Синтаксические анализаторы

Основные принципы работы синтаксических анализаторов

Назначение синтаксических анализаторов

Синтаксический анализатор (синтаксический разбор) — это часть компилятора, которая отвечает за выявление и проверку синтаксических конструкций входного языка. В задачу синтаксического анализа входит:

- найти и выделить синтаксические конструкции в тексте исходной программы;
- установить тип и проверить правильность каждой синтаксической конструкции;
- представить синтаксические конструкции в виде, удобном для дальнейшей генерации текста результирующей программы.

Синтаксический анализатор — это основная часть компилятора на этапе анализа. Без выполнения синтаксического разбора работа компилятора бессмысленна, в то время как лексический разбор, в принципе, является необязательной фазой. Все задачи по проверке синтаксиса входного языка могут быть решены на этапе синтаксического разбора. Лексический анализатор только позволяет избавить сложный по структуре синтаксический анализатор от решения примитивных задач по выявлению и запоминанию лексем исходной программы.

Синтаксический анализатор воспринимает выход лексического анализатора и разбирает его в соответствии с грамматикой входного языка. Однако в грамматике входного языка программирования обычно не уточняется, какие конструкции следует считать лексемами. Примерами конструкций, которые обычно распознаются во время лексического анализа, служат ключевые слова, константы и идентификаторы. Но эти же конструкции могут распознаваться и синтаксическим анализатором. На практике не существует жесткого правила, определяющего, какие конструкции должны распознаваться на лексическом уровне, а какие надо оставлять синтаксическому анализатору. Обычно это определяет разработчик компилятора, исходя из технологических аспектов программирования, а также синтаксиса и семантики входного языка. Принципы взаимодействия лексического и синтаксического анализаторов были рассмотрены ранее в главе «Лексические анализаторы».

В основе синтаксического анализатора лежит распознаватель текста исходной программы, построенный на основе грамматики входного языка. Как правило, синтаксические конструкции языков программирования могут быть описаны с помощью КС-грамматик, реже встречаются языки, которые могут быть описаны с помощью регулярных грамматик.

ПРИМЕЧАНИЕ

> Чаще всего регулярные грамматики применимы к языкам ассемблера, а синтаксис языков высокого уровня основан на грамматике КС-языков.

Главную роль в том, как функционирует синтаксический анализатор и какой алгоритм лежит в его основе, играют принципы построения распознавателей для КС-языков. Без применения этих принципов невозможно выполнить эффективный синтаксический разбор предложений входного языка.

Распознавателями для КС-языков являются автоматы с магазинной памятью — МП-автоматы — односторонние недетерминированные распознаватели с линейно-ограниченной магазинной памятью (классификация распознавателей приведена в соответствующем разделе главы «Формальные языки и грамматики»). Поэтому важно рассмотреть, как функционирует МП-автомат и как для КС-языков решается задача разбора — построение распознавателя языка на основе заданной грамматики. Ниже в этой главе рассмотрены технические аспекты, связанные с реализацией синтаксических анализаторов.

Автоматы с магазинной памятью (МП-автоматы)

Определение МП-автомата

Контекстно-свободными (КС) называются языки, определяемые грамматиками типа G(**VT**,**VN**,**P**,S), в которых правила **P** имеют вид: $A \to \beta$, где $A \in \mathbf{VN}$ и $\beta \in \mathbf{V}^*$, $\mathbf{V}=\mathbf{VT} \cup \mathbf{VN}$. Распознавателями КС-языков служат автоматы с магазинной памятью (*МП-автоматы*).

В общем виде МП-автомат можно определить следующим образом:

R(**Q**,**V**,**Z**,δ,q_0,z_0,**F**), где

Q — множество состояний автомата;

V — алфавит входных символов автомата;

Z — специальный конечный алфавит магазинных символов автомата, $\mathbf{V} \subseteq \mathbf{Z}$;

δ — функция переходов автомата, которая отображает множество $\mathbf{Q} \times (\mathbf{V} \cup \{\lambda\}) \times \mathbf{Z}$ на конечное множество подмножеств $P(\mathbf{Q} \times \mathbf{Z}^*)$;

$q_0 \in \mathbf{Q}$ — начальное состояние автомата;

$z_0 \in \mathbf{Z}$ — начальный символ магазина;

$\mathbf{F} \subseteq \mathbf{Q}$ — множество конечных состояний.

МП-автомат в отличие от обычного КА имеет стек (магазин), в который можно помещать специальные «магазинные» символы (обычно это терминальные и нетерминальные символы грамматики языка). Переход из одного состояния в другое зависит не только от входного символа, но и от символа на верхушке стека. Таким образом, конфигурация автомата определяется тремя параметрами: состоянием автомата, текущим символом входной цепочки (положением указателя в цепочке) и содержимым стека.

МП-автомат условно можно представить в виде схемы на рис. 4.1.

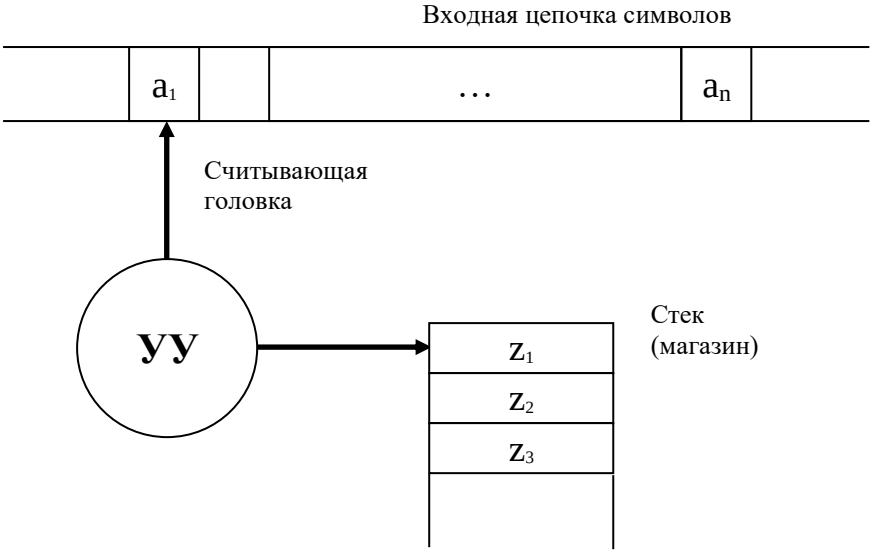


Рис. 4.1. Общая условная схема автомата с магазинной памятью (МП-автомата)

Конфигурация МП-автомата описывается в виде тройки $(q,\alpha,\omega)\in\mathbf{Q}\times\mathbf{V}^*\times\mathbf{Z}^*$, которая определяет текущее состояние автомата q, цепочку еще непрочитанных символов α на входе автомата и содержимое магазина (стека) ω. Вместо α в конфигурации можно указать пару (β,n), где $\beta\in\mathbf{V}^*$ — вся цепочка входных символов, а $n\in\mathbf{N}\cup\{0\}$, $n\geq 0$ — положение считывающего указателя в цепочке.

Тогда один такт работы автомата можно описать в виде $(q,a\alpha,z\omega) \div (q',\alpha,\gamma\omega)$, если $(q',\gamma)\in\delta(q,a,z)$, где: $q,q'\in\mathbf{Q}$, $a\in\mathbf{V}\cup\{\lambda\}$, $\alpha\in\mathbf{V}^*$, $z\in\mathbf{Z}$, $\gamma,\omega\in\mathbf{Z}^*$. При выполнении такта (перехода) из стека удаляется верхний символ, соответствующий условию перехода, и добавляется цепочка, соответствующая правилу перехода. Первый символ цепочки становится верхушкой стека. Допускаются переходы, при которых входной символ игнорируется (и, тем самым, он будет входным символом при следующем переходе). Эти переходы (такты) называются *λ-переходами* (λ-тактами).

Начальная конфигурация МП-автомата, очевидно, определяется как (q_0,α,z_0), $\alpha\in\mathbf{V}^*$. Множество конечных конфигураций автомата — (q,λ,ω), $q\in\mathbf{F}$, $\omega\in\mathbf{Z}^*$.

МП-автомат допускает (принимает) цепочку символов, если, получив эту цепочку на вход, он может перейти в одну из конечных конфигураций — когда при окончании цепочки автомат находится в одном из конечных состояний, а стек пуст. Иначе цепочка символов не принимается.

Язык, определяемый МП-автоматом — это множество всех цепочек символов, которые допускает данный автомат. Язык, определяемый МП-автоматом **R**, обозначается как L(**R**). Два МП-автомата называются эквивалентными, если они определяют один и тот же язык: $\mathbf{R}_1\equiv\mathbf{R}_2 \Leftrightarrow L(\mathbf{R}_1)=L(\mathbf{R}_2)$.

МП-автомат допускает цепочку символов с опустошением магазина, если при окончании разбора цепочки автомат находится в одном из конечных состояний, а стек пуст — конфигурация (q,λ,λ), $q\in\mathbf{F}$. Если язык задан МП-автоматом **R**, который допускает цепочки с опустошением стека, это обозначается так: $L_\lambda(\mathbf{R})$. Доказано, что для любого МП-автомата всегда можно построить эквивалентный ему МП-автомат, допускающий цепочки заданного языка с опустошением стека [4 т.1, 5]. То есть $\forall$ МП-автомата **R**: $\exists$ МП-автомат **R'**, такой, что: $L(\mathbf{R})=L_\lambda(\mathbf{R}')$.

Расширенные МП-автоматы

Кроме обычного МП-автомата существует также понятие расширенного МП-автомата.

Расширенный МП-автомат может заменять цепочку символов конечной длины в верхней части стека на другую цепочку символов конечной длины, в отличие от обычного МП-автомата. В отличие от обычного МП-автомата, который на каждом такте работы может изымать из стека только один символ, расширенный МП-автомат может изымать за один такт цепочку символов, находящуюся на вершине стека. Функция переходов δ для расширенного МП-автомата отображает множество $\mathbf{Q}\times(\mathbf{V}\cup\{\lambda\})\times\mathbf{Z}^*$ на конечное множество подмножеств $P(\mathbf{Q}\times\mathbf{Z}^*)$.

Доказано, что для любого расширенного МП-автомата всегда можно построить эквивалентный ему обычный МП-автомат (обратное утверждение очевидно, так как любой обычный МП-автомат является и расширенным МП-автоматом) [4 т.1, 5]. Таким образом, классы МП-автоматов и расширенных МП-автоматов эквивалентны и задают один и тот же тип языков.

Доказано, что для произвольной КС-грамматики всегда можно построить МП-автомат, распознающий заданный этой грамматикой язык [4 т.1, 5, 15]. Поэтому можно говорить, что МП-автоматы распознают КС-языки. Существует также доказательство того, что для произвольного МП-автомата всегда можно построить КС-грамматику, которая задает язык, распознаваемый этим автоматом. Таким образом, КС-грамматики и МП-автоматы задают один и тот же тип языков — КС-языки.

Поскольку класс расширенных МП-автоматов эквивалентен классу обычных МП-автоматов и задает тот же самый тип языков, то можно утверждать, что и расширенные МП-автоматы распознают языки из типа КС-языков. Следовательно, язык, который может распознавать расширенный МП-автомат, также может быть задан с помощью КС-грамматики.

Детерминированные МП-автоматы

МП-автомат называется *детерминированным*, если из каждой его конфигурации возможно не более одного перехода в следующую конфигурацию. В противном случае МП-автомат называется недетерминированным.

Формально для детерминированного МП-автомата (ДМП-автомата) $R(\mathbf{Q},\mathbf{V},\mathbf{Z},\delta,q_0,z_0,\mathbf{F})$ функция переходов δ может $\forall q\in\mathbf{Q}$, $\forall a\in\mathbf{V}$, $\forall z\in\mathbf{Z}$ иметь один из следующих трех видов:

1. $\delta(q,a,z)$ содержит один элемент: $\delta(q,a,z)=\{(q',\gamma)\}$, $\gamma\in\mathbf{Z}^*$, и $\delta(q,\lambda,z)=\varnothing$;
2. $\delta(q,a,z)=\varnothing$ и $\delta(q,\lambda,z)$ содержит один элемент: $\delta(q,\lambda,z)=\{(q',\gamma)\}$, $\gamma\in\mathbf{Z}^*$;
3. $\delta(q,a,z)=\varnothing$ и $\delta(q,\lambda,z)=\varnothing$.

Класс ДМП-автоматов и соответствующих им языков значительно уже, чем весь класс МП-автоматов и КС-языков.

ВНИМАНИЕ

В отличие от КА, когда для любого недетерминированного КА можно построить эквивалентный ему детерминированный КА, не для каждого МП-автомата можно построить эквивалентный ему ДМП-автомат. Иными словами, в общем случае невозможно преобразовать недетерминированный МП-автомат в детерминированный.

Детерминированные МП-автоматы определяют очень важный класс среди всех КС-языков, называемый детерминированными КС-языками (ДКС-языками). Доказано, что все языки, принадлежащие к классу ДКС-языков, могут быть построены с помощью однозначных КС-грамматик. Поскольку однозначность — это важное и обязательное требование в грамматике любого языка программирования, ДМП-автоматы представляют особый интерес для создания компиляторов.

ПРИМЕЧАНИЕ

Любой ДКС-язык может быть задан однозначной КС-грамматикой, но обратное неверно: не всякий язык, заданный однозначной КС-грамматикой является детерминированным. Например, однозначная грамматика G({a,b}, {S,A,B}, {S→A|B, A→aAb|ab, B→aBbb|abb}, S) задает КС-язык, который не является детерминированным.

Все без исключения синтаксические конструкции языков программирования задаются с помощью однозначных КС-грамматик. Следовательно, синтаксические структуры этих языков относятся к классу ДКС-языков и могут распознаваться с помощью ДМП-автоматов. Поэтому большинство распознавателей, которые будут рассмотрены далее, относятся к классу ДКС-языков.

Построение синтаксических анализаторов

Проблемы построения синтаксических анализаторов

Синтаксический анализатор должен распознавать весь текст исходной программы. Поэтому, в отличие от лексического анализатора, ему нет необходимости искать границы распознаваемой строки символов. Он должен воспринимать всю информацию, поступающую ему на вход, и либо подтвердить ее принадлежность входному языку, либо сообщить об ошибке в исходной программе.

Но, как и в случае лексического анализа, задача синтаксического анализа не ограничивается только проверкой принадлежности цепочки заданному языку. Необходимо оформить найденные синтаксические конструкции для дальнейшей генерации текста результирующей программы. Синтаксический анализатор должен иметь некий выходной язык, с помощью которого он передает следующим фазам компиляции информацию о найденных и разобранных синтаксических структурах. В таком случае он уже является не разновидностью МП-автомата, а преобразователем с магазинной памятью — МП-преобразователем [3, 4 т.1, 5, 15].

ПРИМЕЧАНИЕ

> Во всех рассматриваемых далее примерах результатом работы МП-автомата являлся не только ответ на вопрос о принадлежности входной цепочки заданному языку («да» или «нет»), но и последовательность номеров правил грамматики, использованных для построения входной цепочки. Затем на основе этих правил строятся цепочка вывода и дерево вывода. Поэтому, строго говоря, все рассмотренные примеры МП-автоматов являются не только МП-автоматами, но и МП-преобразователями.

Вопросы, связанные с представлением информации, являющейся результатом работы синтаксического анализатора, и с порождением на основе этой информации текста результирующей программы рассмотрены в следующей главе, поэтому здесь на них останавливаться не будем.

Однако, проблемы, связанные с созданием синтаксического анализатора, не ограничиваются только двумя перечисленными, как это было для лексического анализа. Дело в том, что процесс построения синтаксического анализатора гораздо сложнее аналогичного процесса для лексического анализатора. Так происходит, поскольку КС-грамматики и МП-автоматы, лежащие в основе синтаксического анализа, сложнее, чем регулярные грамматики и КА, лежащие в основе лексического анализа.

Алгоритм создания МП-автомата на основе произвольной КС-грамматики прост [4 т.1, 5, 15], но если напрямую воспользоваться этим алгоритмом, то может оказаться, что работу полученного МП-автомата невозможно будет промоделировать на компьютере. Следовательно, такой автомат практического применения в компиляторе иметь не может. Но даже если реализовать распознаватель на основе МП-автомата удастся, может оказаться, что время его работы непомерно велико и, в худшем случае, экспоненциально зависит от длины входной цепочки.

С этими проблемами столкнулись в свое время разработчики первых компиляторов с языков высокого уровня. Для их решения были предприняты определенные исследования в теории формальных языков и, в частности, в области КС-языков.

Во-первых, очевидно, что поскольку любой язык программирования должен быть задан однозначной КС-грамматикой, а распознавателями для КС-языков, заданных однозначными КС-грамматиками являются ДМП-автоматы, то нужно стремится построить синтаксический анализатор на основе ДМП-автомата. Моделировать работу ДМП-автомата существенно проще, чем работу произвольного МП-автомата.

Во-вторых, было установлено, что существуют преобразования правил КС-грамматик, выполнив которые можно привести грамматику к такому виду, когда построение и моделирование работы МП-автомата, распознающего язык, заданный грамматикой, существенно упрощаются. Были найдены и описаны специальные формы правил КС-грамматик, называемые «нормальными формами», для которых построены соответствующие МП-автоматы [4 т.1, 5, 15]. Преобразовав произвольную КС-грамматику к одной из известных нормальных форм можно сразу же построить соответствующий МП-автомат (нормальные формы КС-грамматик в данном учебнике не рассматриваются).

Наконец, было найдено множество различных классов КС-грамматик, для которых возможно построить ДМП-автомат, имеющий линейную зависимость времени разбора входной цепочки от ее длины. Такие распознаватели для КС-языков

называются *линейными распознавателями*. Часть таких классов КС-грамматик будет рассматриваться далее в этой главе.

В принципе, было бы достаточно знать хотя бы один такой класс, если бы для КС-грамматик были разрешимы проблема преобразования и проблема эквивалентности. Но поскольку, в общем случае, это не так, то одним классом КС-грамматик, для которого существуют линейные распознаватели, ограничиться не удается. По этой причине для всех классов КС-грамматик существует принципиально важное ограничение: в общем случае, невозможно преобразовать произвольную КС-грамматику к виду, требуемому данным классом КС-грамматик, либо же доказать, что такого преобразования не существует. То, что проблема неразрешима в общем случае, не говорит о том, что она не решается в каждом конкретном частном случае, и зачастую удается найти такие преобразования. И чем шире набор классов КС-грамматик с линейными распознавателями, тем проще их искать.

Среди универсальных распознавателей лучшими по эффективности являются табличные, функционирование которых построено на иных принципах, нежели моделирование работы МП-автомата [4 т.1, 5, 15]. Табличные распознаватели обладают полиномиальными характеристиками требуемых вычислительных ресурсов в зависимости от длины входной цепочки: для КС-языков, заданных однозначной грамматикой, удается добиться квадратичной зависимости, для всех прочих КС-языков — кубической зависимости. Но в данном учебнике табличные распознаватели не рассматриваются, поскольку они достаточно сложны, требуют значительного объема памяти (объем необходимой памяти квадратично зависит от длины входной цепочки), но при этом не позволяют добиться практически значимых результатов (квадратичная зависимость требуемых вычислительных ресурсов от длины входной цепочки неприемлема для компилятора).

Варианты синтаксических анализаторов

Построение синтаксического анализатора — это более творческий процесс, чем построение лексического анализатора. Этот процесс не всегда может быть полностью формализован.

Имея грамматику входного языка, разработчик синтаксического анализатора должен в первую очередь выполнить ряд формальных преобразований над этой грамматикой, облегчающих построение распознавателя. После этого он должен проверить, подпадает ли полученная грамматика под один из известных классов КС-языков, для которых существуют линейные распознаватели. Если такой класс найден, можно строить распознаватель (если найдено несколько классов — выбрать тот, для которого построение распознавателя проще, либо построенный распознаватель обладает лучшими характеристиками). Если же такой класс КС-языков найти не удалось, то разработчик должен попытаться выполнить над грамматикой некоторые преобразования, чтобы привести ее к одному из известных классов. Вот эти преобразования не могут быть описаны формально и в каждом конкретном случае разработчик должен попытаться найти их сам. Иногда преобразования имеет смысл искать даже в том случае, когда грамматика подпадает под один из известных классов КС-языков, с целью найти другой класс, для которого можно построить лучший по характеристикам распознаватель.

Только в том случае, когда в результате всех этих действий не удалось найти соответствующий класс КС-языков, разработчик вынужден строить универсальный распознаватель. Характеристики такого распознавателя будут существенно хуже, чем у линейного распознавателя — в лучшем случае, удается достичь квадратичной зависимости времени работы распознавателя от длины входной цепочки. Такие случаи бывают редко, поэтому все современные компиляторы построены на основе линейных распознавателей (иначе время их работы было бы недопустимо велико).

Для каждого класса КС-языков существует свой класс распознавателей, но все они функционируют на основе общих принципов, на которых основано моделирование работы МП-автоматов. Все распознаватели для КС-языков можно разделить на две большие группы: нисходящие и восходящие.

Нисходящие распознаватели просматривают входную цепочку символов слева направо и порождают левосторонний вывод. При этом получается, что дерево вывода строится таким распознавателем от корня к листьям (сверху вниз), откуда и происходит название распознавателя.

Восходящие распознаватели также просматривают входную цепочку символов слева направо, но порождают при этом правосторонний вывод. Дерево вывода строится восходящим распознавателем от листьев к корню (снизу вверх), откуда и происходит название распознавателя.

Для моделирования работы этих двух групп распознавателей используются два алгоритма: алгоритм с подбором альтернатив — для нисходящих распознавателей и алгоритм «сдвиг-свертка» — для восходящих распознавателей. В общем случае, эти два алгоритма универсальны. Они строятся на основе любой КС-грамматики после некоторых формальных преобразований и поэтому могут быть использованы для разбора цепочки любого КС-языка. В этом случае время разбора входной цепочки имеет экспоненциальную зависимость от длины цепочки.

Однако для линейных распознавателей эти алгоритмы могут быть модифицированы так, чтобы время разбора имело линейную зависимость от длины входной цепочки. Указанные модификации не влияют на принципы, на основе которых построена работа алгоритмов, но позволяют оптимизировать их выполнение при условии, что грамматика входного языка принадлежит к определенному классу КС-грамматик. Для каждого такого класса предусматривается своя модификация.

Далее будут рассмотрены формальные преобразования, применимые к КС-грамматикам, которые позволяют строить распознаватели на основе произвольной КС-грамматики. Затем изучаются основы двух названных алгоритмов — алгоритма с подбором альтернатив и алгоритма «сдвиг-свертка». Вначале они даны для распознавателей с возвратами — хотя такие распознаватели КС-языков имеют ограниченное применение, но их рассмотрение позволяет понять принципы, лежащие в основе этих алгоритмов. А потом следуют разделы, посвященные некоторым наиболее известным классам КС-грамматик. Для каждого класса дается его определение, основные ограничения, а также принципы и алгоритмы, лежащие в основе распознавателя для этого класса КС-языков.

Далеко не все известные распознаватели с линейными характеристиками рассматриваются в данном пособии. Более полный набор распознавателей, а также

описание связанных с ними классов КС-грамматик и КС-языков можно найти в [4 т.1, 5, 18, 58, 63].

Выбор распознавателя для синтаксического анализа

Часто одна и та же КС-грамматика может быть отнесена не к одному, а сразу к нескольким классам КС-грамматик, допускающих построение линейных распознавателей. В этом случае необходимо решить, какой из возможных распознавателей выбрать для практической реализации.

Ответить на этот вопрос не всегда легко, поскольку могут быть построены два принципиально разных распознавателя, алгоритмы работы которых несопоставимы. В первую очередь, речь идет именно о восходящих и нисходящих распознавателях: в основе первых лежит алгоритм подбора альтернатив, в основе вторых — алгоритм «сдвиг-свертка».

На вопрос о том, какой распознаватель — нисходящий или восходящий — выбрать для построения синтаксического анализатора, нет однозначного ответа. Эту проблему необходимо решать, опираясь на некую дополнительную информацию о том, как будут использованы или каким образом будут обработаны результаты работы распознавателя.

СОВЕТ

> Следует помнить, что синтаксический анализатор — это один из этапов компиляции. И с этой точки зрения результаты работы распознавателя служат исходными данными для следующих этапов компиляции. Поэтому выбор того или иного распознавателя во многом зависит от реализации компилятора, от того, какие принципы положены в его основу.

Восходящий синтаксический анализ, как правило, привлекательнее нисходящего, так как для языка программирования часто легче построить правосторонний (восходящий) распознаватель. Класс языков, заданных восходящими распознавателями, шире, чем класс языков, заданных нисходящими распознавателями (хотя следует сказать, что не все здесь столь однозначно).

С другой стороны, как будет показано далее, левосторонний (нисходящий) синтаксический анализ предпочтителен с точки зрения процесса трансляции, поскольку на его основе легче организовать процесс порождения цепочек результирующего языка. Ведь в задачу компилятора входит не только распознать (проанализировать) входную программу на входном языке, но и построить (синтезировать) результирующую программу. Более подробную информацию об этом можно получить в главе «Генерация и оптимизация кода» или в работе [4 т.1,2, 5]. Левосторонний анализ, основанный на нисходящем распознавателе, оказывается предпочтительным также при учете вопросов, связанных с обнаружением и локализацией ошибок в тексте исходной программы [4 т.1,2, 5, 58, 65].

Желание использовать более простой класс грамматик для построения распознавателя может потребовать каких-то манипуляций с заданной грамматикой, необходимых для ее преобразования к требуемому классу. При этом нередко грамматика становится неестественной и малопонятной, что в дальнейшем

затрудняет ее использование в схеме синтаксически управляемого перевода и трансляции на этапе генерации результирующего кода (см. главу «Генерация и оптимизация кода»). Поэтому бывает удобным использовать исходную грамматику такой, как она есть, не стремясь преобразовать ее к более простому классу.

В целом следует отметить, что с учетом всего сказанного выше, интерес представляют как левосторонний, так и правосторонний анализ. Конкретный выбор зависит от реализации конкретного компилятора, а также от сложности грамматики входного языка программирования.

Преобразование КС-грамматик. Приведенные грамматики

Цель преобразования КС-грамматик. Приведенные грамматики

Цель преобразования КС-грамматик

Как было сказано выше, для КС-грамматик невозможно в общем случае проверить их однозначность и эквивалентность. Но очень часто правила КС-грамматик можно и нужно преобразовать к некоторому заданному виду таким образом, чтобы получить новую грамматику, эквивалентную исходной. Заранее определенный вид правил грамматики облегчает создание распознавателей.

Можно выделить две основных цели преобразований КС-грамматик: упрощение правил грамматики и облегчение создания распознавателя языка. Не всегда эти две цели можно совместить. В случае с языками программирования, когда итогом работы с грамматикой является создание компилятора, именно вторая цель преобразования является основной. Поэтому упрощениями правил пренебрегают, если при этом удается упростить построение распознавателя языка [12, 24].

Все преобразования условно можно разбить на две группы:

- первая группа — это преобразования, связанные с исключением из грамматики избыточных правил и символов (именно эти преобразования позволяют выполнить упрощения грамматики);
- вторая группа — это преобразования, в результате которых изменяется вид и состав правил грамматики, при этом грамматика может дополняться новыми правилами, а ее словарь нетерминальных символов — новыми символами (то есть преобразования второй группы не связаны с упрощениями).

Следует еще раз подчеркнуть, что всегда в результате преобразований мы получаем новую КС-грамматику, эквивалентную исходной, то есть определяющую тот же самый язык.

Тогда формально преобразование можно определить следующим образом:

$$G(\mathbf{VT},\mathbf{VN},\mathbf{P},S)\rightarrow G'(\mathbf{VT}',\mathbf{VN}',\mathbf{P}',S'):\ L(\mathbf{G})=L(\mathbf{G}')$$

Приведенные грамматики

Приведенные грамматики — это КС-грамматики, которые не содержат недостижимых и бесплодных символов, циклов и λ-правил (правил с пустыми цепочками).

Для того, чтобы преобразовать произвольную КС-грамматику к приведенному виду, необходимо выполнить следующие действия:

- удалить все бесплодные символы;
- удалить все недостижимые символы;
- удалить λ-правила;
- удалить цепные правила.

Следует подчеркнуть, что шаги преобразования должны выполняться именно в указанном порядке.

Удаление бесплодных символов

В грамматике $G(\mathbf{VT},\mathbf{VN},\mathbf{P},S)$ символ $A\in\mathbf{VN}$ называется *бесплодным*, если для него выполняется: $\{\alpha \mid A\Rightarrow^*\alpha,\ \alpha\in\mathbf{VT}^*\} = \varnothing$. То есть нетерминальный символ является бесплодным тогда, когда из него нельзя вывести ни одной цепочки терминальных символов.

В простейшем случае символ является бесплодным, если во всех правилах, где этот символ стоит в левой части, он также встречается и в правой части. Более сложные варианты предполагают зависимости между цепочками бесплодных символов, когда они в любой последовательности вывода порождают друг друга.

Алгоритм удаления бесплодных символов работает со специальным множеством нетерминальных символов $\mathbf{Y}_i$. Первоначально в это множество попадают только те символы, из которых непосредственно можно вывести терминальные цепочки, затем оно пополняется на основе правил грамматики **G**.

Алгоритм удаления бесплодных символов по шагам

1. $\mathbf{Y}_0 = \varnothing$, i:=1;
2. $\mathbf{Y}_i = \{A \mid (A \to \alpha)\in\mathbf{P},\ \alpha\in(\mathbf{Y}_{i-1}\cup\mathbf{VT})^*\}\cup\mathbf{Y}_{i-1}$;
3. Если $\mathbf{Y}_i \neq \mathbf{Y}_{i-1}$, то i:=i+1 и перейти к шагу 2, иначе перейти к шагу 4;
4. $\mathbf{VN}'=\mathbf{Y}_i$, $\mathbf{VT}'=\mathbf{VT}$, в $\mathbf{P}'$ входят те правила из **P**, которые содержат только символы из множества $(\mathbf{VT}\cup\mathbf{Y}_i)$, S'=S.

Удаление недостижимых символов

Символ $x\in(\mathbf{VT}\cup\mathbf{VN})$ называется *недостижимым*, если он не встречается ни в одной сентенциальной форме грамматики $G(\mathbf{VT},\mathbf{VN},\mathbf{P},S)$. То есть, символ недостижимый, если он не участвует ни в одной цепочке вывода из целевого символа грамматики. Очевидно, что такой символ в грамматике не нужен.

Алгоритм удаления недостижимых символов строит множество достижимых символов грамматики G(**VT**,**VN**,**P**,S) — $\mathbf{V}_i$. Первоначально в это множество входит только целевой символ S, затем оно пополняется на основе правил грамматики. Все символы, которые не войдут в данное множество, являются недостижимыми и могут быть исключены в новой грамматике **G'** из словаря и из правил.

Алгоритм удаления недостижимых символов по шагам

1. $\mathbf{V}_0 = \{S\}$, i:=1;
2. $\mathbf{V}_i = \{x \mid x \in (\mathbf{VT} \cup \mathbf{VN})$ и $(A \to \alpha x \beta) \in \mathbf{P}$, $A \in \mathbf{V}_{i-1}$, $\alpha,\beta \in (\mathbf{VT} \cup \mathbf{VN})^*\} \cup \mathbf{V}_{i-1}$;
3. Если $\mathbf{V}_i \neq \mathbf{V}_{i-1}$, то i:=i+1 и перейти к шагу 2, иначе перейти к шагу 4;
4. $\mathbf{VN}' = \mathbf{VN} \cap \mathbf{V}_i$, $\mathbf{VT}' = \mathbf{VT} \cap \mathbf{V}_i$, в **P'** входят те правила из **P**, которые содержат только символы из множества $\mathbf{V}_i$, S'=S.

Пример удаления недостижимых и бесплодных символов

Рассмотрим работу алгоритмов удаления недостижимых и бесплодных символов на примере грамматики:

G({a,b,c},{A,B,C,D,E,F,G,S},**P**,S)

P:

S → aAB | E

A → aA | bB

B → ACb | b

C → A | bA | cC | aE

E → cE | aE | Eb | ED | FG

D → a | c | Fb

F → BC | EC | AC

G → Ga | Gb

ВНИМАНИЕ

Для правильного выполнения преобразований необходимо сначала удалить бесплодные символы, а потом — недостижимые символы, но не наоборот.

Удалим бесплодные символы:

1. $\mathbf{Y}_0 = \varnothing$, i:=1 (шаг 1);
2. $\mathbf{Y}_1 = \{B,D\}$, $\mathbf{Y}_1 \neq \mathbf{Y}_0$: i:=2 (шаги 2 и 3);
3. $\mathbf{Y}_2 = \{B,D,A\}$, $\mathbf{Y}_2 \neq \mathbf{Y}_1$: i:=3 (шаги 2 и 3);

4. $\mathbf{Y}_3$ = {B,D,A,S,C}, $\mathbf{Y}_3 \neq \mathbf{Y}_2$: i:=4 (шаги 2 и 3);
5. $\mathbf{Y}_4$ = {B,D,A,S,C,F}, $\mathbf{Y}_4 \neq \mathbf{Y}_3$: i:=5 (шаги 2 и 3);
6. $\mathbf{Y}_5$ = {B,D,A,S,C,F}, $\mathbf{Y}_5 = \mathbf{Y}_4$ (шаги 2 и 3);
7. Строим множества **VN**' = {A,B,C,D,F,S}, **VT**' = {a,b,c} и **P**' (шаг 4).

Получили грамматику:

G'({a,b,c},{A,B,C,D,F,S},**P'**,S)

P':

S → aAB

A → aA | bB

B → ACb | b

C → A | bA | cC

D → a | c | Fb

F → BC | AC

Удалим недостижимые символы:

1. $\mathbf{V}_0$ = {S}, i:=1 (шаг 1);
2. $\mathbf{V}_1$ = {S,A,B}, $\mathbf{V}_1 \neq \mathbf{V}_0$: i:=2 (шаги 2 и 3);
3. $\mathbf{V}_2$ = {S,A,B,C}, $\mathbf{V}_2 \neq \mathbf{V}_1$: i:=3 (шаги 2 и 3);
4. $\mathbf{V}_3$ = {S,A,B,C}, $\mathbf{V}_3 = \mathbf{V}_2$ (шаги 2 и 3);
5. Строим множества **VN**" = {A,B,C,S}, **VT**" = {a,b,c} и **P**" (шаг 4).

В итоге получили грамматику:

G"({a,b,c},{A,B,C,S},**P"**,S)

P":

S → aAB

A → aA | bB

B → ACb | b

C → A | bA | cC

Алгоритмы удаления бесплодных и недостижимых символов относятся к первой группе преобразований КС-грамматик. Они всегда ведут к упрощению грамматики, сокращению количества символов алфавита и правил грамматики.

Устранение λ-правил

λ-правилами называются все правила грамматики вида A→λ, где A∈**VN**.

КС-грамматика G(**VT**,**VN**,**P**,S) называется грамматикой без λ-правил, если в ней не существует правил (A→λ)∈**P**, A≠S и существует только одно правило (S→λ)∈**P**, в том случае, когда λ∈L(**G**), и при этом S не встречается в правой части ни одного правила.

Для того, чтобы упростить построение распознавателей языка L(**G**), грамматику **G** часто целесообразно преобразовать к виду без λ-правил. Существует алгоритм преобразования произвольной КС-грамматики к виду без λ-правил. Он работает с некоторым множеством нетерминальных символов $\mathbf{W}_i$.

Алгоритм устранения λ-правил по шагам

1. $\mathbf{W}_0$ = {A: (A→λ)∈**P** }, i:=1;
2. $\mathbf{W}_i = \mathbf{W}_{i-1} \cup \{A: (A \to \alpha) \in \mathbf{P}, \alpha \in \mathbf{W}_{i-1}^*\}$;
3. Если $\mathbf{W}_i \neq \mathbf{W}_{i-1}$, то i:=i+1 и перейти к шагу 2, иначе перейти к шагу 4;
4. **VN'** = **VN**, **VT'** = **VT**, в **P'** входят все правила из **P**, кроме правил вида A→λ;
5. Если (A→α)∈**P** и в цепочку α входят символы из множества $\mathbf{W}_i$, тогда на основе цепочки α строится множество цепочек $\{\alpha'_1,\ldots, \alpha'_k\}$, k>0 путем исключения из α всех возможных комбинаций символов $\mathbf{W}_i$, все правила вида $A \to \alpha'_i$, k≥i>0 добавляются в **P'** (при этом надо отбрасывать дубликаты правил и бессмысленные правила вида A→A);
6. Если S∈$\mathbf{W}_i$, то значит λ∈L(**G**), и тогда в **VN'** добавляется новый символ S', который становится целевым символом грамматики **G'**, а в **P'** добавляются два новых правила: S'→λ|S; иначе S'=S (целевой символ грамматики не меняется).

Данный алгоритм часто ведет к увеличению количества правил грамматики, но позволяет упростить построение распознавателя для заданного языка.

