Контекстно-свободные грамматики, вывод, лево- и правосторонний вывод, дерево разбора

Содержание

Основные определения

Определение:

Контекстно-свободной грамматикой (англ. *context-free grammar*) называется грамматика, у которой в левых частях всех правил стоят только одиночные нетерминалы.

Определение:

Контекстно-свободный язык (англ. *context-free language*) — язык, задаваемый контекстно-свободной грамматикой.

Лево- и правосторонний вывод слова

Определение:

Выводом слова (англ. *derivation of a word*) α называется последовательность строк, состоящих из терминалов и нетерминалов. Первая строка последовательности состоит из одного стартового нетерминала. Каждая последующая строка получена из предыдущей путем замены любого нетерминала по одному (любому) из правил, а последней строкой в последовательности является слово α.

Пример:

Рассмотрим грамматику, выводящую все правильные скобочные последовательности.

$($ и $)$ — терминальные символы
S — стартовый нетерминал

Правила:

1. $S \to (S)S$
2. $S \to S(S)$
3. $S \to \varepsilon$

Выведем слово $(()(()))()$:

$$\boldsymbol{S} \Rightarrow (S)\boldsymbol{S} \Rightarrow (S)(\boldsymbol{S})S \Rightarrow (S)()\boldsymbol{S} \Rightarrow (\boldsymbol{S})() \Rightarrow (\boldsymbol{S}(S))() \Rightarrow (S(S)(\boldsymbol{S}))() \Rightarrow (S(S)(\boldsymbol{S}(S)))() \Rightarrow (S($$

Определение:

Левосторонним выводом слова (англ. *leftmost derivation*) α называется такой вывод слова α, в котором каждая последующая строка получена из предыдущей путем замены по одному из правил самого левого встречающегося в строке нетерминала.

Определение:

Правосторонним выводом слова (англ. *rightmost derivation*) α называется такой вывод слова α, в котором каждая последующая строка получена из предыдущей путем замены по одному из правил самого правого встречающегося в строке нетерминала.

Рассмотрим левосторонний вывод скобочной последовательности из примера:

$$\boldsymbol{S} \Rightarrow (\boldsymbol{S})S \Rightarrow ((\boldsymbol{S})S)S \Rightarrow (()\boldsymbol{S})S \Rightarrow (()\boldsymbol{S}(S))S \Rightarrow (()(\boldsymbol{S}))S \Rightarrow (()(\boldsymbol{S}(S)))S \Rightarrow (()((\boldsymbol{S})))S \Rightarrow (()($$

Дерево разбора

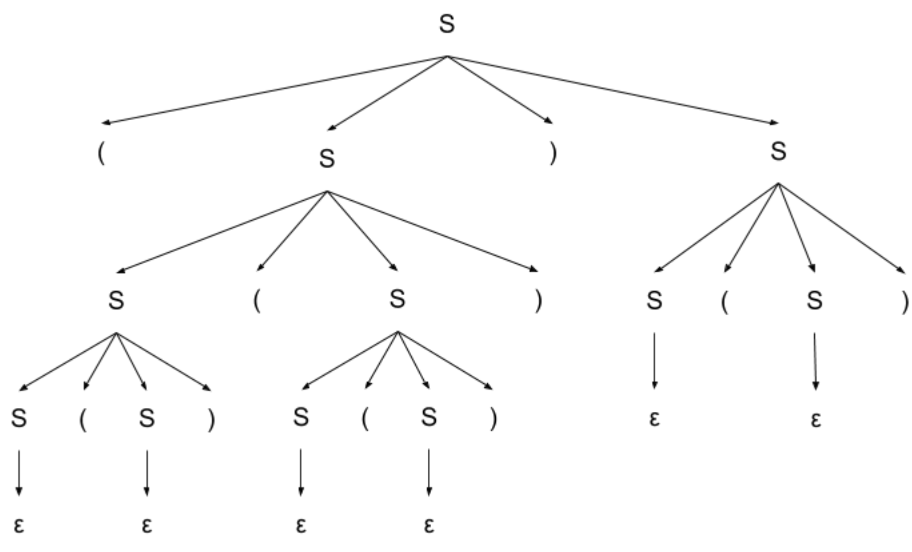
Определение:

Деревом разбора грамматики (англ. *parse tree*) называется дерево, в вершинах которого записаны терминалы или нетерминалы. Все вершины, помеченные терминалами, являются листьями. Все вершины, помеченные нетерминалами, имеют детей. Дети вершины, в которой записан нетерминал, соответствуют раскрытию нетерминала по одному любому правилу (в левой части которого стоит этот нетерминал) и упорядочены так же, как в правой части этого правила.

Определение:

Крона дерева разбора (англ. *leaves of the parse tree*) — множество терминальных символов, упорядоченное в соответствии с номерами их достижения при обходе дерева в глубину из корня. Крона дерева разбора представляет из себя слово языка, которое выводит это дерево.

Построим дерево разбора скобочной последовательности из примера.



Теорема:

Пусть $\Gamma = \langle \Sigma, N, S, P \rangle$ — КС-грамматика. Предположим, что существует дерево разбора с корнем, отмеченным A, и кроной ω, где $\omega \in N^*$. Тогда в грамматике Γ существует левое порождение $A \Rightarrow_{lm}^* \omega$

Доказательство:

$\triangleright$

Используем индукцию по высоте дерева.

База: Базисом является высота 1, наименьшая из возможных для дерева разбора с терминальной кроной.

Поскольку это дерево является деревом разбора, $A \to \omega$ должно быть продукцией. Таким образом, $A \Rightarrow_{lm} \omega$ есть одношаговое левое порождение ω из A.

Индукционный переход: Существует корень с отметкой A и сыновьями, отмеченными слева направо $X_1 X_2 \ldots X_k$. Символы X могут быть как терминалами, так и переменными.

1. Если X_i — терминал, то определим ω_i как цепочку, состоящую из одного X_i.
2. Если X_i — переменная, то она должна быть корнем некоторого поддерева с терминальной кроной, которую обозначим ω_i. Заметим, что в этом случае высота поддерева меньше n, поэтому к нему применимо предположение индукции. Следовательно, существует левое порождение $X_i \Rightarrow^*_{lm} \omega_i$.

Заметим, что $\omega = \omega_1 \omega_2 \ldots \omega_k$. Построим левое порождение цепочки ω следующим образом:

> Начнем с шага $A \Rightarrow_{lm} X_1 X_2 \ldots X_k$.
> Затем для $i = 1, 2, \ldots, k$ покажем, что имеет место следующее порождение:
> $A \Rightarrow^*_{lm} \omega_1 \omega_2 \ldots \omega_i X_{i+1} X_{i+2} \ldots X_k$

Данное доказательство использует в действительности еще одну индукцию, на этот раз по i. Для базиса $i = 0$ мы уже знаем, что $A \Rightarrow_{lm} X_1 X_2 \ldots X_k$.

Для индукции предположим, что существует следующее порождение: $A \Rightarrow^*_{lm} \omega_1 \omega_2 \ldots \omega_{i-1} X_i X_{i+1} \ldots X_k$

1. Если X_i — терминал, то не делаем ничего, но в дальнейшем рассматриваем X_i как терминальную цепочку ω_i. Таким образом, приходим к существованию следующего порождения. $A \Rightarrow^*_{lm} \omega_1 \omega_2 \ldots \omega_i X_{i+1} X_{i+2} \ldots X_k$
2. Если X_i является переменной, то продолжаем порождением ω_i из X_i в контексте уже построенного порождения. Таким образом, если этим порождением является: $X_i \Rightarrow_{lm} \alpha_1 \Rightarrow_{lm} \alpha_2 \ldots \Rightarrow_{lm} \omega_i$, то продолжаем следующими порождениями:

$$\omega_1 \omega_2 \ldots \omega_{i-1} X_i X_{i+1} \ldots X_k \Rightarrow_{lm}$$

$$\omega_1 \omega_2 \ldots \omega_{i-1} \alpha_1 X_{i+1} \ldots X_k \Rightarrow_{lm}$$

$$\omega_1 \omega_2 \ldots \omega_{i-1} \alpha_2 X_{i+1} \ldots X_k \Rightarrow_{lm}$$

$$\ldots$$

$$\omega_1 \omega_2 \ldots \omega_i X_{i+1} X_{i+2} \ldots X_k$$

Результатом является порождение $A \Rightarrow^*_{lm} \omega_1 \omega_2 \ldots \omega_i X_{i+1} X_{i+2} \ldots X_k$.

Когда $i = k$, результат представляет собой левое порождение ω из A.

◁

Теорема:

> Для каждой грамматики $\Gamma = \langle \Sigma, N, S, P \rangle$ и ω из N^* цепочка ω имеет два разных дерева разбора тогда и только тогда, когда ω имеет два разных левых порождения из P.

Доказательство:

▷

$\Longrightarrow$

> Внимательно рассмотрим построение левого порождения по дереву разбора в доказательстве теоремы. В любом случае, если у двух деревьев разбора впервые появляется узел, в котором применяются различные продукции, левые порождения, которые строятся, также используют разные продукции и, следовательно, являются различными.

$\Longleftarrow$

> Хотя мы предварительно не описали непосредственное построение дерева разбора по левому порождению, идея его проста. Начнем построение дерева с корня, отмеченного стартовым символом. Рассмотрим порождение пошагово. На каждом шаге заменяется переменная, и эта переменная будет соответствовать построенному крайнему слева узлу дерева, не имеющему сыновей, но отмеченному этой переменной. По продукции, использованной на этом шаге левого порождения, определим, какие сыновья должны быть у этого узла. Если существуют два разных порождения, то на первом шаге, где они различаются, построенные узлы получат разные списки сыновей, что гарантирует различие деревьев разбора.

◁

Однозначные грамматики

Определение:

Грамматика называется **однозначной** (англ. *unambiguous grammar*), если у каждого слова имеется не более одного дерева разбора в этой грамматике.

Лемма:

Пусть Γ — однозначная грамматика. Тогда $\forall \omega \in \mathbb{L}(\Gamma)$ существует ровно один левосторонний (правосторонний) вывод.

Доказательство:

▷

Очевидно, что по дереву разбора однозначно восстанавливается левосторонний(правосторонний) вывод. Поскольку каждое слово из языка выводится только одним деревом разбора, то существует только один левосторонний(правосторонний) вывод этого слова.

◁

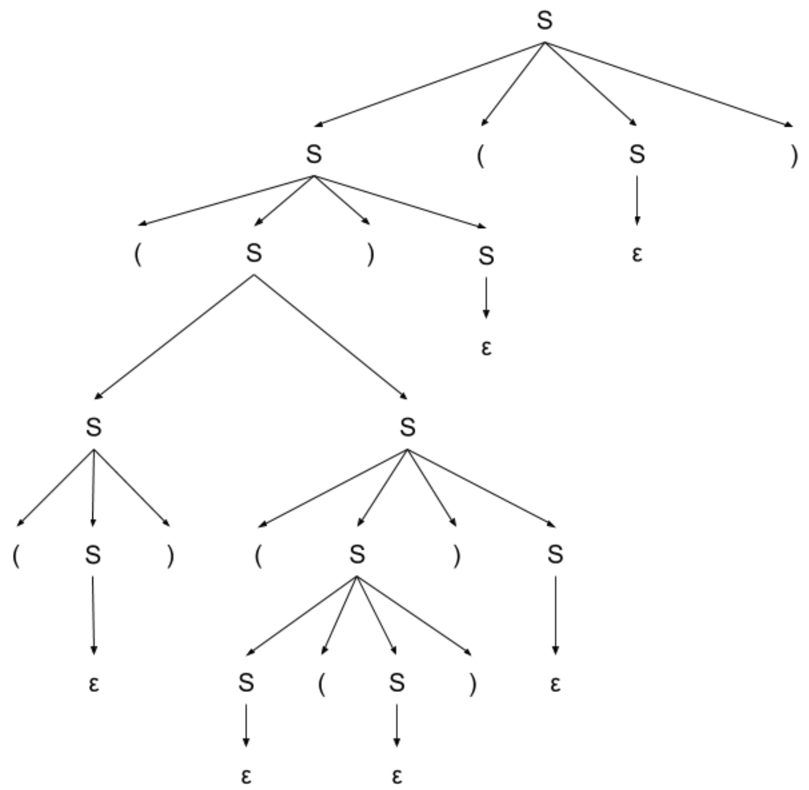
Утверждение:

Грамматика из примера не является однозначной.

▷

Выше уже было построено дерево разбора для слова $(()(()))()$. Построим еще одно дерево разбора для данного слова.

Например, оно будет выглядеть так:



Таким образом, существует слово, у которого есть более одного дерева разбора в данной грамматике $\Rightarrow$ эта грамматика не является однозначной.

◁

Утверждение:

Существуют языки, которые можно задать одновременно как однозначными, так и неоднозначными грамматиками.

▷

Для доказательства достаточно привести однозначную грамматику для языка правильных скобочных последовательностей (неоднозначной грамматикой для данного языка является грамматика из примера выше).

Рассмотрим грамматику:

$($ и $)$ — терминальные символы
S — стартовый нетерминал

Правила:

1. $S \to (S)S$
2. $S \to \varepsilon$

Покажем, что эта грамматика однозначна. Для этого, используя индукцию, докажем, что для любого слова ω, являющегося правильной скобочной последовательностью, в данной грамматике существует только одно дерево разбора.

База: Если $\omega = \varepsilon$, то оно выводится только по второму правилу $\Rightarrow$ для него существует единственное дерево разбора.

Индукционный переход: Пусть $|\omega| = n$ и $\forall \upsilon$: $|\upsilon| < n$ и υ — правильная скобочная последовательность, у которой $\exists!$ дерево разбора.

Найдем в слове ω минимальный индекс $i \neq 0$ такой, что слово $\omega[0 \ldots i]$ является правильной скобочной последовательностью. Так как $i \neq 0$ минимальный, то $\omega[0 \ldots i] = (\alpha)$. Из того, что ω является правильной скобочной последовательностью $\Rightarrow \alpha$ и $\beta = \omega[i+1 \ldots n-1]$ — правильные скобочные последовательности, при этом $|\alpha| < n$ и $|\beta| < n \Rightarrow$ по индукционному предположению предположению у α и β существуют единственные деревья разбора.

Если мы покажем, что из части (S) первого правила можно вывести только слово (α), то утверждение будет доказано (так как из первой части первого правила выводится α, а из второй только β и для каждого из них по предположению существуют единственные деревья разбора).

Пусть из (S) была выведена часть слова $\omega[0 \ldots j] = (\gamma)$, где $j < i$, при этом γ является правильной скобочной последовательностью, но тогда как минимальный индекс мы должны были выбрать j, а не i — противоречие.

Аналогично из (S) не может быть выведена часть слова $\omega[0 \ldots j]$, где $j > i$, потому что тогда $\omega[0 \ldots i] = (\alpha)$ не будет правильной скобочной последовательностью, так как в позиции $i-1$ баланс скобок будет отрицательный.

