Содержание

Аннотация

Как читать это пособие? Каждый раздел разбит условно на 3 части: основную, первую часть, которая, как правило повествуется на лекции; затем, опционально, идут комментарии к практике, а затем – некоторые важные ремарки и дополнительные примеры. Некоторые подразделы также могут быть разбиты на 3 части, как правило, это подразделы с большим объемом материала, по каждому из которых было отдельное занятие.

1 Языки и их свойства, операции над языками

1.1 Введение понятия языка

Назовём множество абстрактных объектов – символов – алфавитом Σ. Пусть алфавит конечный. Пустой и бесконечный алфавиты нам неинтересны.

Введём слово над алфавитом $\Sigma : w(A) = a_i, a_i \in \Sigma, \forall i = 0..|w(A)|$ – последовательность (строка) символов из алфавита $, 0 \leq |w(\Sigma)| < +\infty$.

Чтобы оперировать словами длины 0, вводят специальный символ длины $0 - \varepsilon : |\varepsilon^n| = 0, n = 0.. + \infty$; Его называют пустым.

Обозначим множество таких последовательностей из символов алфавита Σ, включая слово длины 0, как Σ^*.Тогда некоторый язык $L(\Sigma)$ над алфавитом Σ можно задать как подмножество слов над алфавитом: $Ł(\Sigma) \subseteq (\Sigma^*)$. Таким образом, математически мы определили объекты, с которыми будем работать, – это последовательности конечной длины и множества.

Теория формальных языков – математический способ конструктивного описания множеств последовательностей (слов) элементов некоторых множеств (алфавитов). Почему конструктивного? Потому что, в принципе, все слова языка можно просто перечислить, если:

1. любое слово – конечной длины.
2. множество слов конечно.
3. нет ограничений на временную сложность алгоритмов, используемых в работе с таким языком.

Нарушения 1) и 2), соответственно, говорят о том, что мы будем перечислять слова бесконечно, 3) это пожелание для применения на практике – нам нужны алгоритмы, которые работают, по крайней мере, за полином

небольшой степени и по времени, и по памяти, а лучше за линию, так как мы хотим иметь дело с относительно мощными языками, и нам важна масштабируемость.

В нашем курсе:

1. условие 1) будет всегда выполняться: считаем, что любое слово языка – конечной длины.

2. пусть 2) не выполняется, а 3) нас просят строго соблюсти.

Тогда задача конструктивного, то есть 'сжатого' и точного описания множества слов обретает куда более глубокую практическую значимость.

Кроме перечисления, можно предложить еще 2 способа задания языка, сконструировав:

1. Распознаватель – все слова языка можно распознать некоторой вычислительной машиной.

2. Генератор – все слова языка можно вывести посредством формальной процедуры переписывания строк по системе правил.

Система математических объектов, позволяющих и то и другое сделать, называется формальной грамматикой.

Опр. 1.1 *Грамматикой называется кортеж* $G = \langle \Sigma, N, R, S \rangle$, *где* Σ – *множество терминальных, или конечных символов,* N – *множество нетерминальных, или промежуточных символов,* $\Sigma \cap N = \emptyset$, $S \in N$ – *стартовый нетерминал,* $R = \{u \vdash w | u, w \in (\Sigma \cup N)^*\}$ – *множество правил вывода, то есть замены подстрок, входящих в слово в процессе его вывода/распознавания, такой, что язык* $L(G)$, *задаваемый* G, *получается посредством всевозможных применений правил* R, *начиная со строки из* S, *и заканчивая строками, состоящими только из терминалов:* $L(G) = \{\alpha \in T^* | S \vdash^* \alpha\}$, *где* $\vdash$ – *операция применения одного правила из* R, *то есть замены левой части правила, встреченной соответствующей позиции в строке, на правую, а* $\vdash^*$ – *последовательность таких применений различных правил из* R.

Как только мы перешли к понятию грамматики, мы получили конструктивный способ описания любого языка, даже бесконечного.

Итак, способы задать язык:

- Перечислить все элементы. Такой способ работает только для конечных языков.

- Задать распознаватель — процедуру, которая по данному слову может определить, принадлежит оно заданному языку или нет.

- Задать генератор — процедуру, которая возвращает очередное слово языка.

С последними двумя способами теория формальных языков и работает. Мы начнём с первого, в последствии переключимся на второй, а затем синхронно двинемся дальше с обеими способами, усложняя и рассматриваемые методы, подходы и задачи.

1.2 Операции над языками

Начнём с базовых операций над элементами языков – словами.

1.2.1 Операции над словами

Опр. 1.2 *Конкатенация – склеивание*[1] *строк. Если* $u = a_1 \dots a_m$ *и* $v = b_1 \dots b_n$ *— две строки, то их конкатенация — это строка* $u \cdot v = uv = a_1 \dots a_m b_1 \dots b_n$*. Знак* $\cdot$*, как правило, опускают.*

Конкатенация строки сама с собой обозначается как возведение в степень: w^n – n раз повторяемая w. $w^1 = w, w^0 = \varepsilon$, то есть конкатенация играет роль умножения с единицей ε, и превращает язык в свободную группу.