Пример устранения λ-правил

Рассмотрим грамматику:

G({a,b,c},{A,B,C,S},**P**,S)

P:

S → AaB | aB | cC

A → AB | a | b | B

B → Ba | λ

C → AB | c

Удалим λ-правила:

1. $\mathbf{W}_0$ = {B}, i:=1 (шаг 1);
2. $\mathbf{W}_1$ = {B,A}, $\mathbf{W}_1 \neq \mathbf{W}_0$, i:=2 (шаги 2 и 3);

3. $\mathbf{W}_2$ = {B,A,C}, $\mathbf{W}_2 \neq \mathbf{W}_1$, i:=3 (шаги 2 и 3);
4. $\mathbf{W}_3$ = {B,A,C}, $\mathbf{W}_3 = \mathbf{W}_2$ (шаги 2 и 3);
5. Построим **VN'** = {A,B,C,S}, **VT'** = {a,b,c} и множество правил **P'** (шаг 4):

   P':

   S → AaB | aB | cC

   A → AB | a | b | B

   B → Ba

   C → AB | c

6. Рассмотрим все правила из множества **P'** (шаг 5):
   - Из правил S→AaB|aB|cC исключим все комбинации $\mathbf{W}_3$={B,A,C} и получим новые правила S→Aa|aB|a|a|c, добавим их в **P'**, исключая дубликаты, получим: S→AaB|aB|cC|Aa|aB|a|c;
   - Из правил A→AB|a|b|B исключим все комбинации $\mathbf{W}_3$={B,A,C} и получим новые правила A→A|B, в **P'** их добавлять не надо, поскольку правило A→B там уже есть, а правило A→A бессмысленно;
   - Из правила B→Ba исключим все комбинации $\mathbf{W}_3$={B,A,C} и получим новое правило B→a, добавим его в **P'**, получим: B→Ba|a;
   - Из правил C→AB|c исключим все комбинации $\mathbf{W}_3$={B,A,C} и получим новые правила C→A|B, добавим их в **P'**, получим: C→AB|A|B|c.
7. S$\notin \mathbf{W}_3$, поэтому в грамматику **G'** не надо добавлять новый целевой символ: S'=S (шаг 6).

Получим грамматику:

G'({a,b,c},{A,B,C,S},**P'**,S)

P':

S → AaB | aB | cC | Aa | a | c

A → AB | a | b | B

B → Ba | a

C → AB | A | B | c

Устранение цепных правил

Циклом (циклическим выводом) в грамматике G(**VT**,**VN**,**P**,S) называется вывод вида A⇒*A, A∈**VN**. Очевидно, что такой вывод абсолютно бесполезен. В распознавателях КС-языков целесообразно избегать возможности появления циклов.

Циклы возможны только в том случае, если в КС-грамматике присутствуют *цепные правила* вида $A \to B$, $A,B \in \mathbf{VN}$. Чтобы исключить возможность появления циклов в цепочках вывода, достаточно устранить цепные правила из правил грамматики.

Чтобы устранить цепные правила в КС-грамматике $G(\mathbf{VT},\mathbf{VN},\mathbf{P},S)$ для каждого нетерминального символа $X \in \mathbf{VN}$ строится специальное множество цепных символов $\mathbf{N}^X$, а затем на основании построенных множеств выполняются преобразования правил **P**. Поэтому алгоритм устранения цепных правил надо выполнить для всех нетерминальных символов грамматики из множества **VN**.

Алгоритм устранения цепных правил по шагам

Шаг 1. Для всех символов X из **VN** повторять шаги 2–4, затем перейти к шагу 5;

Шаг 2. $\mathbf{N}^X_0 = \{X\}$, i:=1;

Шаг 3. $\mathbf{N}^X_i = \mathbf{N}^X_{i-1} \cup \{B: (A \to B) \in \mathbf{P}, B \in \mathbf{N}^X_{i-1}\}$;

Шаг 4. Если $\mathbf{N}^X_i \neq \mathbf{N}^X_{i-1}$, то i:=i+1 и перейти к шагу 3, иначе $\mathbf{N}^X = \mathbf{N}^X_i - \{X\}$ и продолжить цикл по шагу 1;

Шаг 5. **VN'** = **VN**, **VT'** = **VT**, в **P'** входят все правила из **P**, кроме правил вида $A \to B$, S' = S;

Шаг 6. Для всех правил $(A \to \alpha) \in \mathbf{P'}$, если $B \in \mathbf{N}^A$, $B \neq A$, то в **P'** добавляются правила вида $B \to \alpha$.

Данный алгоритм, также как и алгоритм устранения λ-правил, ведет к увеличению числа правил грамматики, но упрощает построение распознавателей.

Пример устранения цепных правил

Рассмотрим в качестве примера грамматику арифметических выражений над символами `a` и `b`, которая уже рассматривалась ранее в этом учебном пособии:

`G({+,-,/,*,a,b},{S,T,E},P,S):`

`P:`

`S → S+T | S-T | T`

`T → T*E | T/E | E`

`E → (S) | a | b`

Устраним цепные правила:

1. $\mathbf{N}^S_0 = \{S\}$, i:=1;
2. $\mathbf{N}^S_1 = \{S,T\}$, $\mathbf{N}^S_1 \neq \mathbf{N}^S_0$, i:=2;
3. $\mathbf{N}^S_2 = \{S,T,E\}$, $\mathbf{N}^S_2 \neq \mathbf{N}^S_1$, i:=3;
4. $\mathbf{N}^S_3 = \{S,T,E\}$, $\mathbf{N}^S_3 = \mathbf{N}^S_2$, $\mathbf{N}^S = \{T,E\}$;
5. $\mathbf{N}^T_0 = \{T\}$, i:=1;

6. $\mathbf{N}^{T}_{1} = \{T,E\}$, $\mathbf{N}^{T}_{1} \neq \mathbf{N}^{T}_{0}$: i:=2;
7. $\mathbf{N}^{T}_{2} = \{T,E\}$, $\mathbf{N}^{T}_{2} = \mathbf{N}^{T}_{1}$, $\mathbf{N}^{T} = \{E\}$;
8. $\mathbf{N}^{E}_{0} = \{E\}$, i:=1;
9. $\mathbf{N}^{E}_{1} = \{E\}$, $\mathbf{N}^{E}_{1} = \mathbf{N}^{E}_{0}$, $\mathbf{N}^{E} = \varnothing$;
10. Получили: $\mathbf{N}^{S} = \{T,E\}$, $\mathbf{N}^{T} = \{E\}$, $\mathbf{N}^{E} = \varnothing$, S'=S, построим множества **V**N' = {S,T,E}, **V**T' = {+,-,/,*,a,b}, и множество правил **P**';
11. Рассмотрим все правила из множества **P**' — интерес представляют только правила для символов T и E, так как $\mathbf{N}^{S} = \{T,E\}$ и $\mathbf{N}^{T} = \{E\}$:
    - Для правил `T→T*E|T/E` имеем новые правила `S→T*E|T/E`, поскольку $T \in \mathbf{N}^{S}$;
    - Для правил `E→(S)|a|b` имеем новые правила `S→(S)|a|b` и `T→(S)|a|b`, поскольку $E \in \mathbf{N}^{S}$ и $E \in \mathbf{N}^{T}$.

Получим новую грамматику:

`G'({+,-,/,*,a,b},{S,T,E},P',S)`

P':

```
S → S+T | S-T | T*E | T/E | (S) | a | b

T → T*E | T/E | (S) | a | b

E → (S) | a | b
```

Эта грамматика далее будет использована для построения распознавателей КС-языков.

Устранение левой рекурсии

Определение левой рекурсии

Символ $A \in \mathbf{VN}$ в КС-грамматике G(**VT**,**VN**,P,S) называется рекурсивным, если для него существует цепочка вывода вида $A \Rightarrow^{+} \alpha A \beta$, где $\alpha,\beta \in (\mathbf{VT} \cup \mathbf{VN})^{*}$.

Если $\alpha=\lambda$ и $\beta \neq \lambda$, то рекурсия называется левой, а грамматика **G** — *леворекурсивной*; если $\alpha \neq \lambda$ и $\beta=\lambda$, то рекурсия называется правой, а грамматика **G** — *праворекурсивной*. Если $\alpha=\lambda$ и $\beta=\lambda$, то рекурсия представляет собой цикл. Алгоритм исключения циклов был рассмотрен выше, поэтому далее циклы не рассматриваются.

Любая КС-грамматика может быть как леворекурсивной, так и праворекурсивной, а также леворекурсивной и праворекурсивной одновременно.

КС-грамматика называется *нелеворекурсивной*, если она не является леворекурсивной. Аналогично, КС-грамматика является *неправорекурсивной*, если не является праворекурсивной.

Некоторые алгоритмы левостороннего разбора для КС-языков не работают с леворекурсивными грамматиками, поэтому возникает необходимость исключить

левую рекурсию из выводов грамматики. Далее будет рассмотрен алгоритм, который позволяет преобразовать правила произвольной КС-грамматики таким образом, чтобы в выводах не встречалась левая рекурсия.

Следует отметить, что поскольку рекурсия лежит в основе построения языков на основе правил грамматики в форме Бэкуса-Наура, полностью исключить рекурсию из выводов грамматики невозможно. Можно избавится только от одного вида рекурсии — левого или правого, то есть преобразовать исходную грамматику **G** к одному из видов: нелеворекурсивному (избавиться от левой рекурсии) или неправорекурсивному (избавиться от правой рекурсии). Для левосторонних распознавателей интерес представляет избавление от левой рекурсии — то есть преобразование грамматики к нелеворекурсивному виду.

Причина устранения именно левой рекурсии кроется в том, что все существующие в реальных компиляторах распознаватели читают входной текст слева направо.

Доказано, что любую КС-грамматику можно преобразовать к нелеворекурсивному или неправорекурсивному виду [4 т.1, 5, 58].

Алгоритм устранения левой рекурсии

Условие: дана КС-грамматика G(**VT**,**VN**,**P**,S), необходимо построить эквивалентную ей нелеворекурсивную грамматику G'(**VN**',**VT**,**P**',S'): L(**G**)=L(**G**').

Алгоритм преобразования работает с множеством правил исходной грамматики **P**, множеством нетерминальных символов **VN** и двумя переменными счетчиками: i и j.

Шаг 1. Обозначим нетерминальные символы грамматики так: **VN**=$\{A_1,A_2,\ldots,A_n\}$, i:=1.

Шаг 2. Рассмотрим правила для символа A_i. Если эти правила не содержат левой рекурсии, то перенесем их во множество правил **P'** без изменений, а символ A_i добавим во множество нетерминальных символов **VN'**.

Иначе запишем правила для A_i в виде $A_i \to A_i\alpha_1|A_i\alpha_2|\ldots|A_i\alpha_m|\beta_1|\beta_2|\ldots|\beta_p$, где $\forall p \geq j \geq 1$ ни одна из цепочек β_j не начинается с символов A_k, таких, что $k \leq i$.

Вместо этого правила во множество **P'** запишем два правила вида:

$$A_i \to \beta_1|\beta_2|\ldots|\beta_p|\beta_1A_i'|\beta_2A_i'|\ldots|\beta_pA_i'$$

$$A_i' \to \alpha_1|\alpha_2|\ldots|\alpha_m|\alpha_1A_i'|\alpha_2A_i'|\ldots|\alpha_mA_i'$$

Символы A_i и A_i' включаем во множество **VN'**.

Теперь все правила для A_i начинаются либо с терминального символа, либо с нетерминального символа A_k, такого, что $k>i$.

Шаг 3. Если i=n, то грамматика **G'** построена, перейти к шагу 6, иначе i:=i+1, j:=1 и перейти к шагу 4.

Шаг 4. Для символа A_j во множестве правил **P'** заменить все правила вида $A_i \to A_j\alpha$, где $\alpha \in (\mathbf{VT} \cup \mathbf{VN})^*$, на правила вида $A_i \to \beta_1\alpha|\beta_2\alpha|\ldots|\beta_m\alpha$, причем $A_j \to \beta_1|\beta_2|\ldots|\beta_m$ — все правила для символа A_j.

Так как правая часть правил $A_j \to \beta_1|\beta_2|\ldots|\beta_m$ уже начинается с терминального символа или нетерминального символа A_k, k>j, то теперь и правая часть правил для символа A_i будет удовлетворять этому условию.

Шаг 5. Если j=i−1, то перейти к шагу 2, иначе j:=j+1 и перейти к шагу 4.

Шаг 6. Целевым символом грамматики **G'** становится символ A_k, соответствующий символу S исходной грамматики **G**.

Рассмотрим в качестве примера грамматику для арифметических выражений над символами a и b:

G({+,-,/,*,a,b},{S,T,E},**P**,S)**:**

P:

S → S+T | S−T | T

T → T*E | T/E | E

E → (S) | a | b

Эта грамматика является леворекурсивной. Построим эквивалентную ей нелеворекурсивную грамматику **G'**.

Шаг 1. Обозначим **VN**={A_1,A_2,A_3}. i:=1.

Тогда правила грамматики **G** будут иметь вид:

$A_1 \to A_1+A_2 \mid A_1-A_2 \mid A_2$

$A_2 \to A_2*A_3 \mid A_2/A_3 \mid A_3$

$A_3 \to (A_1) \mid a \mid b$

Шаг 2. Для A_1 имеем правила $A_1 \to A_1+A_2|A_1-A_2|A_2$. Их можно записать в виде $A_1 \to A_1\alpha_1|A_1\alpha_2|\beta_1$, где $\alpha_1=+A_2$, $\alpha_2=-A_2$, $\beta_1=A_2$.

Запишем новые правила для множества **P'**:

$A_1 \to A_2 \mid A_2A_1'$

$A_1' \to +A_2 \mid -A_2 \mid +A_2A_1' \mid -A_2A_1'$

Добавив эти правила в **P'**, а символы A_1 и A_1' во множество нетерминальных символов, получим: **VN'**={A_1,A_1'}.

Шаг 3. i=1 < 3. Построение не закончено: i:=i+1 = 2, j:=1.

Шаг 4. Для символа A_2 во множестве правил **P'** нет правила вида $A_2 \to A_1\alpha$, поэтому на этом шаге никаких действий не выполняем.

Шаг 5. j = 1 = i−1, переходим опять к шагу 2.

Шаг 2. Для A_2 имеем правила $A_2 \to A_2*A_3|A_2/A_3|A_3$. Их можно записать в виде $A_2 \to A_2\alpha_1|A_2\alpha_2|\beta_1$, где $\alpha_1=*A_3$, $\alpha_2=/A_3$, $\beta_1=A_3$.

Запишем новые правила для множества **P'**:

$A_2 \rightarrow A_3 \mid A_3A_2'$

$A_2' \rightarrow *A_3 \mid /A_3 \mid *A_3A_2' \mid /A_3A_2'$

Добавим эти правила в **P'**, а символы A_2 и A_2' во множество нетерминальных символов, получим: **VN'**={A_1,A_1',A_2,A_2'}.

Шаг 3. i=2 < 3. Построение не закончено: i:=i+1 = 3, j:=1.

Шаг 4. Для символа A_3 во множестве правил **P'** нет правила вида $A_3 \rightarrow A_1\alpha$, поэтому на этом шаге никаких действий не выполняем.

Шаг 5. j=1 < i–1, j:=j+1 = 2, переходим к шагу 4.

Шаг 4. Для символа A_3 во множестве правил **P'** нет правила вида $A_3 \rightarrow A_2\alpha$, поэтому на этом шаге никаких действий не выполняем.

Шаг 5. j=2 = i–1, переходим опять к шагу 2.

Шаг 2. Для A_3 имеем правила $A_3 \rightarrow (A_1)|a|b$. Эти правила не содержат левой рекурсии. Переносим их в **P'**, а символ A_3 добавляем в **VN'**. Получим: **VN'**={A_1,A_1',A_2,A_2',A_3}.

Шаг 3. i=3 = 3. Построение грамматики **G'** закончено.

В результате выполнения алгоритма преобразования получили нелеворекурсивную грамматику G({+,–,/,*,a,b}, {A_1,A_1',A_2,A_2',A_3}, **P'**, A_1) с правилами:

P':

$A_1 \rightarrow A_2 \mid A_2A_1'$

$A_1' \rightarrow +A_2 \mid –A_2 \mid +A_2A_1' \mid –A_2A_1'$

$A_2 \rightarrow A_3 \mid A_3A_2'$

$A_2' \rightarrow *A_3 \mid /A_3 \mid *A_3A_2' \mid /A_3A_2'$

$A_3 \rightarrow (A_1) \mid a \mid b$

Синтаксические распознаватели с возвратом

Принципы работы распознавателей с возвратом

Распознаватели с возвратом — это самый примитивный тип распознавателей для КС-языков. Логика их работы основана на моделировании функционирования недетерминированного МП-автомата.

Поскольку моделируется недетерминированный МП-автомат, то на некотором шаге работы моделирующего алгоритма существует возможность возникновения нескольких возможных следующих состояний автомата. В таком случае есть два варианта реализации алгоритма [15].

В первом варианте на каждом шаге работы алгоритм должен запоминать все возможные следующие состояния МП-автомата, выбирать одно из них, переходить в это состояние и действовать так до тех пор, пока либо не будет достигнуто конечное состояние автомата, либо автомат не перейдет в такую конфигурацию, когда

следующее состояние будет не определено. Если достигнуто одно из конечных состояний — входная цепочка принята, работа алгоритма завершается. В противном случае алгоритм должен вернуть автомат на несколько шагов назад, когда еще был возможен выбор одного из набора следующих состояний, выбрать другой вариант и промоделировать поведение автомата с этим условием. Алгоритм завершается с ошибкой, когда все возможные варианты работы автомата перебраны, и при этом не было достигнуто ни одно из возможных конечных состояний.

Во втором варианте алгоритм моделирования МП-автомата должен на каждом шаге работы при возникновении неоднозначности с несколькими возможными следующими состояниями автомата запускать новую свою копию для обработки каждого из этих состояний. Алгоритм завершается, если хотя бы одна из выполняющихся его копий достигнет одно из конечных состояний. При этом работа всех остальных копий алгоритма прекращается. Если ни одна из копий алгоритма не достигла конечного состояния МП-автомата, то алгоритм завершается с ошибкой.

Второй вариант реализации алгоритма связан с управлением параллельными процессами в вычислительных системах, поэтому сложен в реализации. Кроме того, на каждом шаге работы МП-автомата альтернатив следующих состояний может быть много, а количество возможных параллельно выполняющихся процессов в операционных системах ограничено, поэтому применение второго варианта алгоритма осложнено. По этим причинам большее распространение получил первый вариант алгоритма, который предусматривает возврат к ранее запомненным состояниям МП-автомата — отсюда и название «разбор с возвратами».

Следует отметить, что, хотя МП-автомат является односторонним распознавателем, алгоритм моделирования его работы предусматривает возврат назад, к уже прочитанной части цепочки символов, чтобы исключить недетерминизм в поведении автомата (который иначе невозможно промоделировать).

Есть еще одна особенность в моделировании МП-автомата: любой практически ценный алгоритм должен завершаться за конечное число шагов (успешно или неуспешно). Алгоритм моделирования работы произвольного МП-автомата в общем случае не удовлетворяет этому условию.

Чтобы избежать таких ситуаций, алгоритмы разбора с возвратами строят не для произвольных МП-автоматов, а для МП-автоматов, удовлетворяющим некоторым заданным условиям. Как правило, эти условия связаны с тем, что МП-автомат должен строиться на основе грамматики заданного языка только после того, как она подвергнется некоторым преобразованиям. Поскольку преобразования грамматик сами по себе не накладывают каких-либо ограничений на входной класс КС-языков, то они и не ограничивают применимости алгоритмов разбора с возвратами. Следовательно, эти алгоритмы применимы для любого КС-языка, заданного произвольной КС-грамматикой.

Алгоритмы разбора с возвратами обладают экспоненциальными характеристиками. Это значит, что вычислительные затраты алгоритмов экспоненциально зависят от длины входной цепочки символов: α, $\alpha \in \mathbf{VT}^*$, $n=|\alpha|$. Конкретная зависимость определяется вариантом реализации алгоритма [4 т.1, 5].

Доказано, что в общем случае при первом варианте реализации для произвольной КС-грамматики $G(\mathbf{VT},\mathbf{VN},\mathbf{P},S)$ время выполнения данного алгоритма $T_э$ будет иметь

экспоненциальную зависимость от длины входной цепочки, а необходимый объем памяти $M_э$ — линейную зависимость от длины входной цепочки: $T_э=O(e^n)$ и $M_э=O(n)$. При втором варианте реализации, наоборот, время выполнения данного алгоритма $T_э$ будет иметь линейную зависимость от длины входной цепочки, а необходимый объем памяти $M_э$ — экспоненциальную зависимость от длины входной цепочки: $T_э=O(n)$ и $M_э=O(e^n)$.

Экспоненциальная зависимость вычислительных затрат от длины входной цепочки существенно ограничивает применимость алгоритмов разбора с возвратами. Они тривиальны в реализации, но имеют неудовлетворительные характеристики, поэтому могут использоваться только для КС-языков с малой длиной входных предложений языка.[1] Для многих классов КС-языков существуют более эффективные алгоритмы распознавания, поэтому алгоритмы разбора с возвратами применяются редко.

Далее рассмотрены два основных варианта таких алгоритмов.

Нисходящий распознаватель с подбором альтернатив

Принцип работы нисходящего распознавателя с подбором альтернатив

Нисходящий распознаватель с подбором альтернатив моделирует работу МП-автомата с одним состоянием q: R({q},**V**,**Z**,δ,q,S,{q}). Автомат распознает цепочки КС-языка, заданного КС-грамматикой G(**VT**,**VN**,**P**,S). Входной алфавит автомата содержит терминальные символы грамматики: **V**=**VT**, а алфавит магазинных символов строится из терминальных и нетерминальных символов грамматики: **Z**=**VT**∪**VN**.

Начальная конфигурация автомата определяется так: (q,α,S) — автомат пребывает в своем единственном состоянии q, считывающая головка находится в начале входной цепочки символов $\alpha\in\mathbf{VT}^*$, в стеке лежит символ, соответствующий целевому символу грамматики S.

Конечная конфигурация автомата определяется так: (q,λ,λ) — автомат пребывает в своем единственном состоянии q, считывающая головка находится за концом входной цепочки символов, стек пуст.

Функция переходов МП-автомата строится на основе правил грамматики:

1. $(q,\alpha)\in\delta(q,\lambda,A)$, $A\in\mathbf{VN}$, $\alpha\in(\mathbf{VT}\cup\mathbf{VN})^*$, если правило $A\to\alpha$ содержится во множестве правил **P** грамматики **G**: $A\to\alpha\in\mathbf{P}$;
2. $(q,\lambda)\in\delta(q,a,a)$ $\forall a\in\mathbf{VT}$.

Работу данного МП-автомата можно неформально описать следующим образом: если на верхушке стека автомата находится нетерминальный символ A, то его можно заменить на цепочку символов α, если в грамматике языка есть правило $A\to\alpha$, не

---

[1] Возможность использовать эти алгоритмы в реальных компиляторах весьма сомнительна, поскольку длина входной цепочки может достигать нескольких тысяч и даже десятков тысяч символов. Очевидно, что время работы алгоритма будет в таком варианте явно неприемлемым даже на самых современных компьютерах.

сдвигая при этом считывающую головку автомата (этот шаг работы называется «подбор альтернативы»); если же на верхушке стека находится терминальный символ *a*, который совпадает с текущим символом входной цепочки, то этот символ можно выбросить из стека и передвинуть считывающую головку на одну позицию вправо (этот шаг работы называется «выброс»). Данный МП-автомат может быть недетерминированным, поскольку при подборе альтернативы в грамматике языка может оказаться более одного правила вида $A \to \alpha$, тогда функция $\delta(q,\lambda,A)$ будет содержать более одного следующего состояния — у МП-автомата будет несколько альтернатив.

Решение о том, выполнять ли на каждом шаге работы МП-автомата выброс или подбор альтернативы, принимается однозначно. Моделирующий алгоритм должен обеспечивать выбор одной из возможных альтернатив и хранение информации о том, какие альтернативы на каком шаге уже были выбраны, чтобы иметь возможность вернуться к этому шагу и подобрать другие альтернативы. Такой алгоритм разбора называется алгоритмом с подбором альтернатив.

Данный МП-автомат строит левосторонние выводы для грамматики G(**VT**,**VN**,**P**,S). Для моделирования такого автомата необходимо, чтобы грамматика G(**VT**,**VN**,**P**,S) не была леворекурсивной — в противном случае, очевидно, алгоритм может войти в бесконечный цикл. Поскольку, как было показано выше, произвольную КС-грамматику всегда можно преобразовать к нелеворекурсивному виду, этот алгоритм применим для любой КС-грамматики. Следовательно, им можно распознавать цепочки любого КС-языка.

Реализация алгоритма распознавателя с подбором альтернатив

Существует масса способов реализации алгоритма с подбором альтернатив [4 т.1, 5, 33, 42, 58]. Рассмотрим один из примеров реализации алгоритма нисходящего распознавателя с возвратом.

Для удобства работы все правила из множества **P** в грамматике **G** представим в виде $A \to \alpha_1|\alpha_2|\ldots|\alpha_n$, то есть пронумеруем все возможные альтернативы для каждого нетерминального символа $A \in \mathbf{VN}$. Входная цепочка символов имеет вид $\alpha = a_1a_2\ldots a_n$, $|\alpha|=n$. Алгоритм использует также дополнительное состояние b (от «back» — «назад»), которое сигнализирует о выполнении возврата к уже прочитанной части входной цепочки.[2] Для хранения уже выбранных альтернатив используется дополнительный стек возвратов $\mathbf{L}_2$, который может содержать следующую информацию:

- символы $a \in \mathbf{VT}$ входного языка автомата;
- символы вида A_j, где $A \in \mathbf{VN}$ — это означает, что среди всех возможных правил для символа A была выбрана альтернатива с номером j.

---

[2] Сам автомат имеет только одно состояние q, которого достаточно для его функционирования, однако нет возможности моделировать на компьютере работу недетерминированного автомата, поэтому приходится выполнять возврат к уже прочитанной части цепочки и вводить для этой цели дополнительное состояние.

В итоге алгоритм работает с двумя стеками: $\mathbf{L}_1$ — стек МП-автомата и $\mathbf{L}_2$ — стек возвратов. Оба они представлены в виде цепочек символов. Символы в цепочку стека $\mathbf{L}_1$ помещаются слева, а в цепочку стека $\mathbf{L}_2$ — справа. В целом состояние алгоритма на каждом шаге определяется четырьмя параметрами: $(Q,i,\mathbf{L}_1,\mathbf{L}_2)$, где

- Q — текущее состояние автомата (q или b);
- i — положение считывающей головки во входной цепочке символов α;
- $\mathbf{L}_1$ — содержимое стека МП-автомата;
- $\mathbf{L}_2$ — содержимое дополнительного стека.

Начальным состоянием алгоритма является состояние $(q,1,S,\lambda)$, где S — целевой символ грамматики. Алгоритм начинает свою работу с начального состояния и циклически выполняет шаги до тех пор, пока не перейдет в конечное состояние или не обнаружит ошибку. На каждом шаге алгоритма проверяется, соответствует ли текущее состояние алгоритма заданному для данного шага исходному состоянию, и выполняются ли заданные дополнительные условия. Если это требование выполняется, алгоритм переходит в следующее состояние, установленное для этого шага, если нет — шаг пропускается, алгоритм переходит к следующему шагу.

Алгоритм предусматривает циклическое выполнение следующих шагов:

Шаг 1 (Разрастание). $(q,i,A\beta,\alpha) \rightarrow (q,i,\gamma_1\beta,\alpha A_1)$, если $A\rightarrow\gamma_1$ — это первая из всех возможных альтернатив для символа A.

Шаг 2 (Успешное сравнение). $(q,i,a\beta,\alpha) \rightarrow (q,i+1,\beta,\alpha a)$, если $a=a_i$, $a\in\mathbf{VT}$.

Шаг 3 (Завершение). Если состояние соответствует $(q,n+1,\lambda,\alpha)$, то разбор завершен, алгоритм заканчивает работу, иначе $(q,i,\lambda,\alpha)\rightarrow(b,i,\lambda,\alpha)$, когда $i \neq n+1$.

Шаг 4 (Неуспешное сравнение). $(q,i,a\beta,\alpha)\rightarrow(b,i,a\beta,\alpha)$, если $a\neq a_i$, $a\in\mathbf{VT}$.

Шаг 5 (Возврат по входу). $(b,i,\beta,\alpha a) \rightarrow (q,i-1,a\beta,\alpha)$, $\forall a\in\mathbf{VT}$.

Шаг 6 (Другая альтернатива). Исходное состояние $(b,i,\gamma_j\beta,\alpha A_j)$, действия:

- перейти в состояние $(q,i,\gamma_{j+1}\beta,\alpha A_{j+1})$, если еще существует альтернатива $A\rightarrow\gamma_{j+1}$ для символа $A\in\mathbf{VN}$;
- сигнализировать об ошибке и прекратить выполнение алгоритма, если $A\equiv S$ и не существует больше альтернатив для символа S;
- иначе перейти в состояние $(q,i,A\beta,\alpha)$.

В случае успешного завершения алгоритма цепочку вывода можно построить на основе содержимого стека $\mathbf{L}_2$, полученного в результате выполнения алгоритма. Цепочка вывода строится следующим образом: поместить в цепочку номер правила m, соответствующий альтернативе $A\rightarrow\gamma_j$, если в стеке содержится символ A_j (все символы $a\in\mathbf{VT}$, содержащиеся в стеке $\mathbf{L}_2$, игнорируются).

ВНИМАНИЕ

> Следует помнить, что для применения этого алгоритма исходная грамматика не должна быть леворекурсивной. Если это условие не выполняется, то грамматику предварительно надо преобразовать к нелеворекурсивному виду.

Рассмотрим в качестве примера грамматику `G({+,-,/,*,a,b}, {S,R,T,F,E}, P, S)` с правилами:

P:

`S → T | TR`

`R → +T | −T | +TR | −TR`

`T → E | EF`

`F → *E | /E | *EF | /EF`

`E → (S) | a | b`

Это нелеворекурсивная грамматика для арифметических выражений (она была построена ранее в разделе «Устранение левой рекурсии»).

Проследим разбор цепочки `a+(a*b)` из языка, заданного этой грамматикой с помощью алгоритма нисходящего распознавателя с возвратами. Работу алгоритма будем представлять в виде последовательности его состояний, взятых в скобки `{}` (фигурные скобки используются, чтобы не путать их с круглыми скобками, предусмотренными в правилах грамматики). Альтернативы будем номеровать слева направо. Для пояснения каждый шаг работы сопровождается номером шага алгоритма, который был применен для перехода в очередное состояние (записывается слева через символ **:** — двоеточие).

Алгоритм работы нисходящего распознавателя с возвратами при разборе цепочки `a+(a*b)` будет выполнять следующие шаги:

1. 0: {q,1,S,λ}
2. 1: {q,1,T,S_1}
3. 1: {q,1,E,S_1T_1}
4. 1: {q,1,(S),$S_1T_1E_1$}
5. 4: {b,1,(S),$S_1T_1E_1$}
6. 6: {q,1,a,$S_1T_1E_2$}
7. 2: {q,2,λ,$S_1T_1E_2$a}
8. 3: {b,2,λ,$S_1T_1E_2$a}
9. 5: {b,1,a,$S_1T_1E_2$}
10. 6: {q,1,b,$S_1T_1E_3$}
11. 4: {b,1,b,$S_1T_1E_3$}
12. 6: {b,1,E,S_1T_1}

13. 6: {q,1,EF,S_1T_2}
14. 1: {q,1,(S)F,$S_1T_2E_1$}
15. 4: {b,1,(S)F,$S_1T_2E_1$}
16. 6: {q,1,aF,$S_1T_2E_2$}
17. 2: {q,2,F,$S_1T_2E_2$a}
18. 1: {q,2,*E,$S_1T_2E_2aF_1$}
19. 4: {b,2,*E,$S_1T_2E_2aF_1$}
20. 6: {q,2,/E,$S_1T_2E_2aF_2$}
21. 4: {b,2,/E,$S_1T_2E_2aF_2$}
22. 6: {q,2,*EF,$S_1T_2E_2aF_3$}
23. 4: {b,2,*EF,$S_1T_2E_2aF_3$}
24. 6: {q,2,/EF,$S_1T_2E_2aF_4$}
25. 4: {b,2,/EF,$S_1T_2E_2aF_4$}
26. 6: {b,2,F,$S_1T_2E_2$a}
27. 5: {b,1,aF,$S_1T_2E_2$}
28. 6: {q,1,bF,$S_1T_2E_3$}
29. 4: {b,1,bF,$S_1T_2E_3$}
30. 6: {b,1,EF,S_1T_2}
31. 6: {b,1,T,S_1}
32. 6: {q,1,TR,S_2}
33. 1: {q,1,ER,S_2T_1}
34. 1: {q,1,(S)R,$S_2T_1E_1$}
35. 4: {b,1,(S)R,$S_2T_1E_1$}
36. 6: {q,1,aR,$S_2T_1E_2$}
37. 2: {q,2,R,$S_2T_1E_2$a}
38. 1: {q,2,+T,$S_2T_1E_2aR_1$}
39. 2: {q,3,T,$S_2T_1E_2aR_1$+}
40. 1: {q,3,E,$S_2T_1E_2aR_1+T_1$}
41. 1: {q,3,(S),$S_2T_1E_2aR_1+T_1E_1$}
42. 2: {q,4,S),$S_2T_1E_2aR_1+T_1E_1$(}
43. 1: {q,4,T),$S_2T_1E_2aR_1+T_1E_1(S_1$}
44. 1: {q,4,E),$S_2T_1E_2aR_1+T_1E_1(S_1T_1$}

45. 1: {q,4,(S)),$S_2T_1E_2aR_1+T_1E_1(S_1T_1E_1$}

46. 4: {b,4,(S)),$S_2T_1E_2aR_1+T_1E_1(S_1T_1E_1$}

47. 6: {q,4,a),$S_2T_1E_2aR_1+T_1E_1(S_1T_1E_2$}

48. 2: {q,5,),$S_2T_1E_2aR_1+T_1E_1(S_1T_1E_2a$}

49. 4: {b,5,),$S_2T_1E_2aR_1+T_1E_1(S_1T_1E_2a$}

50. 5: {b,4,a),$S_2T_1E_2aR_1+T_1E_1(S_1T_1E_2$}

51. 6: {q,4,b),$S_2T_1E_2aR_1+T_1E_1(S_1T_1E_3$}

52. 4: {b,4,b),$S_2T_1E_2aR_1+T_1E_1(S_1T_1E_3$}

53. 6: {b,4,E),$S_2T_1E_2aR_1+T_1E_1(S_1T_1$}

54. 6: {q,4,EF),$S_2T_1E_2aR_1+T_1E_1(S_1T_2$}

55. 1: {q,4,(S)F),$S_2T_1E_2aR_1+T_1E_1(S_1T_2E_1$}

56. 4: {b,4,(S)F),$S_2T_1E_2aR_1+T_1E_1(S_1T_2E_1$}

57. 6: {q,4,aF),$S_2T_1E_2aR_1+T_1E_1(S_1T_2E_2$}

58. 2: {q,5,F),$S_2T_1E_2aR_1+T_1E_1(S_1T_2E_2a$}

59. 1: {q,5,*E),$S_2T_1E_2aR_1+T_1E_1(S_1T_2E_2aF_1$}

60. 2: {q,6,E),$S_2T_1E_2aR_1+T_1E_1(S_1T_2E_2aF_1$*}

61. 1: {q,6,(S)),$S_2T_1E_2aR_1+T_1E_1(S_1T_2E_2aF_1$*$E_1$}

62. 4: {b,6,(S)),$S_2T_1E_2aR_1+T_1E_1(S_1T_2E_2aF_1$*$E_1$}

63. 6: {q,6,a),$S_2T_1E_2aR_1+T_1E_1(S_1T_2E_2aF_1$*$E_2$}

64. 4: {b,6,a),$S_2T_1E_2aR_1+T_1E_1(S_1T_2E_2aF_1$*$E_2$}

65. 6: {q,6,b), $S_2T_1E_2aR_1+T_1E_1(S_1T_2E_2aF_1$*$E_3$}

66. 2: {q,7,),$S_2T_1E_2aR_1+T_1E_1(S_1T_2E_2aF_1$*$E_3$b}

67. 2: {q,8,λ,$S_2T_1E_2aR_1+T_1E_1(S_1T_2E_2aF_1$*$E_3$b)}

68. 3: Разбор закончен, алгоритм завершен.

На основании полученной цепочки номеров альтернатив $S_2T_1E_2R_1T_1E_1S_1T_2E_2F_1E_3$ построим последовательность номеров примененных правил: 2, 7, 14, 3, 7, 13, 1, 8, 14, 9, 15. Получаем левосторонний вывод:

S⇒TR⇒ER⇒aR⇒a+T⇒a+E⇒a+(S)⇒a+(T)⇒a+(EF)⇒a+(aF)⇒a+(a*E)⇒a+(a*b)

Соответствующее ему дерево вывода приведено на рис. 4.2.

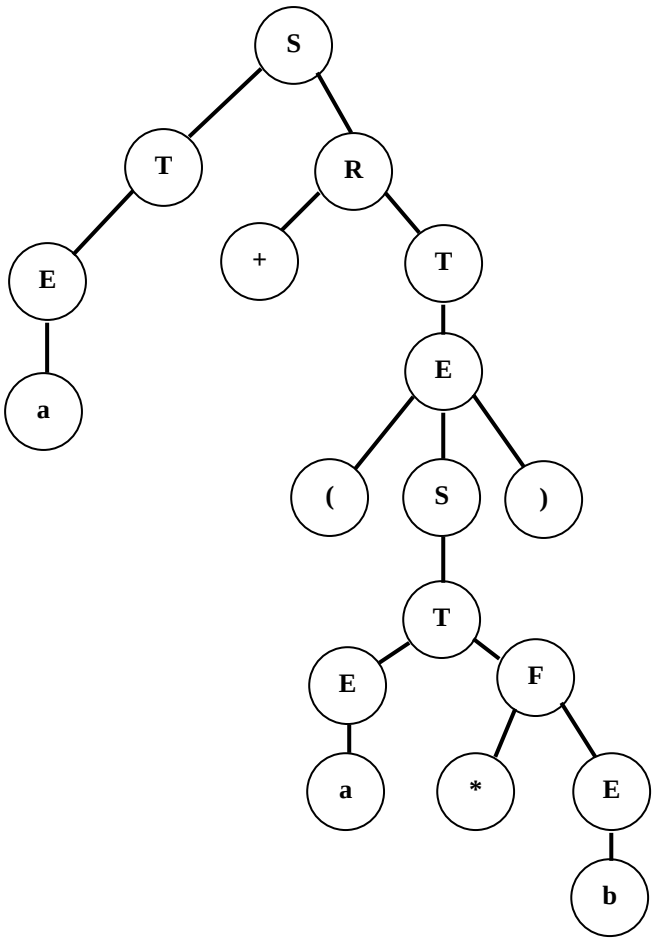


Рис. 4.2. Дерево вывода для грамматики без левых рекурсий

Из приведенного примера очевиден недостаток алгоритма нисходящего разбора с возвратами — значительная временная емкость: для разбора достаточно короткой входной цепочки (всего 7 символов) потребовалось 68 шагов работы алгоритма. Такого результата и следовало ожидать.