Значит, из (S) была выведена часть слова $\omega[0 \ldots i] \Rightarrow \omega$ имеет единственное дерево разбора $\Rightarrow$ данная грамматика однозначная.

Таким образом, для языка правильных скобочных последовательностей мы привели пример как однозначной, так и неоднозначной грамматики.
◁

Однако, есть КС-языки, для которых не существует однозначных КС-грамматик. Такие языки и грамматики их порождающие называют **существенно неоднозначными**.

См. также

- Формальные грамматики
- Иерархия Хомского формальных грамматик
- Замкнутость КС-языков относительно различных операций
- Существенно неоднозначные языки

Источники информации

- Wikipedia — Context-free grammar
- Википедия — Контекстно-свободная грамматика

- *Хопкрофт Д., Мотвани Р., Ульман Д.* — **Введение в теорию автоматов, языков и вычислений**, 2-е изд. : Пер. с англ. — Москва, Издательский дом «Вильямс», 2002. — 528 с. : ISBN 5-8459-0261-4 (рус.)

Устранение левой рекурсии

Содержание

Определение:

Говорят, что контекстно-свободная (КС) грамматика Γ содержит **непосредственную левую рекурсию** (англ. *direct left recursion*), если она содержит правило вида $A \to A\alpha$.

Определение:

Говорят, что КС-грамматика Γ содержит **левую рекурсию** (англ. *left recursion*), если в ней существует вывод вида $A \Rightarrow^* A\alpha$.

Методы нисходящего разбора не в состоянии работать с леворекурсивными грамматиками. Проблема в том, что продукция вида $A \Rightarrow^* A\alpha$ может применяться бесконечно долго, так и не выработав некий терминальный символ, который можно было бы сравнить со строкой. Поэтому требуется преобразование грамматики, которое бы устранило левую рекурсию.

Устранение непосредственной левой рекурсии

Опишем процедуру, устраняющую все правила вида $A \to A\alpha$, для фиксированного нетерминала A.

1. Запишем все правила вывода из A в виде:
   $A \to A\alpha_1 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$, где
   - α — непустая последовательность терминалов и нетерминалов ($\alpha \nrightarrow \varepsilon$);
   - β — непустая последовательность терминалов и нетерминалов, не начинающаяся с A.
2. Заменим правила вывода из A на
   $A \to \beta_1 A' \mid \ldots \ \mid \beta_m A' \mid \beta_1 \mid \ldots \mid \beta_m$.
3. Создадим новый нетерминал $A' \to \alpha_1 A' \mid \ldots \mid \alpha_n A' \mid \alpha_1 \mid \ldots \mid \alpha_n$.

Изначально нетерминал A порождает строки вида $\beta\alpha_{i0}\alpha_{i1} \ldots \alpha_{ik}$. В новой грамматике нетерминал A порождает $\beta A'$, а A' порождает строки вида $\alpha_{i0}\alpha_{i1} \ldots \alpha_{ik}$. Из этого очевидно, что изначальная грамматика эквивалентна новой.

Пример

$$A \to S\alpha \mid A\alpha$$

$$S \to A\beta$$

Есть непосредственная левая рекурсия $A \to A\alpha$. Добавим нетерминал A' и добавим правила $A \to S\alpha A'$, $A' \to \alpha A'$.

Новая грамматика:

$$A \to S\alpha A' \mid S\alpha$$

$$A' \to \alpha A' \mid \alpha$$

$$S \to A\beta$$

В новой грамматике нет непосредственной левой рекурсии, но нетерминал A леворекурсивен, так как есть $A \Rightarrow S\alpha A' \Rightarrow A\beta\alpha A'$

Алгоритм устранения произвольной левой рекурсии

Воспользуемся алгоритмом удаления ε-правил. Получим грамматику без ε-правил для языка $L(\Gamma) \setminus \{\varepsilon\}$.

Упорядочим нетерминалы, например по возрастанию индексов, и будем добиваться того, чтобы не было правил вида $A_i \to A_j\alpha$, где $j \leqslant i$. Если данное условие выполняется для всех A_i, то в грамматике нет $A_i \Rightarrow^* A_i$, а значит не будет левой рекурсии.

Пусть $N = \{A_1, A_2, \ldots, A_n\}$ — упорядоченное множество всех нетерминалов.

```
for A_i ∈ N
  for A_j ∈ {N | 1 ⩽ j < i}
    for p ∈ {P | A_i → A_j γ}
      удалить продукцию p
      for Q → x_i ∈ {A_j → δ_1 | ... | δ_k}
        добавить правило A_i → x_i γ
  устранить непосредственную левую рекурсию для A_i
```

Если ε присутствовал в языке исходной грамматики, добавим новый начальный символ S' и правила $S' \to S \mid \varepsilon$.

После i итерации внешнего цикла в любой продукции внешнего цикла в любой продукции вида $A_k \to A_l\alpha, k < i$, должно быть $l > k$. В результате при следующей итерации внутреннего цикла растет нижний предел m всех продукций вида $A_i \to A_m\alpha$ до тех пор, пока не будет достигнуто $i \leqslant m$.

После i итерации внешнего цикла в грамматике будут только правила вида $A_i \to A_j\alpha$, где $j > i$. Можно заметить, что неравенство становится строгим только после применения алгоритма устранения непосредственной левой рекурсии. При этом добавляются новые нетерминалы. Пусть ${A'}_i$ новый нетерминал. Можно заметить, что нет правила вида $\ldots \to {A'}_i$, где ${A'}_i$ самый левый нетерминал, а значит новые нетерминалы можно не рассматривать во внешнем цикле. Для строгого поддержания инвариантов цикла можно считать, что созданный на i итерации в процессе устранения непосредственной левой рекурсии нетерминал имеет номер A_{-i} (т.е. имеет номер, меньший всех имеющихся на данный момент нетерминалов).

На i итерации внешнего цикла все правила вида $A_i \rightarrow A_j\gamma$ где $j < i$ заменяются на $A_i \rightarrow \delta_1\gamma \mid \ldots \mid \delta_k\gamma$ где $A_j \rightarrow \delta_1 \mid \ldots \mid \delta_k$. Очевидно, что одна итерация алгоритма не меняет язык, а значит язык получившийся в итоге грамматики совпадает с исходным.

Оценка времени работы

Пусть a_i количество правил для нетерминала A_i. Тогда i итерация внешнего цикла будет выполняться за $O\left(\sum\limits_{A_i \rightarrow A_j, j<i} a_j\right) + O(a_i)$, что меньше чем $O\left(\sum\limits_{i=1}^{n} a_j\right)$, значит асимптотика алгоритма $O\left(n\sum\limits_{i=1}^{n} a_j\right)$.

Худший случай

Проблема этого алгоритма в том, что в зависимости от порядка нетерминалов в множестве размер грамматки может получиться экспоненциальным.

Пример грамматики для которой имеет значение порядок нетерминалов

$$A_1 \rightarrow 0 \mid 1$$

$$A_{i+1} \rightarrow A_i0 \mid A_i1 \text{ для } 1 \leqslant i < n$$

Упорядочим множество нетерминалов по возрастанию индексов. Легко заметить, что правила для A_i будут представлять из себя все двоичные вектора длины i, а значит размер грамматики будет экспоненциальным.

Порядок выбора нетерминалов

Определение:

Говорят, что нетерминал X — **прямой левый вывод** (англ. *direct left corner*) из A, если существует правило вида $A \rightarrow X\alpha$.

Определение:

Левый вывод (англ. *left corner*) — транзитивное, рефлексивное замыкание отношения «быть прямым левым выводом».

Во внутреннем цикле алгоритма для всех нетерминалов A_i и A_j, таких что $j < i$ и A_j — прямой левый вывод из A_i заменяем все прямые левые выводы A_j из A_i на все выводы из A_j.

Это действие удаляет левую рекурсию только если A_i — леворекурсивный нетерминал и A_j содержится в выводе A_i (то есть A_i — левый вывод из A_j,в то время как A_j — левый вывод из A_i).

Перестанем добавлять бесполезные выводы, которые не участвуют в удалении левой рекурсии, упорядочив нетерминалы так: если $j < i$ и A_j — прямой левый вывод из A_i, то A_i — левый вывод из A_j. Упорядочим их по уменьшению количества различных прямых левых выводов из них.

Так как отношение «быть левым выводом» транзитивно,то если C — прямой левый вывод из B, то каждый левый вывод из C также будет левым выводом из B. А так как отношение «быть левым выводом» рефлексивно, B явлеяется своим левым выводом, а значит если C — прямой левый вывод из B — он должен следовать за B в упорядоченном множестве, если только B не является левым выводом из C.

Пример

Дана грамматика:

$$A \to S\alpha$$

$$S \to S\beta \mid A\gamma \mid \beta$$

Среди правил A непосредственной рекурсии нет, поэтому во время первой итерации внешнего цикла ничего не происходит. Во время второй итерации внешнего цикла правило $S \to A\gamma$ переходит в $S \to S\alpha\gamma$.

Грамматика имеет вид

$$A \to S\alpha$$

$$S \to S\beta \mid S\alpha\gamma$$

Устраняем левую рекурсию для S

$$S \to \beta S_1 \mid \beta$$

$$S_1 \to \beta S_1 \mid \alpha\gamma S_1 \mid \beta \mid \alpha\gamma$$

См. также

- Контекстно-свободные грамматики
- Нормальная форма Хомского
- Удаления ε-правил из грамматики

Источники информации

- *Хопкрофт Д., Мотвани Р., Ульман Д.* — **Введение в теорию автоматов, языков и вычислений**, 2-е изд. : Пер. с англ. — Москва, Издательский дом «Вильямс», 2002. — 528 с. : ISBN 5-8459-0261-4 (рус.)
- *Robert C. Moore* — Removing Left Recursion from Context-Free Grammars (https://aclanthology.org/A00-2033.pdf)

Формальные грамматики

Содержание

Определения

Определение:

Формальная грамматика (англ. *Formal grammar*) — способ описания формального языка, представляющий собой четверку

$\Gamma = \langle \Sigma, N, S \in N, P \subset ((\Sigma \cup N)^* N (\Sigma \cup N)^*) \times (\Sigma \cup N)^* \rangle$, где:

- Σ — алфавит, элементы которого называют **терминалами** (англ. *terminals*);
- N — множество, элементы которого называют **нетерминалами** (англ. *nonterminals*);
- S — начальный символ грамматики (англ. *start symbol*);
- P — набор правил вывода (англ. *production rules* или *productions*) $\alpha \rightarrow \beta$.

Определение:

β **выводится из** α **за один шаг** $(\alpha \Rightarrow \beta)$:

1. $\alpha = \alpha_1\alpha_2\alpha_3$
2. $\beta = \beta_1\beta_2\beta_3$
3. $\alpha_1 = \beta_1, \alpha_3 = \beta_3, \alpha_2 \rightarrow \beta_2 \in P$.

Определение:

β **выводится из** α **за ноль или более шагов** $(\alpha \Rightarrow^* \beta)$: $\exists \gamma_1, \gamma_2, \ldots, \gamma_n : \alpha \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_n \Rightarrow \beta$ (Рефлексивно-транзитивное замыкание отношения $\Rightarrow$).

Определение:

Языком грамматики (англ. *Language of grammar*) называется $L(\Gamma) = \{\omega \in \Sigma^* \mid S \Rightarrow^* \omega\}$.

Определение:

Сентенциальная форма (англ. *Sentential form*) — последовательность терминалов и нетерминалов, выводимых из начального символа.

Обозначения

- Нетерминалы обозначаются заглавными буквами латинского алфавита (например: A, B, C).
- Терминалы обозначаются строчными буквами из начала латинского алфавита (например: a, b, c).
- Последовательности из терминалов (слова) обозначают строчными буквами из конца латинского или греческого алфавита (например: ω).
- Последовательности из терминалов и нетерминалов обозначаются строчными буквами из начала греческого алфавита (например: β, α).

Примеры грамматик

Правильные скобочные последовательности

$\Sigma = \{(,)\}$

$S \to (S)$

$S \to SS$

$S \to \varepsilon$

Вывод строки $(()())$:
$S \Rightarrow (\boldsymbol{S}) \Rightarrow (\boldsymbol{S}S) \Rightarrow ((S)\boldsymbol{S}) \Rightarrow ((\boldsymbol{S})(S)) \Rightarrow (()(\boldsymbol{S})) \Rightarrow (()())$.

Вывод строки $((()())(()))$:
$S \Rightarrow (\boldsymbol{S}) \Rightarrow (\boldsymbol{S}S) \to ((S)\boldsymbol{S}) \to ((\boldsymbol{S})(S)) \to\to ((\boldsymbol{S}S)((S))) \to (((\boldsymbol{S})S)((S))) \to ((()\boldsymbol{S})((S))) \to$
$\to ((()(\boldsymbol{S}))((S))) \to ((()())((\boldsymbol{S}))) \to ((()())(()))$.

Арифметические выражения

$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, *, /, -, (,)\}$

$S \to SOS$

$S \to (S)$

$S \to 0$

$S \to DN$

$O \to + \mid - \mid * \mid /$

$D \to 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

$N \to NN \mid \varepsilon$

$N \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9.$

Вывод строки $2 + 2 * 2$:
$S \Rightarrow SO\boldsymbol{S} \Rightarrow \boldsymbol{S}OSOS \Rightarrow 2O\boldsymbol{S}OS \Rightarrow 2O2O\boldsymbol{S} \Rightarrow 2\boldsymbol{O}2O2 \Rightarrow 2 + 2\boldsymbol{O}2 \Rightarrow 2 + 2 * 2$.

Левосторонний вывод этой же строки:
$S \Rightarrow \boldsymbol{S}OS \Rightarrow 2\boldsymbol{O}S \Rightarrow 2 + \boldsymbol{S} \Rightarrow 2 + \boldsymbol{S}OS \Rightarrow 2 + 2\boldsymbol{O}S \Rightarrow 2 + 2 * \boldsymbol{S} \Rightarrow 2 + 2 * 2$.

Язык $0^n 1^n 2^n$

Данный язык является контекстно-зависимым. КЗ-грамматика для языка приведена ниже, а через лемму о разрастании доказывается его неконтекстно-свободность.

$\Sigma = \{0, 1, 2\}$

$S \to 012$
$S \to 0TS2$
$T0 \to 0T$
$T1 \to 11$

Вывод строки 000111222 :

$$S \Rightarrow 0T\boldsymbol{S}2 \Rightarrow 0T0T\boldsymbol{S}22 \Rightarrow 0T0\boldsymbol{T0}1222 \Rightarrow 0\boldsymbol{T0}0T1222 \Rightarrow 00\boldsymbol{T0}T1222 \Rightarrow 000T\boldsymbol{T1}222 \Rightarrow 000\boldsymbol{T}1$$

Данная грамматика описывает этот язык, так как мы можем вывести любую строку одним методом. $n-1$ раз выполняем правило вывода $S \to 0TS2$. Потом выполняем правило $S \to 012, n-1$ раз выполняем $T0 \to 0T$. После этого у нас получается строка $0^nT^{n-1}2^n$. Выполняем $n-1$ раз последнее правило и в результате получаем искомую строку.