Следует заметить, что человечество активно работает над обобщением операции конкатенации, изобретая таким образом различные языки, призванные описывать неодномерные конструкции, например графы. Обобщение может выглядеть как, например, «подцепить к строке u строку w не справа, а сверху или снизу». Подробнее о таких языках можно узнать, например, в обзорной видеолекции по графовым грамматикам [15].

Опр. 1.3 *Взятие префикса – из любой строки* s *длины* l *можно взять префикс* $s[:n]$ *длины* $n, n \in 0..l$*,* $s[:0] = \varepsilon, s[:l] = s$*.*

Опр. 1.4 *Взятие суффикса* $s[n:]$ *вводится по аналогии с взятием префикса.*

Опр. 1.5 *Взятие подстроки* $s[n:m], n \leq m$ *можно ввести, например, как* $(s[n:])[:(m-n)]$*, либо* $(s[:m])[n:]$*.*

Конечно, существует множество других интересных, широкоиспользуемых либо экзотических операций, вроде инверсии слова, но оставим их за рамками данного пособия.

1.2.2 Операции над языками как множествами

Операции объединения, пересечения, вычитания, дополнения для языков определяются как для обычных множеств. Эти операции понадобятся нам, в особенности, при проверке свойств принадлежности языка некоторому классу.

[1] устоявшегося русского термина пока нет, увы

1.2.3 Операции над языками как множествами, содержащими последовательности

Опр. 1.6 *Конкатенация языков* $L_1(\Sigma_1), L_2(\Sigma_2) \subset (\Sigma_1 \cup \Sigma_2)^*$ *– это операция склеивания всех возможных слов языков:* $L_1 \cdot L_2 = \{uv | u \in L_1, v \in L_2\}$.

Конечно, можно сконкатенировать любое неотрицательное количество языков k. Если язык конкатенируют сам с собой, то это обозначают L^k. Для $k < 2$ операцию определяют так: если $k = 0$, то это будет язык $\{\varepsilon\}$, что соответствует определению $x^0 = 1$ для чисел. Если $k = 1$, то это будет сам L. Заметим, что конкатенация играет роль умножения[2].

Опр. 1.7 *Итерация языка* L *:* $L^* = \bigcup_{k=0}^{\infty} L^k$.

Заметим, что множество слов Σ^* – итерация языка Σ.

1.3 О приложениях теории формальных языков

В общем и целом, формальные языки имеют приложения таких направлениях науки и техники, как:

- Построение и синтаксический анализ языков программирования
- Построение автоматизированных систем управления [3]
- Извлечение информации из текста (на естественном или формальном языке) с учётом его синтаксической структуры
- Вычисление запросов к базам данных
- Анализ цепочек аминокислот
- Статический анализ ПО
- Прочее...

Здесь следует внести ремарку о моделях, применяемых для отдельных классов приложений. Существует огромное количество исследовательских работ, и немного меньшее количество промышленных реализаций обобщений «обыкновенных» методов из теории формальных языков для строк.

Так, к примеру,зачастую для запросов к графовым БД и иногда для статического анализа ПО используют обобщение КС-языков со строк на графы. В химии и биоинформатике исторически «любят» грамматики, описывающие не только последовательности, но деревья и даже графы, и т.д.

[2] это и правда умножение в некотором полукольце с единицей ε (вопрос: а какая операция – сложение в этом полукольце?)

[3] особенно широко применяются автоматные языки, выразительно эквивалентные вычислительным машинам с конечным числом состояний

2 Конечные автоматы

Конечный автомат – математическая модель вычислителя с конечной памятью.

Опр. 2.1 *Недетерминированный конечный автомат (НКА) – это кортеж* $\langle Q, \Sigma, \Delta, q_0, F \rangle$*:*

- $Q, |Q| < \infty$ – *множество состояний*
- Σ – *алфавит*
- $\Delta \subset Q \times \Sigma^* \times Q$ – *множество переходов*[4]
- $q_0 \in Q$ – *стартовое состояние*
- $F \subset Q$ – *множество финальных состояний*

Существует эквивалентное определение автомата, где вместо Δ задают функцию перехода $\delta : Q \times \Sigma^* \to 2^Q$; будем пользоваться «более графовым» определением через Δ, хотя функция перехода нам ещё понадобится.

Способ распознавания строки автоматом лежит в его определении: представим граф автомата. Вершины – это состояния, рёбра – переходы. Если мы находимся в стартовом состоянии, и нам подают на вход строку, то нам достаточно брать по символу/слову из Σ^*, смотреть, по каким рёбрам графа мы можем перейти (если ε – перейти можем спонтанно), совершать переход(ы), брать следующий символ/слово из Σ^*, смотреть, куда мы по нему можем перейти из текущего состояния, и так далее. Слово распозналось, если мы дошли до какого либо финишного состояния и обработали всё слово. Итого, распознавание строки автоматом – суть проверка достижимости по рёбрам его графа из q_0 в одно из состояний в F.

