правила с левой рекурсией при нисходящем разборе?
10. Каким образом можно использовать правила с левой рекурсией в нисходящем разборе?
11. Каким образом можно проверить две грамматики на эквивалентность?
12. Можно ли с помощью синтаксиса задавать семантику?
13. Как синтаксически задать приоритет арифметических операций?
14. Как синтаксически задать порядок выполнения операций?
15. Каким образом доказывается принадлежность к КС(1) грамматике?
16. Как можно упростить доказательство принадлежности к КС(1) грамматике?
17. Протестируйте предложенную выше программу распознавания вложенности круглых скобок. Опишите, что произойдет, если в функции **Parser** заменить вызов **Z** на **S**.