См. также

- Возможность порождения формальной грамматикой произвольного перечислимого языка
- Иерархия Хомского формальных грамматик
- Неукорачивающие и контекстно-зависимые грамматики, эквивалентность
- Правоконтекстные грамматики, эквивалентность автоматам

Источники информации

- Wikipedia — Formal grammar
- Wikipedia — Formal language
- *Хопкрофт Д., Мотвани Р., Ульман Д.* — Введение в теорию автоматов, языков и вычислений, 2-е изд. : Пер. с англ. — Москва, Издательский дом «Вильямс», 2002. — 528 с. : ISBN 5-8459-0261-4 (рус.)

Нормальная форма Хомского

Определение:

Грамматикой в **нормальной форме Хомского** (англ. *Chomsky normal form*) называется контекстно-свободная грамматика, в которой могут содержаться правила только следующего вида:

$$A \to BC,$$

$$A \to a,$$

$$S \to \varepsilon,$$

где a — терминал, A, B, C — нетерминалы, S — стартовая вершина, ε — пустая строка, стартовая вершина не содержится в правых частях правил.

Содержание

Приведение грамматики к нормальной форме Хомского

Теорема:

Любую контекстно-свободную грамматику можно привести к нормальной форме Хомского.

Доказательство:

$\triangleright$

Рассмотрим контекстно-свободную грамматику Γ. Для приведения ее к нормальной форме Хомского необходимо выполнить пять шагов. На каждом шаге мы строим новую Γ_i, которая допускает тот же язык, что и Γ.

1. Уберём длинные правила.

   Воспользуемся алгоритмом удаления длинных правил из грамматики. Получим грамматику Γ_1, эквивалентную исходной, содержащую правила длины $0, 1$ и 2.

2. Удаление ε-правил.

   Воспользуемся алгоритмом удаления ε-правил из грамматики. Получим грамматику Γ_2, эквивалентную исходной, но в которой нет ε-правил.

3. Удаление цепных правил.

   Воспользуемся алгоритмом удаления цепных правил из грамматики. Алгоритм работает таким образом, что новые ε-правила не образуются. Получим грамматику Γ_3, эквивалентную Γ.

4. Удалим бесполезные символы.

   Воспользуемся алгоритмом удаления бесполезных символов из грамматики. Так как Γ_3 эквивалентна Γ, то бесполезные символы не могли перестать быть бесполезными. Более того, мы только удаляем правила, новые ε-правила и цепные правила не могли появиться.

5. Уберём ситуации, когда в правиле встречаются несколько терминалов.

   Для всех правил вида $A \to u_1 u_2$ (где u_i — терминал или нетерминал) заменим все терминалы u_i на новые нетерминалы U_i и добавим правила $U_i \to u_i$. Теперь правила содержат либо одиночный терминал, либо строку из двух нетерминалов.

Таким образом, мы получили грамматику в нормальной форме Хомского, которая допускает тот же язык, что и Γ.

Стоит заметить, что порядок выполнения операций важен. Первое правило должно быть выполнено перед вторым, иначе время нормализации ухудшится до $O(2^{|\Gamma|})$. Третье правило идет после второго, потому что после удаления ε-правил, могут образоваться новые цепные правила. Также четвертое правило должно быть выполнено позже третьего и второго, так как они могут порождать бесполезные символы.

При таком порядке действий размеры грамматики возрастают полиномиально.

После удалении длинных правил из каждого правила длины $k \geqslant 3$ могло появиться $k - 1$ новых правил, причем их длина не превышает двух. На этом шаге размер грамматики возрастает не более, чем вдвое.

При удалении ε-правил из грамматики, содержащей правила длины $0, 1$ и 2, размеры грамматики могли вырасти не больше, чем в 3 раза.

Всего цепных правил в грамматике не больше, чем n^2, где n — число нетерминалов. При удалении цепных правил мы берем каждую из цепных пар и производим добавление нецепных правил, выводимых из второго нетерминала в паре. Если максимальная суммарная длина всех правил, выводимых из какого-либо нетерминала, равна k, то размер грамматики возрастет не больше, чем на $k \cdot n^2$.

Наконец, на последнем шаге может произойти добавление не более, чем $|\Sigma|$ (Σ — алфавит грамматики) новых правил, причем все они будут длины 1.
$\triangleleft$

Пример

Текущий шаг	Грамматика после применения правила
0. Исходная грамматика	$S \to aXbX \vert aZ$ $X \to aY \vert bY \vert \varepsilon$ $Y \to X \vert cc$ $Z \to ZX$
1. Удаление длинных правил	$S \to aS_1 \vert aZ$ $X \to aY \vert bY \vert \varepsilon$ $Y \to X \vert cc$ $Z \to ZX$ $S_1 \to XS_2$ $S_2 \to yX$
2. Удаление ε-правил	$S \to aS_1 \vert aZ$ $X \to aY \vert bY$ $Y \to aY \vert bY \vert cc$ $Z \to ZX$ $S_1 \to XS_2 \vert S_2$ $S_2 \to yX \vert y$
3. Удаление цепных правил	$S \to aS_1 \vert aZ$ $X \to aY \vert bY$ $Y \to aY \vert bY \vert cc$ $Z \to ZX$ $S_1 \to XS_2 \vert yX \vert y$ $S_2 \to yX \vert y$
4. Удаление бесполезных символов	$S \to aS_1$ $X \to aY \vert bY$ $Y \to aY \vert bY \vert cc$ $S_1 \to XS_2 \vert yX \vert y$ $S_2 \to yX \vert y$
5. Уберём ситуации, когда в правиле встречаются несколько терминалов.	$S \to S_3S_1$ $X \to S_3Y \vert X_1Y$ $Y \to S_3Y \vert X_1Y \vert Y_1Y_1$ $S_1 \to XS_2 \vert S_4X \vert y$ $S_2 \to S_4X \vert y$ $S_3 \to a$ $S_4 \to y$ $X_1 \to b$ $Y_1 \to c$

См. также

- Контекстно-свободные грамматики
- Нормальная форма Куроды
- Приведение грамматики к ослабленной нормальной форме Грейбах

Источники информации

- Wikipedia — Chomsky normal form

- *Хопкрофт Д., Мотвани Р., Ульман Д.* — **Введение в теорию автоматов, языков и вычислений**, 2-е изд. : Пер. с англ. — Москва, Издательский дом «Вильямс», 2002. — 528с. : ISBN 5-8459-0261-4 (рус.)

Удаление eps-правил из грамматики

Содержание

Используемые определения

Определение:

Правила вида $A \to \varepsilon$ называются ε**-правилами** (англ. ε-*rule*).

Определение:

Нетерминал A называется ε**-порождающим** (англ. ε-*generating*), если $A \Rightarrow^* \varepsilon$.

Алгоритм поиска ε-порождающих нетерминалов

Вход: КС-грамматика $\Gamma = \langle N, \Sigma, P, S \rangle$.
Выход: множество ε-порождающих нетерминалов.

1. Найти все ε-правила. Составить множество, состоящее из нетерминалов, входящих в левые части таких правил.
2. Перебираем правила грамматики Γ. Если найдено правило $A \to C_1C_2 \ldots C_k$, для которого верно, что каждый C_i принадлежит множеству, то добавить A в множество.
3. Если на шаге 2 множество изменилось, то повторить шаг 2.

Доказательство корректности

Теорема:

Описанный выше алгоритм находит все ε-порождающие нетерминалы грамматики Γ.

Доказательство:

$\triangleright$

Для доказательства корректности алгоритма достаточно показать, что, если множество ε-порождающих нетерминалов на очередной итерации алгоритма не изменялось, то алгоритм нашел все ε-порождающие нетерминалы.

Пусть после завершения алгоритма существуют нетерминалы такие, что они являются ε-порождающими, но не были найдены алгоритмом. Выберем из этих нетерминалов нетерминал B, из которого выводится ε за наименьшее число шагов. Тогда в грамматике есть правило $B \to C_1C_2 \ldots C_k$, где каждый нетерминал C_i — ε-порождающий. Каждый C_i входит в множество ε -порождающих нетерминалов, так как иначе вместо B необходимо было взять C_i. Следовательно, на одной из итераций алгоритма B уже добавился в множество ε -порождающих нетерминалов. Противоречие. Следовательно, алгоритм находит все ε-порождающие нетерминалы.

$\triangleleft$

Модификация с очередью

Заведем несколько структур:

- $\mathtt{isEpsilon[nonterm_i]}$ — для каждого нетерминала будем хранить пометку, является он ε-порождающим или нет.
- $\mathtt{concernedRules[nonterm_i]}$ — для каждого нетерминала будем хранить список номеров тех правил, в правой части которых он встречается;
- $\mathtt{counter[rule_i]}$ — для каждого правила будем хранить счетчик количества нетерминалов в правой части, которые еще не помечены ε -порождающими;

- `Q` — очередь нетерминалов, помеченных ε-порождающими, но еще не обработанных.

Сначала проставим `false` в `isEpsilon` для всех нетерминалов, а в `counter` для каждого правила запишем количество нетерминалов справа от него. Те правила, для которых `counter` сразу же оказался нулевым, добавим в `Q` и объявим истинным соответствующий `isEpsilon`, так как это ε-правила. Теперь будем доставать из очереди по одному нетерминалу, смотреть на список `concernedRules` для него и уменьшать `counter` для всех правил оттуда. Если `counter` какого-то правила в этот момент обнулился, то нетерминал из левой части этого правила помечается ε-порождающим, если еще не был помечен до этого, и добавляется в `Q`. Продолжаем, пока очередь не станет пустой.

Время работы алгоритма

Базовый алгоритм работает за $O(|\Gamma|^2)$. В алгоритме с модификацией нетерминал попадает в очередь ровно один раз, соответственно ровно один раз мы пройдемся по списку правил, в правой части которых он лежит. Суммарно получается $O(|\Gamma|)$.

Пример

Рассмотрим грамматику, причем сразу пронумеруем правила:

1. $S \to ABC$
2. $S \to DS$
3. $A \to \varepsilon$
4. $B \to AC$
5. $C \to \varepsilon$
6. $D \to d$

Поскольку правило 6 содержит справа терминалы, оно заведомо не будет влиять на ответ, поэтому мы не будем его учитывать.

Построим массив списков `concernedRules`.

`concernedRules`				
S	A	B	C	D
2	$1, 4$	1	$1, 4$	2

Q	isEpsilon					counter					Комментарий
$\{\}$	S	A	B	C	D	**1**	**2**	**3**	**4**	**5**	Зададим начальные значения массивам `counter` и `isEpsilon`.
	0	0	0	0	0	3	2	0	2	0	
$\{A, C\}$	S	A	B	C	D	**1**	**2**	**3**	**4**	**5**	Заметим, что правила 3 и 5 являются ε-правилами. Пометим левые нетерминалы из этих правил и добавим их в очередь. После этого в `Q` лежит A и C, а `counter` остался без изменений.
	0	1	0	1	0	3	2	0	2	0	
$\{C\}$	S	A	B	C	D	**1**	**2**	**3**	**4**	**5**	Достанем из очереди A, декрементируем те счетчики, которые относятся к связанным с ним правилам. К очереди ничего не добавится.
	0	1	0	1	0	2	2	0	1	0	
$\{B\}$	S	A	B	C	D	**1**	**2**	**3**	**4**	**5**	Достанем из очереди C. После проведения действий из алгоритма в очередь добавится B.
	0	1	1	1	0	1	2	0	0	0	
$\{S\}$	S	A	B	C	D	**1**	**2**	**3**	**4**	**5**	Достанем из очереди B. После действий алгоритма в очередь добавится S.
	1	1	1	1	0	0	2	0	0	0	
$\{\}$	S	A	B	C	D	**1**	**2**	**3**	**4**	**5**	Достанем из очереди S. Ничего не добавится в очередь и она останется пустой. Алгоритм закончил свое выполнение. Итого в множество ε-правил входят все нетерминалы, кроме D.
	1	1	1	1	0	0	1	0	0	0	

Если применять алгоритм без модификации с очередью, то действия будут следующие:

1. Возьмём множество состоящее из ε-порождающих нетерминалов $\{A, C\}$.
2. Добавим B в множество, так как правая часть правила $B \to AC$ состоит только из нетерминалов из множества.
3. Повторим второй пункт для правила $S \to ABC$ и получим множество $\{A, B, C, S\}$.
4. Больше нет нерассмотренных правил, содержащих справа только нетерминалы из множества.

Таким образом ε-порождающими нетерминалами являются A, B, C и S.

Алгоритм удаления ε-правил из грамматики

Вход: КС-грамматика $\Gamma = \langle N, \Sigma, P, S \rangle$.
Выход: КС-грамматика $\Gamma' = \langle N, \Sigma, P', S' \rangle$ без ε-правил (может присутствовать правило $S \to \varepsilon$, но в этом случае S не встречается в правых частях правил); $L(\Gamma') = L(\Gamma)$.

1. Добавить все правила из P в P'.
2. Найти все ε-порождаюшие нетерминалы.
3. Для каждого правила вида $A \to \alpha_0 B_1 \alpha_1 B_2 \alpha_2 \dots B_k \alpha_k$ (где α_i — последовательности из терминалов и нетерминалов, B_j — ε-порождающие

нетерминалы) добавить в P' все возможные варианты правил, в которых либо присутствует, либо удалён каждый из нетерминалов B_j $(1 \leqslant j \leqslant k)$.

4. Удалить все ε-правила из P'.
5. Если в исходной грамматике Γ выводилось ε, то необходимо добавить новый нетерминал S', сделать его стартовым, добавить правило $S' \to S|\varepsilon$.

Доказательство корректности

Теорема:

> Если грамматика Γ' была построена с помощью описанного выше алгоритма по грамматике Γ, то $L(\Gamma') = L(\Gamma)$.

Доказательство:

$\triangleright$

Сначала докажем, что, если не выполнять шаг 5 алгоритма, то получится грамматика $\Gamma' : L(\Gamma') = L(\Gamma) \setminus \{\varepsilon\}$.

Для этого достаточно доказать, что $A \underset{\Gamma'}{\Rightarrow}^* w$ тогда и только тогда, когда $A \underset{\Gamma}{\Rightarrow}^* w$ и $w \neq \varepsilon$ (*).

$\Rightarrow$ Пусть $A \underset{\Gamma'}{\Rightarrow}^* w$ и $w \neq \varepsilon$.

Докажем индукцией по длине порождения в грамматике Γ', что $A \underset{\Gamma}{\Rightarrow}^* w$.

База. $A \underset{\Gamma'}{\Rightarrow} w$.

В этом случае в Γ' есть правило $A \to w$. По построению Γ' в Γ есть правило $A \to \alpha$, причем α — цепочка w, элементы которой, возможно, перемежаются ε -порождающими нетерминалами. Тогда в Γ есть порождения $A \underset{\Gamma}{\Rightarrow} \alpha \underset{\Gamma}{\Rightarrow}^* w$.