Преимуществом данного алгоритма можно считать простоту его реализации. Практически этот алгоритм разбора можно использовать только тогда, когда известно, что длина исходной цепочки символов заведомо невелика. Для реальных компиляторов такое условие невыполнимо, но для некоторых распознавателей вполне допустимо, и тогда данный алгоритм разбора может найти применение именно благодаря своей простоте.

Еще одно преимущество алгоритма — его универсальность. На его основе можно распознавать входные цепочки языка, заданного любой КС-грамматикой, достаточно лишь привести ее к нелеворекурсивному виду (а это можно сделать с любой грамматикой). Интересно, что грамматика даже не обязательно должна быть однозначной — для неоднозначной грамматики алгоритм найдет один из возможных левосторонних выводов.

Сам по себе алгоритм разбора с подбором альтернатив, использующий возвраты, не находит применения в реальных компиляторах. Однако его основные принципы лежат в основе многих нисходящих распознавателей, строящих левосторонние

выводы и работающих без использования возвратов. Методы, позволяющие строить такие распознаватели для некоторых классов КС-языков, рассмотрены далее.

Восходящий распознаватель на основе алгоритма «сдвиг-свертка»

Принцип работы восходящего распознавателя по алгоритму «сдвиг-свертка»

Восходящий распознаватель по алгоритму «сдвиг-свертка» строится на основе расширенного МП-автомата с одним состоянием q: R({q},**V**,**Z**,δ,q,S,{q}). Автомат распознает цепочки КС-языка, заданного КС-грамматикой G(**VT**,**VN**,**P**,S). Входной алфавит автомата содержит терминальные символы грамматики: **V**=**VT**; а алфавит магазинных символов строится из терминальных и нетерминальных символов грамматики: **Z**=**VT**∪**VN**.

Начальная конфигурация автомата определяется так: (q,α,λ) — автомат пребывает в своем единственном состоянии q, считывающая головка находится в начале входной цепочки символов $\alpha \in \mathbf{VT}^*$, стек пуст.

Конечная конфигурация автомата определяется так: (q,λ,S) — автомат пребывает в своем единственном состоянии q, считывающая головка находится за концом входной цепочки символов, в стеке лежит символ, соответствующий целевому символу грамматики S.

Функция переходов МП-автомата строится на основе правил грамматики:

1. $(q,A) \in \delta(q,\lambda,\gamma)$, $A \in \mathbf{VN}$, $\gamma \in (\mathbf{VT} \cup \mathbf{VN})^*$, если правило $A \to \gamma$ содержится во множестве правил **P** грамматики **G**: $A \to \gamma \in \mathbf{P}$;
2. $(q,a) \in \delta(q,a,\lambda)$ $\forall a \in \mathbf{VT}$.

Неформально работу этого расширенного автомата можно описать так: если на верхушке стека находится цепочка символов γ, то ее можно заменить на нетерминальный символ A, если в грамматике языка существует правило вида $A \to \gamma$, не сдвигая при этом считывающую головку автомата (этот шаг работы называется «свертка»); с другой стороны, если считывающая головка автомата обозревает некоторый символ входной цепочки a, то его можно поместить в стек, сдвинув при этом головку на одну позицию вправо (этот шаг работы называется «сдвиг» или «перенос»). Алгоритм, моделирующий работу такого расширенного МП-автомата, называется алгоритмом «сдвиг-свертка» или «перенос-свертка».

Данный расширенный МП-автомат строит правосторонние выводы для грамматики G(**VT**,**VN**,**P**,S). Для моделирования такого автомата необходимо, чтобы грамматика G(**VT**,**VN**,**P**,S) не содержала λ-правил и цепных правил — в противном случае алгоритм может войти в бесконечный цикл из сверток. Поскольку, как было доказано выше, произвольную КС-грамматику всегда можно преобразовать к виду без λ-правил и цепных правил, то этот алгоритм применим для любой КС-грамматики. Следовательно, им можно распознавать цепочки любого КС-языка.

Данный расширенный МП-автомат потенциально имеет больше неоднозначностей, чем рассмотренный выше МП-автомат, основанный на алгоритме подбора альтернатив. На каждом шаге работы автомата надо решать следующие вопросы:

1. что необходимо выполнять: сдвиг или свертку;
2. если выполнять свертку, то какую цепочку γ выбрать для поиска правил (цепочка γ должна встречаться в правой части правил грамматики);
3. какое правило выбрать для свертки, если окажется, что существует несколько правил вида $A \to \gamma$ (несколько правил с одинаковой правой частью).

Чтобы промоделировать работу этого расширенного МП-автомата, надо на каждом шаге запоминать все предпринятые действия, чтобы иметь возможность вернуться к уже сделанному шагу и выполнить эти же действия по-другому. Этот процесс должен повторяться до тех пор, пока не будут перебраны все возможные варианты.

Реализация распознавателя с возвратами на основе алгоритма «сдвиг-свертка»

Существует несколько реализаций алгоритма «сдвиг-свертка» с возвратами [4 т.1, 5, 15, 58]. Один из вариантов рассмотрен ниже.

Для работы алгоритма всем правилам грамматики G(**VT**,**VN**,**P**,S), на основе которой построен автомат, необходимо дать порядковые номера. Будем номеровать правила грамматики в направлении слева направо и сверху вниз в порядке их записи в форме Бэкуса-Наура. Входная цепочка символов имеет вид $\alpha = a_1 a_2 \ldots a_n$, $|\alpha| = n$.

Алгоритм моделирования расширенного МП-автомата, аналогично алгоритму нисходящего распознавателя, использует дополнительное состояние b и дополнительный стек возвратов $\mathbf{L}_2$. В стек помещаются номера правил грамматики, использованных для свертки, если на очередном шаге алгоритма была выполнена свертка, или 0, если на очередном шаге алгоритма был выполнен сдвиг.

В итоге алгоритм работает с двумя стеками: $\mathbf{L}_1$ — стек МП-автомата и $\mathbf{L}_2$ — стек возвратов. Первый представлен в виде цепочки символов, второй — цепочки целых чисел от 0 до m, где m — количество правил грамматики **G**, причем 0 соответствует операции сдвига. Символы в цепочку стека $\mathbf{L}_1$ помещаются справа, числа в стек $\mathbf{L}_2$ — слева. В целом состояние алгоритма на каждом шаге определяется четырьмя параметрами: (Q,i,$\mathbf{L}_1$,$\mathbf{L}_2$), где:

- Q — текущее состояние автомата (q или b);
- i — положение считывающей головки во входной цепочке символов α;
- $\mathbf{L}_1$ — содержимое стека МП-автомата;
- $\mathbf{L}_2$ — содержимое дополнительного стека возвратов.

Начальным состоянием алгоритма является состояние (q,1,λ,λ). Алгоритм начинает свою работу с начального состояния и циклически выполняет пять шагов до тех пор, пока не перейдет в конечное состояние или не обнаружит ошибку.

Алгоритм предусматривает циклическое выполнение следующих шагов:

Шаг 1 (Попытка свертки). $(q,i,\alpha\beta,\gamma)\rightarrow(q,i,\alpha A,j\gamma)$, если $A\rightarrow\beta$ — это первое из всех возможных правил из множества правил **P** с номером j для подцепочки β, причем оно есть первое подходящее правило для цепочки $\alpha\beta$, для которой правило вида $A\rightarrow\beta$ существует. Если удалось выполнить свертку — возвращаемся к шагу 1, иначе — переходим к шагу 2.

Шаг 2 (Перенос — сдвиг). Если $i<n+1$, то $(q,i,\alpha,\gamma)\rightarrow(q,i+1,\alpha a_i,0\gamma)$, $a_i\in$**VT**. Если $i=n+1$, то перейти к шагу 3, иначе перейти к шагу 1.

Шаг 3 (Завершение). Если состояние соответствует $(q,n+1,S,\gamma)$, то разбор завершен, алгоритм заканчивает работу, иначе перейти к шагу 4.

Шаг 4 (Переход к возврату). $(q,n+1,\alpha,\gamma)\rightarrow(b,n+1,\alpha,\gamma)$.

Шаг 5 (Возврат). Если исходное состояние $(b,i,\alpha A,j\gamma)$, то:

- перейти в состояние $(q,i,\alpha'B,k\gamma)$, если $j>0$, и $A\rightarrow\beta$ — это правило с номером j и существует правило $B\rightarrow\beta'$ с номером k, $k>j$, такое, что $\alpha\beta=\alpha'\beta'$; после чего надо вернуться к шагу 1;
- перейти в состояние $(b,n+1,\alpha\beta,\gamma)$, если $i=n+1$, $j>0$, $A\rightarrow\beta$ — это правило с номером j, и не существует других правил из множества **P** с номером $k>j$, таких, что их правая часть является правой подцепочкой из цепочки $\alpha\beta$; после этого вернуться к шагу 5;
- перейти в состояние $(q,i+1,\alpha\beta a_i,0\gamma)$, $a_i\in$**VT**, если $i\neq n+1$, $j>0$, $A\rightarrow\beta$ — это правило с номером j, и не существует других правил из множества **P** с номером $k>j$, таких, что их правая часть является правой подцепочкой из цепочки $\alpha\beta$; после этого перейти к шагу 1;
- иначе сигнализировать об ошибке и прекратить выполнение алгоритма.

Если исходное состояние $(b,i,\alpha a,0\gamma)$, $a\in$**T**, то если $i>1$, тогда перейти в следующее состояние $(b,i-1,\alpha,\gamma)$, и вернуться к шагу 5; иначе сигнализировать об ошибке и прекратить выполнение алгоритма.

В случае успешного завершения алгоритма цепочку вывода можно построить на основе содержимого стека $\mathbf{L}_2$, полученного в результате выполнения алгоритма. Для этого достаточно удалить из стека $\mathbf{L}_2$ все цифры 0.

ВНИМАНИЕ

> Следует помнить, что для применения этого алгоритма исходная грамматика не должна содержать циклов и λ-правил. Если это условие не удовлетворяется, то грамматику надо предварительно преобразовать к приведенной форме.

Возьмем в качестве примера грамматику `G({+,-,/,*,a,b},{S,T,E},P,S)`:

`P:`

`S → S+T | S-T | T*E | T/E | (S) | a | b`

T → T*E | T/E | (S) | a | b

E → (S) | a | b

Это грамматика арифметических выражений, в которой устранены цепные правила (ее уже рассматривали в разделе «Преобразование КС-грамматик. Приведенные грамматики»). Следовательно, в ней не может быть циклов.[3] Кроме того, видно, что в ней нет λ-правил. Таким образом, цепочки языка, заданного этой грамматикой, можно распознавать с помощью алгоритма «сдвиг-свертка» с возвратами.

Проследим разбор цепочки `a+(a*b)` из языка, заданного этой грамматикой. Работу алгоритма будем представлять в виде последовательности его состояний, взятых в скобки {} (фигурные скобки используются, чтобы не путать их с круглыми скобками, предусмотренными в правилах грамматики). Правила будем номеровать слева направо и сверху вниз (всего в грамматике получается 15 правил). Для пояснения каждый шаг работы сопровождается номером шага алгоритма, который был применен для перехода в очередной состояние (записывается слева перед состоянием через символ «:» — двоеточие).

Алгоритм работы восходящего распознавателя с возвратами при разборе цепочки `a+(a*b)` будет выполнять следующие шаги:

1. 0: {q,1,λ,λ}
2. 2: {q,2,a,[0]}
3. 1: {q,2,S,[6,0]}
4. 2: {q,3,S+,[0,6,0]}
5. 2: {q,4,S+(,[0,0,6,0]}
6. 2: {q,5,S+(a,[0,0,0,6,0]}
7. 1: {q,5,S+(S,[6,0,0,0,6,0]}
8. 2: {q,6,S+(S*,[0,6,0,0,0,6,0]}
9. 2: {q,7,S+(S*b,[0,0,6,0,0,0,6,0]}
10. 1: {q,7,S+(S*S,[7,0,0,6,0,0,0,6,0]}
11. 2: {q,8,S+(S*S),[0,7,0,0,6,0,0,0,6,0]}
12. 4: {b,8,S+(S*S),[0,7,0,0,6,0,0,0,6,0]}
13. 5: {b,7,S+(S*S,[7,0,0,6,0,0,0,6,0]}
14. 5: {q,7,S+(S*T,[12,0,0,6,0,0,0,6,0]}

---

[3] На самом деле исходная грамматика для арифметических выражений, которая была рассмотрена в разделе «Проблемы однозначности и эквивалентности грамматик», тоже не содержит циклов (это легко заметить из вида ее правил), однако для чистоты утверждения цепные правила были исключены.

15. 2: {q,8,S+(S*T),[0,12,0,0,6,0,0,0,6,0]}
16. 4: {b,8,S+(S*T),[0,12,0,0,6,0,0,0,6,0]}
17. 5: {b,7,S+(S*T,[12,0,0,6,0,0,0,6,0]}
18. 5: {q,7,S+(S*E,[15,0,0,6,0,0,0,6,0]}
19. 2: {q,8,S+(S*E),[0,15,0,0,6,0,0,0,6,0]}
20. 4: {b,8,S+(S*E),[0,15,0,0,6,0,0,0,6,0]}
21. 5: {b,7,S+(S*E,[15,0,0,6,0,0,0,6,0]}
22. 5: {q,8,S+(S*a),[0,0,0,6,0,0,0,6,0]}
23. 4:{b,8,S+(S*a),[0,0,0,6,0,0,0,6,0]}
24. 5:{b,7,S+(S*a,[0,0,6,0,0,0,6,0]}
25. 5:{b,6,S+(S*,[0,6,0,0,0,6,0]}
26. 5:{b,5,S+(S,[6,0,0,0,6,0]}
27. 5:{q,5,S+(T,[11,0,0,0,6,0]}
28. 2:{q,6,S+(T*,[0,11,0,0,0,6,0]}
29. 2:{q,7,S+(T*b,[0,0,11,0,0,0,6,0]}
30. 1:{q,7,S+(T*S,[7,0,0,11,0,0,0,6,0]}
31. 2:{q,8,S+(T*S),[0,7,0,0,11,0,0,0,6,0]}
32. 4:{b,8,S+(T*S),[0,7,0,0,11,0,0,0,6,0]}
33. 5:{b,7,S+(T*S,[7,0,0,11,0,0,0,6,0]}
34. 5:{q,7,S+(T*T,[12,0,0,11,0,0,0,6,0]}
35. 2:{q,8,S+(T*T),[0,12,0,0,11,0,0,0,6,0]}
36. 4:{b,8,S+(T*T),[0,12,0,0,11,0,0,0,6,0]}
37. 5:{b,7,S+(T*T,[12,0,0,11,0,0,0,6,0]}
38. 5:{q,7,S+(T*E,[15,0,0,11,0,0,0,6,0]}
39. 1:{q,7,S+(S,[3,15,0,0,11,0,0,0,6,0]}
40. 2:{q,8,S+(S),[0,3,15,0,0,11,0,0,0,6,0]}
41. 1:{q,8,S+S,[5,0,3,15,0,0,11,0,0,0,6,0]}
42. 4:{b,8,S+S,[5,0,3,15,0,0,11,0,0,0,6,0]}
43. 5:{q,8,S+T,[10,0,3,15,0,0,11,0,0,0,6,0]}
44. 1:{q,8,S,[1,10,0,3,15,0,0,11,0,0,0,6,0]}
45. 3: Разбор закончен, алгоритм завершен.

На основании полученной цепочки номеров правил: 1, 10, 3, 15, 11, 6 получаем правосторонний вывод:

S⇒S+T⇒S+(S)⇒S+(T*E)⇒S+(T*b)⇒S+(a*b)⇒a+(a*b)

Соответствующее ему дерево вывода приведено на рис. 4.3.

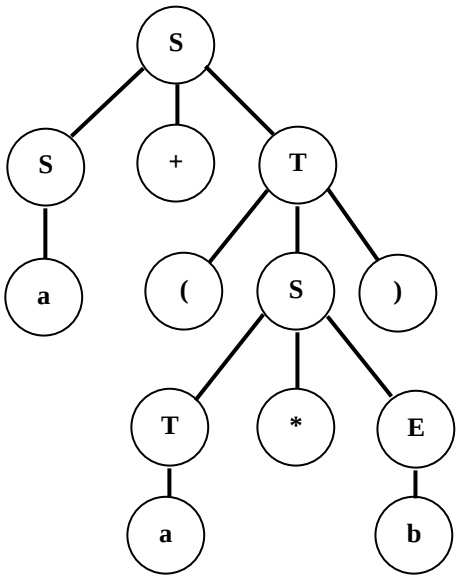


Рис. 4.3. Дерево вывода для грамматики без цепных правил

В приведенном примере очевиден тот же недостаток алгоритма восходящего разбора с возвратами, что и у алгоритма нисходящего разбора с возвратами — значительная временная емкость: для разбора достаточно короткой входной цепочки (всего 7 символов) потребовалось 45 шагов работы алгоритма.

Преимущество у данного алгоритма то же, что и у алгоритма нисходящего разбора с возвратами — простота реализации. Поэтому и использовать его можно практически в тех же случаях — когда известно, что длина исходной цепочки символов заведомо не будет большой.

Этот алгоритм также универсален. На его основе можно распознавать входные цепочки языка, заданного любой КС-грамматикой, достаточно лишь преобразовать ее к приведенному виду, чтобы она не содержала цепных правил и λ-правил.

Сам по себе алгоритм «сдвиг-свертка» с возвратами не находит применения в реальных компиляторах. Однако его базовые принципы лежат в основе многих восходящих распознавателей, строящих правосторонние выводы и работающих без использования возвратов. Методы, позволяющие строить такие распознаватели для некоторых классов КС-языков, рассмотрены далее.

В принципе, два рассмотренных алгоритма — нисходящего и восходящего разбора с возвратами — имеют схожие характеристики по потребным вычислительным

ресурсам и одинаково просты в реализации. То, какой из них лучше взять для реализации простейшего распознавателя в том или ином случае, зависит прежде всего от грамматики языка. В рассмотренном примере восходящий алгоритм смог построить вывод за меньшее число шагов, чем нисходящий — но это еще не значит, что он во всех случаях будет эффективнее для рассмотренного языка арифметических выражений. Вопрос о выборе типа распознавателя — нисходящий либо восходящий — был рассмотрен выше.

Нисходящие распознаватели КС-языков без возвратов

Стремление улучшить алгоритм с подбором альтернатив для нисходящего разбора заключается в первую очередь в определении метода, по которому на каждом шаге алгоритма можно было бы однозначно выбрать одну из всего множества возможных альтернатив. В таком случае алгоритм не требовал бы возврата на предыдущие шаги и за счет этого обладал бы линейными характеристиками. В случае неуспеха выполнения алгоритма входная цепочка не принимается, повторные итерации разбора не выполняются.

Левосторонний разбор по методу рекурсивного спуска

Наиболее очевидным методом выбора одной из множества альтернатив является выбор ее на основании символа $a \in \mathbf{VT}$, обозреваемого считывающей головкой автомата на каждом шаге его работы. Поскольку в процессе нисходящего разбора именно этот символ должен появиться на верхушке магазина для продвижения считывающей головки автомата на один шаг (условие $\delta(q,a,a)=\{(q,\lambda)\}$, $\forall a \in \mathbf{VT}$ в функции переходов МП-автомата), разумно искать альтернативу, где он присутствует в начале цепочки, стоящей в правой части правила грамматики.

По такому принципу действует алгоритм разбора по методу рекурсивного спуска.

Алгоритм разбора по методу рекурсивного спуска

В реализации этого алгоритма для каждого нетерминального символа $A \in \mathbf{VN}$ грамматики $G(\mathbf{VN},\mathbf{VT},\mathbf{P},S)$ строится процедура разбора, которая получает на вход цепочку символов α и положение считывающей головки в цепочке i. Если для символа A в грамматике **G** определено более одного правила, то процедура разбора ищет среди них правило вида $A \to a\gamma$, $a \in \mathbf{VT}$, $\gamma \in (\mathbf{VN} \cup \mathbf{VT})^*$, первый символ правой части которого совпадал бы с текущим символом входной цепочки $a=\alpha_i$. Если такого правила не найдено, то цепочка не принимается.

Название алгоритма происходит из его реализации, которая заключается в последовательности рекурсивных вызовов процедур разбора. Для начала разбора входной цепочки нужно вызвать процедуру для символа S с параметром i=1.

Условия применимости метода можно получить из описания алгоритма — в грамматике $G(\mathbf{VN},\mathbf{VT},\mathbf{P},S)$ $\forall A \in \mathbf{VN}$ возможны только два варианта правил:

$A \to \gamma$, $\gamma \in (\mathbf{VN} \cup \mathbf{VT})^*$ и это единственное правило для A;

$A \to a_1\beta_1 | a_2\beta_2 | \ldots | a_n\beta_n$, $\forall i$: $a_i \in \mathbf{VT}$, $\beta_i \in (\mathbf{VN} \cup \mathbf{VT})^*$ и если $i \neq j$, то $a_i \neq a_j$.

Этим условиям удовлетворяет незначительное количество реальных грамматик.

Это достаточные, но не необходимые условия. Если грамматика не удовлетворяет этим условиям, еще не значит, что заданный ею язык не может распознаваться с помощью метода рекурсивного спуска. Возможно, над грамматикой просто необходимо выполнить ряд дополнительных преобразований.

К сожалению, не существует алгоритма, который бы позволил преобразовать произвольную КС-грамматику к указанному выше виду, равно как не существует и алгоритма, который бы позволял проверить, возможны ли такого рода преобразования. То есть для произвольной КС-грамматики нельзя сказать, анализируем ли заданный ею язык методом рекурсивного спуска или нет.

СОВЕТ

> Можно рекомендовать ряд преобразований, которые способствуют приведению грамматики к требуемому виду (но не гарантируют его достижения).

Эти преобразования заключаются в следующем:

1. исключение λ-правил;
2. исключение левой рекурсии;
3. добавление новых нетерминальных символов (левая факторизация);

например, если правило имеет вид: $A \to a\alpha_1|a\alpha_2|\ldots|a\alpha_n|b_1\beta_1|b_2\beta_2|\ldots|b_m\beta_m$ и ни одна цепочка символов β_j не начинается с символа a, то заменяем его на два: $A \to aA'|b_1\beta_1|b_2\beta_2|\ldots|b_m\beta_m$ и $A' \to \alpha_1|\alpha_2|\ldots|\alpha_n$.

4. замена нетерминальных символов в правилах на цепочки их выводов;

например, если имеются правила:

$A \to B_1|B_2|\ldots|B_n|b_1\beta_1|b_2\beta_2|\ldots|b_m\beta_m$

$B_1 \to \alpha_{11}|\alpha_{12}|\ldots|\alpha_{1k}$

…

$B_n \to \alpha_{n1}|\alpha_{n2}|\ldots|\alpha_{np}$.

заменяем первое правило на:

$A \to \alpha_{11}|\alpha_{12}|\ldots|\alpha_{1k}|\ldots|\alpha_{n1}|\alpha_{n2}|\ldots|\alpha_{np}|b_1\beta_1|b_2\beta_2|\ldots|b_m\beta_m$.

Левую факторизацию можно применять к правилам грамматики несколько раз с целью исключить для каждого нетерминального символа правила, начинающиеся с одних и тех же терминальных символов.

В целом алгоритм рекурсивного спуска эффективен и прост в реализации, но имеет очень ограниченную применимость.

Пример реализации метода рекурсивного спуска

Дана грамматика G({a,b,c},{A,B,C,S},**P**,S):

P:

S → aA | bB

A → a | bA | cC

B → b | aB | cC

C → AaBb

Необходимо построить распознаватель, работающий по методу рекурсивного спуска.

Видно, что грамматика удовлетворяет условиям, необходимым для построения такого распознавателя.

Напишем процедуры на языке программирования C, которые будут обеспечивать разбор входных цепочек языка, заданного данной грамматикой. Согласно алгоритму, необходимо построить процедуру разбора для каждого нетерминального символа грамматики, поэтому дадим процедурам соответствующие наименования. Входные данные для процедур разбора будут следующие:

- цепочка входных символов;
- положение указателя (считывающей головки МП-автомата) во входной цепочке;
- массив для записи номеров примененных правил;
- порядковый номер очередного правила в массиве.

Результатом работы каждой процедуры может быть число, отличное от нуля («истина»), или 0 («ложь»). В первом случае входная цепочка символов принимается распознавателем, во втором случае — не принимается. Для удобства реализации в том случае, если цепочка принимается распознавателем, будем возвращать текущее положение указателя в цепочке. Кроме того, потребуется еще одна дополнительная процедура для ведения записей в массиве последовательности правил (назовем ее *WriteRules*).

```
void WriteRules(int* piRul, int* iP, int iRule)
{
 piRul[*iP] = iRule;
 *iP = *iP + 1;
}


int proc_S (char* szS, int iN, int* piRul, int* iP)
{
 switch (szS[iN])
 {
```

```
    case 'a':
       WriteRules(piRul,iP,1);
       return proc_A(szS,iN+1,piRul,iP);
    case 'b':
       WriteRules(piRul,iP,2);
       return proc_B(szS,iN+1,piRul,iP);
 }
 return 0;
}

int proc_A (char* szS, int iN, int* piRul, int* iP)
{
 switch (szS[iN])
 {
    case 'a':
       WriteRules(piRul,iP,3);
       return iN+1;
    case 'b':
       WriteRules(piRul,iP,4);
       return proc_A(szS,iN+1,piRul,iP);
    case 'c':
       WriteRules(piRul,iP,5);
       return proc_C(szS,iN+1,piRul,iP);
 }
 return 0;
}

int proc_B (char* szS, int iN, int* piRul, int* iP)
{
 switch (szS[iN])
 {
    case 'b':
       WriteRules(piRul,iP,6);
```

```
        return iN+1;
    case 'a':
        WriteRules(piRul,iP,7);
        return proc_B(szS,iN+1,piRul,iP);
    case 'c':
        WriteRules(piRul,iP,8);
        return proc_B(szS,iN+1,piRul,iP);
 }
 return 0;
}

int proc_C (char* szS, int iN, int* piRul, int* iP)
{ int i;
 WriteRules(piRul,iP,9);
 i = proc_A(szS,iN,piRul,iP);
 if (i == 0) return 0;
 if (szS[i] != 'a') return 0;
 i++;
 i = proc_B(szS,i,piRul,iP);
 if (i == 0) return 0;
 if (szS[i] != 'b') return 0;
 return i+1;
}
```

Теперь для распознавания входной цепочки необходимо иметь целочисленный массив `Rules` достаточного объема для хранения номеров правил. Тогда работа распознавателя заключается в вызове процедуры `proc_S(Str,0,Rules,&N)`, где `Str` — это входная цепочка символов, `N` — переменная для запоминания количества примененных правил (первоначально `N=0`). Затем требуется обработка полученного результата: если результат на 1 превышает длину цепочки — цепочка принята, иначе — цепочка не принята. В первом случае в массиве `Rules` будем иметь последовательность номеров правил грамматики, необходимых для вывода цепочки, а в переменной `N` — количество этих правил.

Объем массива `Rules` заранее не известен, так как заранее не известно количество шагов вывода. Чтобы избежать проблем с недостаточным объемом статического массива, приведенные выше процедуры распознавателя можно модифицировать так,

чтобы они работали с динамическим распределением памяти. На логику работы распознавателя это никак не повлияет.[4]

Из приведенного примера видно, что алгоритм рекурсивного спуска удобен и прост в реализации. Главным препятствием в его применении является то, что класс грамматик, допускающих разбор на основе этого алгоритма, сильно ограничен.

Расширенные варианты метода рекурсивного спуска

Метод рекурсивного спуска позволяет выбрать альтернативу, ориентируясь на текущий символ входной цепочки, обозреваемый считывающей головкой МП-автомата. Если имеется возможность просматривать не один, а несколько символов вперед от текущего положения считывающей головки, то можно расширить область применимости метода рекурсивного спуска. В этом случае уже можно искать правила на основе некоторого терминального символа, входящего в правую часть правила. Естественно, и в таком варианте выбор должен быть однозначным — для каждого нетерминального символа в левой части правила необходимо, чтобы в правой части правила не встречалось двух одинаковых терминальных символов.

Этот метод требует также анализа типа присутствующей в правилах рекурсии, поскольку один и тот же терминальный символ может встречаться во входной строке несколько раз, и в зависимости от типа рекурсии следует искать его крайнее левое или крайнее правое вхождение в строке.

Рассмотрим грамматику арифметических выражений для символов `a` и `b`:

`G({+,-,/,*,a,b},{S,T,E},`**`P`**`,S):`

P:

S → S+T | S-T | T

T → T*E | T/E | E

E → (S) | a | b

Это грамматика для арифметических выражений, которая уже была рассмотрена в разделе «Проблемы однозначности и эквивалентности грамматик» и служила основой для построения распознавателей в разделе «Распознаватели КС-языков с возвратом».

Запишем правила этой грамматики в форме с применением метасимволов. Получим:

P:

S → T{(+T,-T)}

---

[4] Следует помнить также, что метод рекурсивного спуска основан на рекурсивном вызове множества процедур, что при значительной длине входной цепочки символов может потребовать соответствующего объема стека вызовов для хранения адресов процедур, их параметров и локальных переменных. Более подробно этот вопрос рассмотрен в разделе «Распределение памяти» данного учебника.

```
T → E{(*E,/E)}

E → (S) | a | b
```

При такой форме записи процедура разбора для каждого нетерминального символа становится тривиальной.

Для символа `S` распознаваемая строка должна всегда начинаться со строки, допустимой для символа `T`, за которой может следовать любое количество символов + или −, и если они найдены, то за ними опять должна быть строка, допустимая для символа `T`. Аналогично, для символа `T` распознаваемая строка должна всегда начинаться со строки, допустимой для символа `E`, за которой может следовать любое количество символов `*` или `/`, и если они найдены, то за ними опять должна быть строка, допустимая для символа `E`. С другой стороны, для символа `E` строка должна начинаться строго с символов `(`, `a` или `b`, причем в первом случае за символом `(` должна следовать строка, допустимая для символа `S`, а за ней — обязательно символ `)`.

Исходя из этого, построены процедуры разбора входной строки на языке Pascal (используется Borland Pascal или Object Pascal, которые допускают тип `string` — строка). Входными данными для них являются:

- исходная строка символов;
- текущее положение указателя в исходной строке;
- длина исходной строки (в принципе, этот параметр можно опустить, но он введен для удобства);
- результирующая строка правил.

Процедуры построены так, что в результирующую строку правил помещаются номера примененных правил в строковом формате, перечисленные через запятую (`,`). Правила номеруются в грамматике, записанной в форме Бэкуса-Наура, в порядке слева направо и сверху вниз (всего в исходной грамматике 9 правил). Распознаватель строит левосторонний вывод, поэтому на основе строки номеров правил всегда можно получить цепочку вывода или дерево вывода.

Для начала разбора нужно вызвать процедуру `proc_S(S,1,N,Pr)`, где:

- `S` — входная строка символов;
- `N` — длина входной строки (в языке Borland Pascal вместо `N` можно взять `Length(S)`);
- `Pr` — строка, куда будет помещена последовательность примененных правил.

Результатом выполнения `proc_S(S,1,N,Pr)` будет `N+1`, если строка `S` принимается, и некоторое число, меньшее `N+1`, если строка не принимается. Если

строка S принимается, то строка Pr будет содержать последовательность номеров правил, которые необходимо применить для того, чтобы вывести S.[5]

```
procedure proc_S (S: string; i,n: integer; var pr: string): integer;
var s1 : string;
begin
 i := proc_T(S,i,n,s1);
 if i > 0 then
 begin
    pr := '3,' + s1;
    while (i <= n) and (i <> 0) do
    case S[i] of
       '+': begin
          if i = n then i := 0
          else
          begin
             i := proc_T(S,i+1,n,s1);
             pr := '1,' + pr +','+ s1;
          end;
       end;
       '-': begin
          if i = n then i := 0
          else
          begin
             i := proc_T(S,i+1,n,s1);
             pr := '2,' + pr +','+ s1;
          end;
```

[5] Использование языка программирования Borland Pascal накладывает определенные технические ограничения на данный распознаватель — длина строки в этом языке не может превышать 255 символов. Однако данные ограничения можно снять, если реализовать свой тип данных строка. При использовании Borland Delphi эти ограничения отпадают. Конечно, такого рода ограничения не имеют принципиального значения при теоретическом исследовании работы распознавателя, тем не менее, автор считает необходимым упомянуть о них.

```
        end;
        else break;
    end;{case}
 end;{if}
 proc_S := i;
end;


procedure proc_S (S: string; i,n: integer; var pr: string):
integer;
var s1 : string;
begin
 i := proc_E(S,i,n,s1);
 if i > 0 then
 begin
    pr := '6,' + s1;
    while (i <= n) and (i <> 0) do
    case S[i] of
        '*': begin
            if i = n then i := 0
            else
            begin
                i := proc_E(S,i+1,n,s1);
                pr := '4,' + pr +','+ s1;
            end;
        end;
        '/': begin
            if i = n then i := 0
            else
            begin
                i := proc_E(S,i+1,n,s1);
                pr := '5,' + pr +','+ s1;
            end;
        end;
```

```
        else break;
    end;{case}
 end;{if}
 proc_S := i;
end;


procedure proc_E (S: string; i,n: integer; var pr: string):
integer;
var s1 : string;
begin
 case S[i] of
    'a': begin
       pr := '8';
       proc_E := i+1;
    end;
    'b': begin
       pr := '9';
       proc_E := i+1;
    end;
    '(': begin
       proc_E := 0;
       if i < n then
       begin
          i := proc_S(S,i+1,n,s1);
          if (i > 0) and (i < n) then
          begin
             pr := '7,' + s1;
             if S[i] = ')' then proc_E := i+1;
          end;
       end;
    end;
    else proc_E := 0;
 end;{case}
```

```
end;
```

Конечно, и в данном случае алгоритм рекурсивного спуска позволил построить достаточно простой распознаватель, однако, прежде чем удалось его применить, потребовался неформальный анализ правил грамматики. Далеко не всегда такого рода неформальный анализ является возможным, особенно если грамматика содержит десятки и даже сотни правил — человек не всегда в состоянии уловить их смысл и взаимосвязь. Поэтому модификации алгоритма рекурсивного спуска, хотя просты и удобны, но не всегда применимы. Даже понять сам факт того, можно или нет в заданной грамматике построить такого рода распознаватель, бывает очень непросто [4 т.1, 5, 12, 15, 18, 21, 58, 59].

Далее будут рассмотрены распознаватели и алгоритмы, которые основаны на строго формальном подходе. Они предваряют построение распознавателя рядом обоснованных действий и преобразований, с помощью которых подготавливаются необходимые исходные данные. В этом случае подготовку всех исходных данных для распознавателя можно формализовать и автоматизировать. Тогда эти предварительные действия можно выполнить с помощью компьютера, в то время как расширенная трактовка рекурсивного спуска предполагает неформальный анализ грамматики человеком, и в этом серьезный недостаток метода.

LL(k)-грамматики

Логическим продолжением идеи, положенной в основу метода рекурсивного спуска, является предложение использовать для выбора одной из множества альтернатив не один, а несколько символов входной цепочки. Однако напрямую переложить алгоритм выбора альтернативы для одного символа на такой же алгоритм для цепочки символов не удастся — два соседних символа в цепочке на самом деле могут быть выведены с использованием различных правил грамматики, поэтому неверным будет напрямую искать их в одном правиле. Тем не менее, существует класс грамматик, основанный именно на этом принципе — выборе одной альтернативы из множества возможных на основе нескольких очередных символов в цепочке. Это LL(k)-грамматики. Правда, алгоритм работы распознавателя для них не так очевидно прост, как рассмотренный выше алгоритм рекурсивного спуска.

Грамматика обладает свойством LL(k), $k>0$, если на каждом шаге вывода для однозначного выбора очередной альтернативы достаточно знать символ на верхушке стека и рассмотреть первые k символов от текущего положения считывающей головки во входной цепочке символов.

Грамматика называется *LL(k)-грамматикой*, если она обладает свойством LL(k) для некоторого $k>0$[6].

---

[6] Требование $k > 0$, безусловно, является разумным — для принятия решения о выборе той или иной альтернативы МП-автомату надо рассмотреть хотя бы один символ входной цепочки. Если представить себе LL-грамматику с $k = 0$, то в такой грамматике вывод совсем не будет зависеть от входной цепочки. В принципе, такая грамматика возможна, но в ней будет всего одна единственная цепочка вывода. Поэтому практическое применение языка, заданного такого рода грамматикой, представляется весьма сомнительным.

Название «LL(k)» несет определенный смысл. Первая литера «L» происходит от слова «left» и означает, что входная цепочка символов читается в направлении слева направо. Вторая литера «L» также происходит от слова «left» и означает, что при работе распознавателя используется левосторонний вывод. Вместо «k» в названии класса грамматики стоит некоторое число, которое показывает, сколько символов надо рассмотреть, чтобы однозначно выбрать одну из множества альтернатив. Так, существуют LL(1)-грамматики, LL(2)-грамматики и другие классы.