Предположение индукции. Пусть из $A \underset{\Gamma'}{\Rightarrow}^* w \neq \varepsilon$ менее, чем за n шагов, следует, что $A \underset{\Gamma}{\Rightarrow}^* w$.

Переход. Пусть в порождении n шагов, $n > 1$. Тогда оно имеет вид $A \underset{\Gamma'}{\Rightarrow} X_1 X_2 \ldots X_k \underset{\Gamma'}{\Rightarrow}^* w$, где $X_i \in N \cup \Sigma$. Первое использованное правило должно быть построено по правилу грамматики Γ $A \to Y_1 Y_2 \ldots Y_m$, где последовательность $Y_1 Y_2 \ldots Y_m$ совпадает с последовательностью $X_1 X_2 \ldots X_k$, символы которой, возможно, перемежаются ε-порождающими нетерминалами.

Цепочку w можно разбить на $w_1 w_2 \dots w_k$, где $X_i \underset{\Gamma'}{\Rightarrow^*} w_i$. Если X_i — терминал, то $w_i = X_i$, а если нетерминал, то порождение $X_i \underset{\Gamma'}{\Rightarrow^*} w_i$ содержит менее n шагов. По предположению $X_i \underset{\Gamma}{\Rightarrow^*} w_i$, значит

$$A \underset{\Gamma}{\Rightarrow} Y_1 Y_2 \dots Y_m \underset{\Gamma}{\Rightarrow^*} X_1 X_2 \dots X_k \underset{\Gamma}{\Rightarrow^*} w_1 w_2 \dots w_k = w.$$

$\Leftarrow$

Пусть $A \underset{\Gamma}{\Rightarrow^*} w$ и $w \neq \varepsilon$.

Докажем индукцией по длине порождения в грамматике Γ, что $A \underset{\Gamma'}{\Rightarrow^*} w$.

База. $A \underset{\Gamma}{\Rightarrow} w$.

Правило $A \to w$ присутствует в Γ. Поскольку $w \neq \varepsilon$, это же правило будет и в Γ', поэтому $A \underset{\Gamma'}{\Rightarrow^*} w$.

Предположение индукции. Пусть из $A \underset{\Gamma}{\Rightarrow^*} w \neq \varepsilon$ менее, чем за n шагов, следует, что $A \underset{\Gamma'}{\Rightarrow^*} w$.

Переход. Пусть в порождении n шагов, $n > 1$. Тогда оно имеет вид $A \underset{\Gamma}{\Rightarrow} Y_1 Y_2 \dots Y_m \underset{\Gamma}{\Rightarrow^*} w$, где $Y_i \in N \cup \Sigma$. Цепочку w можно разбить на $w_1 w_2 \dots w_m$, где $Y_i \underset{\Gamma}{\Rightarrow^*} w_i$.

Пусть $Y_{i_1}, Y_{i_2}, \dots, Y_{i_p}$ — подпоследовательность, состоящая из всех элементов, таких, что $w_{i_k} \neq \varepsilon$, то есть $Y_{i_1} Y_{i_2} \dots Y_{i_p} \underset{\Gamma}{\Rightarrow^*} w$. $p \geqslant 1$, поскольку $w \neq \varepsilon$. Значит, $A \to Y_{i_1} Y_{i_2} \dots Y_{i_p}$ является правилом в Γ' по построению Γ'.

Так как каждое из порождений $Y_i \underset{\Gamma}{\Rightarrow^*} w_i$ содержит менее n шагов, к ним можно применить предположение индукции и заключить, что, если $w_i \neq \varepsilon$, то $Y_i \underset{\Gamma'}{\Rightarrow^*} w_i$.

Таким образом, $A \underset{\Gamma'}{\Rightarrow} Y_{i_1} Y_{i_2} \dots Y_{i_p} \underset{\Gamma'}{\Rightarrow^*} w$.

Подставив S вместо A в утверждение (*), видим, что $w \in L(\Gamma)$ для $w \neq \varepsilon$ тогда и только тогда, когда $w \in L(\Gamma')$. Так как после выполнения шага 5 алгоритма в Γ' могло добавиться только пустое слово ε, то язык, задаваемый КС-грамматикой Γ', совпадает с языком, задаваемым КС-грамматикой Γ.

$\triangleleft$

Время работы алгоритма

Рассмотрим грамматику Γ:

$$
\begin{array}{l}
S \to T_1 T_2 T_3 \ldots T_n \\
T_1 \to t_1 | \varepsilon \\
T_2 \to t_2 | \varepsilon \\
T_n \to t_n | \varepsilon
\end{array}
$$

$|\Gamma| = O(n)$. Из нетерминала S можно вывести 2^n сочетаний нетерминалов T_i. Таким образом в худшем случае алгоритм работает за $O(2^{|\Gamma|})$.
Рассмотрим теперь грамматику с устраненными длинными правилами. После применения данного алгоритма, который работает за $O(|\Gamma|)$, в грамматике станет на $O(|\Gamma|)$ больше правил, но при этом все они будут размером $O(1)$. Итого по-прежнему $|\Gamma| = O(n)$. Однако алгоритм удаления ε-правил будет работать за $O(|\Gamma|)$, поскольку для каждого правила можно будет добавить только $O(1)$ сочетаний нетерминалов.

Пример

Рассмотрим грамматику:

$$
\begin{array}{l}
S \to ABCd \\
A \to a | \varepsilon \\
B \to AC \\
C \to c | \varepsilon
\end{array}
$$

В ней A, B и C являются ε-порождающими нетерминалами.

1. Переберём для каждого правила все возможные сочетания ε-порождающих нетерминалов и добавим новые правила:
   - $S \to Ad | ABd | ACd | Bd | BCd | Cd | d$ для $S \to ABCd$
   - $B \to A | C$ для $B \to AC$
2. Удалим праила $A \to \varepsilon$ и $C \to \varepsilon$

В результате мы получим новую грамматику без ε-правил:

$$
\begin{array}{l}
S \to Ad | ABd | ACd | ABCd | Bd | BCd | Cd | d \\
A \to a \\
B \to A | AC | C
\end{array}
$$

$$C \to c$$

См. также

- Контекстно-свободные грамматики
- Нормальная форма Хомского

Источники информации

- *Хопкрофт Д., Мотвани Р., Ульман Д.* **Введение в теорию автоматов, языков и вычислений**, 2-е изд. : Пер. с англ. — Москва, Издательский дом «Вильямс», 2002. — С. 273: ISBN 5-8459-0261-4 (рус.)
- Wikipedia — Chomsky normal form (http://en.wikipedia.org/wiki/Chomsky_normal_form)

Реализации алгоритмов/Метод рекурсивного спуска

< Реализации алгоритмов

Метод рекурсивного спуска — алгоритм нисходящего синтаксического анализа, реализуемый путём взаимного вызова процедур парсинга, где каждая процедура соответствует одному из правил контекстно-свободной грамматики или БНФ. Применения правил последовательно, слева-направо поглощают токены, полученные от лексического анализатора. Это один из самых простых алгоритмов парсинга, подходящий для полностью ручной реализации.

C

```c
typedef enum {ident, number, lparen, rparen, times, slash, plus,
    minus, eql, neq, lss, leq, gtr, geq, callsym, beginsym, semicolon,
    endsym, ifsym, whilesym, becomes, thensym, dosym, constsym, comma,
    varsym, procsym, period, oddsym} Symbol;

Symbol sym;
void getsym(void);
void error(const char msg[]);
void expression(void);

int accept(Symbol s) {
    if (sym == s) {
        getsym();
        return 1;
    }
    return 0;
}

int expect(Symbol s) {
    if (accept(s))
        return 1;
    error("expect: unexpected symbol");
    return 0;
}

void factor(void) {
    if (accept(ident)) {
        ;
    } else if (accept(number)) {
        ;
    } else if (accept(lparen)) {
        expression();
        expect(rparen);
    } else {
        error("factor: syntax error");
        getsym();
    }
}

void term(void) {
    factor();
    while (sym == times || sym == slash) {
        getsym();
        factor();
    }
}
```

```c
void expression(void) {
    if (sym == plus || sym == minus)
        getsym();
    term();
    while (sym == plus || sym == minus) {
        getsym();
        term();
    }
}

void condition(void) {
    if (accept(oddsym)) {
        expression();
    } else {
        expression();
        if (sym == eql || sym == neq || sym == lss || sym == leq || sym == gtr || sym == geq) {
            getsym();
            expression();
        } else {
            error("condition: invalid operator");
            getsym();
        }
    }
}

void statement(void) {
    if (accept(ident)) {
        expect(becomes);
        expression();
    } else if (accept(callsym)) {
        expect(ident);
    } else if (accept(beginsym)) {
        do {
            statement();
        } while (accept(semicolon));
        expect(endsym);
    } else if (accept(ifsym)) {
        condition();
        expect(thensym);
        statement();
    } else if (accept(whilesym)) {
        condition();
        expect(dosym);
        statement();
    } else {
        error("statement: syntax error");
        getsym();
    }
}

void block(void) {
    if (accept(constsym)) {
        do {
            expect(ident);
            expect(eql);
            expect(number);
        } while (accept(comma));
        expect(semicolon);
    }
    if (accept(varsym)) {
        do {
            expect(ident);
        } while (accept(comma));
        expect(semicolon);
    }
    while (accept(procsym)) {
        expect(ident);
        expect(semicolon);
        block();
        expect(semicolon);
    }
    statement();
}

void program(void) {
```

```
    getsym();
    block();
    expect(period);
}
```

Парсер на примере программы "калькулятор" (реализация на C++)

```
/*
 *  Калькулятор. Пример из учебника "C++ Programming Language" (основа).
 *  Применяется алгоритм "рекурсивный спуск" (recursive descent).
 *  (Примечание: калькулятор может работать с выражениями. Например, если
 *  на входе подать r = 2.5; area = pi * r * r; то на выходе будет
 *  2.5; 19.635)
 */

#include <cctype>
#include <iostream>
#include <map>
#include <sstream>
#include <string>

enum Token_value : char {
  NAME,               NUMBER,                 END,
  PLUS='+',           MINUS='-',              MUL='*',         DIV='/',
  PRINT=';',          ASSIGN='=',             LP='(',          RP=')'
};

enum Number_value : char {
  NUM0='0', NUM1='1', NUM2='2',
  NUM3='3', NUM4='4', NUM5='5',
  NUM6='6', NUM7='7', NUM8='8',
  NUM9='9', NUMP='.',
};

Token_value curr_tok = PRINT;        // Хранит последний возврат функции get_token().
double number_value;                 // Хранит целый литерал или литерал с плавающей запятой.
std::string string_value;            // Хранит имя.
std::map<std::string, double> table; // Таблица имён.
int no_of_errors;                    // Хранит количество встречаемых ошибок.

double expr(std::istream*, bool);    // Обязательное объявление.

/****************************************************************************/

// Функция error() имеет тривиальный характер: инкрементирует счётчик ошибок.
double error(const std::string& error_message) {
  ++no_of_errors;
  std::cerr << "error: " << error_message << std::endl;
  return 1;
}

Token_value get_token(std::istream* input) {
  char ch;

  do {    // Пропустить все пробельные символы кроме '\n'.
    if (!input->get(ch)) {
      return curr_tok = END; // Завершить работу калькулятора.
    }
  } while (ch != '\n' && isspace(ch));

  switch (ch) {
    case 0: // При вводе символа конца файла, завершить работу калькулятора.
      return curr_tok = END;
    case PRINT:
    case '\n':
      return curr_tok = PRINT;
    case MUL:
```

```
    case DIV:
    case PLUS:
    case MINUS:
    case LP:
    case RP:
    case ASSIGN:
      return curr_tok = Token_value(ch); // Приведение к целому и возврат.
    case NUM0: case NUM1: case NUM2: case NUM3: case NUM4:
    case NUM5: case NUM6: case NUM7: case NUM8: case NUM9:
    case NUMP:
      input->putback(ch); // Положить назад в поток...
      *input >> number_value; // И считать всю лексему.
      return curr_tok = NUMBER;
    default:
      if (isalpha(ch)) {
        string_value = ch;
        while (input->get(ch) && isalnum(ch)) {
          string_value.push_back(ch);
        }
        input->putback(ch);
        return curr_tok = NAME;
      }
      error("Bad Token");
      return curr_tok = PRINT;
  }
}

/* Каждая функция синтаксического анализа принимает аргумент типа bool
 * указывающий, должна ли функция вызывать get_token() для получения
 * очередной лексемы. */

// prim() - обрабатывает первичные выражения.
double prim(std::istream* input, bool get) {
  if (get) {
    get_token(input);
  }

  switch (curr_tok) {
    case NUMBER: {
      double v = number_value;
      get_token(input);
      return v;
    }
    case NAME: {
      double& v = table[string_value];
      if (get_token(input) == ASSIGN) {
          v = expr(input, true);
      }
      return v;
    }
    case MINUS:
      return -prim(input, true);
    case LP: {
      double e = expr(input, true);
      if (curr_tok != RP) {
          return error("')' expected");
      }
      get_token(input);
      return e;
    }
    default:
      return error("primary expected");
  }
}

// term() - умножение и деление.
double term(std::istream* input, bool get) {
  double left = prim(input, get);

  for ( ; ; ) {
    switch (curr_tok) {
      case MUL:
        left *= prim(input, true);
        break;
      case DIV:
```

```
        if (double d = prim(input, true)) {
            left /= d;
            break;
        }
        return error("Divide by 0");
      default:
          return left;
    }
  }
}

// expr() - сложение и вычитание.
double expr(std::istream* input, bool get) {
  double left = term(input, get);

  for ( ; ; ) {
    switch (curr_tok) {
      case PLUS:
        left += term(input, true);
        break;
      case MINUS:
        left -= term(input, true);
        break;
      default:
        return left;
    }
  }
}

int main(int argc, char* argv[]) {
  std::istream* input = nullptr; // Указатель на поток.

  switch (argc) {
    case 1:
      input = &std::cin;
      break;
    case 2:
      input = new std::istringstream(argv[1]);
      break;
    default:
      error("Too many arguments");
      return 1;
  }

  table["pi"] = 3.1415926535897932385;
  table["e"] = 2.7182818284590452354;

  while (*input) {
    get_token(input);
    if (curr_tok == END) {
      break;
    }

    // Снимает ответственность expr() за обработку пустых выражений.
    if (curr_tok == PRINT) {
      continue;
    }

    // expr() -> term() -> prim() -> expr() ...
    std::cout << expr(input, false) << std::endl;
  }

  if (input != &std::cin) {
    delete input;
  }

  return no_of_errors;
}
```

Источник — https://ru.wikibooks.org/w/index.php?title=Реализации_алгоритмов/Метод_рекурсивного_спуска&oldid=163649

ВИКИУЧЕБНИК
Открытые книги
для открытого мира

Реализации алгоритмов/Алгоритм сортировочной станции

< Реализации алгоритмов

Алгоритм сортировочной станции — способ разбора математических выражений, представленных в инфиксной нотации. Может быть использован для получения вывода в виде обратной польской нотации или в виде абстрактного синтаксического дерева. Алгоритм изобретен Эдсгером Дейкстрой и назван им «алгоритм сортировочной станции», поскольку напоминает действие железнодорожной сортировочной станции.

Си

```
#include <string.h>
#include <stdio.h>
#include <stdbool.h>

// Операторы
// Приоритет Оператор Ассоциативность
// 4 ! правая
// 3 * / % левая
// 2 + - левая
// 1 = левая
int op_preced(const char c)
{
    switch(c)
    {
        case '!':
        return 4;

        case '*':
        case '/':
        case '%':
        return 3;

        case '+':
        case '-':
        return 2;

        case '=':
        return 1;
    }
    return 0;
}

bool op_left_assoc(const char c)
{
    switch(c)
    {
        // лево-ассоциативные операторы
        case '*':
        case '/':
        case '%':
        case '+':
        case '-':
        case '=':
        return true;
        // право-ассоциативные операторы
        case '!':
        return false;
```

```
    }
    return false;
}

unsigned int op_arg_count(const char c)
{
    switch(c)
    {
        case '*':
        case '/':
        case '%':
        case '+':
        case '-':
        case '=':
        return 2;
        case '!':
        return 1;

    default:
    return c - 'A';
    }
    return 0;
}

#define is_operator(c) (c == '+' || c == '-' || c == '/' || c == '*' || c == '!' || c == '%' || c ==
'=')
#define is_function(c) (c >= 'A' && c <= 'Z')
#define is_ident(c) ((c >= '0' && c <= '9') || (c >= 'a' && c <= 'z'))

bool shunting_yard(const char *input, char *output)
{
    const char *strpos = input, *strend = input + strlen(input);
    char c, stack[32], sc, *outpos = output;
    unsigned int sl = 0;
    while(strpos < strend)
    {
        c = *strpos;
        if(c != ' ')
        {
            // Если токен является числом (идентификатором), то добавить его в очередь вывода.
            if(is_ident(c))
            {
                *outpos = c; ++outpos;
            }
            // Если токен - функция, то положить его в стек.
            else if(is_function(c))
            {
                stack[sl] = c;
                ++sl;
            }
            //Если токен - разделитель аргументов функции (запятая):
            else if(c == ',')
            {
                bool pe = false;
                while(sl > 0)
                {
                    sc = stack[sl - 1];
                    if(sc == '(')
                    {
                        pe = true;
                        break;
                    }
                    else
                    {
                        // Пока на вершине не левая круглая скобка,
                        // перекладывать операторы из стека в очередь вывода.
                        *outpos = sc; ++outpos;
                        sl--;
                    }
                }
                // Если не была достигнута левая круглая скобка, либо разделитель не в том месте
                // либо была пропущена скобка
                if(!pe)
                {
                    printf("Error: separator or parentheses mismatched\n");
```

```
                return false;
            }
        }
        // Если токен оператор op1, то:
        else if(is_operator(c))
        {
            while(sl > 0)
            {
                sc = stack[sl - 1];
                // Пока на вершине стека присутствует токен оператор op2,
                // а также оператор op1 лево-ассоциативный и его приоритет меньше или такой же чем
у оператора op2,
                // или оператор op1 право-ассоциативный и его приоритет меньше чем у оператора op2
                if(is_operator(sc) &&
                    ((op_left_assoc(c) && (op_preced(c) <= op_preced(sc))) ||
                       (!op_left_assoc(c) && (op_preced(c) < op_preced(sc)))))
                {
                    // Переложить оператор op2 из стека в очередь вывода.
                    *outpos = sc; ++outpos;
                    sl--;
                }
                else
                {
                    break;
                }
            }
            // положить в стек оператор op1
            stack[sl] = c;
            ++sl;
        }
        // Если токен - левая круглая скобка, то положить его в стек.
        else if(c == '(')
        {
            stack[sl] = c;
            ++sl;
        }
        // Если токен - правая круглая скобка:
        else if(c == ')')
        {
            bool pe = false;
            // До появления на вершине стека токена "левая круглая скобка"
            // перекладывать операторы из стека в очередь вывода.
            while(sl > 0)
            {
                sc = stack[sl - 1];
                if(sc == '(')
                {
                    pe = true;
                    break;
                }
                else
                {
                    *outpos = sc; ++outpos;
                    sl--;
                }
            }
            // Если стек кончится до нахождения токена левая круглая скобка, то была пропущена
скобка.
            if(!pe)
            {
                printf("Error: parentheses mismatched\n");
                return false;
            }
            // выкидываем токен "левая круглая скобка" из стека (не добавляем в очередь вывода).
            sl--;
            // Если на вершине стека токен - функция, положить его в очередь вывода.
            if(sl > 0)
            {
                sc = stack[sl - 1];
                if(is_function(sc))
                {
                    *outpos = sc; ++outpos;
                    sl--;
                }
            }
```

```
            }
            else
            {
                printf("Unknown token %c\n", c);
                return false; // Unknown token
            }
        }
        ++strpos;
    }
    // Когда не осталось токенов на входе:
    // Если в стеке остались токены:
    while(sl > 0)
    {
        sc = stack[sl - 1];
        if(sc == '(' || sc == ')')
        {
            printf("Error: parentheses mismatched\n");
            return false;
        }
        *outpos = sc; ++outpos;
        --sl;
    }

    *outpos = 0; // Добавляем завершающий ноль к строке
    return true;
}

bool execution_order(const char *input)
{
    printf("order: (arguments in reverse order)\n");
    const char *strpos = input, *strend = input + strlen(input);
    char c, res[4];
    unsigned int sl = 0, sc, stack[32], rn = 0;
    // Пока на входе остались токены
    while(strpos < strend)
    {
        // Прочитать следующий токен
        c = *strpos;
        // Если токен - значение или идентификатор
        if(is_ident(c))
        {
        // Поместить его в стек
            stack[sl] = c;
            ++sl;
        }
        // В противном случае, токен - оператор (здесь под оператором понимается как оператор, так и
название функции)
        else if(is_operator(c) || is_function(c))
        {
            sprintf(res, "_%02d", rn);
            printf("%s = ", res);
            ++rn;
            // Априори известно, что оператор принимает n аргументов
            unsigned int nargs = op_arg_count(c);
                        unsigned int Tnargs = nargs;
            // Если в стеке значений меньше, чем n
            if(sl < nargs)
            {
                // (ошибка) Недостаточное количество аргументов в выражении.
                return false;
            }
            // В противном случае, взять последние n аргументов из стека
            // Вычислить оператор, взяв эти значения в качестве аргументов
            if(is_function(c))
            {
                printf("%c(", c);
                while(nargs > 0)
                {
                    sc = stack[sl - nargs];
                    if(nargs > 1)
                    {
                        printf("%s, ", &sc);
                    }
                    else
                    {
```

```
                    printf("%s)\n", &sc);
                }
                --nargs;
            }
                            sl -= Tnargs;
        }
        else
        {
            if(nargs == 1)
            {
                sc = stack[sl - 1];
                sl--;
                printf("%c %s;\n", c, &sc);
            }
            else
            {
                sc = stack[sl - 2];
                printf("%s %c ", &sc, c);
                sc = stack[sl - 1];
                printf("%s;\n",&sc);
                                    sl -= 2;

            }
        }
        // Если получены результирующие значения, поместить таковые в стек.
        stack[sl] = *(unsigned int*)res;
        ++sl;
    }
    ++strpos;
  }
  // Если в стеке осталось лишь одно значение,
  // оно будет являться результатом вычислений.
  if(sl == 1)
  {
    sc = stack[sl - 1];
    sl--;
    printf("%s is a result\n", &sc);
    return true;
  }
  // Если в стеке большее количество значений,
  // (ошибка) Пользователь ввёл слишком много значений.
  return false;
}

int main()
{
    // Имена функций: A() B(a) C(a, b), D(a, b, c) ...
    // идентификаторы: 0 1 2 3 ... and a b c d e ...
    // операторы: = - + / * % !
    const char *input = "a = D(f - b * c + d, !e, g)";
    char output[128];
    printf("input: %s\n", input);
    if(shunting_yard(input, output))
    {
        printf("output: %s\n", output);
        execution_order(output);
    }
    return 0;
}
```

Ссылки

- Описание простейшей реализации, реализации унарных операторов и правоассоциативности (http://e-maxx.ru/algo/expressions_parsing)
- Описание и реализация на C++ (http://algolist.ru/syntax/revpn.php)
- Библиотека, реализующая алгоритм, на Java (http://bracer.sourceforge.net)

Реализации алгоритмов/Алгоритм Рутисхаузера

< Реализации алгоритмов

Алгоритм Рутисхаузера — один из ранних формальных алгоритмов разбора выражений со скобками, его особенностью является предположение о правильной скобочной структуре выражения, также алгоритмом не учитывается неявный приоритет операции.

C++

```
std::string Rutishauser::eval(std::string expr){
    int levels[expr.length()];
    int level = 0;
    int maxl = 0;
    int maxs = 0;
    int j = 0;
    bool isInNum = false;
    // Расстановка уровней
    for(int i = 0; i < expr.length(); i++){
        if(!isInNum || !isNum(expr[i])){
            if((expr[i] == '(') || (isNum(expr[i]))){
                level++;
            }else if((expr[i] == ')') || (!isNum(expr[i]))){
                level--;
            }
            levels[j] = level;
            if(level > maxl){
                maxl = level;
                maxs = i;
            }
            j++;
        }
        if(isNum(expr[i])){
            isInNum = true;
        }else{
            isInNum = false;
        }
    }
    if(j == 1){
        return expr;
    }
    int c = 0;
    for(int i = 0; i < j; i++){
        if(levels[i] == maxl){
            c++;
        }
    }
    if(c % 2 != 0){
        for(int i = maxs; i < expr.length(); i++){
            if(!isNum(expr[i])){
                std::string sa, sb, sc;
                if(maxs > 1){
                    sa = expr.substr(0, maxs - 1);
                }
                if(i > maxs){
                    sb = expr.substr(maxs, i - maxs);
                }else{
                    sb = expr[maxs];
                }
                if(i < expr.length() - 1){
```

```
                    sc = expr.substr(i + 1, expr.length());
                }
                return Rutishauser::eval(sa + sb + sc);
            }
        }
    }
    bool isOp = false;
    int p = maxs;
    int p1 = maxs;
    int a, b;
    for(int i = maxs; i < expr.length(); i++){
        if(!isNum(expr[i])){
            if(isOp){
                p1 = i;
                break;
            }
            isOp = true;
            a = atoi(expr.substr(maxs, i - 1).c_str());
            p = i;
        }
    }
    b = atoi(expr.substr(p + 1, p1 - 1).c_str());
    int tmp = 0;
    switch(expr[p]){
        case('+'):
            tmp = a + b;
            break;
        case('-'):
            tmp = a - b;
            break;
        case('*'):
            tmp = a*b;
            break;
        case('/'):
            tmp = a/b;
            break;
    }
    return Rutishauser::eval(expr.substr(0, maxs) + IntToString(tmp) + expr.substr(p1, expr.length() -
p1 + 1));
}
```

Источник — https://ru.wikibooks.org/w/index.php?title=Реализации_алгоритмов/Алгоритм_Рутисхаузера&oldid=105281

Иерархия Хомского формальных грамматик

Определение:

Иерархия Хомского (англ. *Chomsky hierarchy*) — классификация формальных грамматик и задаваемых ими языков, согласно которой они делятся на 4 класса по их условной сложности.

Содержание

Класс 0

К нулевому классу относятся все формальные грамматики. Элементы этого класса называются **неограниченными грамматиками** (англ. *unrestricted grammars*), поскольку на них не накладывается никаких ограничений. Они задают все языки, которые могут быть распознаны машиной Тьюринга. Эти языки также известны как **рекурсивно перечислимые** (англ. *recursively enumerable*).

Правила можно записать в виде:

$\alpha \rightarrow \beta$, где α — любая непустая цепочка, содержащая хотя бы один нетерминальный символ, а β — любая цепочка символов из алфавита.

Практического применения в силу своей сложности такие грамматики не имеют.

Пример

Продукции:

$S \rightarrow aBcc$
$B \rightarrow A$
$BAA \rightarrow d$
$Ac \rightarrow B$
$A \rightarrow AAA \mid dB$

Выведем в данной грамматике строку $addd$:

$$\boldsymbol{S} \Rightarrow a\boldsymbol{B}cc \Rightarrow a\boldsymbol{Ac}c \Rightarrow a\boldsymbol{B}c \Rightarrow a\boldsymbol{Ac} \Rightarrow a\boldsymbol{B} \Rightarrow a\boldsymbol{A} \Rightarrow ad\boldsymbol{B} \Rightarrow ad\boldsymbol{A} \Rightarrow ad\boldsymbol{A}AA \Rightarrow add\boldsymbol{BAA} \Rightarrow a$$

Класс 1

Первый класс представлен **неукорачивающими** и **контекстно-зависимыми** грамматиками.

Определение:

Неукорачивающая грамматика (англ. *noncontracting grammar*) — это формальная грамматика, всякое правило из P которой имеет вид $\alpha \rightarrow \beta$, где $\alpha, \beta \in \{\Sigma \cup N\}^+$ и $|\alpha| \leqslant |\beta|$ (возможно правило $S \rightarrow \varepsilon$, но тогда S не встречается в правых частях правил).

Определение:

Контекстно-зависимая грамматика (англ. *context-sensitive grammar*) — это формальная грамматика, всякое правило из P которой имеет вид $\alpha A\beta \to \alpha\gamma\beta$, где $\alpha, \beta \in \{\Sigma \cup N\}^*$, $A \in N$ и $\gamma \in \{\Sigma \cup N\}^+$ (возможно правило $S \to \varepsilon$, но тогда S не встречается в правых частях правил).

Языки, заданные этими грамматиками, распознаются с помощью **линейно ограниченного автомата** (англ. *linear bounded automaton*) (недетерминированная машина Тьюринга, чья лента ограничена константой, зависящей от длины входа.)

Известно, что неукорачивающие грамматики эквивалентны контекстно-зависимым.

Пример

$$L = \{w \in \Sigma^* \mid w = 0^n 1^n 2^n, n \geqslant 1\}$$

Продукции:

$$S \to 012$$
$$S \to 0AS2$$
$$A0 \to 0A$$
$$A1 \to 11$$

Класс 2

Второй класс составляют контекстно-свободные грамматики, которые задают контекстно-свободные языки. Эти языки распознаются с помощью автоматов с магазинной памятью.

Определение:

Контекстно-свободная грамматика (англ. *context-free grammar*) — это формальная грамматика, всякое правило из P которой имеет вид $A \to \beta$, где $A \in N, \beta \in \{\Sigma \cup N\}^+$.

То есть грамматика допускает появление в левой части правила только одного нетерминального символа.

Пример

$L = \{w \in \Sigma^* \mid w = w^R\}$ (язык палиндромов).

Продукции: $S \to \alpha S\alpha \mid \alpha \mid \varepsilon, \alpha \in \Sigma$

Класс 3

К третьему типу относятся **автоматные** или **регулярные грамматики** (англ. *regular grammars*) — самые простые из формальных грамматик, которые задают регулярные языки. Они являются контекстно-свободными, но с ограниченными возможностями.