В совокупности все LL(k)-грамматики для всех k>0 образуют класс LL-грамматик.

На рис. 4.4 схематично показано частичное дерево вывода для некоторой LL(k)-грамматики. В нем ω обозначает уже разобранную часть входной цепочки α, которая построена на основе левой части дерева `y`. Правая часть дерева `x` — это еще не разобранная часть, а `A` — текущий нетерминальный символ на верхушке стека МП-автомата. Цепочка `x` представляет собой незавершенную часть цепочки вывода, содержащую как терминальные, так и нетерминальные символы. После завершения вывода символ `A` раскрывается в часть входной цепочки υ, а правая часть дерева `x` преобразуется в часть входной цепочки τ. Свойство LL(k) предполагает, что однозначный выбор альтернативы для символа `A` может быть сделан на основе k первых символов цепочки υτ, являющейся частью входной цепочки α.

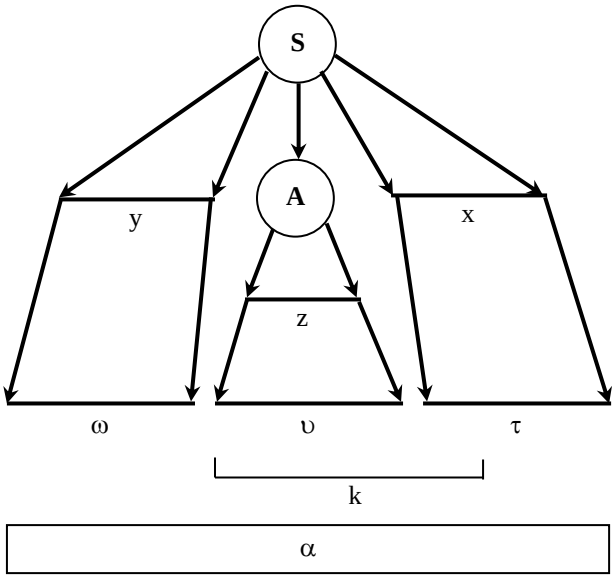


Рис. 4.4. Схема построения дерева вывода для LL(k)-грамматики

Алгоритм разбора входных цепочек для LL(k)-грамматики носит название «k-предсказывающего алгоритма». Принципы его выполнения во многом соответствуют функционированию МП-автомата с той разницей, что на каждом шаге работы этот алгоритм может просматривать k символов вперед от текущего положения считывающей головки автомата.

Для LL(k)-грамматик известны следующие полезные свойства:

- всякая LL(k)-грамматика для любого k>0 является однозначной;
- существует алгоритм, позволяющий проверить, является ли заданная грамматика LL(k)-грамматикой для строго определенного числа k.

Кроме того, известно, что все грамматики, допускающие разбор по методу рекурсивного спуска, являются подклассом LL(1)-грамматик. То есть любая грамматика, допускающая разбор по методу рекурсивного спуска, является LL(1)-грамматикой (но не наоборот!).

Есть, однако, неразрешимые проблемы для LL(k)-грамматик. Это общие проблемы, характерные для всех классов КС-грамматик:

- не существует алгоритма, который бы мог проверить, является ли заданная КС-грамматика LL(k)-грамматикой для некоторого произвольного числа k;
- не существует алгоритма, который бы мог преобразовать произвольную КС-грамматику к виду LL(k)-грамматики для некоторого k или доказать, что преобразование невозможно.

Для LL(k)-грамматики при k>1 совсем не обязательно, чтобы все правые части правил грамматики для каждого нетерминального символа начинались с k различных терминальных символов. Принципы распознавания предложений входного языка такой грамматики накладывают менее жесткие ограничения на правила грамматики, поскольку k соседних символов, по которым однозначно выбирается очередная альтернатива, могут встречаться в нескольких правилах грамматики.

ПРИМЕЧАНИЕ

> Грамматики, у которых все правые части правил для всех нетерминальных символов начинаются с k различных терминальных символов, носят название «сильно LL(k)-грамматики». Метод построения распознавателей для них достаточно прост, алгоритм разбора очевиден, но, к сожалению, такие грамматики встречаются крайне редко.

Поскольку все LL(k)-грамматики используют левосторонний нисходящий распознаватель, основанный на алгоритме с подбором альтернатив, очевидно, что они не могут допускать левую рекурсию.

ВНИМАНИЕ

> Никакая леворекурсивная грамматика не может быть LL-грамматикой. Следовательно, первым делом при попытке преобразовать грамматику к виду LL-грамматики необходимо устранить в ней левую рекурсию.

Класс LL-грамматик широк, но все же он недостаточен для того, чтобы покрыть все возможные синтаксические конструкции в языках программирования. Известно, что существуют ДКС-языки, которые не могут быть заданы LL(k)-грамматикой ни при каких k. Однако LL-грамматики удобны для использования, поскольку позволяют построить линейные распознаватели.

Методы построения распознавателей для LL(k)-грамматик при k>1 в данной книге не рассматриваются, с ними можно ознакомиться в работах [4 т.1].

Синтаксический разбор для LL(1)-грамматик

Очевидно, для каждого нетерминального символа LL(1)-грамматики не может быть двух правил, начинающихся с одного и того же терминального символа. Однако это менее жесткое условие, чем то, которое накладывает распознаватель по методу рекурсивного спуска, поскольку LL(1)-грамматика может допускать в правой части правил цепочки, начинающиеся с нетерминальных символов, а также λ-правила. LL(1)-грамматики позволяют построить достаточно простой и эффективный распознаватель.

Для построения распознавателей языков, заданных LL(k)-грамматиками используются два множества, определяемые следующим образом:

- FIRST(k,α) — множество терминальных цепочек, выводимых из $\alpha \in (\mathbf{VT} \cup \mathbf{VN})^*$, укороченных до k символов;
- FOLLOW(k,A) — множество укороченных до k символов терминальных цепочек, которые могут следовать непосредственно за $A \in \mathbf{VN}$ в цепочках вывода.

Формально эти два множества могут быть определены следующим образом:

$FIRST(k,\alpha) = \{\omega \in \mathbf{VT}^* \mid$ либо $|\omega| \le k$ и $\alpha \Rightarrow^* \omega$, либо $|\omega| > k$ и $\alpha \Rightarrow^* \omega x$, $x \in (\mathbf{VT} \cup \mathbf{VN})^*\}$, $\alpha \in (\mathbf{VT} \cup \mathbf{VN})^*$, $k>0$.

$FOLLOW(k,A) = \{\omega \in \mathbf{VT}^* \mid S \Rightarrow^* \alpha A \gamma$ и $\omega \in FIRST(\gamma,k)\}$, $\alpha \in \mathbf{VT}^*\}$, $A \in \mathbf{VN}$, $k>0$.

Эти множества используются не только для построения распознавателей языков, заданных LL(k)-грамматиками, но и для ряда других классов грамматик, которые будут рассматриваться далее. В случае LL(1)-грамматик используются множества FIRST(1,α) и FOLLOW(1,A).

Для LL(1)-грамматик алгоритм работы распознавателя предельно прост. Он заключается всего в двух условиях, проверяемых на шаге выбора альтернативы. Исходными данными для этих условий являются символ $a \in \mathbf{VT}$, обозреваемый считывающей головкой, и символ $A \in \mathbf{VN}$, находящийся на верхушке стека.

Эти условия можно сформулировать так:

1. необходимо выбрать в качестве альтернативы правило $A \to \alpha$, если $a \in FIRST(1,\alpha)$;
2. необходимо выбрать в качестве альтернативы правило $A \to \lambda$, если оно есть и $a \in FOLLOW(1,A)$.

Если ни одно из этих условий не выполняется, то цепочка не принадлежит заданному языку, и алгоритм должен сигнализировать об ошибке.

Действия алгоритма на шаге «выброса» остаются без изменений.

Кроме того, чтобы убедиться, является ли заданная грамматика $G(\mathbf{VT},\mathbf{VN},\mathbf{P},S)$ LL(1)-грамматикой, необходимо и достаточно проверить следующее условие: для каждого символа $A \in \mathbf{VN}$, для которого в грамматике существует более одного правила вида $A \to \alpha_1|\alpha_2|\ldots|\alpha_n$, должно выполняться требование $FIRST(1,\alpha_i FOLLOW(1,A)) \cap$

FIRST(1,α_jFOLLOW(1,A)) = $\varnothing$ $\forall i \neq j$, $n \geq i > 0$, $n \geq j > 0$. Очевидно, что если для символа $A \in$ **VN** отсутствует правило вида $A \to \lambda$, то согласно этому требованию все множества FIRST(1,α_1), FIRST(1,α_2), …, FIRST(1,α_n) должны попарно не пересекаться, если же присутствует правило $A \to \lambda$, то они не должны также пересекаться с множеством FOLLOW(1,A).

ВНИМАНИЕ

> Очевидно, что LL(1)-грамматика не может содержать для любого нетерминального символа $A \in$ VN двух правил, начинающихся с одного и того же терминального символа.

Условие, накладываемое на правила LL(1)-грамматики, является довольно жестким. Очень немногие реальные грамматики могут быть отнесены к классу LL(1)-грамматик. Например, даже довольно простая грамматика `G({a}, {S}, {S→a|aS}, S)` не удовлетворяет этому условию (хотя она является регулярной праволинейной грамматикой).

Иногда удается преобразовать правила грамматики так, чтобы они удовлетворяли требованию LL(1)-грамматик. Например, приведенная выше грамматика может быть преобразована к виду `G'({a}, {S,A},{S→aA,A→λ|S}, S)`[7]. В такой форме она уже является LL(1)-грамматикой. Но формального метода преобразовать произвольную КС-грамматику к виду LL(1)-грамматики или убедится в том, что такое преобразование невозможно, не существует.

СОВЕТ

> Для приведения произвольной грамматики к виду LL(1)-грамматики можно рекомендовать преобразования, рассмотренные выше при знакомстве с методом рекурсивного спуска. Но применение этих преобразований не гарантирует, что произвольную КС-грамматику удастся привести к виду LL(1)-грамматики.

Для того чтобы запрограммировать работу МП-автомата, выполняющего разбор входных цепочек символов языка, заданного LL(1)-грамматикой, надо научиться строить множества символов FIRST(1,α) и FOLLOW(1,A). Для множества FIRST(1,α) все очевидно, если цепочка α начинается с терминального символа b ($\alpha = b\beta$, $b \in$ **VT**, $\alpha \in$ (**VT**$\cup$**VN**)$^+$, $\beta \in$ (**VT**$\cup$**VN**)*) — в этом случае FIRST(1,α)={b}, если же она начинается с нетерминального символа B ($\alpha = B\beta$, $B \in$ **VN**, $\alpha \in$ (**VT**$\cup$**VN**)$^+$,

---

[7] Можно убедиться, что эти две грамматики задают один и тот же язык: L(G) = L(G'). Это легко сделать, поскольку обе они являются не только КС-грамматиками, но и регулярными праволинейными грамматиками. Кроме того, формальное преобразование G' в G существует — достаточно устранить в грамматике G' λ-правила и цепные правила, и будет получена исходная грамматика G. А вот формального преобразования G в G' нет. В общем случае все может быть гораздо сложнее.

$\beta \in (\mathbf{VT} \cup \mathbf{VN})^*$), то FIRST(1,α)=FIRST(1,B). Следовательно, для LL(1)-грамматик остается только найти алгоритм построения множеств FIRST(1,B) и FOLLOW(1,A) для всех нетерминальных символов $A,B \in \mathbf{VN}$.

Исходными данными для этих алгоритмов служат правила грамматики.

Алгоритм построения множества FIRST(1,A)

Алгоритм строит множества FIRST(1,A) сразу для всех нетерминальных символов грамматики G(**VT**,**VN**,**P**,S), $A \in \mathbf{VN}$. Для выполнения алгоритма надо предварительно преобразовать исходную грамматику G(**VT**,**VN**,**P**,S) в грамматику G'(**VT**,**VN**',**P**',S'), не содержащую λ-правил. На основании полученной грамматики **G**' и выполняется построение множеств FIRST(1,A) для всех $A \in \mathbf{VN}$ (если $A \in \mathbf{VN}$, то согласно алгоритму преобразования, также справедливо $A \in \mathbf{VN}'$). Множества строятся методом последовательного приближения. Если в результате преобразования грамматики **G** в грамматику **G**' множество **VN**' содержит новый символ S', то при построении множества FIRST(1,A) он не учитывается.

Алгоритм состоит из нескольких шагов:

Шаг 1. Для всех $A \in \mathbf{VN}$: $FIRST0(1,A)=\{X \mid A \to X\alpha \in \mathbf{P},\ X \in (\mathbf{VT} \cup \mathbf{VN}),\ \alpha \in (\mathbf{VT} \cup \mathbf{VN})^*\}$ (первоначально вносим во множество первых символов для каждого нетерминального символа A все символы, стоящие в начале правых частей правил для этого символа A); i:=0.

Шаг 2. Для всех $A \in \mathbf{VN}$: $FIRST_{i+1}(1,A)=FIRST_i(1,A) \cup FIRST_i(1,B)$, для всех нетерминальных символов $B \in (FIRST_i(1,A) \cap \mathbf{VN})$.

Шаг 3. Если $\exists A \in \mathbf{VN}$: $FIRST_{i+1}(1,A) \neq FIRST_i(1,A)$, то i:=i+1 и вернуться к шагу 2, иначе перейти к шагу 4.

Шаг 4. Для всех $A \in \mathbf{VN}$: $FIRST(1,A)=FIRST_i(1,A) \setminus \mathbf{VN}$ (исключаем из построенных множеств все нетерминальные символы).

Алгоритм построения множества FOLLOW(1,A)

Алгоритм строит множества FOLLOW(1,A) сразу для всех нетерминальных символов грамматики G(**VT**,**VN**,**P**,S), $A \in \mathbf{VN}$. Для выполнения алгоритма предварительно надо построить все множества FIRST(1,A), $\forall A \in \mathbf{VN}$. Множества строятся методом последовательного приближения. Алгоритм состоит из нескольких шагов:

Шаг 1. Для всех $A \in \mathbf{VN}$: $FOLLOW_0(1,A)=\{X \mid \exists B \to \alpha A X \beta \in \mathbf{P},\ B \in \mathbf{VN},\ X \in (\mathbf{VT} \cup \mathbf{VN}),\ \alpha,\beta \in (\mathbf{VT} \cup \mathbf{VN})^*\}$ (первоначально вносим во множество последующих символов для каждого нетерминального символа A все символы, которые в правых частях правил встречаются непосредственно за символом A); i:=0.

Шаг 2. $FOLLOW_0(1,S)=FOLLOW_0(1,S) \cup \{\lambda\}$ (вносим пустую цепочку во множество последующих символов для целевого символа S — это означает, что в конце разбора за целевым символом цепочка кончается, иногда для этой цели используется специальный символ конца цепочки: $\perp_к$).

Шаг 3. Для всех $A \in \mathbf{VN}$: $FOLLOW'_i(1,A)=FOLLOW_i(1,A) \cup FIRST(1,B)$, для всех нетерминальных символов $B \in (FOLLOW_i(1,A) \cap \mathbf{VN})$.

Шаг 4. Для всех $A \in \mathbf{VN}$: $FOLLOW''_i(1,A) = FOLLOW'_i(1,A) \cup FOLLOW'_i(1,B)$, для всех нетерминальных символов $B \in (FOLLOW'_i(1,A) \cap \mathbf{VN})$, если существует правило $B \to \lambda$.

Шаг 5. Для всех $A \in \mathbf{VN}$: $FOLLOW_{i+1}(1,A)=FOLLOW''_i(1,A) \cup FOLLOW''_i(1,B)$, для всех нетерминальных символов $B \in \mathbf{VN}$, если существует правило $B \to \alpha A$, $\alpha \in (\mathbf{VT} \cup \mathbf{VN})^*$.

Шаг 6. Если $\exists A \in \mathbf{VN}$: $FOLLOW_{i+1}(1,A) \neq FOLLOW_i(1,A)$, то i:=i+1 и вернуться к шагу 3, иначе перейти к шагу 7.

Шаг 7. Для всех $A \in \mathbf{VN}$: $FOLLOW(1,A)=FOLLOW_i(1,A) \setminus \mathbf{VN}$ (исключаем из построенных множеств все нетерминальные символы).

Пример построения распознавателя для LL(1)-грамматики

Рассмотрим в качестве примера грамматику

G({+,-,/,*,a,b},{S,R,T,F,E},**P**,S) с правилами:

P:

S → T | TR

R → +T | −T | +TR | −TR

T → E | EF

F → *E | /E | *EF | /EF

E → (S) | a | b

Это нелеворекурсивная грамматика для арифметических выражений (ранее она была построена в разделе «Распознаватели КС-языков с возвратом»).

Эта грамматика не является LL(1)-грамматикой. Чтобы убедиться в этом, достаточно обратить внимание на правила для символов R и F — для них имеется по два правила, начинающихся с одного и того же терминального символа.

Преобразуем ее в другой вид, добавив λ-правила. В результате получим новую грамматику

G'({+,-,/,*,a,b},{S,R,T,F,E},**P'**,S) с правилами:

P':

S → TR

R → λ | +TR | -TR

T → EF

F → λ | *EF | /EF

E → (S) | a | b

Построенная грамматика **G'** эквивалентна исходной грамматике **G**. В этом можно убедиться, если воспользоваться алгоритмом устранения λ-правил из раздела «Преобразование КС-грамматик». Применив его к грамматике **G'** получим грамматику **G**, а по условиям данного алгоритма: L(**G'**)=L(**G**). Таким образом, мы получили эквивалентную грамматику, хотя она и построена неформальным методом (следует помнить, что не существует формального алгоритма, преобразующего произвольную КС-грамматику в LL(k)-грамматику для заданного k).

Эта грамматика является LL(1)-грамматикой. Чтобы убедиться в этом, построим множества **FIRST** и **FOLLOW** для нетерминальных символов этой грамматики (поскольку речь заведомо идет об LL(1)-грамматике, цифру 1 в обозначении множеств опустим для сокращения записи).

Для построения множества **FIRST** будем использовать исходную грамматику **G**, так как именно она получается из **G'** при устранении λ-правил.

Построение множества **FIRST**:

Шаг 1. $FIRST_0(S) = \{T\}$;

$FIRST_0(R) = \{+,–\}$;

$FIRST_0(T) = \{E\}$;

$FIRST_0(F) = \{*,/\}$;

$FIRST_0(E) = \{(,a,b\}$;

i = 0.

Шаг 2. $FIRST_1(S) = \{T,E\}$;

$FIRST_1(R) = \{+,–\}$;

$FIRST_1(T) = \{E,(,a,b\}$;

$FIRST_1(F) = \{*,/\}$;

$FIRST_1(E) = \{(,a,b\}$.

Шаг 3. i = 1, возвращаемся к шагу 2.

Шаг 2. $FIRST_2(S) = \{T,E,(,a,b\}$;

$FIRST_2(R) = \{+,–\}$;

$FIRST_2(T) = \{E,(,a,b\}$;

$FIRST_2(F) = \{*,/\}$;

$FIRST_2(E) = \{(,a,b\}$.

Шаг 3. i = 2, возвращаемся к шагу 2.

Шаг 2. $FIRST_3(S) = \{T,E,(,a,b\}$;

$FIRST_3(R) = \{+,–\}$;

$FIRST_3(T) = \{E,(,a,b\}$;

$FIRST_3(F) = \{*,/\}$;

$FIRST_3(E) = \{(,a,b\}$.

Шаг 3. i = 2, переходим к шагу 4.

Шаг 4. $FIRST(S) = \{(,a,b\}$;

$FIRST(R) = \{+,–\}$;

$FIRST(T) = \{ (,a,b\}$;

$FIRST(F) = \{*,/\}$;

$FIRST(E) = \{(,a,b\}$;

Построение закончено.

Построение множества **FOLLOW**:

Шаг 1. $FOLLOW_0(S) = \{)\}$;

$FOLLOW_0(R) = \varnothing$;

$FOLLOW_0(T) = \{R\}$;

$FOLLOW_0(F) = \varnothing$;

$FOLLOW_0(E) = \{F\}$;

i = 0.

Шаг 2. $FOLLOW_0(S) = \{),\lambda\}$;

$FOLLOW_0(R) = \varnothing$;

$FOLLOW_0(T) = \{R\}$;

$FOLLOW_0(F) = \varnothing$;

$FOLLOW_0(E) = \{F\}$.

Шаг 3. $FOLLOW'_0(S) = \{),\lambda\}$;

$FOLLOW'_0(R) = \varnothing$;

$FOLLOW'_0(T) = \{R,+,–\}$;

$FOLLOW'_0(F) = \varnothing$;

$FOLLOW'_0(E) = \{F,*,/\}$.

Шаг 4. $FOLLOW''_0(S) = \{),\lambda\}$;

$FOLLOW''_0(R) = \varnothing$;

$FOLLOW''_0(T) = \{R,+,–\}$;

$FOLLOW''_0(F) = \varnothing$;

$FOLLOW''_0(E) = \{F,*,/\}$;

Шаг 5. $FOLLOW_1(S) = \{),\lambda\}$;

$FOLLOW_1(R) = \{),\lambda\};$

$FOLLOW_1(T) = \{R,+,–\};$

$FOLLOW_1(F) = \{R,+,–\};$

$FOLLOW_1(E) = \{F,*,/\};$

Шаг 6. i = 1, возвращаемся к шагу 3.

Шаг 3. $FOLLOW'_1(S) = \{),\lambda\};$

$FOLLOW'_1(R) = \{),\lambda\};$

$FOLLOW'_1(T) = \{R,+,–\};$

$FOLLOW'_1(F) = \{R,+,–\};$

$FOLLOW'_1(E) = \{F,*,/\};$

Шаг 4. $FOLLOW''_1(S) = \{),\lambda\};$

$FOLLOW''_1(R) = \{),\lambda\};$

$FOLLOW''_1(T) = \{R,+,–,),\lambda\};$

$FOLLOW''_1(F) = \{R,+,–,),\lambda\};$

$FOLLOW''_1(E) = \{F,R,*,/,+,–,),\lambda\};$

Шаг 5. $FOLLOW_2(S) = \{),\lambda\};$

$FOLLOW_2(R) = \{),\lambda\};$

$FOLLOW_2(T) = \{R,+,–,),\lambda\};$

$FOLLOW_2(F) = \{R,+,–,),\lambda\};$

$FOLLOW_2(E) = \{F,R,*,/,+,–,),\lambda\};$

Шаг 6. i = 2, возвращаемся к шагу 3.

Шаг 3. $FOLLOW'_2(S) = \{),\lambda\};$

$FOLLOW'_2(R) = \{),\lambda\};$

$FOLLOW'_2(T) = \{R,+,–,),\lambda\};$

$FOLLOW'_2(F) = \{R,+,–,),\lambda\};$

$FOLLOW'_2(E) = \{F,R,*,/,+,–,),\lambda\};$

Шаг 4. $FOLLOW''_2(S) = \{),\lambda\};$

$FOLLOW''_2(R) = \{),\lambda\};$

$FOLLOW''_2(T) = \{R,+,–,),\lambda\};$

$FOLLOW''_2(F) = \{R,+,–,),\lambda\};$

$FOLLOW''_2(E) = \{F,R,*,/,+,–,),\lambda\};$

Шаг 5. $FOLLOW_3(S) = \{),\lambda\}$;

$FOLLOW_3(R) = \{),\lambda\}$;

$FOLLOW_3(T) = \{R,+,–,),\lambda\}$;

$FOLLOW_3(F) = \{R,+,–,),\lambda\}$;

$FOLLOW_3(E) = \{F,R,*,/,+,–,),\lambda\}$;

Шаг 6. i = 2, переходим к шагу 7.

Шаг 7. $FOLLOW(S) = \{),\lambda\}$;

$FOLLOW(R) = \{),\lambda\}$;

$FOLLOW(T) = \{+,–,),\lambda\}$;

$FOLLOW(F) = \{+,–,),\lambda\}$;

$FOLLOW(E) = \{*,/,+,–,),\lambda\}$;

Построение закончено.

В результате выполнения построений можно увидеть, что необходимое и достаточное условие принадлежности КС-грамматики к классу LL(1)-грамматик выполняется.

Построенные множества **FIRST** и **FOLLOW** можно представить в виде таблицы. Результат выполненных построений отражает табл. 4.1.

Таблица 4.1 Множества FIRST и FOLLOW для грамматики G'

Символ A∈VN	FIRST(1,A)	FOLLOW(1,A)
S	(a b	) λ
R	+ –	) λ
T	(a b	+ –) λ
F	* /	+ –) λ
E	(a b	* / + –) λ

Рассмотрим работу распознавателя. Ход разбора будем отражать по шагам работы алгоритма в виде конфигурации МП-автомата, к которой добавлена цепочка, содержащая последовательность примененных правил грамматики. Состояние автомата q, указанное в его конфигурации, можно опустить, так как оно единственное: (α, Z, γ), где

α — непрочитанная часть входной цепочки символов;

Z — содержимое стека (верхушка стека находится слева);

γ — последовательность номеров примененных правил (последовательность дополняется слева, так как автомат порождает левосторонний вывод).

Примем, что правила в грамматике номеруются в порядке слева направо и сверху вниз. На основе номеров примененных правил при успешном завершении разбора можно построить цепочку вывода и дерево вывода.

В качестве примера возьмем две правильных цепочки символов a+a*b и (a+a)*b и две ошибочных цепочки символов a+a* и (+a)*b.

Разбор цепочки a+a*b.

1. (a+a*b,S,λ)
2. (a+a*b,TR,1), так как a∈FIRST(1,TR)
3. (a+a*b,EFR,1,5), так как a∈FIRST(1,EF)
4. (a+a*b,aFR,1,5,10), так как a∈FIRST(1,a)
5. (+a*b, FR, 1,5,10)
6. (+a*b,R,1,5,10,6), так как +∈FOLLOW(1,F)
7. (+a*b,+TR,1,5,10,6,3), так как +∈FIRST(1,+TR)
8. (a*b,TR,1,5,10,6,3)
9. (a*b,EFR,1,5,10,6,3,5), так как a∈FIRST(1,EF)
10. (a*b,aFR,1,5,10,6,3,5,10), так как a∈FIRST(1,a)
11. (*b,FR,1,5,10,6,3,5,10)
12. (*b,*EFR,1,5,10,6,3,5,10,7), так как *∈FIRST(1,*EF)
13. (b,EFR,1,5,10,6,3,5,10,7)
14. (b,bFR,1,5,10,6,3,5,10,7,11), так как b∈FIRST(1,b)
15. (λ,FR,1,5,10,6,3,5,10,7,11)
16. (λ,R,1,5,10,6,3,5,10,7,11,6), так как λ∈FOLLOW(1,F)
17. (λ,λ,1,5,10,6,3,5,10,7,11,6,2), так как λ∈FOLLOW(1,R), разбор закончен. Цепочка принимается.

Получили цепочку вывода:

S ⇒ TR ⇒ EFR ⇒ aFR ⇒ aFR ⇒ aR ⇒ a+TR ⇒ a+EFR ⇒ a+aFR ⇒ a+a*EFR ⇒ a+a*bFR ⇒ a+a*bR ⇒ a+a*b

Соответствующее ей дерево вывода приведено на рис. 4.5.

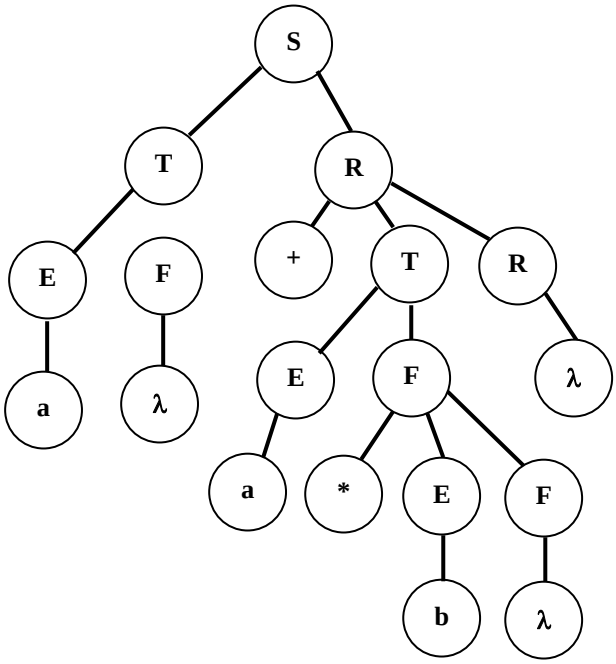


Рис. 4.5. Дерево вывода в LL(1)-грамматике для цепочки «a+a*b»

Разбор цепочки (a+a)*b.

1. ((a+a)*b,S,λ)
2. ((a+a)*b,TR,1), так как (∈FIRST(1,TR)
3. ((a+a)*b,EFR,1,5), так как (∈FIRST(1,EF)
4. ((a+a)*b,(S)FR,1,5,9), так как (∈FIRST(1,(S))
5. (a+a)*b,S)FR,1,5,9)
6. (a+a)*b,TR)FR,1,5,9,1), так как a∈FIRST(1,TR)
7. (a+a)*b,EFR)FR,1,5,9,1,5), так как a∈FIRST(1,EF)
8. (a+a)*b,aFR)FR,1,5,9,1,5,10), так как a∈FIRST(1,a)
9. (+a)*b,FR)FR,1,5,9,1,5,10)
10. (+a)*b,R)FR,1,5,9,1,5,10,6), так как +∈FOLLOW(1,F)
11. (+a)*b,+TR)FR,1,5,9,1,5,10,6,3), так как +∈FIRST(1,+TR)
12. (a)*b,TR)FR,1,5,9,1,5,10,6,3)
13. (a)*b,EFR)FR,1,5,9,1,5,10,6,3,5), так как a∈FIRST(1,EF)
14. (a)*b,aFR)FR,1,5,9,1,5,10,6,3,5,10), так как a∈FIRST(1,a)
15. (q,)*b,FR)FR,1,5,9,1,5,10,6,3,5,10)
16. ()*b,R)FR,1,5,9,1,5,10,6,3,5,10,6), так как)∈FOLLOW(1,F)
17. ()*b,)FR,1,5,9,1,5,10,6,3,5,10,6,2), так как)∈FOLLOW(1,R)
18. (*b,FR,1,5,9,1,5,10,6,3,5,10,6,2)

19. (*b,*EFR,1,5,9,1,5,10,6,3,5,10,6,2,7), так как *∈FOLLOW(1,*EF)
20. (b,EFR,1,5,9,1,5,10,6,3,5,10,6,2,7)
21. (b,bFR,1,5,9,1,5,10,6,3,5,10,6,2,7,11), так как b∈FIRST(1,b)
22. (λ,FR,1,5,9,1,5,10,6,3,5,10,6,2,7,11)
23. (λ,R,1,5,9,1,5,10,6,3,5,10,6,2,7,11,6), так как λ∈ FOLLOW(1,F)
24. (λ,λ,1,5,9,1,5,10,6,3,5,10,6,2,7,11,6,2), так как λ∈FOLLOW(1,R), разбор закончен. Цепочка принимается.

Получили цепочку вывода:

S ⇒ TR ⇒ EFR ⇒ (S)FR ⇒ (TR)FR ⇒ (EFR)FR ⇒ (aFR)FR ⇒ (aR)FR ⇒ (a+TR)FR ⇒ (a+EFR)FR ⇒ (a+aFR)FR ⇒ (a+aR)FR ⇒ (a+a)FR ⇒ (a+a)*EFR ⇒ (a+a)*bFR ⇒ (a+a)*bR ⇒ (a+a)*b

Соответствующее ей дерево вывода приведено на рис. 4.6.

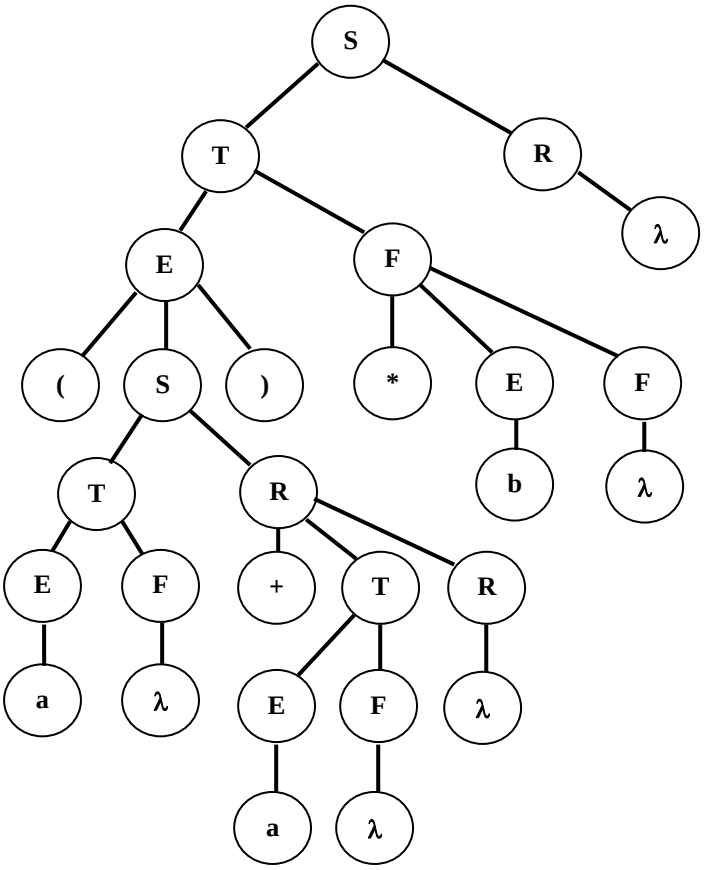


Рис. 4.6. Дерево вывода в LL(1)-грамматике для цепочки «(a+a)*b»

Разбор цепочки a+a*.

1. (a+a*,S,λ)
2. (a+a*,TR,1), так как a∈FIRST(1,TR)
3. (a+a*,EFR,1,5), так как a∈FIRST(1,EF)
4. (a+a*,aFR,1,5,10), так как a∈FIRST(1,a)
5. (+a*,FR,1,5,10)
6. (+a*,R,1,5,10,6), так как +∈FOLLOW(1,F)
7. (+a*,+TR,1,5,10,6,3), так как +∈FIRST(1,+TR)
8. (a*,TR,1,5,10,6,3)
9. (a*,EFR,1,5,10,6,3,5), так как a∈FIRST(1,EF)
10. (a*,aFR,1,5,10,6,3,5,10), так как a∈FIRST(1,a)
11. (*,FR,1,5,10,6,3,5,10)
12. (*,*EFR,1,5,10,6,3,5,10,7), так как *∈FIRST(1,*EF)
13. (λ,EFR,1,5,10,6,3,5,10,7)
14. Ошибка, так как λ∈FOLLOW(1,E), но нет правила вида E → λ. Цепочка не принимается.

Разбор цепочки (+a)*b.

1. ((+a)*b,S,λ)
2. ((+a)*b,TR,1), так как (∈FIRST(1,TR)
3. ((+a)*b,EFR,1,5), так как (∈FIRST(1,EF)
4. ((+a)*b,(S)FR,1,5,9), так как (∈FIRST(1,(S))
5. (+a)*b,S)FR,1,5,9)
6. Ошибка, так как нет правил для S вида S → α таких, чтобы +∈FIRST(1,α) и +∉FOLLOW(1,S). Цепочка не принимается.

Из рассмотренных примеров видно, что алгоритму разбора цепочек, построенному на основе распознавателя для LL(1)-грамматик, требуется гораздо меньше шагов на принятие решения о принадлежности цепочки языку, чем рассмотренному выше алгоритму разбора с возвратами. Надо отметить, что оба алгоритма распознают цепочки одного и того же языка. Данный алгоритм имеет большую эффективность, поскольку при росте длины цепочки количество шагов его растет линейно, а не экспоненциально. Кроме того, ошибка обнаруживается этим алгоритмом сразу, в то время как разбор с возвратами будет просматривать для неверной входной цепочки возможные варианты до конца, пока не переберет их все.

Очевидно, что этот алгоритм является более эффективным, но жесткие ограничения на правила для LL(1)-грамматик сужают возможности его применения.

Восходящие синтаксические распознаватели без возвратов

Восходящие распознаватели выполняют построение дерева вывода снизу вверх. Результатом их работы является правосторонний вывод. Функционирование таких распознавателей основано на модификациях алгоритма «сдвиг-свертка» (или «перенос-свертка»), который был рассмотрен выше. При их создании применяются методы, которые позволяют однозначно выбрать между выполнением «сдвига» («переноса») или «свертки» на каждом шаге алгоритма, а при выполнении свертки однозначно выбрать правило, по которому будет производиться свертка.

LR(k)-грамматики

Идея состоит в том, чтобы модифицировать алгоритм «сдвиг-свертка» таким образом, чтобы на каждом шаге его работы можно было однозначно дать ответ на следующие вопросы:

- что следует выполнять: сдвиг (перенос) или свертку;
- какую цепочку символов α выбрать из стека для выполнения свертки;
- какое правило выбрать для выполнения свертки (в том случае, если существует несколько правил вида $A_1 \to \alpha$, $A_2 \to \alpha$, … $A_n \to \alpha$).

Тогда восходящий алгоритм распознавания цепочек КС-языка не требовал бы выполнения возвратов. Конечно, как уже было сказано, это нельзя сделать в общем случае, для всех КС-языков. Но среди всех КС-языков можно выделить такие классы языков, для которых подобная реализация распознающего алгоритма возможна.

В первую очередь можно использовать тот же самый подход, который был положен в основу определения LL(k)-грамматик. Тогда мы получим другой класс КС-грамматик, который носит название LR(k)-грамматик.

КС-грамматика обладает свойством LR(k), $k \geq 0$, если на каждом шаге вывода для однозначного решения вопроса о выполняемом действии в алгоритме «сдвиг-свертка» («перенос-свертка») достаточно знать содержимое верхней части стека и рассмотреть первые k символов от текущего положения считывающей головки автомата во входной цепочке символов.

Грамматика называется *LR(k)-грамматикой,* если она обладает свойством LR(k) для некоторого $k \geq 0$[8].

---

[8] Существование LR(0)-грамматик уже не является бессмыслицей в отличие от LL(0)-грамматик. Расширенный МП-автомат анализирует несколько символов, находящихся на верхушке стека. Поэтому даже если при k = 0 автомат и не будет смотреть на текущий символ входной цепочки, построенный им вывод все равно будет зависеть от содержимого стека, а значит, и от содержимого входной цепочки.

Название «LR(k)», как и рассмотренное выше «LL(k)», несет определенный смысл. Первая литера «L» обозначает порядок чтения входной цепочки символов: слева—направо. Вторая литера «R» происходит от слова «right» и означает, что в результате работы распознавателя получается правосторонний вывод. Вместо «k» в названии грамматики стоит число, которое показывает, сколько символов входной цепочки надо рассмотреть, чтобы принять решение о действии на каждом шаге алгоритма «сдвиг-свертка». Существуют LR(0)-грамматики, LR(1)-грамматики и другие классы.

В совокупности все LR(k)-грамматики для всех $k \geq 0$ образуют класс LR-грамматик.

На рис. 4.7 схематично показано частичное дерево вывода для некоторой LR(k)-грамматики. В нем ω обозначает уже разобранную часть входной цепочки α, на основе которой построена левая часть дерева `y`. Правая часть дерева `x` — это еще не разобранная часть, а `A` — это нетерминальный символ, к которому на очередном шаге будет свернута цепочка символов `z`, находящаяся на верхушке стека МП-автомата. В эту цепочку уже входит прочитанная, но еще не разобранная часть входной цепочки υ. Правая часть дерева `x` будет построена на основе части входной цепочки τ. Свойство LR(k) предполагает, что однозначный выбор действия, выполняемого на каждом шаге алгоритма «сдвиг-свертка», может быть сделан на основе цепочки υ и `k` первых символов цепочки τ, являющихся частью входной цепочки α. Этим очередным действием может быть свертка цепочки `z` к символу `A` или перенос первого символа из цепочки τ и добавление его к цепочке `z`.

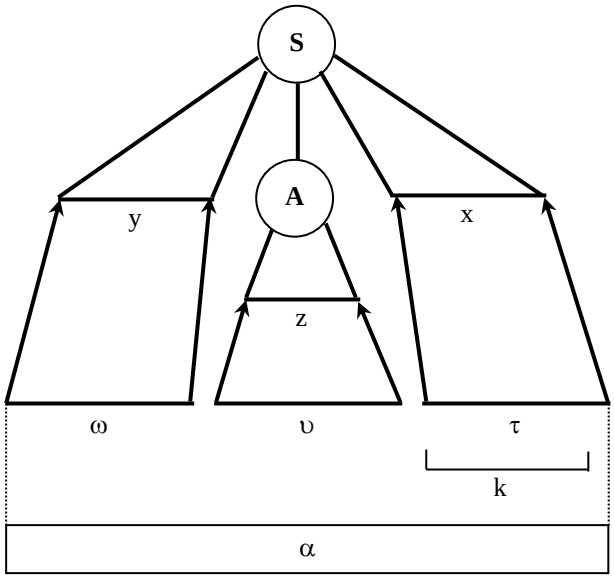


Рис. 4.7. Схема построения дерева вывода для LR(k)-грамматики

Рассмотрев схему построения дерева вывода для LR(k)-грамматики на рис. 4.7 и сравнив ее с приведенной выше на рис. 4.4 схемой для LL(k)-грамматики, можно предположить, что класс LR-грамматик является более широким, чем класс LL-

грамматик. Основанием для такого предположения служит тот факт, что на каждом шаге работы распознавателя LR(k)-грамматики обрабатывается больше информации, чем на шаге работы распознавателя LL(k)-грамматики. Действительно, для принятия решения на каждом шаге алгоритма распознавания LL(k)-грамматики используются первые k символов из цепочки υτ, а для принятия решения на шаге распознавания LR(k)-грамматики — вся цепочка υ и еще первые k символов из цепочки τ. Очевидно, что во втором случае можно проанализировать больший объем информации и, таким образом, построить вывод для более широкого класса КС-языков.

Приведенное выше довольно нестрогое утверждение имеет строго обоснованное доказательство. Доказано, что класс LR-грамматик является более широким, чем класс LL-грамматик [4 т.1, 5]. То есть, для каждого КС-языка, заданного LL-грамматикой, может быть построена LR-грамматика, задающая тот же язык, но не наоборот[9].

Для LR(k)-грамматик известны следующие полезные свойства:

- всякая LR(k)-грамматика для любого k≥0 является однозначной;
- существует алгоритм, позволяющий проверить, является ли заданная грамматика LR(k)-грамматикой для строго определенного числа k.

Есть, однако, неразрешимые проблемы для LR(k)-грамматик. Это общие проблемы, характерные для всех классов КС-грамматик:

- не существует алгоритма, который бы мог проверить, является ли заданная КС-грамматика LR(k)-грамматикой для некоторого произвольного числа k;
- не существует алгоритма, который бы мог преобразовать (или доказать, что преобразование невозможно) произвольную КС-грамматику к виду LR(k)-грамматики для некоторого k.

Формальное определение LR(k) свойства для КС-грамматик можно найти в [4 т.1, 5]. Оно основано на понятии *пополненной КС-грамматики*. Грамматика **G'** является пополненной грамматикой, построенной на основании исходной грамматики G(**VT**,**VN**,**P**,S), если выполняются следующие условия:

- грамматика **G'** совпадает с грамматикой **G**, если целевой символ S не встречается нигде в правых частях правил грамматики **G**;
- грамматика **G'** строится как грамматика G'(**VT**,**VN**∪{S'},**P**∪{S'→S},S'), если целевой символ S встречается в правой части хотя бы одного правила из множества **P** в исходной грамматике **G**.

Фактически пополненная КС-грамматика строится таким образом, чтобы ее целевой символ не встречался в правой части ни одного правила. Если нужно, то в исходную

---

[9] Говоря о соотношении классов LL-грамматик и LR-грамматик, мы не затрагиваем вопрос о значениях k для этих грамматик. Если для некоторой LL(k)-грамматики всегда существует эквивалентная ей LR-грамматика, то это вовсе не значит, что она будет LR(k)-грамматикой с тем же значением k. Но существуют LR-грамматики, для которых нет эквивалентных им LL-грамматик для всех возможных значений k > 0.

грамматику **G** добавляется новый целевой символ S' и новое правило S'→S. Очевидно, что пополненная грамматика **G**' эквивалентна исходной грамматике **G**.

Понятие «пополненная грамматика» введено, чтобы в процессе работы алгоритма «сдвиг-свертка» выполнение свертки к целевому символу S' служило сигналом к завершению алгоритма (поскольку в пополненной грамматике символ S' в правых частях правил не встречается). Поскольку построение пополненных грамматик выполняется элементарно и не накладывает никаких ограничений на исходную КС-грамматику, то дальше будем считать, что все распознаватели для LR(k)-грамматик работают с пополненными грамматиками.

Распознаватель для LR(k)-грамматик функционирует на основе управляющей таблицы **T**. Эта таблица состоит из двух частей, называемых «Действия» и «Переходы». По строкам таблицы распределены все цепочки символов на верхушке стека, которые могут приниматься во внимание в процессе работы распознавателя. По столбцам в части «Действия» распределены все части входной цепочки символов длиной не более k (аванцепочки), которые могут следовать за считывающей головкой автомата в процессе выполнения разбора; а в части «Переходы» — все терминальные и нетерминальные символы грамматики, которые могут появляться на верхушке стека автомата при выполнении действий (сдвигов или сверток).

Клетки управляющей таблицы **T** в части «Действия» содержат следующие данные:

- «*сдвиг*» — если в данной ситуации требуется выполнение сдвига (переноса текущего символа из входной цепочки в стек);
- «*успех*» — если возможна свертка к целевому символу грамматики S, и разбор входной цепочки завершен;
- целое число («*свертка*») — если возможно выполнение свертки (целое число обозначает номер правила грамматики, по которому должна выполняться свертка);
- «*ошибка*» — во всех других ситуациях.

Действия, выполняемые распознавателем, можно вычислять всякий раз на основе состояния стека и текущей аванцепочки. Однако этого вычисления можно избежать, если после выполнения действия сразу же определять, какая строка таблицы **T** будет использована для выбора следующего действия. Тогда эту строку можно поместить в стек вместе с очередным символом и выбрать из стека в момент, когда она понадобится. Таким образом, автомат будет хранить в стеке не только символы алфавита, но и связанные с ними строки управляющей таблицы **T**.

Клетки управляющей таблицы **T** в части «Переходы» как раз и служат для выяснения номера строки таблицы, которая будет использована для определения выполняемого действия на очередном шаге. Эти клетки содержат следующие данные:

- *целое число* — номер строки таблицы **T**;
- «*ошибка*» — во всех других ситуациях.

Для удобства работы распознаватель LR(k)-грамматики использует также два специальных символа $\perp_н$ и $\perp_к$. Считается, что входная цепочка символов всегда начинается символом $\perp_н$ и завершается символом $\perp_к$. Тогда в начальном состоянии

работы распознавателя символ $\perp_н$ находится на верхушке стека, а считывающая головка обозревает первый символ входной цепочки. В конечном состоянии в стеке должны находится символы S (целевой символ) и $\perp_н$, а считывающая головка автомата должна обозревать символ $\perp_к$.

Алгоритм функционирования распознавателя LR(k)-грамматики можно описать следующим образом:

Шаг 1. Поместить в стек символ $\perp_н$ и начальную (нулевую) строку управляющей таблицы **T**. В конец входной цепочки поместить символ $\perp_к$. Перейти к шагу 2.

Шаг 2. Прочитать с вершины стека строку управляющей таблицы **T**. Выбрать из этой строки часть «действие» в соответствии с аванцепочкой, обозреваемой считывающей головкой автомата. Перейти к шагу 3.

Шаг 3. В соответствии с типом действия выполнить выбор из четырех вариантов:

- *«сдвиг»* — если входная цепочка не прочитана до конца, прочитать и запомнить как «новый символ» очередной символ из входной цепочки, сдвинуть считывающую головку на одну позицию вправо, иначе прервать выполнение алгоритма и сообщить об ошибке;
- целое число (*«свертка»*) — выбрать правило в соответствии с номером, удалить из стека цепочку символов, составляющую правую часть выбранного правила, взять символ из левой части правила и запомнить его как «новый символ»;
- *«ошибка»* — прервать выполнение алгоритма, сообщить об ошибке;
- *«успех»* — выполнить свертку к целевому символу S, прервать выполнение алгоритма, сообщить об успешном разборе входной цепочки символов, если входная цепочка прочитана до конца, иначе сообщить об ошибке.

Конец выбора. Перейти к шагу 4.

Шаг 4. Прочитать с вершины стека строку управляющей таблицы **T**. Выбрать из этой строки часть «переход» в соответствии с символом, который был запомнен как «новый символ» на предыдущем шаге. Перейти к шагу 5.

Шаг 5. Если часть «переход» содержит вариант «ошибка», тогда прервать выполнение алгоритма и сообщить об ошибке, иначе (если там содержится номер строки управляющей таблицы **T**) положить в стек «новый символ» и строку таблицы **T** с выбранным номером. Вернуться к шагу 2.

Для работы алгоритма кроме управляющей таблицы **T** используется также временная переменная («новый символ»), хранящая значение терминального или нетерминального символа. В программной реализации алгоритма вовсе не обязательно помещать в стек сами строки управляющей таблицы — поскольку таблица неизменна, достаточно запоминать соответствующие ссылки.

Доказано, что данный алгоритм имеет линейную зависимость необходимых для его выполнения вычислительных ресурсов от длины входной цепочки символов. Следовательно, распознаватель для LR(k)-грамматики является линейным распознавателем.

На практике используются LR(0)-грамматики и LR(1)-грамматики. LR(k)-грамматики со значениями k>1 практически не применяются. Это вызвано двумя причинами:

1. построение распознавателей для LR(k)-грамматик при k>1 связано со значительными сложностями, а управляющая таблица **T** имеет большой объем;
2. доказано, что для любого детерминированного КС-языка может быть построена LR(1)-грамматика, задающая этот язык, поэтому не имеет смысла использовать LR(k)-грамматики со значениями k > 1.

ВНИМАНИЕ

> Класс языков, заданных LR(1)-грамматиками, полностью совпадает с классом детерминированных КС-языков.

То есть, любая LR(1)-грамматика задает ДКС-язык (это очевидно следует из однозначности всех LR-грамматик), и для любого ДКС-языка можно построить LR(1)-грамматику, задающую этот язык. Второе утверждение уже не столь очевидно, но доказано в теории формальных языков [4 т.1, 5].

ПРИМЕЧАНИЕ

> Класс LR(1)-грамматик мог бы быть универсальным для построения синтаксических распознавателей, если бы была разрешима проблема преобразования для КС-грамматик.

Синтаксические конструкции всех языков программирования относятся к классу ДКС-языков. Но ДКС-язык может быть задан грамматикой, которая не относится к классу LR(1)-грамматик. В таком случае совсем не очевидно, что для этого языка удастся построить распознаватель на основе LR(1)-грамматики, потому что, в общем случае, нет алгоритма, который бы позволил эту грамматику получить, хотя и известно, что она существует. То, что проблема не разрешима в общем случае, совсем не означает, что ее не удастся решить в конкретной ситуации. Факт существования LR(1)-грамматики для каждого ДКС-языка играет важную роль — всегда есть смысл попытаться построить такую грамматику.

Синтаксический разбор для LR(0)-грамматик

Построение управляющей таблицы для LR(0)-грамматик

Простейшим случаем LR(k)-грамматик являются LR(0)-грамматики. При k=0 распознающий расширенный МП-автомат совсем не принимает во внимание текущий символ, обозреваемый считывающей головкой. Решение о выполняемом действии принимается только на основании содержимого стека автомата. При этом не должно возникать конфликтов между выполняемым действием (сдвиг или свертка), а также между различными вариантами при выполнении свертки.

Для построения управляющей таблицы **T** заданной LR(0)-грамматики введем понятие последовательности ситуаций.

Ситуация представляет собой множество правил КС-грамматики, записанных с учетом положения считывающей головки МП-автомата, которое может возникнуть в процессе выполнения разбора сентенциальной формы этой грамматики. Положение считывающей головки автомата обозначается в правой части правила $A\to\alpha$, где $\alpha\in(\mathbf{VT}\cup\mathbf{VN})^*$ специальным символом $\bullet$, который может стоять в произвольном месте цепочки α: $\alpha = \gamma\bullet\beta$ (причем любая из цепочек γ и β или обе эти цепочки могут быть пустыми). Этот символ не входит в алфавит грамматики (он не является ни терминальным, ни нетерминальным символом).

Если S — целевой символ грамматики **G**, и правила $S\to\alpha_1|\alpha_2|\ldots|\alpha_n$ входят во множество правил этой грамматики, то ситуация, содержащая множество правил $\{S\to\bullet\alpha_1;\ S\to\bullet\alpha_2;\ \ldots\ S\to\bullet\alpha_n\}$ называется *начальной ситуацией*. Смысл начальной ситуации очевиден: считывающая головка распознающего МП-автомата находится в начале одной из возможных сентенциальных форм грамматики.

Последовательность ситуаций строится по определенным правилам, начиная от начальной ситуации. Правила построения последовательности ситуаций следующие:

- если ситуация содержит правило вида $A\to\gamma\bullet B\beta$, где $A,B\in\mathbf{VN}$, $\gamma,\beta\in(\mathbf{VN}\cup\mathbf{VT})^*$ и при этом правила $B\to\alpha_1|\alpha_2|\ldots|\alpha_m$ входят во множество правил грамматики, то правила вида $B\to\bullet\alpha_1;\ B\to\bullet\alpha_2;\ \ldots\ B\to\bullet\alpha_m$ должны быть добавлены в ситуацию;
- если ситуация содержит правила вида $A\to\gamma\bullet x\beta$, где $A\in\mathbf{VN}$, $\gamma,\beta\in(\mathbf{VN}\cup\mathbf{VT})^*$, а x – произвольный символ ($x\in\mathbf{VN}\cup\mathbf{VT}$), то из нее может быть построена новая ситуация, содержащая правила вида $A\to\gamma x\bullet\beta$, при этом новая ситуация должна проистекать из исходной ситуации на основании символа x.

Ситуации, проистекающие друг из друга на основании различных символов, образуют последовательность ситуаций. Последовательность ситуаций может быть изображена в виде графа взаимосвязи ситуаций, вершины которого соответствуют ситуациям, а дуги — взаимосвязям между ситуациями. Дуги графа взаимосвязи ситуаций будут помечены терминальными и нетерминальными символами грамматики, по которым ситуации связаны между собой.

Поскольку количество правил грамматики конечно, то конечно и количество их комбинаций с символом $\bullet$. Следовательно, конечно и множество всех возможных подмножеств таких комбинаций, то есть и количество возможных ситуаций также конечно. Поэтому последовательность ситуаций всегда может быть построена и никогда не будет бесконечной.

Обозначим все ситуации в последовательности ситуаций $\mathbf{R}_i$, где i — номер ситуации от 0 до N–1 (где N — общее количество ситуаций в последовательности). Тогда начальная ситуация будет обозначаться $\mathbf{R}_0$.

После построения последовательности ситуаций на ее основе строится управляющая таблица **T**. Построение таблицы происходит следующим образом:

- количество строк в таблице **T** соответствует количеству N ситуаций **R** в последовательности ситуаций, причем каждой строке T_i в таблице **T** соответствует ситуация $\mathbf{R}_i$ в последовательности ситуаций;
- для всех ситуаций в последовательности от $\mathbf{R}_0$ до $\mathbf{R}_{N-1}$ выполняется следующее:

- если ситуация $\mathbf{R}_i$ содержит правило вида $S \to \gamma\bullet$, $\gamma \in (\mathbf{VN} \cup \mathbf{VT})^*$, где S — целевой символ грамматики, то в графе *«Действия»* строки T_i таблицы **T** должно быть записано *«успех»*;
- если ситуация $\mathbf{R}_i$ содержит правило вида $A \to \gamma\bullet$, где $A \in \mathbf{VN}$, $\gamma \in (\mathbf{VN} \cup \mathbf{VT})^*$, A не является целевым символом и грамматика содержит правило вида $A \to \gamma$ с номером *m*, то в графе *«Действия»* строки T_i таблицы **T** должно быть записано число *m* (*свертка* по правилу с номером *m*);
- если ситуация $\mathbf{R}_i$ содержит правила вида $A \to \gamma\bullet x\beta$, где $A \in \mathbf{VN}$, а x – произвольный символ ($x \in \mathbf{VN} \cup \mathbf{VT}$), $\gamma,\beta \in (\mathbf{VN} \cup \mathbf{VT})^*$, то в графе *«Действия»* строки T_i таблицы **T** должно быть записано *«сдвиг»*;
- если ситуация $\mathbf{R}_j$ проистекает из ситуации $\mathbf{R}_i$ и связана с нею через символ x ($x \in \mathbf{VN} \cup \mathbf{VT}$), то в графе *«Переходы»*, соответствующей символу x, строки T_i таблицы **T** должно быть записано число *j*;

- клетки таблицы, оставшиеся пустыми после заполнения таблицы T на основании последовательности ситуаций, соответствуют состоянию *«ошибка»*.

Если удается непротиворечивым образом заполнить управляющую таблицу **T** на основании последовательности ситуаций, то рассматриваемая грамматика является LR(0)-грамматикой. Иначе, когда при заполнении таблицы возникают противоречия, грамматика не является LR(0)-грамматикой, и для нее нельзя построить распознаватель типа LR(0) [4 т.1, 5, 12, 59].

Пример построения распознавателя для LR(0)-грамматики

Рассмотрим КС-грамматику `G({a,b}, {S}, {S→aSS|b}, S)`. Пополненная грамматика для нее будет иметь вид: `G'({a,b}, {S,S'}, {S'→S,S→aSS|b}, S')`. Построим последовательность ситуаций для грамматики **`G'`**.

Начальная ситуация $\mathbf{R}_0$ будет содержать правило `S'→ •S`, но в нее также должны быть включены правила `S→•aSS` и `S→•b`, поскольку в грамматике для символа `S` есть правила `S→aSS|b`. Получим ситуацию $\mathbf{R}_0$=`{S'→•S,S→•aSS,S→•b}`.

Из начальной ситуации $\mathbf{R}_0$ по символу `S` можно получить ситуацию $\mathbf{R}_1$, содержащую правило `S'→S•`. Других правил в нее не может быть добавлено, поэтому получаем $\mathbf{R}_1$=`{S'→S•}`.

Из начальной ситуации $\mathbf{R}_0$ по символу `a` можно получить ситуацию $\mathbf{R}_2$, содержащую правило `S→a•SS`, но в нее также должны быть включены правила `S→•aSS` и `S→•b`, поскольку в грамматике для символа `S` есть правила `S→aSS|b`. Получаем $\mathbf{R}_2$=`{S→a•SS,S→•aSS,S→•b}`.

Из начальной ситуации $\mathbf{R}_0$ по символу `b` можно получить ситуацию $\mathbf{R}_3$, содержащую правило `S→b•`. Других правил в нее не может быть добавлено, поэтому получаем $\mathbf{R}_3$=`{S→b•}`.

Из ситуации $\mathbf{R}_2$ по символу `S` можно получить ситуацию $\mathbf{R}_4$, содержащую правило `S→aS•S`, но в нее также должны быть включены правила `S→•aSS` и `S→•b`,

поскольку в грамматике для символа S есть правила S→aSS|b. Получаем **R**$_4$={S→aS•S,S→•aSS,S→•b}.

Из ситуации **R**$_4$ по символу S можно получить ситуацию **R**$_5$, содержащую правило S→aSS•. Других правил в нее не может быть добавлено, поэтому получаем **R**$_5$={S→aSS•}.

Из ситуации **R**$_2$ по символу a можно получить ситуацию **R**$_6$, содержащую правило S→a•SS, но в нее также должны быть включены правила S→•aSS и S→•b, поскольку в грамматике для символа S есть правила S→aSS|b. Получаем **R**$_6$={S→a•SS,S→•aSS,S→•b}. Можно заметить, что эта ситуация совпадает с ситуацией **R**$_2$: **R**$_6$=**R**$_2$.

Из ситуации **R**$_2$ по символу b можно получить ситуацию **R**$_7$, содержащую правило S→b•. Других правил в нее не может быть добавлено, поэтому получаем **R**$_7$={S→b•}. Можно заметить, что эта ситуация совпадает с ситуацией **R**$_3$: **R**$_7$=**R**$_3$.

Из ситуации **R**$_4$ по символу a можно получить ситуацию **R**$_8$, содержащую правило S→a•SS, но в нее также должны быть включены правила S→•aSS и S→•b, поскольку в грамматике для символа S есть правила S→aSS|b. Получаем **R**$_8$={S→a•SS,S→•aSS,S→•b}. Можно заметить, что эта ситуация совпадает с ситуацией **R**$_2$: **R**$_8$=**R**$_2$.

Из ситуации **R**$_4$ по символу b можно получить ситуацию **R**$_9$, содержащую правило S→b•. Других правил в нее не может быть добавлено, поэтому получаем **R**$_9$={S→b•}. Можно заметить, что эта ситуация совпадает с ситуацией **R**$_3$: **R**$_9$=**R**$_3$.

Всего получилось 6 различных ситуаций от **R**$_0$ до **R**$_5$. Граф взаимосвязи ситуаций показан на рис. 4.8.

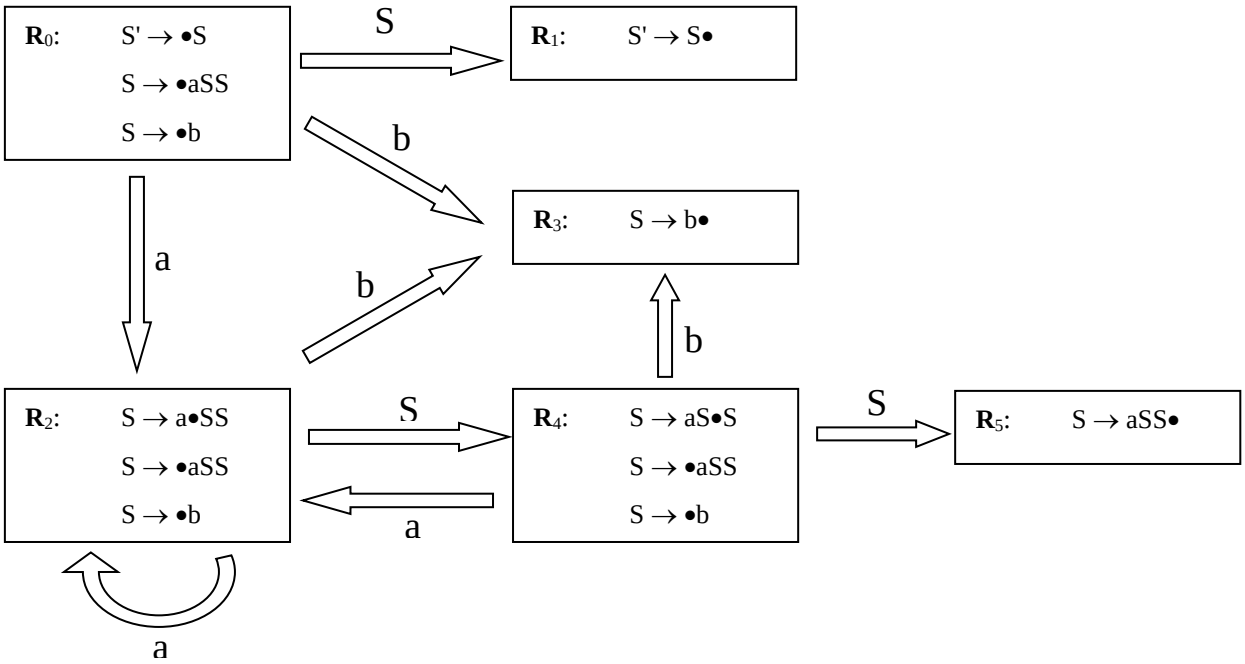


Рис. 4.8. Пример графа взаимосвязи ситуаций для LR(0)-грамматики

На основе найденной последовательности ситуаций можно построить управляющую таблицу распознавателя для LR(0)-грамматики. Поскольку при построении управляющей таблицы не возникает противоречий, то рассмотренная грамматика является LR(0)-грамматикой. Управляющая таблица для нее приведена в табл. 4.2.

Таблица 4.2. Пример управляющей таблицы для LR(0)-грамматики

№	Стек	Действия	Переходы		
			S	A	b
0	$\perp_н$	сдвиг	1	2	3
1	S	успех, 1			
2	a	сдвиг	4	2	3
3	b	свертка, 3			
4	aS	сдвиг	5	2	3
5	aSS	свертка, 2			

Колонка *«Стек»*, присутствующая в таблице, не нужна для распознавателя. Она введена для пояснения состояния стека автомата. Правила грамматики пронумерованы от 1 до 3. Распознаватель работает, не взирая на текущий символ, обозреваемый считывающей головкой расширенного МП-автомата, поэтому колонка *«Действия»* в таблице имеет один столбец, не помеченный никаким символом.

Рассмотрим примеры распознавания цепочек языка, заданного этой грамматикой. Конфигурацию расширенного МП-автомата будем отображать в виде трех компонент: непрочитанная часть входной цепочки символов, содержимое стека МП-автомата, последовательность номеров примененных правил грамматики (поскольку автомат имеет только одно состояние, его можно не учитывать). В стеке МП-автомата вместе с помещенными туда символами показаны и номера строк управляющей таблицы, соответствующие этим символам в формате {*символ*, *номер строки*}.

Разбор цепочки ababab:

1. (abababb$\perp_к$,{$\perp_н$,0},λ)
2. (bababb$\perp_к$,{$\perp_н$,0}{a,2},λ)
3. (ababb$\perp_к$,{$\perp_н$,0}{a,2}{b,3},λ)
4. (ababb$\perp_к$,{$\perp_н$,0}{a,2}{S,4},3)
5. (babb$\perp_к$,{$\perp_н$,0}{a,2}{S,4}{a,2},3)
6. (abb$\perp_к$,{$\perp_н$,0}{a,2}{S,4}{a,2}{b,3},3)
7. (abb$\perp_к$,{$\perp_н$,0}{a,2}{S,4}{a,2}{S,4},3,3)
8. (bb$\perp_к$,{$\perp_н$,0}{a,2}{S,4}{a,2}{S,4}{a,2},3,3)

9. (b⊥к,{⊥н,0}{a,2}{S,4}{a,2}{S,4}{a,2}{b,3},3,3)
10. (b⊥к,{⊥н,0}{a,2}{S,4}{a,2}{S,4}{a,2}{S,4},3,3,3)
11. (⊥к,{⊥н,0}{a,2}{S,4}{a,2}{S,4}{a,2}{S,4}{b,3},3,3,3)
12. (⊥к,{⊥н,0}{a,2}{S,4}{a,2}{S,4}{a,2}{S,4}{S,5},3,3,3,3)
13. (⊥к,{⊥н,0}{a,2}{S,4}{a,2}{S,4}{S,5},3,3,3,3,2)
14. (⊥к,{⊥н,0}{a,2}{S,4}{S,5},3,3,3,3,2,2)
15. (⊥к,{⊥н,0}{S,1},3,3,3,3,2,2,2)
16. (⊥к,{⊥н,0}{S',*},3,3,3,3,2,2,2,1) — разбор завершен.

Соответствующая цепочка вывода будет иметь вид (используется правосторонний вывод):
S'⇒S⇒aSS⇒aSaSS⇒aSaSaSS⇒aSaSaSb⇒aSaSabb⇒aSababb⇒abababb.

Разбор цепочки aabbb:

1. (aabbb⊥к,{⊥н,0},λ)
2. (abbb⊥к,{⊥н,0}{a,2},λ)
3. (bbb⊥к,{⊥н,0}{a,2}{a,2},λ)
4. (bb⊥к,{⊥н,0}{a,2}{a,2}{b,3},λ)
5. (bb⊥к,{⊥н,0}{a,2}{a,2}{S,4},3)
6. (b⊥к,{⊥н,0}{a,2}{a,2}{S,4}{b,3},3)
7. (b⊥к,{⊥н,0}{a,2}{a,2}{S,4}{S,5},3,3)
8. (b⊥к,{⊥н,0}{a,2}{S,4},3,3,2)
9. (⊥к,{⊥н,0}{a,2}{S,4}{b,3},3,3,2)
10. (⊥к,{⊥н,0}{a,2}{S,4}{S,5},3,3,2,3)
11. (⊥к,{⊥н,0}{S,1},3,3,2,3,2)
12. (⊥к,{⊥н,0}{S',*},3,3,2,3,2,1) — разбор завершен.

Соответствующая цепочка вывода будет иметь вид (используется правосторонний вывод): S'⇒S⇒aSS⇒aSb⇒aaSSb⇒aaSbb⇒aabbb.

Разбор цепочки aabb:

1. (aabb⊥к,{⊥н,0},λ)
2. (abb⊥к,{⊥н,0}{a,2},λ)
3. (bb⊥к,{⊥н,0}{a,2}{a,2},λ)
4. (b⊥к,{⊥н,0}{a,2}{a,2}{b,3},λ)

5. (b⊥к,{⊥н,0}{a,2}{a,2}{S,4},3)
6. (⊥к,{⊥н,0}{a,2}{a,2}{S,4}{b,3},3)
7. (⊥к,{⊥н,0}{a,2}{a,2}{S,4}{S,5},3,3)
8. (⊥к,{⊥н,0}{a,2}{S,4},3,3,2)
9. ошибка, невозможно выполнить сдвиг.

Распознаватель для LR(0)-грамматики достаточно прост. Приведенный выше пример можно сравнить с методом рекурсивного спуска или с распознавателем для LL(1)-грамматики — оба эти метода применимы к описанной выше грамматике. По количеству шагов работы распознавателя эти методы сопоставимы, но по реализации нисходящие распознаватели в данном случае немного проще.

Ограничения LR(0)-грамматик

Управляющая таблица **T** для LR(0)-грамматики строится на основании понятия «левых контекстов» нетерминальных символов: после выполнения свертки для нетерминального символа A в стеке МП-автомата ниже этого символа будут располагаться только те символы, которые могут встречаться в цепочке вывода слева от A. Эти символы и составляют «левый контекст» для A. Поскольку выбор между сдвигом или сверткой, а также между типом свертки в LR(0)-грамматиках выполняется только на основании содержимого стека, то LR(0)-грамматика должна допускать однозначный выбор на основе левого контекста для каждого символа [4 т.2, 5].

Однако класс LR(0)-грамматик сильно ограничен, поскольку существует очень мало грамматик, позволяющих выполнять разбор на основании только левого контекста.

Рассмотрим КС-грамматику G({a,b}, {S}, {S→SaSb|λ}, S). Пополненная грамматика для нее будет иметь вид G'({a,b}, {S,S'}, {S'→S,S→SaSb|λ}, S').

Начнем построение последовательности ситуаций для грамматики **G'**.

Начальная ситуация $\mathbf{R}_0$ будет содержать правило S'→•S, но в нее также должны быть включены правила S→•SaSb и S→•, поскольку в грамматике для символа S есть правила S→SaSb|λ (в правиле S→• пустое место соответствует пустой цепочке λ, и это правило в равной степени можно рассматривать как S→•λ или S→λ•). Получим ситуацию $\mathbf{R}_0$={S'→•S,S→•SaSb,S→•}.

При построении управляющей таблицы **T** для этой грамматики уже на шаге для ситуации $\mathbf{R}_0$ возникнут противоречия: с одной стороны, ситуация содержит правило S→•, и для нее в управляющей таблице в графе «Действия» должно быть записано *«свертка»* с номером правила *3*: S→λ (правила в грамматике пронумерованы последовательно от 0 до 3); но с другой стороны, ситуация содержит правила S'→•S и S→•SaSb, поэтому для нее в той же графе «Действия» должно быть записано *«сдвиг»*. Возникшее противоречие говорит о том, что рассматриваемая грамматика не является LR(0)-грамматикой, для нее невозможно выполнить разбор входной цепочки только на основании левого контекста.

ВНИМАНИЕ

> Очевидно, что любая грамматика, содержащая λ–правила не может быть LR(0)-грамматикой.

Можно попытаться выполнить преобразования (например, исключить λ-правила), чтобы привести грамматику к виду LR(0)-грамматики, но это не всегда возможно.

В тех случаях, когда невозможно построить восходящий распознаватель на основе использования только левого контекста, для построения распознавателя используется не только левый, но и правый контекст. Правым контекстом является символ входной цепочки, обозреваемый считывающей головкой расширенного МП-автомата. В этом случае мы получаем распознаватель на основе LR(1)-грамматики[10].

Синтаксический разбор для LR(1)-грамматик

Построение управляющей таблицы для LR(1)-грамматики

Как и для LR(0)-грамматик управляющую таблицу **T** для LR(1)-грамматик можно построить на основании последовательности ситуаций [4 т.2, 5]. Однако ситуация для LR(1)-грамматики имеет более сложную форму, чем для LR(0)-грамматики.

Правило в ситуации для LR(1)-грамматики должно иметь вид: $A\to\gamma\bullet\beta/a$, где $A\in\mathbf{VN}$, $\gamma,\beta\in(\mathbf{VN}\cup\mathbf{VT})^*$, $a\in\mathbf{VT}$. Символ •, а также цепочки γ и β имеют тот же смысл, что и для LR(0)-грамматик, а символ $a\in\mathbf{VT}$ определяет правый контекст.

Начальная ситуация в последовательности ситуаций для LR(1)-грамматики должна содержать правила $\{S\to\bullet\alpha_1/\perp_к, S\to\bullet\alpha_2/\perp_к, \dots S\to\bullet\alpha_n/\perp_к\}$, если правила $S\to\alpha_1|\alpha_2|\dots|\alpha_n$ входят в множество правил этой грамматики, а символ S является целевым символом. Символ $\perp_к$ обозначает символ конца входной цепочки.

Правила построения последовательности ситуаций для LR(1)-грамматики следующие:

- если ситуация содержит правило вида $A\to\gamma\bullet Bb\beta/a$, где $A,B\in\mathbf{VN}$, $a,b\in\mathbf{VT}$, $\gamma,\beta\in(\mathbf{VN}\cup\mathbf{VT})^*$, и при этом правила $B\to\alpha_1|\alpha_2|\dots|\alpha_m$ входят во множество правил грамматики, то правила вида $B\to\bullet\alpha_1/b$; $B\to\bullet\alpha_2/b$; … $B\to\bullet\alpha_m/b$ должны быть добавлены в ситуацию;
- если ситуация содержит правило вида $A\to\gamma\bullet B/a$, где $A,B\in\mathbf{VN}$, $a\in\mathbf{VT}$, $\gamma\in(\mathbf{VN}\cup\mathbf{VT})^*$, и при этом правила $B\to\alpha_1|\alpha_2|\dots|\alpha_m$ входят во множество правил грамматики, то правила вида $B\to\bullet\alpha_1/a$; $B\to\bullet\alpha_2/a$; … $B\to\bullet\alpha_m/a$ должны быть добавлены в ситуацию;
- если ситуация содержит правило вида $A\to\gamma\bullet BC\beta/a$, где $A,B,C\in\mathbf{VN}$, $a,b\in\mathbf{VT}$, $\gamma,\beta\in(\mathbf{VN}\cup\mathbf{VT})^*$, и при этом правила $B\to\alpha_1|\alpha_2|\dots|\alpha_m$ входят во множество правил

---

[10] Как уже было сказано выше, распознаватели для LR(k)-грамматик с $k > 1$ практически не используются.