Все регулярные грамматики могут быть разделены на два эквивалентных класса следующего вида:

Определение:

Леволинейная грамматика (англ. *left-regular grammar*) — это формальная грамматика, всякое правило из P которой имеет вид $A \to B\gamma$ или $A \to \gamma$, где $\gamma \in \Sigma, A, B \in N$.

Определение:

Праволинейная грамматика (англ. *right-regular grammar*) — это формальная грамматика, всякое правило из P которой имеет вид $A \to \gamma B$; или $A \to \gamma$, где $\gamma \in \Sigma, A, B \in N$.

Оба вида задают одинаковые языки. При этом если правила леволинейной и праволинейной грамматик объединить, то язык уже не обязан быть регулярным.

Также можно показать, что множество языков, задаваемых праволинейными грамматиками, совпадает со множеством языков, задаваемых конечными автоматами.

Пример

L для регулярного выражения a^*bc^*.

Продукции:

$S \to aS \mid bA$
$A \to \varepsilon \mid cA$

См. также

- Правоконтекстные грамматики, эквивалентность автоматам
- Возможность порождения формальной грамматикой произвольного перечислимого языка

Источники информации

- *А. Ахо, Дж. Ульман.* Теория синтаксического анализа, перевода и компиляции. Синтаксический анализ. Том 2. Пер. с англ. — М.: Книга по Требованию, 2012. — ISBN 978-5-458-27407-4
- Wikipedia — Chomsky hierarchy
- Википедия — Иерархия Хомского

Существенно неоднозначные языки

Содержание

Неоднозначные грамматики

Определение:

Неоднозначной грамматикой (англ. *ambiguous grammar*) называется грамматика, в которой можно вывести некоторое слово более чем одним способом (то есть для строки есть более одного дерева разбора).

Пример:

Рассмотрим грамматику $E \to E + E|E * E|N$ и выводимое слово $N + N * N$. Его можно вывести двумя способами:

$$E \Rightarrow E + E \Rightarrow E + E * E \Rightarrow N + N * N$$

$$E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow N + N * N$$

Эта грамматика неоднозначна.

В данном случае мы нашли пример слова из языка (который задается грамматикой), которое имеет более одного вывода, и показали, что грамматика является существенно неоднозначной. Однако в общем случае проверка грамматики на неоднозначность является алгоритмически неразрешимой задачей.

Существенно неоднозначные языки

Определение:

Язык называется **существенно неоднозначным** (англ. *inherently ambiguous language*), если любая грамматика, порождающая его, является неоднозначной.

Пример:

Язык $0^a1^b2^c$, где либо $a = b$, либо $b = c$, является существенно неоднозначным.

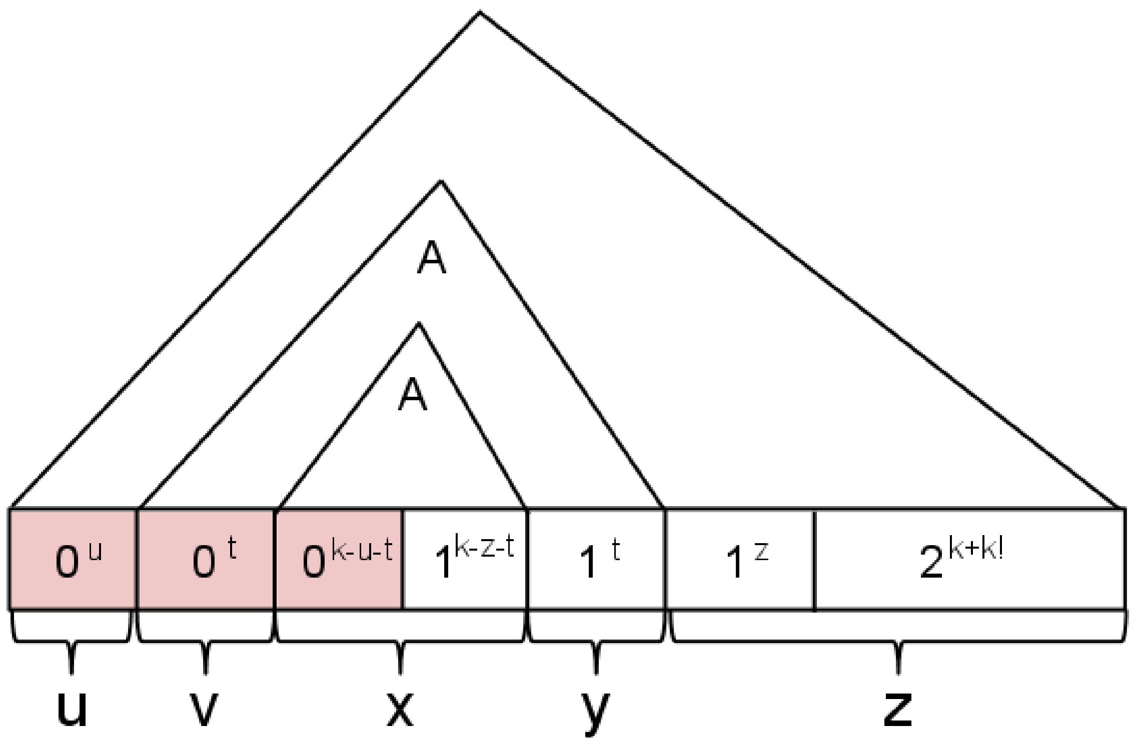
Докажем, что для любой грамматики Γ $\exists n : 0^n1^n2^n$ имеет хотя бы 2 дерева разбора в грамматике Γ.

Возьмем k и рассмотрим слово $0^k1^k2^{k+k!}$.

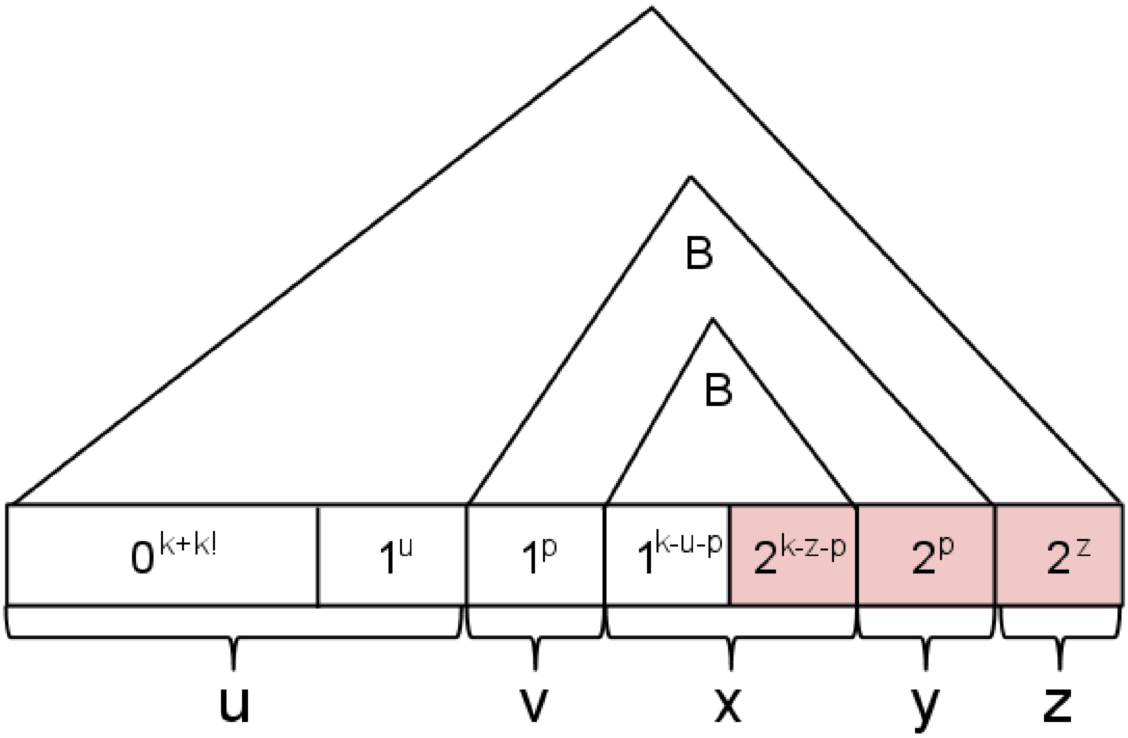
Пометим первые k нулей, по лемме Огдена данное слово можно разбить на 5 частей: $0^k1^k2^{k+k!} = uvxyz$.

Понятно, что v состоит полностью из нулей, а y состоит полностью из единиц, а также длины v и y равны, так как иначе при накачке мы можем получить слово, не принадлежащее языку.

Пусть $|v| = |y| = t$, тогда возьмём слово $q = uv^{k!/t+1}xy^{k!/t+1}z$. По лемме Огдена слово q принадлежит языку, а также существует нетерминал A такой, что с помощью него можно породить слово q, то есть в грамматике можно вывести uAz, и из A можно вывести vAy и x. (Заметим, что $q = 0^{k!+k}1^{k!+k}2^{k!+k}$, то есть $n = k! + k$.)



Теперь рассмотрим слово $0^{k+k!}1^k2^k$, в котором отмечены все двойки. Аналогичными рассуждениями мы получаем, что слово q принадлежит языку, а также существует нетерминал B такой, что с помощью него можно породить слово q, где $|v| = |y| = p$.



Заметим, что поддеревья, соответствующие A и B — разные деревья и одно не является потомком другого, иначе или в поддереве A были бы двойки, или в поддереве B были бы нули — что не является правдой.

Пусть в этих двух случаях дерево разбора было одно и тоже, тогда с помощью A и B можно породить слово вида $0^{k+k!+t}1^{k+k!+t+p}2^{k+k!+p}$, которое не принадлежит языку.

В результате мы имеем два дерева разбора для одного слова. Значит, язык существенно неоднозначен.

См. также

- Лемма Огдена
- Лемма о разрастании для КС-грамматик

- Теорема Парика

Источники информации

- Википедия — Алгоритмически неразрешимая задача (http://ru.wikipedia.org/wiki/Алгоритмически_неразрешимая_задача)
- Wikipedia — Ambiguous grammar (http://en.wikipedia.org/wiki/Ambiguous_grammar)

Замкнутость КС-языков относительно различных операций

В отличие от регулярных языков, КС-языки не замкнуты относительно всех теоретико-множественных операций. К примеру, дополнение и пересечение КС-языков не обязательно являются КС-языками.

Здесь и далее считаем, что L_1 и L_2 — КС-языки.

Содержание

Операции с КС-языками

Объединение

Утверждение:

$L_1 \cup L_2$ является КС-языком.

▷

Построим КС-грамматику для языка $L_1 \cup L_2$. Для этого рассмотрим соответствующие КС-грамматики для языков L_1 и L_2. Пусть стартовые символы в них имеют имена S и T соответственно. Тогда стартовый символ для $L_1 \cup L_2$ обозначим за S' и добавим правило $S' \to S \mid T$.

Покажем, что $S' \Rightarrow^* w \iff S \Rightarrow^* w \vee T \Rightarrow^* w$.

$\Rightarrow$

> Поскольку $S \Rightarrow^* w$ и есть правило $S' \to S$, то, по определению $\Rightarrow^*$ получаем, что $S' \Rightarrow^* w$. Аналогично и для T.

$\Leftarrow$

> Пусть $S' \Rightarrow^* w$. Поскольку $S' \to S \mid T$ — единственные правила, в которых нетерминал S' присутствует в правой части, то это означает, что либо $S' \Rightarrow S \Rightarrow^* w$, либо $S' \Rightarrow T \Rightarrow^* w$.

$\triangleleft$

Конкатенация

Утверждение:

$L_1 L_2$ — КС-язык.

$\triangleright$

Аналогично предыдущему случаю построим КС-грамматику для языка $L_1 L_2$. Для этого добавим правило $S' \to ST$, где S и T — стартовые символы языков L_1 и L_2 соответственно.

$\triangleleft$

Замыкание Клини

Утверждение:

$L^* = \bigcup\limits_{i=0}^{\infty} L^i$ — КС-язык.

$\triangleright$

Если S — стартовый символ КС-грамматики для языка L, то добавим в КС-грамматику для языка L^* новый стартовый символ S' и правила
$S' \to SS' \mid \varepsilon$.
$\triangleleft$

Гомоморфизмы

Прямой гомоморфизм

Утверждение:

КС-языки замкнуты относительно прямого гомоморфизма.

$\triangleright$
Построим КС-грамматику, в которой каждый символ $x \in \Sigma$ заменим на $\mathrm{h}(x)$.
$\triangleleft$

Обратный гомоморфизм

Утверждение:

КС-языки замкнуты относительно обратного гомоморфизма.

$\triangleright$

Докажем аналогично соответствующему утверждению для регулярных языков.
Построим МП-автомат для $\mathrm{h}^{-1}(L) = \{w \mid \mathrm{h}(w) \in L\}$ на основе МП-автомата для языка L (назовем его M).
Новый автомат M' будет действовать следующим образом:

1. Если входное слово закончилось, допускаем или не допускаем его по допускающему состоянию.
2. Считываем символ c.
3. Сохраняем $\mathrm{h}(c)$ в буфере (входная лента для автомата M).
4. Запускаем M на слове, находящемся в буфере.
5. После того, как M обработал весь буфер, переходим к пункту 1.

Если рассмотреть более формально, пусть $M = \langle Q, \Sigma, \Gamma, \delta, s, Z_0, T\rangle$, тогда $M' = \langle Q', \Sigma, \Gamma, \delta', (s, \varepsilon), Z_0, T \times \varepsilon\rangle$.

- $Q' = \{(q, x) \mid q \in Q\}$, где x — суффикс (не обязательно собственный) некоторой цепочки $h(c)$ для символа $c \in \Sigma$. Таким образом, первый компонент состояния M' является состоянием M, а второй — компонентом буфера.
- δ' определяется следующими правилами:
  - $\delta'((q, \varepsilon), c, X) = \{(((q, \mathrm{h}(c)), X) \mid c \in \Sigma, q \in Q, X \in \Gamma\}$. Когда буфер пуст, M' может прочитать свой следующий входной символ c и поместить $\mathrm{h}(c)$ в буфер.
  - Если $(p, \gamma) \in \delta(q, b, X), b \in T \cup \varepsilon$, то $((p, x), \gamma) \in \delta'((q, bx), \varepsilon, X)$. Таким образом, M' всегда имеет возможность имитации перехода M, используя голову буфера. Если $b \in T$, то буфер должен быть непустым, но если $b = \varepsilon$, то буфер может быть пустым.
- Начальным состоянием M' является (s, ε), т.е. M' стартует в начальном состоянии M с пустым буфером.
- Допускающими состояниями M' являются состояния (q, ε), где $q \in T$.