грамматики, тогда $\forall c \in \mathbf{VT}$, таких, что $c \in FIRST(1,C)$, правила вида $B \to \bullet\alpha_1/c$; $B \to \bullet\alpha_2/c$; … $B \to \bullet\alpha_m/c$ должны быть добавлены в ситуацию;

- если ситуация содержит правила вида $A \to \gamma\bullet x\beta/a$, где $A \in \mathbf{VN}$, $a \in \mathbf{VT}$, а x – произвольный символ ($x \in \mathbf{VN} \cup \mathbf{VT}$), $\gamma,\beta \in (\mathbf{VN} \cup \mathbf{VT})^*$, то из нее может быть построена новая ситуация, содержащая правила вида $A \to \gamma x\bullet\beta/a$, при этом новая ситуация должна проистекать из исходной ситуации на основании символа x.

Здесь FIRST(1,C) обозначает множество первых символов для нетерминального символа $C \in \mathbf{VN}$. Построение этого множества было рассмотрено выше в разделе, посвященном нисходящим распознавателям.

Очевидно, что поскольку правила в ситуациях LR(1)-грамматики учитывают еще и правый контекст, то общее количество возможных ситуаций в LR(1)-грамматике будет существенно больше, чем количество ситуаций в LR(0)-грамматике.

Управляющая таблица **T** для распознавателя на основе LR(1)-грамматики строится на основе последовательности ситуаций так же, как и для LR(0)-грамматики, с учетом того, что графа *«Действия»* в этой таблице разбита на несколько подграф по количеству возможных терминальных символов, каждая из которых соответствует определенному символу правого контекста. Для всех ситуаций в последовательности от $\mathbf{R}_0$ до $\mathbf{R}_{N-1}$ с учетом правого контекста выполняются следующие действия:

- если ситуация $\mathbf{R}_i$ содержит правило вида $S \to \gamma\bullet/a$, $\gamma \in (\mathbf{VN} \cup \mathbf{VT})^*$, $a \in \mathbf{VT}$, где S — целевой символ грамматики, то в графе *«Действия»* строки T_i таблицы **T** для символа *a* должно быть записано *«успех»*;

- если ситуация $\mathbf{R}_i$ содержит правило вида $A \to \gamma\bullet/a$, где $A \in \mathbf{VN}$, $\gamma \in (\mathbf{VN} \cup \mathbf{VT})^*$, $a \in \mathbf{VT}$, A не является целевым символом и грамматика содержит правило вида $A \to \gamma$ с номером *m*, то в графе *«Действия»* строки T_i таблицы **T** для символа *a* должно быть записано число *m* (*свертка* по правилу с номером *m*);

- если ситуация $\mathbf{R}_i$ содержит правила вида $A \to \gamma\bullet x\beta/a$, где $A \in \mathbf{VN}$, $a \in \mathbf{VT}$, а x – произвольный символ ($x \in \mathbf{VN} \cup \mathbf{VT}$), $\gamma,\beta \in (\mathbf{VN} \cup \mathbf{VT})^*$, то в графе *«Действия»* строки T_i таблицы **T** для символа *a* должно быть записано *«сдвиг»*;

- если ситуация $\mathbf{R}_j$ проистекает из ситуации $\mathbf{R}_i$ и связана с нею через символ x ($x \in \mathbf{VN} \cup \mathbf{VT}$), то в графе *«Переход»*, соответствующей символу x, строки T_i таблицы **T** должно быть записано число *j*.

Если удается непротиворечивым образом заполнить управляющую таблицу **T**, то рассматриваемая грамматика является LR(1)-грамматикой. Иначе, когда при заполнении таблицы возникают противоречия, грамматика не является LR(1)-грамматикой, и для нее нельзя построить распознаватель типа LR(1) [4 т.1,2, 5].

Пример построения распознавателя для LR(1)-грамматики

Возьмем КС-грамматику `G({a,b}, {S}, {S→SaSb|λ}, S)`, которая уже рассматривалась выше. Пополненная грамматика для нее будет иметь вид `G'({a,b}, {S,S'}, {S'→S,S→SaSb|λ}, S')`.

Построим последовательность ситуаций для грамматики **`G'`**.

Начальная ситуация $\mathbf{R}_0$ будет содержать правило $S'\to\bullet S/\perp_к$, но в нее также должны быть включены правила $S\to\bullet SaSb/\perp_к$ и $S\to\bullet/\perp_к$, поскольку в грамматике для символа S есть правила $S\to SaSb|\lambda$, а затем, на основании правила $S\to\bullet SaSb/\perp_к$ еще и правила $S\to\bullet SaSb/a$ и $S\to\bullet/a$. Получим ситуацию $\mathbf{R}_0=\{S'\to\bullet S/\perp_к, S\to\bullet SaSb/\perp_к, S\to\bullet/\perp_к, S\to\bullet SaSb/a, S\to\bullet/a\}$.

Из начальной ситуации $\mathbf{R}_0$ по символу S можно получить ситуацию $\mathbf{R}_1$, содержащую правила $S'\to S\bullet/\perp_к$, $S\to S\bullet aSb/\perp_к$ и $S\to S\bullet aSb/a$. Других правил в нее не может быть добавлено, поэтому получаем $\mathbf{R}_1=\{S'\to S\bullet/\perp_к, S\to S\bullet aSb/\perp_к, S\to S\bullet aSb/a\}$.

Из ситуации $\mathbf{R}_1$ по символу a можно получить ситуацию $\mathbf{R}_2$, содержащую правила $S\to Sa\bullet Sb/\perp_к$ и $S\to Sa\bullet Sb/a$, но в нее также должны быть включены правила $S\to\bullet SaSb/b$ и $S\to\bullet/b$, поскольку в грамматике для символа S есть правила $S\to SaSb|\lambda$, а затем, на основании правила $S\to\bullet SaSb/b$ еще и правила $S\to\bullet SaSb/a$ и $S\to\bullet/a$. Получим ситуацию $\mathbf{R}_2=\{S\to Sa\bullet Sb/\perp_к, S\to Sa\bullet Sb/a, S\to\bullet SaSb/b, S\to\bullet/b, S\to\bullet SaSb/a, S\to\bullet/a\}$.

Продолжив построения, получим 8 различных ситуаций от $\mathbf{R}_0$ до $\mathbf{R}_7$, граф взаимосвязи которых показан на рис. 4.9.

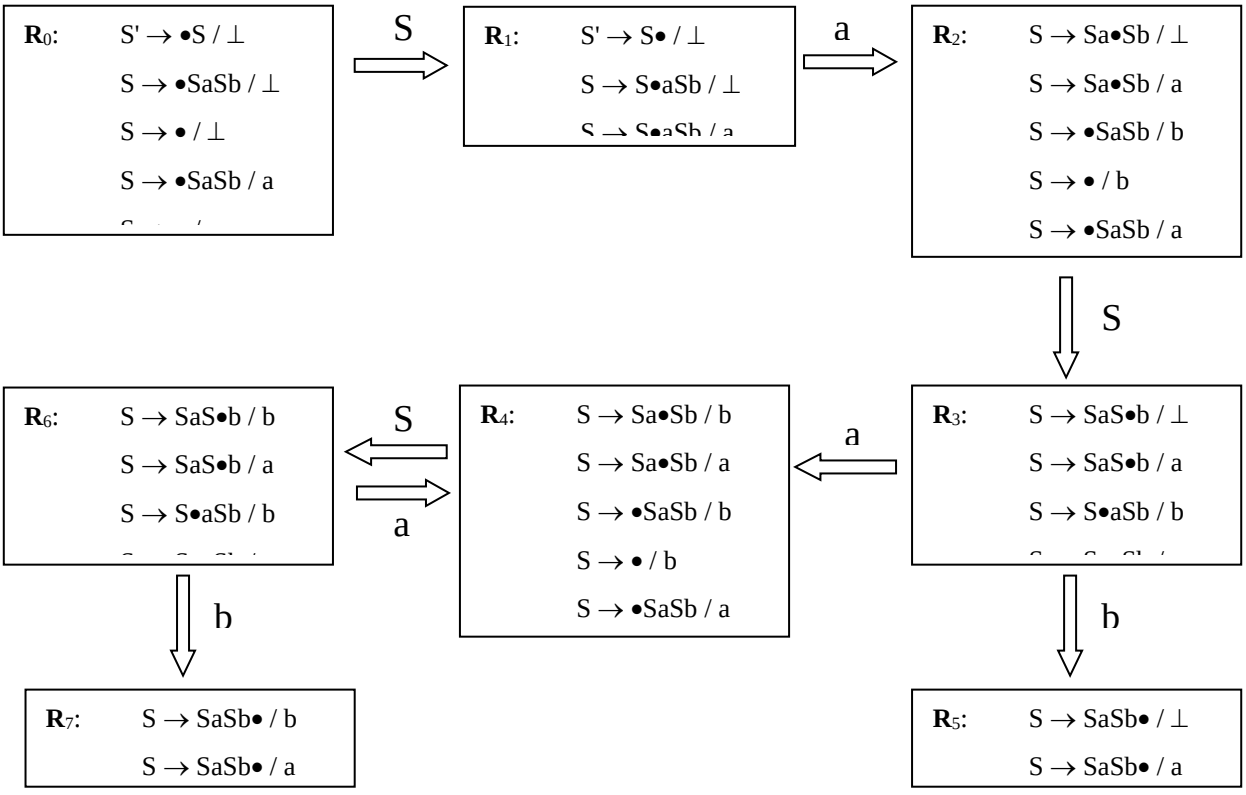


Рис. 4.9. Пример графа взаимосвязи ситуаций для LR(1)-грамматики

На основе найденной последовательности ситуаций можно построить управляющую таблицу распознавателя для LR(1)-грамматики. Поскольку при построении управляющей таблицы не возникает противоречий, то рассмотренная грамматика является LR(1)-грамматикой. Управляющая таблица для нее приведена в табл. 4.3.

Таблица 4.3. Пример управляющей таблицы для LR(1)-грамматики

№	Стек	Действия			Переходы		
		a	b	$\perp_к$	a	B	S
0	$\perp_н$	свертка, 3		свертка, 3			1
1	S	сдвиг		успех, 1	2		
2	Sa	свертка, 3	свертка, 3				3
3	SaS	сдвиг	сдвиг		4	5	
4	SaSa	свертка, 3	свертка, 3				6
5	SaSb	свертка, 2		свертка, 2			
6	SaSaS	сдвиг	сдвиг		4	7	
7	SaSaSb	свертка, 2	свертка, 2				

Колонка *«Стек»*, присутствующая в таблице, не нужна для распознавателя. Она введена для пояснения состояния стека автомата. Пустые клетки в таблице соответствуют состоянию *«ошибка»*. Правила грамматики пронумерованы от 1 до 3. Колонка *«Действия»* в таблице содержит перечень действий, соответствующих текущему входному символу, обозреваемому считывающей головкой расширенного МП-автомата.

Рассмотрим примеры распознавания цепочек языка, заданного этой грамматикой. Конфигурацию расширенного МП-автомата будем отображать в виде трех компонентов, как это было сделано для LR(0)-грамматик.

Разбор цепочки `ababab`:

1. $(ababab\perp_к, \{\perp_н, 0\}, \lambda)$
2. $(ababab\perp_к, \{\perp_н, 0\}\{S, 1\}, 3)$
3. $(babab\perp_к, \{\perp_н, 0\}\{S, 1\}\{a, 2\}, 3)$
4. $(babab\perp_к, \{\perp_н, 0\}\{S, 1\}\{a, 2\}\{S, 3\}, 3, 3)$
5. $(abab\perp_к, \{\perp_н, 0\}\{S, 1\}\{a, 2\}\{S, 3\}\{b, 5\}, 3, 3)$
6. $(abab\perp_к, \{\perp_н, 0\}\{S, 1\}, 3, 3, 2)$
7. $(bab\perp_к, \{\perp_н, 0\}\{S, 1\}\{a, 2\}, 3, 3, 2)$
8. $(bab\perp_к, \{\perp_н, 0\}\{S, 1\}\{a, 2\}\{S, 3\}, 3, 3, 2, 3)$
9. $(ab\perp_к, \{\perp_н, 0\}\{S, 1\}\{a, 2\}\{S, 3\}\{b, 5\}, 3, 3, 2, 3)$

10. (abb⊥к,{⊥н,0}{S,1},3,3,2,3,2)

11. (bb⊥к,{⊥н,0}{S,1}{a,2},3,3,2,3,2)

12. (bb⊥к,{⊥н,0}{S,1}{a,2}{S,3},3,3,2,3,2,3)

13. (b⊥к,{⊥н,0}{S,1}{a,2}{S,3}{b,5},3,3,2,3,2,3)

14. ошибка, нет данных для b в строке 5.

Разбор цепочки aababb:

1. (aababb⊥к,{⊥н,0},λ)

2. (aababb⊥к,{⊥н,0}{S,1},3)

3. (ababb⊥к,{⊥н,0}{S,1}{a,2},3)

4. (ababb⊥к,{⊥н,0}{S,1}{a,2}{S,3},3,3)

5. (babb⊥к,{⊥н,0}{S,1}{a,2}{S,3}{a,4},3,3)

6. (babb⊥к,{⊥н,0}{S,1}{a,2}{S,3}{a,4}{S,6},3,3,3)

7. (abb⊥к,{⊥н,0}{S,1}{a,2}{S,3}{a,4}{S,6}{b,7},3,3,3)

8. (abb⊥к,{⊥н,0}{S,1}{a,2}{S,3},3,3,3,2)

9. (bb⊥к,{⊥н,0}{S,1}{a,2}{S,3}{a,4},3,3,3,2)

10. (bb⊥к,{⊥н,0}{S,1}{a,2}{S,3}{a,4}{S,6},3,3,3,2,3)

11. (b⊥к,{⊥н,0}{S,1}{a,2}{S,3}{a,4}{S,6}{b,7},3,3,3,2,3)

12. (b⊥к,{⊥н,0}{S,1}{a,2}{S,3},3,3,3,2,3,2)

13. (⊥к,{⊥н,0}{S,1}{a,2}{S,3}{b,5},3,3,3,2,3,2)

14. (⊥к,{⊥н,0}{S,1},3,3,3,2,3,2,2)

15. (⊥к,{⊥н,0}{S',*},3,3,3,2,3,2,2,1) — разбор завершен.

Соответствующая цепочка вывода будет иметь вид (используется правосторонний вывод):
S'⇒S⇒SaSb⇒SaSaSbb⇒SaSabb⇒SaSaSbabb⇒SaSababb⇒Saababb⇒aababb.

Сложности построения LR(1)-распознавателей

Класс языков, заданных LR(1)-грамматиками, является самым широким классом КС-языков, допускающим построение линейного восходящего распознавателя. Все детерминированные КС-языки могут быть заданы с помощью LR(1)-грамматик. Однако не всякая однозначная КС-грамматика является LR(1)-грамматикой. И если язык задан грамматикой, не являющейся LR(1)-грамматикой, то в общем случае не существует алгоритма преобразования ее к виду LR(1).

Это проблема общая для всех КС-грамматик. Но для LR(1)-грамматик можно с уверенностью сказать, что если не удалось преобразовать произвольную КС-грамматику к виду LR(1)-грамматики, то нет большого смысла пытаться преобразовать ее к виду LR(k)-грамматики с k>1.

Дело в том, что даже для LR(1)-грамматик управляющая таблица распознавателя имеет значительный объем, а для ее построения необходимо рассмотреть большое количество взаимосвязанных ситуаций. В общем случае, задача имеет комбинаторную сложность, и для реальной LR(1)-грамматики, содержащей несколько десятков правил и несколько десятков терминальных символов, множество рассматриваемых ситуаций будет достаточно велико (сотни возможных ситуаций плюс все взаимосвязи между ними). Сложность построения управляющих таблиц для LR(k)-грамматик при k>1 будет увеличиваться с ростом k в геометрической прогрессии пропорционально количеству терминальных символов грамматики. Поэтому LR(k)-грамматики с k>1 практического применения не имеют.

Например, неоднократно рассмотренная выше грамматика арифметических выражений для символов a и b `G({+,-,/,*,a,b}, {S',S,T,E}, P, S')`:

P:

S' → S

S → S+T | S-T | T

T → T*E | T/E | E

E → (S) | a | b

является LR(1)-грамматикой, но последовательность ситуаций для нее слишком велика, чтобы ее можно было проиллюстрировать в данном учебном пособии (желающие читатели могут построить последовательность ситуаций и управляющую таблицу для данной грамматики, используя приведенный выше алгоритм).

Высокая трудоемкость построения управляющей таблицы для LR(1)-грамматик послужила причиной тому, что для реализации линейных восходящих распознавателей на основе алгоритма «сдвиг-свертка» были предложены другие классы грамматик, такие как SLR(k)-грамматики и LALR(k)-грамматики, грамматики предшествования, грамматики с ограниченным правым контекстом (ОПК) и другие. Не все из этих классов КС-грамматик определяют столь широкий класс КС-языков, как LR(1)-грамматики, но многие из них очень удобны и практичны для построения синтаксических анализаторов.

Далее в этой главе будут рассмотрены еще два класса КС-грамматик — SLR(1)-грамматики и LALR(1)-грамматики, являющиеся незначительной модификацией LR(1)-грамматик. Отдельная глава посвящена грамматикам предшествования. Остальные грамматики из всего множества классов КС-грамматик в данном учебном пособии не рассматриваются, их можно найти в [4 т.2, 5].

SLR(1) и LALR(1)-грамматики

Распознаватели для обоих этих классов грамматик представляют собой модификацию распознавателя для LR(1)-грамматик.

Синтаксический разбор для SLR(1)-грамматик

Идея распознавателей для SLR(k)-грамматик заключается в том, чтобы сократить множество рассматриваемых ситуаций и упростить построение управляющей таблицы **T**. Ведь увеличение количества рассматриваемых ситуаций в LR(1)-грамматике по сравнению с LR(0)-грамматикой происходит из-за того, что в каждом правиле для каждой ситуации учитывается правый контекст. В то же время, реально рассматривать правый контекст необходимо не во всех ситуациях, а только в тех, где возникают конфликты в алгоритме построения таблицы **T**.

Все SLR(k)-грамматики для всех k>1 образуют класс SLR-грамматик (k>1, поскольку SLR(0)-грамматика совпадает по определению с LR(0)-грамматикой). Название SLR-грамматик происходит из самой идеи их построения — SLR означает «Simple LR», то есть, упрощенные LR-грамматики.

В распознавателях на основе SLR(k)-грамматик построение управляющей таблицы **T** выполняется следующим образом:

- строится последовательность ситуаций, соответствующая LR(0)-грамматике;
- на основе построенной последовательности ситуаций заполняется управляющая таблица **T**, причем, если при заполнении таблицы возникают конфликты, то для их разрешения используются правый контекст длиной k и множества символов FIRST(k,α) и FOLLOW(k,A) для правил вида $A\to\alpha$.

Здесь множества FIRST(k,α) и FOLLOW(k,A) представляют собой, соответственно, множество k первых терминальных символов цепочки вывода из α и k первых терминальных символов, которые могут следовать в цепочках вывода за символом A. Эти множества были определены выше в разделе, посвященном LL(1)-грамматикам.

На практике чаще всего применяются SLR(1)-грамматики, поскольку для них построение множеств символов FIRST(1,α) и FOLLOW(1,A) выполняется элементарно просто.

Управляющая таблица **T** для распознавателя на основе SLR(1)-грамматики строится на основе последовательности ситуаций так же, как и для LR(0)-грамматики, но графа «*Действия*» в этой таблице разбита на несколько подграф по количеству возможных терминальных символов, аналогично LR(1)-грамматике**.** Для всех ситуаций в последовательности от $\mathbf{R}_0$ до $\mathbf{R}_{N-1}$ с учетом правого контекста выполняются следующие действия:

- если ситуация $\mathbf{R}_i$ содержит правило вида $S\to\gamma\bullet$, $\gamma\in(\mathbf{VN}\cup\mathbf{VT})^*$, где S — целевой символ грамматики, то в графе *«Действия»* строки T_i таблицы **T** должно быть записано *«успех»* для всех символов $a\in$FOLLOW(1,S) — фактически, таким символом является символ конца строки $\perp_к$;
- если ситуация $\mathbf{R}_i$ содержит правило вида $A\to\gamma\bullet$, где $A\in\mathbf{VN}$, $\gamma\in(\mathbf{VN}\cup\mathbf{VT})^*$, A не является целевым символом и грамматика содержит правило вида $A\to\gamma$ с номером *m*, то в графе *«Действия»* строки T_i таблицы **T** должно быть записано число *m* (*свертка* по правилу с номером *m*) для всех символов $a\in$FOLLOW(1,A);

- если ситуация $\mathbf{R}_i$ содержит правила вида $A \to \gamma \bullet x \beta$, где $A \in \mathbf{VN}$, а x – произвольный символ ($x \in \mathbf{VN} \cup \mathbf{VT}$), $\gamma, \beta \in (\mathbf{VN} \cup \mathbf{VT})^*$, то в графе *«Действия»* строки T_i таблицы **T** должно быть записано *«сдвиг»* для всех символов $a \in \mathrm{FIRST}(1,x)$[11];
- если ситуация $\mathbf{R}_j$ проистекает из ситуации $\mathbf{R}_i$ и связана с нею через символ x ($x \in \mathbf{VN} \cup \mathbf{VT}$), то в графе *«Переход»*, соответствующей символу x, строки T_i таблицы **T** должно быть записано число *j*.

Если удается непротиворечивым образом заполнить управляющую таблицу **T**, то рассматриваемая грамматика является SLR(1)-грамматикой. Иначе, когда при заполнении таблицы возникают противоречия, грамматика не является SLR(1)-грамматикой и для нее нельзя построить распознаватель типа SLR(1) [4 т.2, 5, 58].

Например, рассмотрим грамматику арифметических выражений для символов a и b `G({+,-,/,*,a,b}, {S',S,T,E}, P, S')`:

P:

`S' → S`

`S → S+T | S-T | T`

`T → T*E | T/E | E`

`E → (S) | a | b`

Если начать строить множество ситуаций LR(0)-грамматики для этой грамматики, то начальная ситуация будет выглядеть так: $\mathbf{R}_0$={S'→•S, S→•S+T, S→•S-T, S→•T, T→•T*E, T→•T/E, T→•E, E→•(S), E→•a, E→•b}. Из нее по символу S можно получить ситуацию $\mathbf{R}_1$={S'→S•, S→S•+T, S→S•-T}, а по символу T — ситуацию $\mathbf{R}_2$={S→T•, T→T•*E, T→T•/E}.

При построении управляющей таблицы T для этой грамматики на основе алгоритма для LR(0)-грамматик в состояниях $\mathbf{R}_1$ и $\mathbf{R}_2$ возникнут противоречия, поэтому данная грамматика не является LR(0)-грамматикой. Однако если воспользоваться алгоритмом для SLR(1)-грамматики, то противоречий удается избежать:

- для ситуации $\mathbf{R}_1$, приняв во внимание, что FOLLOW(1,S')={$\perp_к$}, а FIRST(1,+)={+} и FIRST(1,-)={-}, получим, что для правого контекста $\perp_к$ («конец строки») необходимо сигнализировать об успехе, а для правых контекстов + и - — выполнять сдвиг (остальные варианты правого контекста в этой ситуации предполагают ошибку);
- для ситуации $\mathbf{R}_2$, найдем FOLLOW(1,S), получим FOLLOW(1,S)={+,-,)}, FIRST(1,*)={*} и FIRST(1,/)={/}, получим, что для правых контекстов +,-,) необходимо выполнять свертку по правилу S→T, а для правых контекстов * и / — сдвиг (остальные варианты правого контекста в этой ситуации предполагают ошибку).

---

[11] Напоминаем, что для терминального символа b справедливо FIRST(1,b) = {b}.

Построив полностью последовательность ситуаций для этой грамматики и управляющую таблицу **T** на основе данной последовательности, можно убедиться, что рассмотренная грамматика является SLR(1)-грамматикой.

Ограничения SLR(1)-грамматик

Языки, заданные SLR(1)-грамматиками, представляют собой более широкий класс КС-языков, чем языки, заданные LR(0)-грамматиками.

С другой стороны, SLR(1)-грамматики позволяют сократить объем управляющей таблицы **T** распознавателя за счет того, что количество строк в ней соответствует количеству состояний LR(0)-грамматики, которое зачастую существенно меньше, чем количество состояний LR(1)-грамматики. В то же время, они используют более простой алгоритм исключения конфликтных ситуаций при построении этой таблицы, чем LR(1)-грамматики. В этом их преимущество перед LR(1)-грамматиками.

Однако не всякая LR(1)-грамматика является SLR(1)-грамматикой. Оказывается, что класс языков, заданных SLR(1)-грамматиками уже, чем класс языков, заданных LR(1)-грамматиками. Это значит, что он уже, чем класс ДКС-языков.

ВНИМАНИЕ

> Не всякий детерминированный КС-язык может быть задан SLR(1)-грамматикой.

Возьмем рассмотренную ранее КС-грамматику `G({a,b}, {S}, {S→SaSb|λ}, S)` с пополненной грамматикой `G'({a,b}, {S,S'}, {S'→S,S→SaSb|λ}, S')`.

Как было показано выше, начальная ситуация для этой грамматики имеет вид $\mathbf{R}_0$=`{S'→•S,S→•SaSb,S→•}`. При построении управляющей таблицы **T** на основе алгоритма для LR(0)-грамматики возникают противоречия. Если попытаться воспользоваться алгоритмом для SLR(1)-грамматики, то необходимо вычислить множества `FIRST(1,S)` и `FOLLOW(1,S)`. Получим `FIRST(1,S)={a}` и `FOLLOW(1,S)={a,b}`. Оказывается, что для правого контекста `a` конфликта избежать не удается: с одной стороны, в этой ситуации для него необходимо выполнять сдвиг, поскольку `S→•SaSb` входит в ситуацию и `a∈FIRST(1,S)`, но с другой стороны, для него надо выполнять свертку по правилу `S→λ`, поскольку `S→•` входит в ситуацию, и `a∈FOLLOW(1,S)`. Следовательно, эта грамматика не является SLR(1)-грамматикой.

Для того чтобы расширить класс языков, заданных упрощенными LR(1)-грамматиками, был предложен еще один класс грамматик — LALR(1)-грамматики.

Синтаксический разбор для LALR(1)-грамматик

Идея распознавателей для LALR(k)-грамматик та же самая, что и для SLR(k)-грамматик: сократить множество рассматриваемых ситуаций до множества ситуаций LR(0)-грамматики и упростить построение управляющей таблицы **T**, но при этом

найти алгоритм разрешения конфликтов при построении для ситуаций, в которых эти конфликты возникают, на основании правого контекста длиной k.

Все LALR(k)-грамматики для всех k>1 образуют класс LALR-грамматик (k>1, поскольку LALR(0)-грамматика совпадает по определению с LR(0)-грамматикой). Название LALR-грамматик происходит из идеи, которая используется для разрешения конфликтов — LALR означает «Look Ahead», то есть, LR-грамматики с «заглядыванием вперед».

На практике используются только LALR(1)-грамматики, так как применяемый для них метод разрешения конфликтов достаточно сложно распространить на LALR(k)-грамматики с k>1.

В распознавателях на основе LALR(1)-грамматик построение управляющей таблицы **T** выполняется так же, как и для SLR(1)-грамматик, но разрешение конфликтов выполняется не на основании множества FOLLOW(1,A), а путем анализа правого контекста для каждого конфликтного правила в ситуации на основании того, как это правило попало в данную ситуацию. Если анализ контекста допускает выполнение свертки для некоторого правого контекста, то для этого правого контекста в управляющую таблицу **T** записывается действие *«свертка»* на основании конфликтного правила. Действие *«сдвиг»* записывается для всех остальных символов правого контекста, которые входят в множество FIRST(1,β) правил вида A→α•β и не допускают выполнение свертки (более подробное описание анализа правого контекста для LALR(1)-грамматик можно найти в [4 т.2, 5, 58].

Возьмем КС-грамматику `G({a,b}, {S}, {S→SaSb|λ}, S)` с пополненной грамматикой `G'({a,b}, {S,S'}, {S'→S,S→SaSb|λ}, S')`.

Построим для нее последовательность ситуаций:

$\mathbf{R}_0$ = {S'→•S, S → •SaSb, S → •};

$\mathbf{R}_1$ = {S'→S•, S→S•aSb} (порождена из $\mathbf{R}_0$ по символу S);

$\mathbf{R}_2$ = {S→Sa•Sb, S→•SaSb, S→•} (порождена из $\mathbf{R}_1$ и из $\mathbf{R}_3$ по символу a);

$\mathbf{R}_3$ = {S→SaS•b, S→S•aSb} (порождена из $\mathbf{R}_2$ по символу S);

$\mathbf{R}_4$ = {S→SaSb•} (порождена из $\mathbf{R}_3$ по символу b).

Конфликты присутствуют в ситуациях $\mathbf{R}_1$ и $\mathbf{R}_2$.

Рассмотрим ситуацию $\mathbf{R}_1$. В ней возможна свертка по правилу `S'→S•` или сдвиг в соответствии с правилом `S→S•aSb`. Если проанализировать правило `S'→S•`, то видно, что оно появилось в ситуации $\mathbf{R}_1$ из правила `S'→•S` ситуации $\mathbf{R}_0$, следовательно, правым контекстом для этого правила может быть только символ $\perp_к$. Тогда для ситуации $\mathbf{R}_1$ имеем, что для правого контекста $\perp_к$ должна выполняться свертка по правилу `S'→S` (а поскольку в правой части правила стоит целевой символ `S'`, то это соответствует действию *«успех»*), а для правого контекста `a` — сдвиг `(FIRST(1,aSb)={a})`.

Аналогично, для ситуации $\mathbf{R}_2$ возможна свертка по правилу `S→•` или сдвиг в соответствии с правилами `S→Sa•Sb` и `S→•SaSb`. Если проанализировать правило

S→•, то видно, что оно могло появиться в ситуации $\mathbf{R}_2$ из правила S→Sa•Sb этой же ситуации — тогда правым контекстом для него должен быть символ b, но это же правило могло появиться в ситуации $\mathbf{R}_2$ также на основании правила S→•SaSb из этой же ситуации — тогда правым контекстом для него будет символ a. Получаем, что в ситуации $\mathbf{R}_2$ для правых контекстов a и b должна выполняться свертка по правилу S→λ. Сдвиг в этой ситуации выполняться не должен, поскольку FIRST(1,Sb)=FIRST(1,SaSb)={a,b}, но для символов a и b уже выполняется свертка.

Все конфликтные ситуации удалось разрешить. Таким образом, данная грамматика является LALR(1)-грамматикой.

Построим управляющую таблицу для данной грамматики, используя найденную последовательность ситуаций и правила разрешения конфликтов, полученные для ситуаций $\mathbf{R}_1$ и $\mathbf{R}_2$.

Таблица 4.4. Пример управляющей таблицы для LALR(1)-грамматики

№	Стек	Действия			Переходы		
		A	b	$\perp_к$	a	b	S
0	$\perp_н$	свертка, 3		свертка, 3			1
1	S	Сдвиг		успех, 1	2		
2	Sa	свертка, 3	свертка, 3				3
3	SaS	Сдвиг	сдвиг		2	4	
4	SaSb	свертка, 2	свертка, 2	свертка, 2			

Как и в ранее построенных таблицах, колонка *«Стек»* не нужна распознавателю, а введена для пояснения состояния стека. Пустые клетки в таблице соответствуют состоянию *«ошибка»*. Правила грамматики пронумерованы от 1 до 3. Колонка *«Действия»* в таблице содержит перечень действий, соответствующих текущему входному символу, обозреваемому считывающей головкой автомата.

Рассмотрим примеры распознавания цепочек языка, заданного этой грамматикой. Конфигурацию расширенного МП-автомата будем отображать в виде трех компонентов, как это было сделано выше.

Разбор цепочки abababb:

1. (abababb$\perp_к$,{$\perp_н$,0},λ)
2. (abababb$\perp_к$,{$\perp_н$,0}{S,1},3)
3. (bababb$\perp_к$,{$\perp_н$,0}{S,1}{a,2},3)
4. (bababb$\perp_к$,{$\perp_н$,0}{S,1}{a,2}{S,3},3,3)
5. (ababb$\perp_к$,{$\perp_н$,0}{S,1}{a,2}{S,3}{b,4},3,3)
6. (ababb$\perp_к$,{$\perp_н$,0}{S,1},3,3,2)

7. (babb⊥к,{⊥н,0}{S,1}{a,2},3,3,2)
8. (babb⊥к,{⊥н,0}{S,1}{a,2}{S,3},3,3,2,3)
9. (abb⊥к,{⊥н,0}{S,1}{a,2}{S,3}{b,4},3,3,2,3)
10. (abb⊥к,{⊥н,0}{S,1},3,3,2,3,2)
11. (bb⊥к,{⊥н,0}{S,1}{a,2},3,3,2,3,2)
12. (bb⊥к,{⊥н,0}{S,1}{a,2}{S,3},3,3,2,3,2,3)
13. (b⊥к,{⊥н,0}{S,1}{a,2}{S,3}{b,4},3,3,2,3,2,3)
14. (b⊥к,{⊥н,0}{S,1},3,3,2,3,2,3,2)
15. ошибка, нет действий для b в строке 1.

Разбор цепочки aababb:

1. (aababb⊥к,{⊥н,0},λ)
2. (aababb⊥к,{⊥н,0}{S,1},3)
3. (ababb⊥к,{⊥н,0}{S,1}{a,2},3)
4. (ababb⊥к,{⊥н,0}{S,1}{a,2}{S,3},3,3)
5. (babb⊥к,{⊥н,0}{S,1}{a,2}{S,3}{a,2},3,3)
6. (babb⊥к,{⊥н,0}{S,1}{a,2}{S,3}{a,2}{S,3},3,3,3)
7. (abb⊥к,{⊥н,0}{S,1}{a,2}{S,3}{a,2}{S,3}{b,4},3,3,3)
8. (abb⊥к,{⊥н,0}{S,1}{a,2}{S,3},3,3,3,2)
9. (bb⊥к,{⊥н,0}{S,1}{a,2}{S,3}{a,2},3,3,3,2)
10. (bb⊥к,{⊥н,0}{S,1}{a,2}{S,3}{a,2}{S,3},3,3,3,2,3)
11. (b⊥к,{⊥н,0}{S,1}{a,2}{S,3}{a,2}{S,3}{b,4},3,3,3,2,3)
12. (b⊥к,{⊥н,0}{S,1}{a,2}{S,3},3,3,3,2,3,2)
13. (⊥к,{⊥н,0}{S,1}{a,2}{S,3}{b,4},3,3,3,2,3,2)
14. (⊥к,{⊥н,0}{S,1},3,3,3,2,3,2,2)
15. (⊥к,{⊥н,0}{S',*},3,3,3,2,3,2,2,1) — разбор завершен.

Соответствующая цепочка вывода будет иметь вид (используется правосторонний вывод):

S'⇒S⇒SaSb⇒SaSaSbb⇒SaSabb⇒SaSaSbabb⇒SaSababb⇒Saababb⇒aababb.

Можно убедиться, что вывод, найденный с помощью распознавателя на основе LALR(1)-грамматики соответствует выводу, найденному распознавателем на основе LR(1)-грамматики.

Особенности LALR(1)-грамматик

Как и SLR(1)-грамматики, LALR(1)-грамматики позволяют сократить объем управляющей таблицы **T** распознавателя за счет того, что количество строк в ней соответствует количеству состояний LR(0)-грамматики. Но, по сравнению с SLR(1)-грамматиками, они используют более сложный алгоритм исключения конфликтных ситуаций при построении этой таблицы. В целом, построить распознаватель на основе LALR(1)-грамматики проще, чем распознаватель на основе LR(1)-грамматики, но немного сложнее, чем распознаватель на основе SLR(1)-грамматики.

Класс LALR(1)-грамматик шире, чем класс SLR(1)-грамматик. Например, рассмотренная выше пополненная КС-грамматика `G'({a,b}, {S,S'}, {S'→S,S→SaSb|λ}, S')` является LALR(1)-грамматикой, но не является SLR(1)-грамматикой. Однако важно, что и языки, заданные LALR(1)-грамматиками, представляют собой более широкий класс КС-языков, чем языки, заданные SLR(1)-грамматиками.

ПРИМЕЧАНИЕ

Всякая SLR(1)-грамматика является LALR(1)-грамматикой, но не наоборот.

Доказано, что любой ДКС-язык может быть задан LALR(1)-грамматикой, а это значит, что класс языков, заданных LALR(1)-грамматиками, совпадает с классом языков, заданных LR(1)-грамматиками [4 т.2, 5]. Хотя известно, что класс LR(1)-грамматик шире, чем класс LALR(1)-грамматик.

ВНИМАНИЕ

Всякий язык, заданный LR(1)-грамматикой, может быть задан LALR(1)-грамматикой, но не всякая LR(1)-грамматика является LALR(1)-грамматикой.

LALR(1)-грамматики представляют собой более удобный инструмент для построения распознавателей, чем LR(1)-грамматики. Кроме того, этот класс грамматик достаточно мощный, чтобы на его основе можно было построить синтаксический анализатор для любого языка программирования, представляющего практический интерес (а такой интерес представляют только ДКС-языки). Однако этот класс грамматик уже, чем класс LR(1)-грамматик, а значит существуют LR(1)-грамматики, которые не являются LALR(1)-грамматиками. Как и для других классов КС-грамматик, не всегда удается найти преобразование, позволяющее построить LALR(1)-грамматику для языка, заданного LR(1)-грамматикой. В ряде случаев этот факт может ограничить применение распознавателей на основе LALR(1)-грамматик.