Таким образом получаем, что $(s, \mathrm{h}(w), Z_0) \vdash_M^* (p, \varepsilon, \gamma) \Leftrightarrow ((s, \varepsilon), w, Z_0) \vdash_{M'}^* ((p, \varepsilon), \varepsilon, \gamma)$, то есть автомат M' допускает те и только те слова, которые принадлежат языку $\mathrm{h}^{-1}(L)$.

$\triangleleft$

Разворот

Утверждение:

$L^R = \{w^R \mid w \in L\}$ контекстно-свободен.

$\triangleright$

Для того, чтобы построить L^R, необходимо развернуть все правые части правил грамматики для L.

Покажем, что $w \in L \iff w^R \in L^R$.

$\Rightarrow$

Докажем с помощью метода математической индукции по длине порождения в грамматике L.
База. $A \underset{L}{\Rightarrow} w$.

В грамматике L существует правило $A \to w$ и, так как мы развернули все правые части правил, то $A \underset{L^R}{\Rightarrow} w^R$.

Предположение индукции. Пусть $A \underset{L}{\Rightarrow}^* w$ менее чем за n шагов, тогда $A \underset{L^R}{\Rightarrow}^* w^R$.

Переход.

Пусть в порождении n шагов, $n > 1$. Тогда оно имеет вид $A \underset{L}{\Rightarrow} Y_1 Y_2 \dots Y_m \underset{L}{\Rightarrow}^* w$, где $Y_i \in N \cup \Sigma$. Цепочку w можно разбить на $w_1 w_2 \dots w_m$, где $Y_i \underset{L}{\Rightarrow}^* w_i$. Так как каждое из порождений $Y_i \underset{L}{\Rightarrow}^* w_i$ содержит менее n шагов, к ним можно применить предположение индукции и заключить, что $Y_i \underset{L^R}{\Rightarrow}^* w_i^R$. Так как $A \underset{L}{\Rightarrow} Y_1 Y_2 \dots Y_m$, то $A \underset{L^R}{\Rightarrow} Y_m Y_{m-1} \dots Y_1$, откуда следует, что $A \underset{L^R}{\Rightarrow}^* w^R$.

$\Leftarrow$

Доказательство аналогично.

$\triangleleft$

Пример разворота:

Пусть задана КС-грамматика G для языка $L = a^i b^j c^i$ со следующими правилами:

- $A \to bA \mid \varepsilon$
- $B \to aBc \mid A$

В таком случае КС-грамматика G^R для языка $L^R = c^i b^j a^i$ выглядит следующим образом:

- $A \rightarrow Ab \mid \varepsilon$
- $B \rightarrow cBa \mid A$

Дополнение к языку тандемных повторов

Утверждение:

Язык тандемных повторов $L = \{ww \mid w \in \Sigma^*\}$ не является КС-языком.

▷

Это доказывается с помощью леммы о разрастании.

◁

Утверждение:

Дополнение к языку тандемных повторов $\overline{L}$ является КС-языком.

▷

Для упрощения рассмотрим этот язык на бинарном алфавите $\Sigma = \{a, b\}$. Для $\overline{L}$ можно составить следующую КС-грамматику G:

- $S \rightarrow AB \mid BA$
- $S \rightarrow A \mid B$
- $S \rightarrow \varepsilon$
- $A \rightarrow aAa \mid aAb \mid bAa \mid bAb \mid a$
- $B \rightarrow aBa \mid aBb \mid bBa \mid bBb \mid b$

Докажем этот факт.

Сначала заметим, что нетерминал A порождает слова нечётной длины с центральным символом a. В свою очередь нетерминал B порождает слова нечётной длины с центральным символом b. Таким образом, правило $S \rightarrow A \mid B$ порождает все возможные слова нечётной длины.

Докажем, что все слова, порождённые G, есть в $\overline{L}$.

ε, а также все слова нечётной длины не являются тандемными повторами.

Рассмотрим произвольное слово чётной длины, сгенерированное при помощи правила $S \rightarrow AB$. Пусть его часть, соответствующая A, имеет длину $2N + 1$, а часть, соответствующая B, — длину $2M + 1$.

N символов	a	N символов	M симв.	b	M симв.

Таким образом, мы получили слово длины $2N + 2M + 2$. Если оно является тандемным повтором, то символ, стоящий на позиции $N + 1$, должен быть равен символу на позиции $2N + M + 2$. Но по построению это не так.

Для правила $S \to BA$ доказательство аналогично.

Докажем, что все слова из $\overline{L}$ порождаются G.

С помощью G можно вывести ε, а также любое слово нечётной длины.

Далее рассмотрим произвольное слово чётной длины из $\overline{L}$. Докажем, что его можно разбить на два слова нечётной длины, имеющие различные центральные символы. Предположим, что это не так, то есть такого разбиения нет.

Пусть это слово имеет длину $2N$. Тогда рассмотрим все его префиксы нечётной длины. Их центры находятся на позициях $1, 2, \ldots, N$, а центры соответствующих им суффиксов — на позициях $N + 1, N + 2, \ldots, 2N$. Поскольку искомого разбиения не существует, то получается, что символ на позиции 1 равен символу на позиции $N + 1$, символ на позиции 2 равен символу на позиции $N + 2$, и так далее. Следовательно, первая половина слова равна его второй половине, т.е. оно является тандемных повтором.

Получили противоречие, следовательно любое слово чётной длины из $\overline{L}$ можно разделить на два слова нечётной длины с различными центральными символами. В свою очередь, такие слова могут быть сгенерированы при помощи грамматики G и соединены при помощи правила $S \to AB \mid BA$.

$\triangleleft$

Пересечение

Утверждение:

Если $L_1 = a^i b^i c^j, L_2 = a^i b^j c^j$, то $L_1 \cap L_2$ не является КС-языком.

$\triangleright$

$$L_1 = \{a^i b^i\}\{c^j\}, L_2 = \{a^i\}\{b^j c^j\}$$

По замкнутости КС-языков относительно конкатенации получаем, что L_1 и L_2 являются КС-языками.

Но $L_1 \cap L_2 = \{a^i b^i c^i \mid i \in \mathbb{N}\}$, который по лемме о разрастании для КС-языков не является КС-языком.
$\triangleleft$

Разность

Утверждение:

КС-языки не замкнуты относительно разности.

$\triangleright$

$$L_1 \setminus L_2 = L_1 \cap \overline{L_2}$$

$\triangleleft$

Более того, задачи определения того, является ли дополнение КС-языка КС-языком и проверки непустоты пересечения КС-языков являются алгоритмически неразрешимыми.

Половины тандемных повторов

Определение:

$$\mathrm{half}(L) = \{w \mid ww \in L\}$$

Утверждение:

Операция half не сохраняет КС-язык таковым.

$\triangleright$

Покажем это на примере. Рассмотрим язык $L = \{a^n b a^n b a^m b a^l b a^k b a^k b\}$.

Заметим, что он может быть сгенерирован при помощи следующей КС-грамматики:

- $S \to AbBbBbAb$
- $B \to a \mid aB$
- $A \to b \mid aAa$

Докажем, что $\mathrm{half}(L)$ не является КС-языком.

Пусть $\alpha = a^n b a^n b a^m b a^l b a^k b a^k b = ww$. Отсюда следует, что:

- $n = l$
- $n = k$
- $m = k$

А значит, $n = l = k = m$, и $\mathrm{half}(L) = \{a^n b a^n b a^n b\}$, и по лемме о разрастании КС-языком не является.

$\triangleleft$

Операции над КС-языком и регулярным языком

Пересечение

Утверждение:

Пересечение КС-языка и регулярного языка — КС-язык.

$\triangleright$

Построим МП-автомат для пересечения регулярного языка и КС-языка.

Пусть регулярный язык задан своим ДКА, а КС-язык — своим МП-автоматом с допуском по допускающему состоянию. Построим прямое произведение этих автоматов так же, как строилось прямое произведение для двух ДКА.

Более формально, пусть R — регулярный язык, заданный своим ДКА $\langle \Sigma, Q_1, s_1, T_1, \delta_1 \rangle$, и L — КС-язык, заданный своим МП-автоматом: $\langle \Sigma, \Gamma, Q_2, s_2, T_2, z_0, \delta_2 \rangle$. Тогда прямым произведением назовем следующий автомат:

- $Q = \{\langle q_1, q_2 \rangle \mid q_1 \in Q_1, q_2 \in Q_2\}$. Иначе говоря, состояние в новом автомате — пара из состояния первого автомата и состояния второго автомата.
- $s = \langle s_1, s_2 \rangle$
- Стековый алфавит Γ остается неизменным.
- $T = \{\langle t_1, t_2 \rangle \mid t_1 \in T_1, t_2 \in T_2\}$. Допускающие состояния нового автомата — пары состояний, где оба состояния были допускающими в своем автомате.

- $\delta(\langle q_1, q_2 \rangle, c, d) = \langle \delta_1(q_1, c), \delta_2(q_2, c, d) \rangle$. При этом на стек кладется то, что положил бы изначальный МП-автомат при совершении перехода из состояния q_2,

видя на ленте символ c и символ d на вершине стека.

Этот автомат использует в качестве состояний пары из двух состояний каждого автомата, а за операции со стеком отвечает только МП-автомат. Слово допускается этим автоматом $\iff$ слово допускается и ДКА и МП-автоматом, то есть язык данного автомата совпадает с $R \cap L$.
$\triangleleft$

Разность

Утверждение:

Разность КС-языка и регулярного языка — КС-язык.

$\triangleright$

$$L \setminus R = L \cap \overline{R}$$

Регулярные языки замкнуты относительно дополнения, следовательно разность можно выразить через пересечение.
$\triangleleft$

См. также

- Контекстно-свободные грамматики, вывод, лево- и правосторонний вывод, дерево разбора
- Замкнутость регулярных языков относительно различных операций
- Основные определения, связанные со строками

Источники информации

- *Хопкрофт Д., Мотвани Р., Ульман Д.* — Введение в теорию автоматов, языков и вычислений, 2-е изд. : Пер. с англ. — Москва, Издательский дом «Вильямс», 2002. — С. 302-304 : ISBN 5-8459-0261-4 (рус.)

Замкнутость регулярных языков относительно различных операций

Содержание

Теорема

Теорема:

Пусть L_1, L_2 — регулярные языки над одним алфавитом Σ. Тогда следующие языки также являются регулярными:

1. Языки, полученные путём применения следующих теоретико-множественных операций:
   - $L_1 \cup L_2$,
   - $\overline{L_1}$,
   - $L_1 \cap L_2$,
   - $L_1 \setminus L_2$;
2. L_1^*;
3. $L_1 L_2$;
4. $\overleftarrow{L_1}$.

Доказательство:

$\triangleright$

Как известно, классы регулярных и автоматных языков совпадают. Пусть языки L_1 и L_2 распознаются автоматами $A_1 = \langle \Sigma, Q_1, s_1, T_1, \delta_1 : Q_1 \times \Sigma \to 2^{Q_1} \rangle$ и $A_2 = \langle \Sigma, Q_2, s_2, T_2, \delta_2 : Q_2 \times \Sigma \to 2^{Q_2} \rangle$ соответственно.

1. - $L_1 \cup L_2$ является регулярным по определению регулярных языков.
   - Рассмотрим автомат $A_1' = \langle \Sigma, Q_1, s_1, Q_1 \setminus T_1, \delta_1 \rangle$, то есть автомат A, в котором терминальные и нетерминальные состояния инвертированы (при таком построении следует помнить, что если в исходном автомате было опущено дьявольское состояние, его нужно явно добавить и сделать допускающим.) Очевидно, он допускает те и только те слова, которые не допускает автомат A_1, а значит, задаёт язык $\overline{L_1}$. Таким образом, $\overline{L_1}$ — регулярный.
   - $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$. Тогда $L_1 \cap L_2$ — регулярный. Также автомат для пересечения языков можно построить явно, используя конструкцию произведения автоматов.
   - $L_1 \setminus L_2 = L_1 \cap \overline{L_2}$. Тогда $L_1 \setminus L_2$ — регулярный.
2. L_1^* является регулярным по определению регулярных языков.
3. $L_1 L_2$ также является регулярным по определению регулярных языков.
4. Рассмотрим НКА с ε-переходами $A_1' = \langle \Sigma, Q_1, s', \{s_1\}, \delta_1' \rangle$, где $\delta_1'(v, c) = \{u | \delta_1(u, c) = v\}; \delta_1'(s', \varepsilon) = \{T_i\}$. Если в исходном автомате путь по α из s_1 приводил в терминальное состояние, то в новом автомате существует путь по α из этого терминального состояния в s_1 (и наоборот). Следовательно, этот автомат распознает в точности развернутые слова языка L_1. Тогда язык $\overleftarrow{L_1}$ — регулярный.

◁

Примеры доказательств

Гомоморфизм цепочек

Утверждение:

$L \subset \Sigma_1^*$ — регулярный , $\varphi : \Sigma_1^* \to \Sigma_2^*$ — гомоморфизм цепочек. Тогда $\varphi(L)$ — регулярный.

▷

Рассмотрим ДКА, распознающий L. Заменим в нем все переходы по символам на переходы по их образам при гомоморфизме. Полученный автомат (с переходами по строкам) распознает в точности $\varphi(L)$ и имеет эквивалентный ДКА.

◁

Утверждение:

$L \subset \Sigma_2^*$ — регулярный , $\varphi : \Sigma_1^* \to \Sigma_2^*$ — гомоморфизм цепочек. Тогда $\varphi^{-1}(L)$ — регулярный.

▷

Рассмотрим ДКА, распознающий L. Отследим для каждого состояния u и символа c строку $\varphi(c)$: $\langle u, \varphi(c)\rangle \vdash^* \langle v, \varepsilon\rangle$ и положим $\delta(u, c) = v$ в новом автомате (на том же множестве состояний). Автомат с построенной таким образом функцией переходов, очевидно, распознает слова языка $\varphi^{-1}(L)$ и только их.

$\triangleleft$

Язык half(L)

Определение:

Определим $\mathrm{half(L)}$ как множество первых половин цепочек языка L, то есть множество $\{w \mid \exists x : wx \in L \land |w| = |x|\}$.

Например, если $L = \{\varepsilon, 0010, 011, 010110\}$, то $\mathrm{half(L)} = \{\varepsilon, 00, 010\}$. Заметим, что цепочки нечетной длины не влияют на $\mathrm{half(L)}$.

Утверждение:

Пусть L — регулярный язык. Тогда язык $\mathrm{half(L)}$ также регулярен.

$\triangleright$

Так как L — регулярный язык, то существует ДКА $M = \langle \Sigma, Q, q_0, F, \delta\rangle$, допускающий его. Рассмотрим строку x. Для того, чтобы проверить, что $x \in \mathrm{half(L)}$, нам надо убедиться, что существует строка y такой же длины, что и x, которая, будучи сконкатенированной с x, даст строку из L, то есть если на вход автомату подать xy, то в конце обработки мы окажемся в терминальном состоянии. Предположим, что автомат, закончив обработку x, находится в состоянии q_i, то есть $\delta(q_0, x) = q_i$. Мы должны проверить, что существует строка y, $|y| = |x|$, которая ведет из состояния q_i до какого-нибудь терминального состояния M, то есть $\delta(q_i, y) \in F$.