Но распознаватели на основе LALR(1)-грамматик имеют еще один недостаток по сравнению с распознавателями на основе LR(1)-грамматик. Дело в том, что метод анализа правого контекста, применяемый при построении распознавателя для

LALR(1)-грамматики несколько ограничивает возможности распознавателя по обнаружению ошибок во входной цепочке символов. Ошибка, конечно же, будет обнаружена, но распознавателю на основе LALR(1)-грамматики потребуется для этого больше шагов, чем распознавателю на основе LR(1)-грамматики. Это значит, что ошибка будет найдена на более поздней стадии синтаксического анализа[12]. В таком случае у компилятора будет меньше возможностей по диагностике ошибки. Поэтому можно сказать, что распознаватели на основе LALR(1)-грамматик упрощают выполнение синтаксического анализа, но при этом уменьшают возможности компилятора по диагностике ошибок, по сравнению с распознавателями на основе LR(1)-грамматик.

Тем не менее, несмотря на указанные недостатки, распознаватели на основе LALR(1)-грамматик являются одним из самых мощных и эффективных средств для построения синтаксических анализаторов. Именно поэтому данный тип распознавателя используется при решении задачи автоматизации построения синтаксического анализатора.

Примеры построения LR(0), SLR(1) и LALR(1) распознавателей можно найти в книгах [5, 58].

Автоматизация построения синтаксических анализаторов (программа YACC)

При разработке различных прикладных программ часто возникает задача синтаксического разбора некоторого входного текста. Конечно, ее можно всегда решить, полностью самостоятельно построив соответствующий анализатор. И хотя задача выполнения синтаксического разбора встречается не столь часто, как задача выполнения лексического разбора, но все-таки и для ее решения были предложены соответствующие программные средства.

Автоматизированное построение синтаксических анализаторов может быть выполнено с помощью программы YACC.

Программа YACC (Yet Another Compiler Compiler) предназначена для построения синтаксического анализатора КС-языка. Анализируемый язык описывается с помощью грамматики в виде, близком форме Бэкуса-Наура (нормальная форма Бэкуса-Наура — НФБН). Результатом работы YACC является исходный текст программы синтаксического анализатора. Анализатор, который порождается YACC, реализует восходящий распознаватель на основе LALR(1)-грамматики [3, 9, 31, 55, 58].

Как и программа LEX, служащая для автоматизации построения лексических анализаторов, программа YACC тесно связана с историей операционных систем типа UNIX. Эта программа входит в поставку многих версий ОС UNIX или Linux. Поэтому чаще всего результатом работы YACC является исходный текст синтаксического распознавателя на языке C. Однако существуют версии YACC,

---

[12] Этот факт можно заметить даже на примере разбора простейшей ошибочной входной цепочки, который был рассмотрен выше для LR(1)-грамматики, а затем и для LALR(1)-грамматики.

выполняющиеся под управлением ОС, отличных от UNIX и порождающие исходный код на других языках программирования (например, Pascal).

Принцип работы YACC похож на принцип работы LEX: на вход поступает файл, содержащий описание грамматики заданного КС-языка, а на выходе получаем текст программы синтаксического распознавателя, который, естественно, можно дополнять и редактировать, как и любую другую программу на заданном языке программирования.

Исходная грамматика для YACC состоит из трех секций, разделенных двойным символом `%` — `%%`: секции описаний, секции правил, в которой описывается грамматика, и секции программ, содержимое которой просто копируется в выходной файл.

Например, ниже приведено описание простейшей грамматики для YACC, которая соответствует грамматике арифметических выражений, многократно использовавшейся в качестве примера в данном пособии:

```
%token a
%token b
%start e
%%
e : e '+' m | e '-' m | m ;
m : m '*' t | m '/' t | t ;
t : a | b | '(' e ')' ;
%%
```

Секция описаний содержит информацию о том, что идентификатор `a` является лексемой (терминальным символом) грамматики, а символ `e` — ее начальным нетерминальным символом.

Грамматика записана обычным образом — идентификаторы обозначают терминальные и нетерминальные символы; символьные константы типа `'+'` и `'-'` считаются терминальными символами. Символы `: | ;` принадлежат к метаязыку YACC и читаются согласно НФБН «есть по определению», «или» и «конец правила» соответственно.

В отличие от LEX, который всегда способен построить лексический распознаватель, если входной файл содержит правильное регулярное выражение, YACC не всегда может построить распознаватель, даже если входной язык задан правильной КС-грамматикой. Ведь заданная грамматика может и не принадлежать к классу LALR(1)-грамматик. В этом случае YACC выдаст сообщение об ошибке (о наличии неразрешимого конфликта в LALR(1)-грамматике) при построении синтаксического анализатора. Тогда пользователь должен либо преобразовать грамматику, либо задать YACC некоторые дополнительные правила, которые могут облегчить построение анализатора. Например, YACC позволяет указать правила, явно задающие приоритет операций и порядок их выполнения (слева направо или справа налево).

С каждым правилом грамматики может быть связано действие, которое будет выполнено при свертке по данному правилу. Оно записывается в виде заключенной в фигурные скобки последовательности операторов языка, на котором порождается исходный текст программы распознавателя (обычно это язык С). Последовательность должна располагаться после правой части соответствующего правила. Также YACC позволяет управлять действиями, которые будут выполняться распознавателем в том случае, если входная цепочка не принадлежит заданному языку. Распознаватель имеет возможность выдать сообщение об ошибке, остановиться, либо же продолжить разбор, предприняв некоторые действия, связанные с попыткой локализовать либо устранить ошибку во входной цепочке.

YACC в настоящее время не является единственным программным продуктом, предназначенным для автоматизации построения синтаксических анализаторов, так же как и LEX не является единственным программным продуктом для автоматизации построения лексических анализаторов. Сейчас на рынке средств разработки программного обеспечения присутствует целый ряд программных продуктов, ориентированных на решение такого рода задач. Некоторые из них распространяются бесплатно или же бесплатно, но с некоторыми ограничениями, другие являются коммерческими программными продуктами. Многие средства такого рода входят в состав ОС и систем программирования (стоит отметить, что LEX и YACC входят в состав ОС типа UNIX).

СОВЕТ

> Если у читателей существует потребность в программном обеспечении для автоматизированного построения синтаксических анализаторов, автор прежде всего рекомендует обратиться для поиска таких средств во всемирную сеть Интернет. По ключевым словам «YACC» и «LALR» можно получить достаточное количество полезных ссылок.

Более подробные сведения о программе автоматизированного построения синтаксических распознавателей YACC можно получить в [22, 31, 53].

Синтаксические распознаватели на основе грамматик предшествования

Общие принципы грамматик предшествования

Еще одним распространенным классом КС-грамматик, для которых возможно построить восходящий распознаватель без возвратов, являются грамматики предшествования. Также как и распознаватель для рассмотренных выше LR-грамматик, распознаватель для грамматик предшествования строится на основе алгоритма «сдвиг-свертка» («перенос-свертка»), который в общем виде был рассмотрен в разделе «Синтаксические распознаватели с возвратом».

Принцип организации распознавателя на основе грамматики предшествования исходит из того, что для каждой упорядоченной пары символов в грамматике устанавливается отношение, называемое отношением предшествования. В процессе разбора расширенный МП-автомат сравнивает текущий символ входной цепочки с

одним из символов, находящихся на верхушке стека автомата. В процессе сравнения проверяется, какое из возможных отношений предшествования существует между этими двумя символами. В зависимости от найденного отношения выполняется либо сдвиг, либо свертка. При отсутствии отношения предшествования между символами алгоритм сигнализирует об ошибке.

Задача заключается в том, чтобы иметь возможность непротиворечивым образом определить отношения предшествования между символами грамматики. Если это возможно, то грамматика может быть отнесена к одному из классов грамматик предшествования.

Существует несколько видов грамматик предшествования. Они различаются по тому, какие отношения предшествования в них определены и между какими типами символов (терминальными или нетерминальными) могут быть установлены эти отношения. Кроме того, возможны незначительные модификации функционирования самого алгоритма «сдвиг-свертка» в распознавателях для таких грамматик (в основном на этапе выбора правила для выполнения свертки, когда возможны неоднозначности) [4 т.1,2, 5].

Выделяют следующие виды грамматик предшествования:

- простого предшествования;
- расширенного предшествования;
- слабого предшествования;
- смешанной стратегии предшествования;
- операторного предшествования.

Далее будут рассмотрены два наиболее простых и распространенных типа — грамматики простого и операторного предшествования.

Грамматики простого предшествования

Грамматикой простого предшествования называют такую приведенную КС-грамматику[13] $G(\mathbf{VN},\mathbf{VT},\mathbf{P},S)$, $\mathbf{V} = \mathbf{VT} \cup \mathbf{VN}$, в которой:

1. Для каждой упорядоченной пары терминальных и нетерминальных символов выполняется не более чем одно из трех отношений предшествования:
   - $B_i = \cdot B_j$ ($\forall B_i, B_j \in \mathbf{V}$), если и только если $\exists$ правило $A \to xB_iB_jy \in \mathbf{P}$, где $A \in \mathbf{VN}$, $x,y \in \mathbf{V}^*$;
   - $B_i < \cdot B_j$ ($\forall B_i, B_j \in \mathbf{V}$), если и только если $\exists$ правило $A \to xB_iDy \in \mathbf{P}$ и вывод $D \Rightarrow^* S_jz$, где $A,D \in \mathbf{VN}$, $x,y,z \in \mathbf{V}^*$;
   - $B_i \cdot > B_j$ ($\forall B_i, B_j \in \mathbf{V}$), если и только если $\exists$ правило $A \to xCB_jy \in \mathbf{P}$ и вывод $C \Rightarrow^* zB_i$ или $\exists$ правило $A \to xCDy \in \mathbf{P}$ и выводы $C \Rightarrow^* zB_i$ и $D \Rightarrow^* B_jw$, где $A,C,D \in \mathbf{VN}$, $x,y,z,w \in \mathbf{V}^*$.

---

[13] Напоминаем, что КС-грамматика называется приведенной, если она не содержит циклов, бесплодных и недостижимых символов и λ-правил.

2. Различные правила в грамматике имеют разные правые части (то есть в грамматике не должно быть двух различных правил с одной и той же правой частью).

Отношения $=\cdot$, $<\cdot$ и $\cdot>$ называют отношениями предшествования для символов. Отношение предшествования единственно для каждой упорядоченной пары символов. При этом между какими-либо двумя символами может и не быть отношения предшествования — это значит, что они не могут находиться рядом ни в одном элементе разбора синтаксически правильной цепочки. Отношения предшествования зависят от порядка, в котором стоят символы, и в этом смысле их нельзя путать со знаками математических операций — они не обладают ни свойством коммутативности, ни свойством ассоциативности. Например, если известно, что $B_i \cdot > B_j$, то не обязательно выполняется $B_j < \cdot B_i$ (поэтому знаки предшествования иногда помечают специальной точкой: $=\cdot$, $<\cdot$, $\cdot>$)

Для грамматик простого предшествования известны следующие полезные свойства:

- всякая грамматика простого предшествования является однозначной;
- легко проверить, является или нет произвольная КС-грамматика грамматикой простого предшествования (для этого достаточно проверить рассмотренные выше свойства грамматик простого предшествования или воспользоваться алгоритмом построения матрицы предшествования, который рассмотрен далее).

Как и для многих других классов грамматик, для грамматик простого предшествования не существует алгоритма, который бы мог преобразовать произвольную КС-грамматику в грамматику простого предшествования или доказать, что преобразование невозможно.

Метод предшествования основан на том факте, что отношения предшествования между двумя соседними символами распознаваемой строки соответствуют трем следующим вариантам:

- $B_i < \cdot B_{i+1}$, если символ B_{i+1} — крайний левый символ некоторой основы (это отношение между символами можно назвать «предшествует основе» или просто «предшествует»);
- $B_i \cdot > B_{i+1}$, если символ B_i — крайний правый символ некоторой основы (это отношение между символами можно назвать «следует за основой» или просто «следует»);
- $B_i = \cdot B_{i+1}$, если символы B_i и B_{i+1} принадлежат одной основе (это отношение между символами можно назвать «составляют основу»).

Исходя из этих соотношений, выполняется разбор строки для грамматики предшествования.

Суть принципа такого разбора можно понять из рис. 4.10. На нем изображена входная цепочка символов $\alpha\gamma\beta\delta$ в тот момент, когда выполняется свертка цепочки γ. Символ `a` является последним символом подцепочки α, а символ `b` — первым символом подцепочки β. Тогда, если в грамматике удастся установить непротиворечивые отношения предшествования, то в процессе выполнения разбора по алгоритму «сдвиг-свертка» можно всегда выполнять сдвиг до тех пор, пока между

символом на верхушке стека и текущим символом входной цепочки существует отношение <· или =·. А как только между этими символами будет обнаружено отношение ·>, так сразу надо выполнять свертку. Причем для выполнения свертки из стека надо выбирать все символы, связанные отношением =·. То, что все различные правила в грамматике предшествования имеют различные правые части, гарантирует непротиворечивость выбора правила при выполнении свертки.

Таким образом, установление непротиворечивых отношений предшествования между символами грамматики в комплексе с несовпадающими правыми частями различных правил дает ответы на все вопросы, которые надо решить для организации работы алгоритма «сдвиг-свертка» без возвратов.

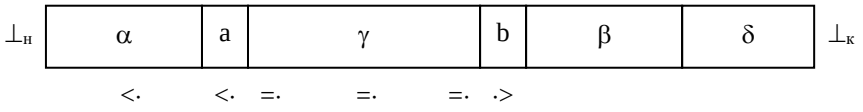


Рис. 4.10. Отношения между символами входной цепочки в грамматике предшествования

На основании отношений предшествования строят матрицу предшествования грамматики. Строки матрицы предшествования помечаются первыми (левыми) символами, столбцы — вторыми (правыми) символами отношений предшествования. В клетки матрицы на пересечении соответствующих столбца и строки помещаются знаки отношений. При этом пустые клетки матрицы говорят о том, что между данными символами нет ни одного отношения предшествования.

Матрицу предшествования грамматики сложно построить, опираясь непосредственно на определения отношений предшествования. Удобнее воспользоваться двумя дополнительными множествами — множеством крайних левых и множеством крайних правых символов относительно нетерминальных символов грамматики G(**VN**,**VT**,**P**,S), **V** = **VT**∪**VN**. Эти множества определяются следующим образом:

- **L**(A) = {X | ∃A⇒*Xz}, A∈**VN**, X∈**V**, z∈**V*** — множество крайних левых символов относительно нетерминального символа A (цепочка z может быть и пустой цепочкой);
- **R**(A) = {X | ∃A⇒*zX}, A∈**VN**, X∈**V**, z∈**V*** — множество крайних правых символов относительно нетерминального символа A.

Иными словами, множество крайних левых символов относительно нетерминального символа A — это множество всех крайних левых символов в цепочках, которые могут быть выведены из символа A. Аналогично, множество крайних правых символов относительно нетерминального символа A — это множество всех крайних правых символов в цепочках, которые могут быть выведены из символа A.

Тогда отношения предшествования можно определить так:

- B_i=·B_j (∀B_i,B_j∈**V**), если ∃ правило A→x$B_i B_j$y∈**P**, где A∈**VN**, x,y∈**V***;
- B_i<·B_j (∀B_i,B_j∈**V**), если ∃ правило A→xB_iDy∈**P** и B_j∈**L**(D), где A,D∈**VN**, x,y∈**V***;

- $B_i \cdot > B_j$ ($\forall B_i, B_j \in \mathbf{V}$) , если $\exists$ правило $A \to xCB_jy \in \mathbf{P}$ и $B_i \in \mathbf{R}(C)$ или $\exists$ правило $A \to xCDy \in \mathbf{P}$ и $B_i \in \mathbf{R}(C)$, $B_j \in \mathbf{L}(D)$, где $A,C,D \in \mathbf{VN}$, $x,y \in \mathbf{V}^*$.

Такое определение отношений удобнее на практике, так как не требует построения выводов, а множества $\mathbf{L}(A)$ и $\mathbf{R}(A)$ могут быть построены для каждого нетерминального символа $A \in \mathbf{VN}$ грамматики $G(\mathbf{VN},\mathbf{VT},\mathbf{P},S)$, $\mathbf{V}=\mathbf{VT} \cup \mathbf{VN}$ по очень простому алгоритму:

Шаг 1. $\forall A \in \mathbf{VN}$:

$\mathbf{R}_0(A) = \{X \mid A \to yX, X \in \mathbf{V}, y \in \mathbf{V}^*\}$,

$\mathbf{L}_0(A) = \{X \mid A \to Xy, X \in \mathbf{V}, y \in \mathbf{V}^*\}$, i:=1.

Для каждого нетерминального символа A ищем все правила, содержащие A в левой части. Во множество $\mathbf{L}(A)$ включаем самый левый символ из правой части правил, а во множество $\mathbf{R}(A)$ — самый крайний правый символ из правой части. Переходим к шагу 2.

Шаг 2. $\forall A \in \mathbf{VN}$:

$\mathbf{R}_i(A) = \mathbf{R}_{i-1}(A) \cup \mathbf{R}_{i-1}(B)$, $\forall B \in (\mathbf{R}_{i-1}(A) \cap \mathbf{VN})$,

$\mathbf{L}_i(A) = \mathbf{L}_{i-1}(A) \cup \mathbf{L}_{i-1}(B)$, $\forall B \in (\mathbf{L}_{i-1}(A) \cap \mathbf{VN})$.

Для каждого нетерминального символа A: если множество $\mathbf{L}(A)$ содержит нетерминальные символы грамматики A', A", ..., то его надо дополнить символами, входящими в соответствующие множества $\mathbf{L}(A')$, $\mathbf{L}(A")$, ... и не входящими в $\mathbf{L}(A)$. Ту же операцию надо выполнить для $\mathbf{R}(A)$.

Шаг 3. Если $\exists A \in \mathbf{VN}$: $\mathbf{R}_i(A) \neq \mathbf{R}_{i-1}(A)$ или $\mathbf{L}_i(A) \neq \mathbf{L}_{i-1}(A)$, то i := i + 1 и вернуться к шагу 2, иначе построение закончено: $\mathbf{R}(A) = \mathbf{R}_i(A)$ и $\mathbf{L}(A) = \mathbf{L}_i(A)$.

Если на предыдущем шаге хотя бы одно множество $\mathbf{L}(A)$ или $\mathbf{R}(A)$ для некоторого символа грамматики изменилось, то надо вернуться к шагу 2, иначе построение закончено.

После построения множеств $\mathbf{L}(A)$ и $\mathbf{R}(A)$ по правилам грамматики создается матрица предшествования. Матрицу предшествования дополняют символами $\perp_н$ и $\perp_к$ (начало и конец цепочки). Для них определены следующие отношения предшествования:

$\perp_н \cdot < X$, $\forall a \in \mathbf{V}$, если $\exists S \Rightarrow^* Xy$, где $S \in \mathbf{VN}$, $y \in \mathbf{V}^*$ или (с другой стороны) если $X \in \mathbf{L}(S)$;

$\perp_к \cdot > X$, $\forall a \in \mathbf{V}$, если $\exists S \Rightarrow^* yX$, где $S \in \mathbf{VN}$, $y \in \mathbf{V}^*$ или (с другой стороны) если $X \in \mathbf{R}(S)$.

Здесь S — целевой символ грамматики.

Матрица предшествования служит основой для работы распознавателя языка, заданного грамматикой простого предшествования.

Алгоритм «сдвиг-свертка» для грамматики простого предшествования.

Отношения предшествования служат для того, чтобы определить в процессе выполнения алгоритма, какое действие — сдвиг или свертка — должно выполняться на каждом шаге алгоритма, и однозначно выбрать цепочку для свертки. Однозначный выбор правила при свертке обеспечивается за счет различия правых

частей всех правил грамматики. В начальном состоянии автомата считывающая головка обозревает первый символ входной цепочки, в стеке МП-автомата находится символ $\perp_н$ (начало цепочки), в конец цепочки помещен символ $\perp_к$ (конец цепочки). Символы $\perp_н$ и $\perp_к$ введены для удобства работы алгоритма, в язык, заданный исходной грамматикой, они не входят.

Разбор считается законченным (алгоритм завершается), если считывающая головка автомата обозревает символ $\perp_к$, и при этом больше не может быть выполнена свертка. Решение о принятии цепочки зависит от содержимого стека. Автомат принимает цепочку, если в результате завершения алгоритма в стеке находятся начальный символ грамматики S и символ $\perp_н$. Выполнение алгоритма может быть прервано, если на одном из его шагов возникнет ошибка. Тогда входная цепочка не принимается.

Алгоритм состоит из следующих шагов:

Шаг 1. Поместить в верхушку стека символ $\perp_н$, считывающую головку — в начало входной цепочки символов.

Шаг 2. Сравнить с помощью отношения предшествования символ, находящийся на вершине стека (левый символ отношения), с текущим символом входной цепочки, обозреваемым считывающей головкой (правый символ отношения).

Шаг 3. Если имеет место отношение $<\cdot$ или $=\cdot$, то произвести сдвиг (перенос текущего символа из входной цепочки в стек и сдвиг считывающей головки на один шаг вправо) и вернуться к шагу 2. Иначе перейти к шагу 4.

Шаг 4. Если имеет место отношение $\cdot>$, то произвести свертку. Для этого надо найти на вершине стека все символы, связанные отношением $=\cdot$ («основу»), удалить эти символы из стека. Затем выбрать из грамматики правило, имеющее правую часть, совпадающую с основой, и поместить в стек левую часть выбранного правила (если символов, связанных отношением $=\cdot$, на верхушке стека нет, то в качестве основы используется один, самый верхний символ стека). Если правило, совпадающее с основой, найти не удалось, то необходимо прервать выполнение алгоритма и сообщить об ошибке, иначе, если разбор не закончен, то вернуться к шагу 2.

Шаг 5. Если не установлено ни одно отношение предшествования между текущим символом входной цепочки и символом на верхушке стека, то надо прервать выполнение алгоритма и сообщить об ошибке.

Ошибка в процессе выполнения алгоритма возникает, когда невозможно выполнить очередной шаг — например, если не установлено отношение предшествования между двумя сравниваемыми символами (на шагах 2 и 4), или если не удается найти нужное правило в грамматике (на шаге 4). Тогда выполнение алгоритма прерывается.

Пример распознавателя для грамматики простого предшествования

Рассмотрим в качестве примера грамматику `G({+,-,/,*,a,b}, {S,R,T,F,E},` **`P`**`, S)` с правилами:

P:

S → TR | T

R → +T | −T | +TR | −TR

T → EF | E

F → *E | /E | *EF | /EF

E → (S) | a | b

Эта нелеворекурсивная грамматика для арифметических выражений над символами a и b уже несколько раз использовалась в качестве примера для построения распознавателей. Хотя эта грамматика и содержит цепные правила, легко увидеть, что она не содержит циклов, совпадающих правых частей правил и λ-правил, следовательно, по формальным признакам ее можно отнести к грамматикам простого предшествования. Осталось определить отношения предшествования.

Построим множества крайних левых и крайних правых символов относительно нетерминальных символов грамматики.

Шаг 1.

$\mathbf{L}_0(S) = \{T\}$ $\mathbf{R}_0(S) = \{R,T\}$

$\mathbf{L}_0(R) = \{+,–\}$ $\mathbf{R}_0(R) = \{R,T\}$

$\mathbf{L}_0(T) = \{E\}$ $\mathbf{R}_0(T) = \{E,F\}$

$\mathbf{L}_0(F) = \{*,/\}$ $\mathbf{R}_0(F) = \{E,F\}$

$\mathbf{L}_0(E) = \{(,a,b\}$ $\mathbf{R}_0(E) = \{),a,b\}$

Шаг 2.

$\mathbf{L}_1(S) = \{T,E\}$ $\mathbf{R}_1(S) = \{R,T,E,F\}$

$\mathbf{L}_1(R) = \{+,–\}$ $\mathbf{R}_1(R) = \{R,T,E,F\}$

$\mathbf{L}_1(T) = \{E,(,a,b\}$ $\mathbf{R}_1(T) = \{E,F,),a,b\}$

$\mathbf{L}_1(F) = \{*,/\}$ $\mathbf{R}_1(F) = \{E,F,),a,b\}$

$\mathbf{L}_1(E) = \{(,a,b\}$ $\mathbf{R}_1(E) = \{),a,b\}$

Шаг 3. Имеется $\mathbf{L}_0(S) \neq \mathbf{L}_1(S)$, возвращаемся к шагу 2.

Шаг 2.

$\mathbf{L}_2(S) = \{T,E,(,a,b\}$ $\mathbf{R}_2(S) = \{R,T,E,F,),a,b\}$

$\mathbf{L}_2(R) = \{+,–\}$ $\mathbf{R}_2(R) = \{R,T,E,F,),a,b\}$

$\mathbf{L}_2(T) = \{E,(,a,b\}$ $\mathbf{R}_2(T) = \{E,F,),a,b\}$

$\mathbf{L}_2(F) = \{*,/\}$ $\mathbf{R}_2(F) = \{E,F,),a,b\}$

$\mathbf{L}_2(E) = \{(,a,b\}$ $\mathbf{R}_2(E) = \{),a,b\}$

Шаг 3. Имеется $\mathbf{L}_1(S) \neq \mathbf{L}_2(S)$, возвращаемся к шагу 2.

Шаг 2.

$\mathbf{L}_3$(S) = {T,E,(,a,b} $\mathbf{R}_3$(S) = {R,T,E,F,),a,b}

$\mathbf{L}_3$(R) = {+,-} $\mathbf{R}_3$(R) = {R,T,E,F,),a,b}

$\mathbf{L}_3$(T) = {E,(,a,b} $\mathbf{R}_3$(T) = {E,F,),a,b}

$\mathbf{L}_3$(F) = {*,/} $\mathbf{R}_3$(F) = {E,F,),a,b}

$\mathbf{L}_3$(E) = {(,a,b} $\mathbf{R}_3$(E) = {),a,b}

Шаг 3: Ни одно множество не изменилось, построение закончено. Результат:

L(S) = {T,E,(,a,b} **R**(S) = {R,T,E,F,),a,b}

L(R) = {+,–} **R**(R) = {R,T,E,F,),a,b}

L(T) = {E,(,a,b} **R**(T) = {E,F,),a,b}

L(F) = {*,/} **R**(F) = {E,F,),a,b}

L(E) = {(,a,b} **R**(E) = {),a,b}

На основании построенных множеств крайних левых и крайних правых символов и правил грамматики построим матрицу предшествования. Результат приведен в табл. 4.5.

Таблица 4.5. Таблица предшествования для грамматики простого предшествования.

	+	**–**	*****	**/**	**(**	**)**	**a**	**b**	**S**	**R**	**T**	**F**	**E**	**$\perp_к$**
+					<·		<·	<·			=·		<·	
–					<·		<·	<·			=·		<·	
*****					<·		<·	<·					=·	
/					<·		<·	<·					=·	
(					<·		<·	<·	=·		<·		<·	
)	·>	·>	·>	·>		·>				·>		·>		·>
a	·>	·>	·>	·>		·>				·>		·>		·>
b	·>	·>	·>	·>		·>				·>		·>		·>
S						=·								
R						·>								·>
T	<·	<·				·>				=·				·>
F	·>	·>				·>				·>				·>
E	·>	·>	<·	<·		·>				·>		=·		·>
$\perp_н$					<·		<·	<·			<·		<·	

Поясним построение таблицы на примере символа +.

Во-первых, в правилах грамматики R→+T|+TR символ + стоит слева от символа T. Значит, в строке символа + в столбце, соответствующем символу T, ставим знак =·. Кроме того, во множество **L**(T) входят символы E, (, a, b. Тогда в строке символа + во всех столбцах, соответствующим этим четырем символам, ставим знак <·.

Во-вторых, символ + входит во множество **L**(R), а в грамматике имеются правила вида S→TR и R→+TR|−TR. Следовательно, надо рассмотреть также множество **R**(T). Туда входят символы E, F,), a, b. Значит, в столбце символа + во всех строках, соответствующим этим пяти символам, ставим знак ·>. Больше символ + ни в какие множества не входит и ни в каких правилах не встречается.

Продолжая эти рассуждения для остальных терминальных и нетерминальных символов грамматики, заполняем все ячейки матрицы предшествования, приведенной выше. Если окажется, что согласно логике рассуждений в какую-либо клетку матрицы предшествования необходимо поместить более чем один знак =·, <· или ·>, то это означает, что исходная грамматика не является грамматикой простого предшествования.

Отдельно можно рассмотреть заполнение строки для символа $\perp_н$ (начало строки) и столбца для символа $\perp_к$ (конец строки). Множество **L**(S), где S — целевой символ, содержит символы T, E, (, a, b. Помещаем знак <· в строку, соответствующую символу $\perp_н$ для всех пяти столбцов, соответствующих этим символам. Аналогично, множество **R**(S) содержит символы R, T, E, F,), a, b. Помещаем знак ·> в столбец, соответствующий символу $\perp_к$ для всех семи строк, соответствующих этим символам.

Рассмотрим работу алгоритма распознавания на примерах. Последовательность разбора будем записывать в виде последовательности конфигураций МП-автомата из трех составляющих:

1. не просмотренная автоматом часть входной цепочки;
2. содержимое стека;
3. последовательность примененных правил грамматики.

Так как автомат имеет только одно состояние, то для определения его конфигурации достаточно двух составляющих — положения считывающей головки во входной цепочке и содержимого стека. Последовательность номеров правил несет дополнительную полезную информацию, по которой можно построить цепочку или дерево вывода. Правила в грамматике номеруются в направлении слева направо и сверху вниз (всего в грамматике имеется 15 правил).

Будем обозначать такт автомата символом ÷. Введем также дополнительное обозначение $÷_п$, если на данном такте выполнялся перенос, и $÷_с$, если выполнялась свертка.

Последовательности разбора цепочек входных символов будут, таким образом, иметь вид:

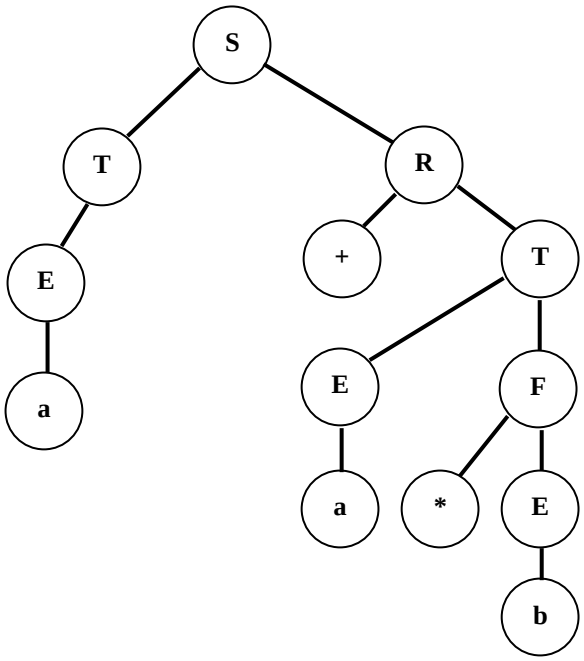
Пример 1. Входная цепочка a+a*b.

1. {a+a*b⊥к;⊥нa;∅} ÷п
2. {+a*b⊥к;⊥нa;∅} ÷с
3. {+a*b⊥к;⊥нE;14} ÷с
4. {+a*b⊥к;⊥нT;14,8} ÷п
5. {a*b⊥к;⊥нT+;14,8} ÷п
6. {*b⊥к;⊥нT+a;14,8} ÷с
7. {*b⊥к;⊥нT+E;14,8,14} ÷п
8. {b⊥к;⊥нT+E*;14,8,14} ÷п
9. {⊥к;⊥нT+E*b;14,8,14} ÷с
10. {⊥к;⊥нT+E*E;14,8,14,15} ÷с
11. {⊥к;⊥нT+EF;14,8,14,15,9} ÷с
12. {⊥к;⊥нT+T;14,8,14,15,9,7} ÷с
13. {⊥к;⊥нTR;14,8,14,15,9,7,3} ÷с
14. {⊥к;⊥нS;14,8,14,15,9,7,3,1} алгоритм завершен, цепочка принята.

Соответствующая цепочка вывода будет иметь вид (используется правосторонний вывод): S⇒TR⇒T+T⇒T+EF⇒T+E*E⇒T+E*b⇒T+a*b⇒E+a*b⇒a+a*b.

Дерево вывода, соответствующее этой цепочке, приведено на рис. 4.11.

Рис. 4.11. Пример дерева вывода для грамматики простого предшествования



Пример 2. Входная цепочка (a+a)*b.

1. {(a+a)*b⊥к;⊥н;∅} ÷п
2. {a+a)*b⊥к;⊥н(;∅} ÷п
3. {+a)*b⊥к;⊥н(a;∅} ÷с
4. {+a)*b⊥к;⊥н(E;14} ÷с
5. {+a)*b⊥к;⊥н(T;14,8} ÷п
6. {a)*b⊥к;⊥н(T+;14,8} ÷п
7. {)*b⊥к;⊥н(T+a;14,8} ÷с
8. {)*b⊥к;⊥н(T+E;14,8,14} ÷с
9. {)*b⊥к;⊥н(T+T;14,8,14,8} ÷с
10. {)*b⊥к;⊥н(TR;14,8,14,8,3} ÷с
11. {)*b⊥к;⊥н(S;14,8,14,8,3,1} ÷п
12. {*b⊥к;⊥н(S);14,8,14,8,3,1} ÷с
13. {*b⊥к;⊥нE;14,8,14,8,3,1,13} ÷п
14. {b⊥к;⊥нE*;14,8,14,8,3,1,13} ÷п
15. {⊥к;⊥нE*b;14,8,14,8,3,1,13} ÷с
16. {⊥к;⊥нE*E;14,8,14,8,3,1,13,15} ÷с
17. {⊥к;⊥нEF;14,8,14,8,3,1,13,15,9} ÷с
18. {⊥к;⊥нT;14,8,14,8,3,1,13,15,9,7} ÷с
19. {⊥к;⊥нS;14,8,14,8,3,1,13,15,9,7,2} алгоритм завершен, цепочка принята.

Соответствующая цепочка вывода будет иметь вид (используется правосторонний вывод):
S⇒T⇒EF⇒E*E⇒E*b⇒(S)*b⇒(TR)*b⇒(T+T)*b⇒(T+E)*b⇒(T+a)*b⇒(E+a)*b⇒(a+a)*b.

Дерево вывода, соответствующее этой цепочке, приведено на рис. 4.12.

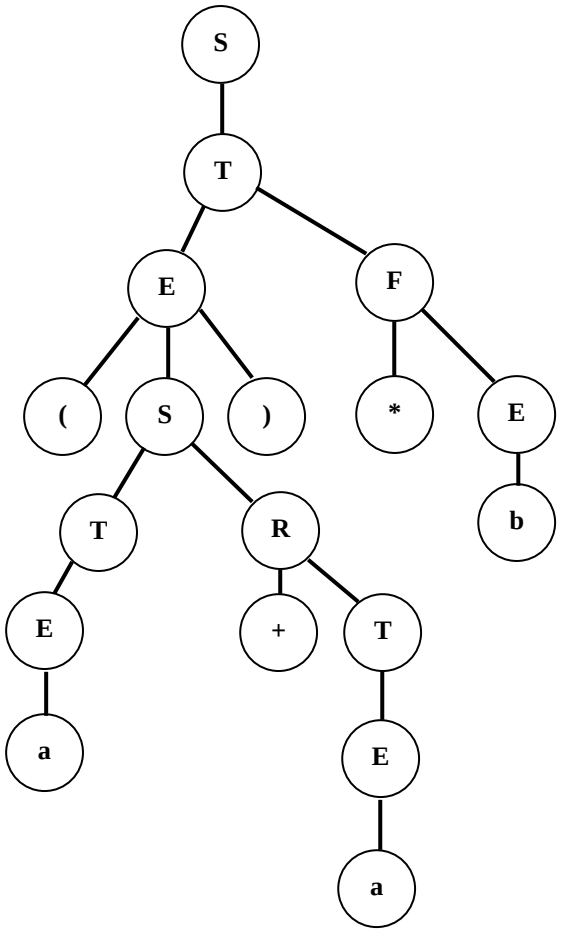


Рис. 4.12. Пример дерева вывода для грамматики простого предшествования

Пример 3. Входная цепочка `a+a*`.

1. {a+a*⊥к;⊥нa;∅} ÷п
2. {+a*⊥к;⊥нa;∅} ÷с
3. {+a*⊥к;⊥нE;14} ÷с
4. {+a*⊥к;⊥нT;14,8} ÷п
5. {a*⊥к;⊥нT+;14,8} ÷п
6. {*⊥к;⊥нT+a;14,8} ÷с
7. {*⊥к;⊥нT+E;14,8,14} ÷п
8. {⊥к;⊥нT+E*;14,8,14} ÷
9. ошибка! (нет отношений предшествования между символами * и ⊥к).

Пример 4. Входная цепочка `a+a)*b`.

1. {a+a)*b⊥к;⊥нa;∅} ÷п
2. {+a)*b⊥к;⊥нa;∅} ÷с

3. {+a)*b⊥к;⊥нE;14} ÷с
4. {+a)*b⊥к;⊥нT;14,8} ÷п
5. {a)*b⊥к;⊥нT+;14,8} ÷п
6. {)*b⊥к;⊥нT+a;14,8} ÷с
7. {)*b⊥к;⊥нT+E;14,8,14} ÷с
8. {)*b⊥к;⊥нT+T;14,8,14,8} ÷с
9. {)*b⊥к;⊥нTR;14,8,14,8,3} ÷с
10. {)*b⊥к;⊥нS;14,8,14,8,3,1} ÷п
11. {*b⊥к;⊥нS);14,8,14,8,3,1} ÷с
12. ошибка! (невозможно выбрать правило для свертки на этом шаге).