Предположим, что мы прошли n вершин автомата, то есть $|x| = n$. Обозначим за S_n множество всех состояний, с которых можно попасть в терминальные за n шагов. Тогда $q_i \in S_n \Leftrightarrow x \in \mathrm{half(L)}$. Если мы сможем отслеживать S_n и q_i, то сможем определять, верно ли, что $x \in \mathrm{half(L)}$. Заметим, что $S_0 \equiv F$. Очевидно мы можем построить S_{n+1} зная S_n и δ:
$S_{n+1} = prev(S_n) = \{q \in Q \mid \exists a \in \Sigma, q' \in S_n, \delta(q, a) = q'\}$ — множество состояний, из которых есть переход в какое-либо состояние из S_n (по единственному символу). Теперь надо найти способ отслеживать и обновлять S_n.

Построим ДКА M', который будет хранить эту информацию в своих состояниях. Определим $Q' = Q \times 2^Q$, то есть каждое состояние M' — это пара из одиночного состояния из M и множества состояний из M. Функцию перехода δ' автомата M' определим так, чтобы если по какой-то строке x длины n в автомате M мы перешли в состояние q_i, то по этой же строке в автомате M' мы перейдем в состояние (q_i, S_n), где S_n — множество состояний из M, определенное выше. Вспомним приведенную выше функцию $prev(S_n) = S_{n+1}$. С ее помощью мы можем определить функцию перехода следующим образом: $\delta'((q, S), a) = (\delta(q, a), prev(S))$. Начальное состояние $q'_0 = (q_0, S_0) = (q_0, F)$. Множество терминальных состояний — $F' = \{(q, S) \mid q \in S, S \in 2^Q\}$.

Теперь по индукции не сложно доказать, что $\delta'(q'_0, x) = (\delta(q_0, x), S_n)$, где $|x| = n$. По определению множества терминальных вершин, автомат M' допускает строку x тогда и только тогда, когда $\delta(q_0, x) \in S_n$. Следовательно, автомат M' допускает язык $\mathrm{half(L)}$.Таким образом, мы построили ДКА, который допускает язык $\mathrm{half(L)}$. Следовательно, данный язык является регулярным.

$\triangleleft$

Язык cycle(L)

Определение:

Определим $\mathrm{cycle(L)}$ как множество $\{w|$ цепочку w можно представить в виде $w = xy$, где $yx \in L\}$.

Например, если $L = \{01, 011\}$, то $\mathrm{cycle(L)} = \{01, 10, 011, 110, 101\}$.

Утверждение:

Пусть L — регулярный язык. Тогда язык $\mathrm{cycle(L)}$ также регулярен.

$\triangleright$

Так как L — регулярный язык, то существует допускающий его ДКА $M = \langle \Sigma, Q, q_0, F, \delta \rangle$. Построим из M недетерминированный автомат с ε-переходами следующим образом: рассмотрим состояние $q \in Q$, из которого есть переходы в другие состояния (то есть начиная с q можно построить непустое слово, заканчивающееся в терминальной вершине). Тогда если какое-то слово

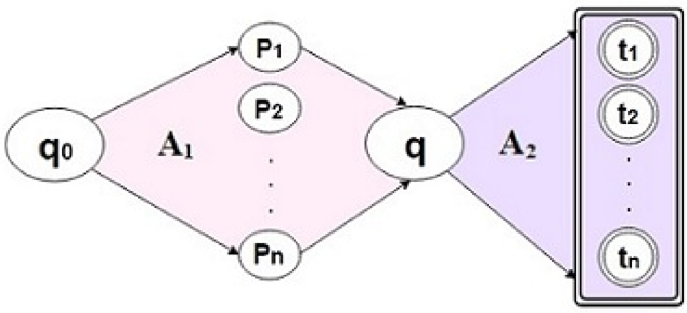


Рис. 1. Разбиение автомата.

проходит через это состояние, оно может быть зациклено таким образом, что его суффикс, начинающийся с q, станет префиксом нового слова, а префикс, заканчивающийся в q — суффиксом. Разделим автомат на две части A_1 и A_2 такие, что A_1 будет содержать все вершины, из которых достижима q, а A_2 — все вершины, которые достижимы из q (см. рис. 1). Заметим, что каждая вершина может содержаться в обеих частях одновременно, такое может случиться, если автомат M содержит циклы. Теперь перестроим автомат так, что он будет принимать слова "зацикленные" вокруг q, то есть начинающиеся с q и после достижения терминальной вершины продолжающиеся с q_0 (см. рис. 2). Для этого стартовой вершиной сделаем q и построим от нее часть A_2. Теперь добавим состояние q_0 и соединим с ним все терминальные состояния из A_2 с помощью ε-переходов. Далее построим от q_0 часть A_1. Добавим вершину q', эквивалентную q, и сделаем ее терминальной. Данный автомат принимает слова, зацикленные вокруг выбранной вершины q. Мы хотим, чтобы автомат принимал слова, зацикленные вокруг любой такой q. Для этого создадим новую стартовую вершину q_0' и свяжем ее ε -переходами со всеми перестроенными автоматами (зацикленными вокруг всех подходящих q), в том числе и с изначальным автоматом. Построенный автомат допускает язык $\mathrm{cycle(L)}$, следовательно, данный язык является регулярным.

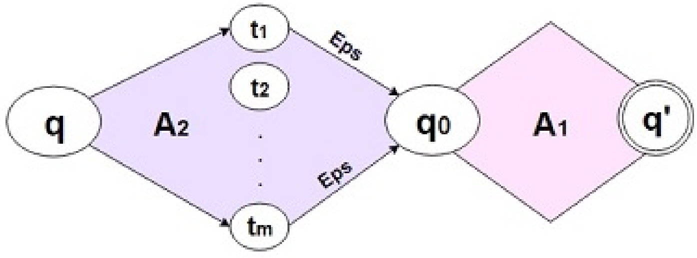


Рис. 2. Перестроение.

◁

Для лучшего понимания алгоритма перестроения автомата рассмотрим пример.

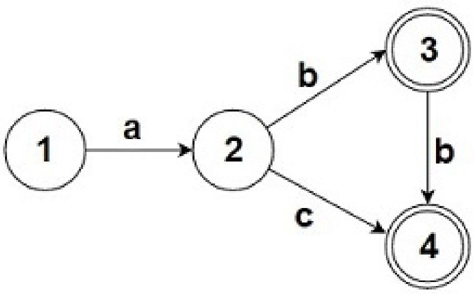


Рис. 3. Автомат, принимающий язык L.

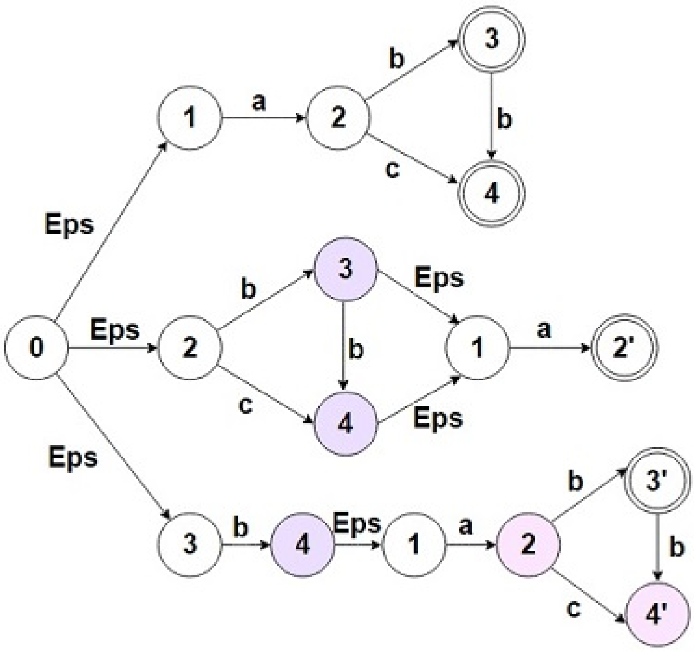


Рис. 4. Автомат, принимающий язык $\mathrm{cycle(L)}$.

На рис. 3 представлен автомат, допускающий язык $L = \{ab, abb, ac\}$. На рис. 4 показано, как этот автомат был перестроен. Были добавлены части, зацикленные относительно вершин 2 и 3. Появилась новая стартовая вершина 0, которая связана ε-переходами с изначальным автоматом и его измененными версиями. Данный автомат распознает язык $\mathrm{cycle(L)} = \{ab, abb, ac, ba, bba, ca, bab\}$: первые три слова распознает первая часть, которая совпадает с изначальным автоматом; следующие три — вторая, перестроенная относительно вершины 2; последнее слово распознает третья часть, зацикленная относительно вершины 3.

Язык alt(L, M)

Определение:

Пусть $w = w_1 w_2 \ldots w_n$ и $x = x_1 x_2 \ldots x_n$. Определим $alternation(w, x) = w_1 x_1 w_2 x_2 \ldots w_n x_n$.

Теперь распространим это определение:

Определение:

Пусть L и M — два языка над одним алфавитом Σ. Тогда $\mathrm{alt(L, M)} = \{alternation(w, x) \mid |w| = |x|, w \in L, x \in M\}$.

Например, если $L = \{10, 00, 111, 1001\}$ и $M = \{11, 0101\}$, то $\mathrm{alt(L, M)} = \{1101, 0101, 10010011\}$.

Утверждение:

Пусть L и M — регулярные языки. Тогда $\mathrm{alt(L, M)}$ также является регулярным.

$\triangleright$

Так как L и M — регулярные языки, то существуют ДКА $D_L = \langle \Sigma, Q_L, q_{0L}, F_L, \delta_L \rangle$, распознающий язык L, и $D_M = \langle \Sigma, Q_M, q_{0M}, F_M, \delta_M \rangle$, распознающий язык M. Построим автомат D_{alt}, который будет распознавать язык $\mathrm{alt(L, M)}$. Идея следующая: каждое состояние этого автомата будем описывать тремя значениями (p, q, b), где $p \in Q_L$, $q \in Q_M$ и $b \in \{1, 0\}$. Нам нужно организовать чередование переходов по состояниям автоматов, то есть если мы на определенном шаге перешли от одного состояния автомата D_L до другого, то на следующем мы обязаны совершить переход по состояниям автомата D_M. Для этого нам нужно обновлять состояние одного автомата и при этом сохранять состояние другого для следующего перехода. Тут мы будем использовать третье значение: если $b = 0$, то будет двигаться по состояниям первого автомата, то есть значение p при переходе в новое состояние автомата D_{alt} поменяется, q останется неизменной, b станет 1, если $b = 1$, то, соответственно, все наоборот. То есть у нас будут две функции перехода, выбирать нужную будем в зависимости от четности третьего параметра. Важно, что на каждом шаге мы инвертируем значение b, что гарантирует чередование. Определим автомат $D_{alt} = \langle \Sigma, Q', q'_0, F', \delta' \rangle$ следующим образом:

1. $Q' = Q_L \times Q_M \times \{0, 1\}$
2. $q'_0 = (q_{0L}, q_{0M}, 0)$
3. $F' = F_L \times F_M \times \{0\}$
4. $\delta'((p, q, 0), a) = (\delta_L(p, a), q, 1)$ и $\delta'((p, q, 1), a) = (p, \delta_M(q, a), 0)$

Стартовая вершина имеет третий параметр $b = 0$, так как первое значение должно быть получено из автомата D_L. Аналогично все терминальные вершины должны иметь то же значение последнего параметра, так как количество переходов должно быть четным и последний переход должен был быть осуществлен по автомату D_M. Функция перехода δ' использует δ_L для получения нечетных символов и δ_M для четных. Таким образом, D_{alt} состоит из чередующихся символов D_L и D_M. При этом D_{alt} принимает w тогда и только тогда, когда D_L последовательно принимает все нечетные символы w и D_M — все четные, а так же w имеет четную длину. Следовательно, D_{alt} распознает язык $\mathrm{alt}(\mathrm{L}, \mathrm{M})$, что доказывает, что $\mathrm{alt}(\mathrm{L}, \mathrm{M})$ является регулярным.

$\triangleleft$

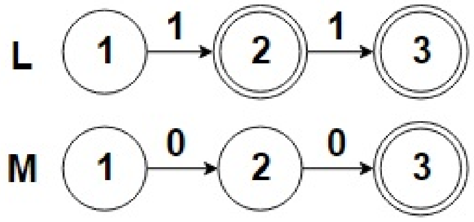


Рис. 5. Автоматы для языков L и M.

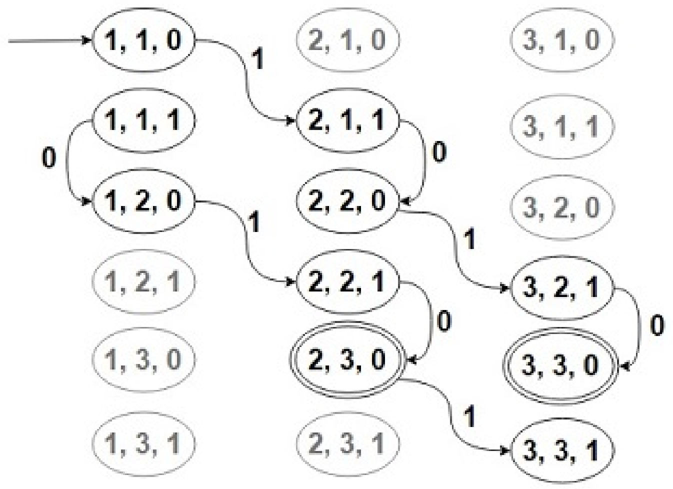


Рис. 6. Автомат, принимающий язык $\mathrm{alt}(\mathrm{L}, \mathrm{M})$.

Чтобы более наглядно показать, как строится автомат D_{alt}, разберем пример. Пусть $L = \{1, 11\}$ и $M = \{00\}$ (см. рис. 5). Все состояния нового автомата представлены на рис. 6. Стартовая вершина $q'_0 = (1, 1, 0)$, множество терминальных вершин — $F' = \{(2, 3, 0), (3, 3, 0)\}$. Мы видим, что построенные по функции δ' переходы на каждом шаге меняют состояние одного из автоматов, а именно того, по которому происходит переход, сохраняя состояние другого для следующего шага. Таким образом, каждый следующий символ получен из автомата, отличного от того, что был использован на предыдущем шаге. Декартово произведение состояний гарантирует, что мы рассмотрим все состояния и переходы изначальных автоматов. Для данного примера мы получаем, что $\mathrm{alt(L, M)} = \{1010\}$.

См. также

- Анализ свойств регулярных языков (пустота, совпадение, включение, конечность, подсчет числа слов)
- Теорема Клини (совпадение классов автоматных и регулярных языков)

Источники

- Хопкрофт Д., Мотвани Р., Ульман Д. "Введение в теорию автоматов, языков и вычислений", 2-е изд. : Пер. с англ. — М.:Издательский дом «Вильямс», 2002. — С. 149 — ISBN 5-8459-0261-4