Грамматики простого предшествования являются удобным механизмом для анализа входных цепочек КС-языков. Распознаватель для этого класса грамматик строить легче, чем для рассмотренных выше LR-грамматик. Однако класс языков, заданных грамматиками простого предшествования уже, чем класс языков, заданных LR-грамматиками. Отсюда ясно, что не всякий ДКС-язык может быть задан грамматикой простого предшествования, а следовательно, не для каждого ДКС-языка можно построить распознаватель по методу простого предшествования.

У грамматик простого предшествования есть еще один недостаток — при большом количестве терминальных и нетерминальных символов в грамматике матрица предшествования будет иметь значительный объем (при этом следует заметить, что значительная часть ее ячеек может оставаться пустой). Поиск в такой матрице может занять некоторое время, что существенно при работе распознавателя — фактически время поиска линейно зависит от числа символов грамматики, а объем матрицы — квадратично. Для того чтобы избежать хранения и обработки таких матриц, можно выполнить «линеаризацию матрицы предшествования». Тогда каждый раз, чтобы установить отношение предшествования между двумя символами, будет выполняться не поиск по матрице, а вычисление некой специально организованной функции. Вопросы линеаризации матрицы предшествования здесь не рассматриваются, с применяемыми при этом методами можно ознакомиться в [3, 4 т.1,2, 18, 24].

Грамматики операторного предшествования

Операторной грамматикой называется КС-грамматика без λ-правил, в которой правые части всех правил не содержат смежных нетерминальных символов. Для операторной грамматики отношения предшествования можно задать на множестве терминальных символов (включая символы $\perp_н$ и $\perp_к$).

Грамматикой операторного предшествования называется операторная КС-грамматика G(**VN**,**VT**,**P**,S), **V**=**VT**∪**VN**, для которой выполняются следующие условия:

1. Для каждой упорядоченной пары терминальных символов выполняется не более чем одно из трех отношений предшествования:
   - a=·b, если и только если существует правило A→xaby∈**P** или правило A→xaCby, где a,b∈**VT**, A,C∈**VN**, x,y∈$\mathbf{V}^*$;
   - a<·b, если и только если существует правило A→xaCy∈**P** и вывод C⇒*bz или вывод C⇒*Dbz, где a,b∈**VT**, A,C,D∈**VN**, x,y,z∈$\mathbf{V}^*$;
   - a·>b, если и только если существует правило A→xCby∈**P** и вывод C⇒*za или вывод C⇒*zaD, где a,b∈**VT**, A,C,D∈**VN**, x,y,z∈$\mathbf{V}^*$.[14]
2. Различные порождающие правила имеют разные правые части, λ-правила отсутствуют.

Отношения предшествования для грамматик операторного предшествования определены таким образом, что для них выполняется еще одна особенность — правила грамматики операторного предшествования не могут содержать двух смежных нетерминальных символов в правой части. То есть в грамматике операторного предшествования G(**VN**,**VT**,**P**,S), **V**=**VT**∪**VN** не может быть ни одного правила вида: A→xBCy, где A,B,C∈**VN**, x,y∈$\mathbf{V}^*$ (здесь x и y — это произвольные цепочки символов, могут быть и пустыми).

Для грамматик операторного предшествования также известны следующие свойства:

- всякая грамматика операторного предшествования задает детерминированный КС-язык (но не всякая грамматика операторного предшествования при этом является однозначной!);
- легко проверить, является или нет произвольная КС-грамматика грамматикой операторного предшествования (точно так же, как и для простого предшествования).

Как и для многих других классов грамматик, для грамматик операторного предшествования не существует алгоритма, который бы мог преобразовать произвольную КС-грамматику в грамматику операторного предшествования или доказать, что преобразование невозможно.

Принцип работы распознавателя для грамматики операторного предшествования аналогичен грамматике простого предшествования, но отношения операторного предшествования проверяются в процессе разбора только между терминальными символами.

---

[14] В литературе отношения операторного предшествования иногда обозначают другими символами, отличными от «<·», «·>» и «=·», чтобы не путать их с отношениями простого предшествования. Например, встречаются обозначения «<∘», «∘>» и «=∘». В данном пособии путаница исключена, поэтому будут использоваться одни и те же обозначения, хотя по сути отношения предшествования несколько различны.

Для грамматики данного вида на основе установленных отношений предшествования также строится матрица предшествования, но она содержит только терминальные символы грамматики.

Для построения этой матрицы удобно ввести множества крайних левых и крайних правых терминальных символов относительно нетерминального символа A — $\mathbf{L}^t(A)$ или $\mathbf{R}^t(A)$:

- $\mathbf{L}^t(A) = \{t \mid \exists A \Rightarrow^* tz$ или $\exists A \Rightarrow^* Ctz\}$, где $t \in \mathbf{VT}$, $A,C \in \mathbf{VN}$, $z \in \mathbf{V}^*$;
- $\mathbf{R}^t(A) = \{t \mid \exists A \Rightarrow^* zt$ или $\exists A \Rightarrow^* ztC\}$, где $t \in \mathbf{VT}$, $A,C \in \mathbf{VN}$, $z \in \mathbf{V}^*$.

Тогда определения отношений операторного предшествования будут выглядеть так:

- $a \doteq b$, если $\exists$ правило $A \to xaby \in \mathbf{P}$ или правило $U \to xaCby$, где $a,b \in \mathbf{VT}$, $A,C \in \mathbf{VN}$, $x,y \in \mathbf{V}^*$;
- $a \lessdot b$, если $\exists$ правило $A \to xaCy \in \mathbf{P}$ и $b \in \mathbf{L}^t(C)$, где $a,b \in \mathbf{VT}$, $A,C \in \mathbf{VN}$, $x,y \in \mathbf{V}^*$;
- $a \gtrdot b$, если $\exists$ правило $A \to xCby \in \mathbf{P}$ и $a \in \mathbf{R}^t(C)$, где $a,b \in \mathbf{VT}$, $A,C \in \mathbf{VN}$, $x,y \in \mathbf{V}^*$.

В данных определениях цепочки символов x, y, z могут быть и пустыми цепочками.

Для нахождения множеств $\mathbf{L}^t(A)$ и $\mathbf{R}^t(A)$ предварительно необходимо выполнить построение множеств $\mathbf{L}(A)$ и $\mathbf{R}(A)$, как это было рассмотрено ранее. Далее для построения $\mathbf{L}^t(A)$ и $\mathbf{R}^t(A)$ используется следующий алгоритм:

Шаг 1. $\forall A \in \mathbf{VN}$:

$\mathbf{R}^t_0(A) = \{t \mid A \to ytB$ или $A \to yt$, $t \in \mathbf{VT}$, $B \in \mathbf{VN}$, $y \in \mathbf{V}^*\}$,

$\mathbf{L}^t_0(A) = \{t \mid A \to Bty$ или $A \to ty$, $t \in \mathbf{VT}$, $B \in \mathbf{VN}$, $y \in \mathbf{V}^*\}$.

Для каждого нетерминального символа A ищем все правила, содержащие A в левой части. Во множество $\mathbf{L}(A)$ включаем самый левый терминальный символ из правой части правил, игнорируя нетерминальные символы, а во множество $\mathbf{R}(A)$ — самый крайний правый терминальный символ из правой части правил. Переходим к шагу 2.

Шаг 2. $\forall A \in \mathbf{VN}$:

$\mathbf{R}^t(A) = \mathbf{R}^t_0(A) \cup \mathbf{R}^t_0(B)$, $\forall B \in (\mathbf{R}(A) \cap \mathbf{VN})$,

$\mathbf{L}^t(A) = \mathbf{L}^t_0(A) \cup \mathbf{L}^t_0(B)$, $\forall B \in (\mathbf{L}(A) \cap \mathbf{VN})$.

Для каждого нетерминального символа A: если множество $\mathbf{L}(A)$ содержит нетерминальные символы грамматики A', A", ..., то его надо дополнить символами, входящими в соответствующие множества $\mathbf{L}^t(A')$, $\mathbf{L}^t(A")$, ... и не входящими в $\mathbf{L}^t(A)$. Ту же операцию надо выполнить для множеств $\mathbf{R}(A)$ и $\mathbf{R}^t(A)$. Построение закончено.

Для практического использования матрицу предшествования дополняют символами $\perp_н$ и $\perp_к$ (начало и конец цепочки). Для них определены следующие отношения предшествования:

$\perp_н \lessdot a$, $\forall a \in \mathbf{VT}$, если $\exists S \Rightarrow^* ax$ или $\exists S \Rightarrow^* Cax$, где $S,C \in \mathbf{VN}$, $x \in \mathbf{V}^*$ или если $a \in \mathbf{L}^t(S)$;

$\perp_к \gtrdot a$, $\forall a \in \mathbf{VT}$, если $\exists S \Rightarrow^* xa$ или $\exists S \Rightarrow^* xaC$, где $S,C \in \mathbf{VN}$, $x \in \mathbf{V}^*$ или если $a \in \mathbf{R}^t(S)$.

Здесь S — целевой символ грамматики.

Матрица предшествования служит основой для работы распознавателя языка, заданного грамматикой операторного предшествования. Поскольку она содержит только терминальные символы, то, следовательно, будет иметь меньший размер, чем аналогичная матрица для грамматики простого предшествования. Следует отметить, что напрямую сравнивать матрицы двух грамматик нельзя — не всякая грамматика простого предшествования является грамматикой операторного предшествования и наоборот. Например, рассмотренная далее в примере грамматика операторного предшествования не является грамматикой простого предшествования (читатели могут это проверить самостоятельно).

ПРИМЕЧАНИЕ

> Размер матрицы для грамматики операторного предшествования всегда будет меньше, чем размер матрицы эквивалентной ей грамматики простого предшествования.

Все, что было сказано выше о способах хранения матриц для грамматик простого предшествования, в равной степени относится также и к грамматикам операторного предшествования, с той только разницей, что объем хранимой матрицы будет меньше.

Алгоритм «сдвиг-свертка» для грамматики операторного предшествования

Этот алгоритм в целом похож на алгоритм для грамматик простого предшествования, рассмотренный выше. Он также выполняется расширенным МП-автоматом и имеет те же условия завершения и обнаружения ошибок. Основное отличие состоит в том, что при определении отношения предшествования этот алгоритм не принимает во внимание находящиеся в стеке нетерминальные символы, и при сравнении ищет ближайший к верхушке стека терминальный символ. Однако после выполнения сравнения и определения границ основы при поиске правила в грамматике безусловно следует принимать во внимание нетерминальные символы.

Алгоритм состоит из следующих шагов:

Шаг 1. Поместить в верхушку стека символ $\perp_н$, считывающую головку — в начало входной цепочки символов.

Шаг 2. Сравнить с помощью отношения предшествования терминальный символ, ближайший к вершине стека (левый символ отношения), с текущим символом входной цепочки, обозреваемым считывающей головкой (правый символ отношения). При этом из стека надо выбрать самый верхний терминальный символ, игнорируя все возможные нетерминальные символы.

Шаг 3. Если имеет место отношение $<\cdot$ или $=\cdot$, то произвести сдвиг (перенос текущего символа из входной цепочки в стек и сдвиг считывающей головки на один шаг вправо) и вернуться к шагу 2. Иначе перейти к шагу 4.

Шаг 4. Если имеет место отношение $\cdot>$, то произвести свертку. Для этого надо найти на вершине стека все терминальные символы, связанные отношением $=\cdot$ («основу»), а также все соседствующие с ними нетерминальные символы (при определении отношения нетерминальные символы игнорируются). Если терминальных символов,

связанных отношением =·, на верхушке стека нет, то в качестве основы используется один, самый верхний в стеке терминальный символ стека. Все (и терминальные, и нетерминальные) символы, составляющие основу, надо удалить из стека, а затем выбрать из грамматики правило, имеющее правую часть, совпадающую с основой, и поместить в стек левую часть выбранного правила. Если правило, совпадающее с основой, найти не удалось, то необходимо прервать выполнение алгоритма и сообщить об ошибке, иначе, если разбор не закончен, то вернуться к шагу 2.

Шаг 5. Если не установлено ни одно отношение предшествования между текущим символом входной цепочки и самым верхним терминальным символом в стеке, то надо прервать выполнение алгоритма и сообщить об ошибке.

Конечная конфигурация данного МП-автомата совпадает с конфигурацией при распознавании цепочек грамматик простого предшествования.

Пример построения распознавателя для грамматики операторного предшествования

Рассмотрим в качестве примера грамматику для арифметических выражений над символами a и b G({+,−,/,*,a,b},{S,T,E},**P**,S):

P:

S → S+T | S−T | T

T → T*E | T/E | E

E → (S) | a | b

Эта грамматика уже много раз использовалась в качестве примера для построения распознавателей.

Видно, что эта грамматика является грамматикой операторного предшествования.

Построим множества крайних левых и крайних правых символов **L**(A), **R**(A) относительно всех нетерминальных символов грамматики. Рассмотрим работу алгоритма построения этих множеств по шагам.

Шаг 1.

$\mathbf{L}_0$(S) = {S,T}, $\mathbf{R}_0$(S) = {T},

$\mathbf{L}_0$(T) = {T,E}, $\mathbf{R}_0$(T) = {E},

$\mathbf{L}_0$(E) = {(,a,b}, $\mathbf{R}_0$(E) = {),a,b}, i = 1

Шаг 2:

$\mathbf{L}_1$(S) = {S,T,E}, $\mathbf{R}_1$(S) = {T,E},

$\mathbf{L}_1$(T) = {T,E,(,a,b}, $\mathbf{R}_1$(T) = {E,),a,b},

$\mathbf{L}_1$(E) = {(,a,b}, $\mathbf{R}_1$(E) = {),a,b}

Шаг 3: так как $\mathbf{L}_0$(S)≠$\mathbf{L}_1$(S), то i=2 и возвращаемся к шагу 2.

Шаг 2:

$\mathbf{L}_2(S) = \{S,T,E,(,a,b\}$, $\mathbf{R}_2(S) = \{T,E,),a,b\}$,

$\mathbf{L}_2(T) = \{T,E,(,a,b\}$, $\mathbf{R}_2(T) = \{E,),a,b\ \}$,

$\mathbf{L}_2(E) = \{(,a,b\}$, $\mathbf{R}_2(E) = \{),a,b\}$

Шаг 3: так как $\mathbf{L}_1(S) \neq \mathbf{L}_2(S)$, то i=3 и возвращаемся к шагу 2.

Шаг 2:

$\mathbf{L}_3(S) = \{S,T,E,(,a,b\}$, $\mathbf{R}_3(S) = \{T,E,),a,b\ \}$,

$\mathbf{L}_3(T) = \{T,E,(,a,b\}$, $\mathbf{R}_3(T) = \{E,),a,b\ \}$,

$\mathbf{L}_3(E) = \{(,a,b\}$, $\mathbf{R}_3(E) = \{),a,b\}$

Построение закончено. Получили результат:

$\mathbf{L}(S) = \{S,T,E,(,a,b\}$, $\mathbf{R}(S) = \{T,E,),a,b\ \}$,

$\mathbf{L}(T) = \{T,E,(,a,b\}$, $\mathbf{R}(T) = \{E,),a,b\ \}$,

$\mathbf{L}(E) = \{(,a,b\}$, $\mathbf{R}(E) = \{),a,b\}$

На основе полученных множеств построим множества крайних левых и крайних правых терминальных символов $\mathbf{L}^t(A)$, $\mathbf{R}^t(A)$ относительно всех нетерминальных символов грамматики. Рассмотрим работу алгоритма по шагам:

Шаг 1.

$\mathbf{L}^t_0(S) = \{+,–\}$, $\mathbf{R}^t_0(S) = \{+,–\}$,

$\mathbf{L}^t_0(T) = \{*,/\}$, $\mathbf{R}^t_0(T) = \{*,/\}$,

$\mathbf{L}^t_0(E) = \{(,a,b\}$, $\mathbf{R}^t_0(E) = \{),a,b\}$

Шаг 2.

$\mathbf{L}^t(S) = \{+,–,*,/,(,a,b\}$, $\mathbf{R}^t(S) = \{+,–,*,/,),a,b\}$,

$\mathbf{L}^t(T) = \{*,/,(,a,b\}$, $\mathbf{R}^t(T) = \{*,/,),a,b\}$,

$\mathbf{L}^t(E) = \{(,a,b\}$, $\mathbf{R}^t(E) = \{),a,b\}$

Построение закончено.

На основе этих множеств и правил грамматики **G** построим матрицу предшествования грамматики (табл. 4.6).

Таблица 4.6. Матрица предшествования грамматики

Символы	+	-	*	/	(	)	A	b	$\perp_к$
+	·>	·>	<·	<·	<·	·>	<·	<·	·>
-	·>	·>	<·	<·	<·	·>	<·	<·	·>
*	·>	·>	·>	·>	<·	·>	<·	<·	·>
/	·>	·>	·>	·>	<·	·>	<·	<·	·>
(	<·	<·	<·	<·	<·	=·	<·	<·	
)	·>	·>	·>	·>		·>			·>
a	·>	·>	·>	·>		·>			·>
b	·>	·>	·>	·>		·>			·>
$\perp_н$	<·	<·	<·	<·	<·		<·	<·	

Поясним, как заполняется матрица предшествования в таблице на примере символа +. В правиле грамматики S→S+T (правило 1) этот символ стоит слева от нетерминального символа T. В множество $\mathbf{L}^t$(T) входят символы: *, /, (, a, b. Ставим знак <· в клетках матрицы, соответствующих этим символам, в строке для символа +. В то же время в этом же правиле символ + стоит справа от нетерминального символа S. Во множество $\mathbf{R}^t$(S) входят символы: +, –, *, /,), a, b. Ставим знак ·> в клетках матрицы, соответствующим этим символам, в столбце для символа +. Больше символ + ни в каком правиле не встречается, значит, заполнение матрицы для него закончено, берем следующий символ и продолжаем заполнять матрицу таким же методом, пока не переберем все терминальные символы.

Отдельно рассмотрим символы $\perp_н$ и $\perp_к$. В строке символа $\perp_н$ ставим знак <· в клетках символов, входящих во множество $\mathbf{L}^t$(S). Это символы +, –, *, /, (, a, b. В столбце символа $\perp_к$ ставим знак ·> в клетках символов, входящих во множество $\mathbf{R}^t$(S). Это символы +, –, *, /,), a, b.

Еще можно отметить, что в клетке соответствующей открывающей скобке (символ `(`) слева и закрывающей скобке (символ `)`) справа помещается знак =· («составляют основу»). Так происходит, поскольку в грамматике присутствует правило `E→(S)`, где эти символы стоят рядом (через нетерминальный символ) в его правой части. Следует отметить, что понятия «справа» и «слева» здесь имеют важное значение: в клетке соответствующей закрывающей скобке (символ `)`) слева и открывающей скобке (символ `(`) справа знак отсутствует — такое сочетание символов недопустимо (отношение `(`=·`)` верно, а отношение `)`=·`(` — неверно).

Алгоритм разбора цепочек грамматики операторного предшествования игнорирует нетерминальные символы. Поэтому имеет смысл преобразовать исходную грамматику таким образом, чтобы оставить в ней только один нетерминальный символ. Тогда получим следующий вид правил:

P:

S → S+S | S−S | S (правила 1, 2 и 3)

S → S*S | S/S | S (правила 4, 5 и 6)

S → (S) | a | b (правила 7, 8 и 9)

Если теперь исключить бессмысленные правила вида S→S, то получим следующее множество правил (нумерацию правил сохраним в соответствии с исходной грамматикой):

P:

S → S+S | S−S (правила 1, 2)

S → S*S | S/S (правила 4, 5)

S → (S) | a | b (правила 7, 8 и 9)

Такое преобразование не ведет к созданию эквивалентной грамматики и выполняется только для упрощения работы алгоритма после построения матрицы предшествования. Полученная в результате преобразования грамматика не является однозначной, но в алгоритм распознавания уже были заложены все необходимые данные о порядке применения правил при создании матрицы предшествования, поэтому распознаватель остается детерминированным. Построенная таким способом грамматика называется «*остовной*» грамматикой. Вывод, полученный при разборе на основе остовной грамматики, называют результатом «остовного» разбора или «остовным» выводом [4 т.1,2, 5].

По результатам остовного разбора можно построить соответствующий ему вывод на основе правил исходной грамматики. Однако эта задача не представляет практического интереса, поскольку остовный вывод отличается от вывода на основе исходной грамматики только тем, что в нем отсутствуют шаги, связанные с применением цепных правил, и не учитываются типы нетерминальных символов. Для компиляторов же распознавание цепочек входного языка заключается не в нахождении того или иного вывода, а в выявлении основных синтаксических конструкций исходной программы с целью построения на их основе цепочек языка результирующей программы. В этом смысле типы нетерминальных символов и цепные правила не несут никакой полезной информации, а напротив, только усложняют обработку цепочки вывода. Поэтому для реального компилятора нахождение остовного вывода является даже более полезным, чем нахождение вывода на основе исходной грамматики. Найденный остовный вывод в дальнейших преобразованиях уже не нуждается[15].

---

[15] Из цепочки (и дерева) вывода удаляются цепные правила, которые, как будет показано далее, все равно не несут никакой полезной семантической (смысловой) нагрузки, а потому для компилятора являются бесполезными. Это положительное свойство распознавателя.

Рассмотрим работу алгоритма распознавания на примерах. Последовательность разбора будем записывать в виде последовательности конфигураций расширенного МП-автомата из трех составляющих:

1. не просмотренная автоматом часть входной цепочки;
2. содержимое стека;
3. последовательность примененных правил грамматики.

Так как автомат имеет только одно состояние, то для определения его конфигурации достаточно двух составляющих — положения считывающей головки во входной цепочке и содержимого стека. Последовательность номеров правил несет дополнительную полезную информацию, по которой можно построить цепочку или дерево вывода.

Как и ранее, будем обозначать такт автомата: $\div_п$, если на данном такте выполнялся перенос, и $\div_с$, если выполнялась свертка.

Последовательности разбора цепочек входных символов будут, таким образом, иметь вид, приведенный ниже.

Пример 1. Входная цепочка `a+a*b`.

1. {a+a*b⊥к;⊥нS;∅}÷п
2. {+a*b⊥к;⊥нa;∅}÷с
3. {+a*b⊥к;⊥нS;8}÷п
4. {a*b⊥к;⊥нS+;8}÷п
5. {*b⊥к;⊥нS+a;8}÷с
6. {*b⊥к;⊥нS+S;8,8}÷п
7. {b⊥к;⊥нS+S*;8,8}÷п
8. {⊥к;⊥нS+S*b;8,8}÷с
9. {⊥к;⊥нS+S*S;8,8,9}÷с
10. {⊥к;⊥нS+S;8,8,9,4}÷с
11. {⊥к;⊥нS;8,8,9,4,1} — разбор завершен, цепочка принята.

Соответствующая цепочка вывода будет иметь вид (используется правосторонний вывод): S⇒S+S⇒S+S*S⇒S+S*b⇒S+a*b⇒a+a*b.

Дерево вывода, соответствующее этой цепочке, приведено на рис. 4.13.

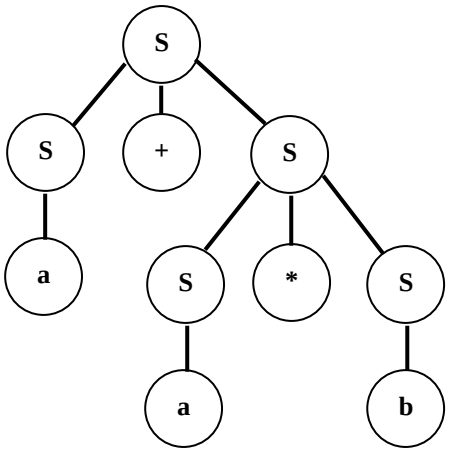


Рис. 4.13. Пример дерева вывода для грамматики операторного предшествования

Пример 2. Входная цепочка `(a+a)*b`.

1. {(a+a)*b⊥к;⊥н;∅}÷п
2. {a+a)*b⊥к;⊥н(;∅}÷п
3. {+a)*b⊥к;⊥н(a;∅}÷с
4. {+a)*b⊥к;⊥н(S;8}÷п
5. {a)*b⊥к;⊥н(S+;8}÷п
6. {)*b⊥к;⊥н(S+a;8}÷с
7. {)*b⊥к;⊥н(S+S;8,8}÷с
8. {)*b⊥к;⊥н(S;8,8,1}÷п
9. {*b⊥к;⊥н(S);8,8,1}÷с
10. {*b⊥к;⊥нS;8,8,1,7}÷п
11. {b⊥к;⊥нS*;8,8,1,7}÷п
12. {⊥к;⊥нS*b;8,8,1,7}÷с
13. {⊥к;⊥нS*S;8,8,1,7,9}÷с
14. {⊥к;⊥нS;8,8,1,7,9,4} — разбор завершен, цепочка принята.

Соответствующая цепочка вывода будет иметь вид (используется правосторонний вывод): S⇒S*S⇒S*b⇒(S)*b⇒(S+S)*b⇒(S+a)*b⇒(a+a)*b.

Дерево вывода, соответствующее этой цепочке, приведено на рис. 4.14.

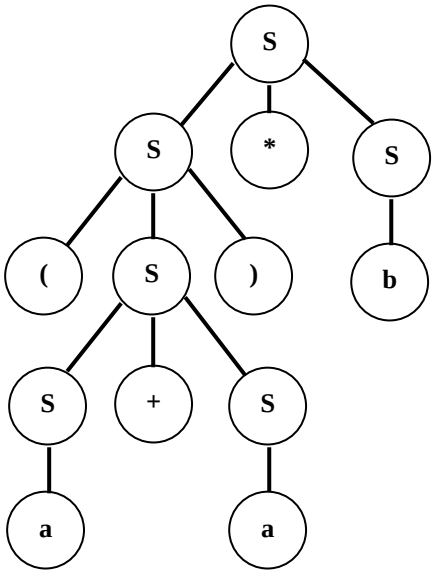


Рис. 4.14. Пример дерева вывода для грамматики операторного предшествования

Пример 3. Входная цепочка `a+a*`.

1. $\{a+a*\perp_к;\perp_н;\varnothing\}\div_п$
2. $\{+a*\perp_к;\perp_н a;\varnothing\}\div_с$
3. $\{+a*\perp_к;\perp_н S;8\}\div_п$
4. $\{a*\perp_к;\perp_н S+;8\}\div_п$
5. $\{*\perp_к;\perp_н S+a;8\}\div_с$
6. $\{*\perp_к;\perp_н S+S;8,8\}\div_п$
7. $\{\perp_к;\perp_н S+S*;8,8\}\div_с$
8. `ошибка!` (нет правила для выполнения свертки на этом шаге).

Пример 4. Входная цепочка `a+a)*b`.

1. $\{a+a)*b\perp_к;\perp_н;\varnothing\}\div_п$
2. $\{+a)*b\perp_к;\perp_н a;\varnothing\}\div_с$
3. $\{+a)*b\perp_к;\perp_н S;8\}\div_п$
4. $\{a)*b;\perp_н S+;7\}\div_п$
5. $\{)*b\perp_к;\perp_н S+a;8\}\div_с$
6. $\{)*b\perp_к;\perp_н S+S;8,8\}\div_с$
7. $\{)*b\perp_к;\perp_н S;8,8,1\}$
8. `ошибка!` (нет отношений предшествования между символами $\perp_н$ и `)`).

Два первых примера наглядно демонстрируют, что приоритет операций, установленный в грамматике, влияет на последовательность разбора и

последовательность применения правил несмотря на то, что нетерминальные символы распознавателем не учитываются.

Более подробно построение распознавателя для грамматик операторного предшествования рассмотрено в книге [42].

Как было сказано выше, матрица для грамматики операторного предшествования всегда имеет меньший объем, чем матрица для эквивалентной ей грамматики простого предшествования. Кроме того, распознаватель грамматики операторного предшествования игнорирует нетерминальные символы в процессе разбора, а значит, не учитывает цепные правила, делает меньше шагов и порождает более короткую цепочку вывода. Поэтому распознаватель для грамматики операторного предшествования всегда проще, чем распознаватель для эквивалентной ей грамматики простого предшествования.

Интересно, что поскольку распознаватель на основе грамматик операторного предшествования не учитывает типы нетерминальных символов, то он может работать даже с неоднозначными грамматиками, в которых есть правила, различающиеся только типами нетерминальных символов. Примером такой грамматики может служить грамматика `G"({a,b},{S,A,B},P,S)` с правилами:

P:

`S → A | B`

`A → aAb | ab`

`B → aBb | ab`

Как и для любой другой грамматики операторного предшествования, распознаватель для этой грамматики будет детерминированным. Остовная грамматика, построенная на ее основе, будет иметь только два правила вида: `S→aSb|ab`. Неоднозначность заключается в том, что каждому найденному остовному выводу будет соответствовать не один, а несколько выводов в исходной грамматике (в данном случае — всегда два вывода в зависимости от того, какое правило из `S→A|B` будет применено на первом шаге вывода).

ПРИМЕЧАНИЕ

> Грамматики, содержащие правила, различающиеся только типами нетерминальных символов, практического значения не имеют, а потому интереса для построения компиляторов не представляют.

Хотя классы грамматик простого и операторного предшествования несопоставимы[16], класс языков операторного предшествования уже, чем класс языков простого предшествования. Поэтому не всегда возможно для языка, заданного грамматикой простого предшествования, построить грамматику операторного предшествования.

---

[16] В том, что эти два класса грамматик несопоставимы, можно убедится, рассмотрев два приведенных выше примера — в них взяты различные по своей сути и классам грамматики, хотя они и являются эквивалентными — задают один и тот же язык.

Поскольку класс языков, заданных грамматиками операторного предшествования, еще более узок, чем даже класс языков, заданных грамматиками простого предшествования, с помощью этих грамматик можно определить далеко не каждый детерминированный КС-язык. Грамматики операторного предшествования — это очень удобный инструмент для построения распознавателей, но они имеют ограниченную область применения.

Контрольные вопросы и задачи

Вопросы

1. Какие из следующих утверждений справедливы:

- если язык задан КС-грамматикой, то он может быть задан с помощью МП-автомата;
- если язык задан КС-грамматикой, то он может быть задан с помощью ДМП-автомата;
- если язык задан КА, то он может быть задан КС-грамматикой;
- если язык задан ДМП-автоматом, то он может быть задан КС-грамматикой;
- если язык задан однозначной КС-грамматикой, то для него можно построить КА;
- если язык задан однозначной КС-грамматикой, то он может быть задан с помощью ДМП-автомата;
- если язык задан расширенным МП-автоматом, то он может быть задан КС-грамматикой?

2. Какие из приведенных ниже МП-автоматов являются детерминированными:

$R_1(\{q_1,q_2\},\{a,b\},\{a,b,A\},\delta_1,q_1,A,\{q_2\}),\delta_1(q_1,a,A) = \{(q_1,AA)\},\delta_1(q_2,a,A) = \{(q_2,\lambda)\},\delta_1(q_1,b,A) = \{(q_2,A)\},\delta_1(q_2,b,A) = \{(q_2,Aab)\},\delta_1(q_1,\lambda,a) = \{(q_1,\lambda)\},\delta_1(q_1,\lambda,b) = \{(q_1,\lambda)\};$

$R_2(\{q\},\{a,b\},\{a,b,A\},\delta_2,q,A,\{q\}),\delta_2(q,a,A) = \{(q,AA)\},\delta_2(q,b,A) = \{(q,A)\},\delta_2(q,\lambda,a) = \{(q,\lambda)\},\delta_2(q,\lambda,b) = \{(q,\lambda)\},\delta_2(q,\lambda,A) = \{(q,A)\};$

$R_3(\{q\},\{a,b\},\{a,b,A\},\delta_3,q,A,\{q\}),\delta_3(q,a,A) = \{(q,AA),(q,a),(q,\lambda)\},\delta_3(q,b,A) = \{(q,A)\},\delta_3(q,\lambda,a) = \{(q,A)\};$

$R_4(\{q\},\{a,b\},\{a,b,A\},\delta_4,q,A,\{q\}),\delta_4(q,a,A) = \{(q,AAb)\},\delta_4(q,b,A) = \{(q,a)\},\delta_4(q,\lambda,a) = \{(q,\lambda)\},\delta_4(q,\lambda,b) = \{(q,\lambda)\}?$

3. Почему синтаксические конструкции языков программирования могут быть распознаны с помощью ДМП-автоматов?

4. Можно ли полностью проверить структуру входной программы с помощью ДМП-автоматов для большинства языков программирования? Если нет, то можно ли решить ту же задачу, используя МП-автоматы или расширенные МП-автоматы?
5. Всегда ли преобразование правил КС-грамматик ведет к упрощению правил?
6. Почему при преобразовании КС-грамматики к приведенному виду сначала необходимо удалить бесплодные символы, а потом — недостижимые символы?
7. Почему для языка, заданного КС-грамматикой, содержащей λ-правила, существуют трудности с моделированием работы расширенного МП-автомата, выполняющего восходящий разбор с правосторонним выводом?
8. Почему необходимо устранить именно левую рекурсию из правил грамматики? Для моделирования работы какого (восходящего или нисходящего) распознавателя представляет сложности левая рекурсия?
9. Можно ли полностью устранить рекурсию из правил грамматики, записанных в форме Бэкуса-Наура?
10. На алгоритме работы какого распознавателя основан метод рекурсивного спуска?
11. Можно ли реализовать распознаватель по методу расширенного рекурсивного спуска для грамматики, содержащей левую рекурсию?
12. За счет чего класс LL(1)-грамматик является более широким, чем класс КС-грамматик, для которых можно построить распознаватель по методу рекурсивного спуска?
13. На каком алгоритме основана работа распознавателя для LL(k)-грамматики?
14. На каком алгоритме основана работа распознавателя для LR(k)-грамматики?
15. Почему в грамматиках простого предшествования и операторного предшествования не могут присутствовать λ-правила?
16. Класс языков, заданных LR(1)-грамматиками совпадает с классом ДКС-языков. Зачем же тогда используются другие классы грамматик?

Задачи

1. Задана грамматика G({".",+,-,0,1}, {<число>,<часть>,<цифра>,<осн>}, **P**, <число>)

   P: <число> → +<осн> | -<осн> | <осн>

   <осн> → <часть>.<часть> | <часть>. | <часть>

   <часть> → <цифра> | <часть><цифра>

   <цифра> → 0 | 1

постройте для нее обычный и расширенный МП-автоматы.

Являются ли построенные автоматы детерминированными?

2. Дана грамматика: G({a,b,c},{A,B,C,D,E,F,G,S},**P**,S)

P: S → aAbB | E

A → BCa | a | λ

B → ACb | b | λ

C → A | B | bA | aB | cC | aE | bE

E → Ea | Eb | Ec | ED | FG | DG

D → c | Fb | Fa | λ

F → BC | AC | DC | EC

G → Ga | Gb | Gc | GD

преобразуйте ее к приведенному виду.

3. Преобразуйте грамматику, построенную в задаче №2, к виду без λ-правил.
4. Дана грамматика G({" ",(,),o,r,a,n,d,t,b},{S,T,E,F},**P**,S)

   P: S → S or T | T

   T → T and E | E

   E → not E | F

   F → (S) | b

- преобразуйте ее к виду без цепных правил;
- исключите левую рекурсию в правилах грамматики.

5. Дана грамматика: G({if,then,else,a,b},{S,T,E,F},**P**,S)

   P: S → if b then T else S | if b then S | a

   T → if b then T else T | a

Выполните для нее разбор цепочки символов if b then if b then if b then a else a.

К первому или к последнему if будет отнесено else в этой цепочке?

Упражнения

1. Постройте нисходящий распознаватель с возвратом для грамматики, заданной в задаче №3. Выполните разбор цепочки символов «a or a and not a and (a or not not a)».
2. Постройте восходящий распознаватель с возвратом для грамматики, заданной в задаче №3. Выполните разбор цепочки символов «a or a and not a and (a or not not a)».
3. Постройте распознаватель, основанный на методе рекурсивного спуска для грамматики: G({a,b,c},{S,A,B,C},**P**,S)

P: S → aABb | bBAa | cCc

A → aA | bB | cC

B → b | aAC

C → aA | bA | cC

выполните разбор цепочки символов aabbabbcabbb.

4. Постройте распознаватель, основанный на расширенном методе рекурсивного спуска для грамматики: G({a,<,=,>,+,-,/,*},{S,T,E},**P**,S)

   P: S → T<T | T>T | T<=T | T>=T

   T → T+E | T-E | E

   E → a*a | a/a | a

выполните разбор цепочки символов a*a+a<a/a-a*a.

5. Дана грамматика G({(,),+,*,a},{S,T,F},**P**,S)

   P: S → S+T | T

   T → T*E | F

   F → (S) | a

постройте для нее распознаватель на основе SLR(1)-грамматики.

6. Дана грамматика G({(,),^,&,~,a},{S,T,E,F},**P**,S)

   P: S → S^T | T

   T → T&E | E

   E → ~E | F

   F → (S) | a

постройте для нее распознаватель на основе грамматики операторного предшествования. Выполните разбор цепочки символов a^a&~a&(a^~~a).

7. Дана грамматика: G({if,then,else,a,b},{S,T,E,F},**P**,S)

   P: S → if b then T else S | if b then S | a

   T → if b then T else T | a

постройте для нее распознаватель на основе грамматики операторного предшествования, считая if, then и else едиными терминальными символами.