Основным недостатком КА служит то, что мы в каждый момент времени знаем только текущее состояние и в какие мы можем из него перейти. У нас нет данных о том, что происходило ранее, и это накладывает ограничения на выразительность[5]. К примеру, нельзя составить КА, распознающий язык $a^n b^n, \forall n \in [0, +\infty)$, хотя для любого фиксированного множества n – можно (Рис. 1).

О достижимости проще говорить в терминах пар $\langle q_x, v \rangle \in Q \times \Sigma^*$, где q_x – текущее состояние, а v – недоразобранная подстрока входной строки. Такая пара называется конфигурацией автомата[6]. Введём отношение достижимости на конфигурациях.

Опр. 2.2 *Достижимость* ($\vdash$) – *наименьшее рефлексивное транзитивное отношение над* $Q \times \Sigma^*$*, такое что:*

[4] Δ задаёт множество двухместных отношений на Q, помеченных элементами Σ^*.

[5] тем не менее, конечные автоматы широко применяются в технике вокруг нас. Примеры: светофор, лифт, кодовый замок, система контроля воздуха в помещении, компьютерная мышь, аудиоплеер, веб-форма и т.д.

[6] по мере усложнения моделей вычислителей, мы будем добавлять новые параметры в конфигурацию – например, появится параметр, описывающий стек, и т.д.

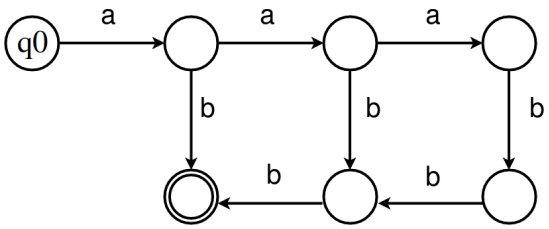


Рис. 1: КА, распознающий язык $a^n b^n, n \in [1, 3]$

1. $\forall w \in \Sigma^* : (\langle q_1, w\rangle \to q_2) \in \Delta \Rightarrow \langle q_1, w\rangle \Rightarrow \langle q_2, \varepsilon\rangle$
2. $\forall u, v \in \Sigma^* : \langle q_1, u\rangle \vdash \langle q_2, \varepsilon\rangle, \langle q_2, v\rangle \vdash \langle q_3, \varepsilon\rangle \Rightarrow \langle q_1, uv\rangle \vdash \langle q_3, \varepsilon\rangle$
3. $\forall u \in \Sigma^* : \langle q_1, u\rangle \vdash \langle q_2, \varepsilon\rangle \Rightarrow \forall v \in \Sigma^* \langle q_1, uv\rangle \vdash \langle q_2, v\rangle$

Используя это определение, несложно задать язык, распознаваемый КА.

Опр. 2.3 *Пусть дан $M = \langle Q, \Sigma, \Delta, q_0, F\rangle$. Язык, распознаваемый автоматом M – $L(M) = \{w \in \Sigma^* | \exists q \in F : \langle q_0, w\rangle \vdash \langle q, \varepsilon\rangle\}$.*

Опр. 2.4 *Язык L называется автоматным, если существует КА M : $L = L(M)$. Множество таких языков L образует класс автоматных языков.*

На практике гораздо приятнее работать с детерминированным конечным автоматом (ДКА).

Опр. 2.5 *(Неформально) НКА $M = \langle Q, \Sigma, \Delta, q_0, F\rangle$ называется детерминированным КА, если*

- *Все переходы – однобуквенные: $\forall(\langle q_1, w\rangle \to q_2) \in \Delta : |w| = 1$*
- *$\forall a \in \Sigma, q \in Q |\delta(q, a)| \leq 1$, где $\delta(q, a)$ – множество состояний, достижимых из q по символу a. Задание: расписать $\delta(q, w)$ аккуратнее через конфигурации.*

Иными словами, для любых фиксированных букв, для любого состояния, переход приводит только в одно результирующее состояние.

Можно ввести ДКА-автоматный язык L_{NFA} по аналогии с тем, как вводили $L(M) = L_{DFA}$. Очевидно, что $L_{DFA} \subseteq L_{NFA}$, так как ДКА – это частный случай НКА.

Если мы покажем, что произвольный НКА сводится к ДКА, то $L_{DFA} = L_{NFA}$.

2.1 Сведение НКА к ДКА

Л. 2.1 *(«Построение подмножеств», Рабин и Скотт [1959]). Пусть* $B = (\Sigma, Q, q_0, \Delta, F)$ *— произвольный. Тогда* $\exists$ *DFA* $A = (\Sigma, 2^Q, Q_0, \Delta', F')$*, состояния которого — помножества* Q*, который распознаёт тот же язык, что и* B*. Его переход в каждом состоянии-подмножестве* $s \subseteq Q$ *по каждому символу* $a \in \Sigma$ *ведет во множество состояний, достижимых по* a *из некоторого состояния* s*.*

Произведём серию упрощений НКА.

Утв. 2.1 *В определении НКА можно считать все переходы – однобуквенными. Для этого нужно перестроить множества* Δ *и* Q*.*

Утв. 2.2 *В определении НКА можно считать* $|F| = 1$*.*

Утв. 2.3 *(ε-замыкание) От переходов по* ε *можно избавиться, применив некоторые преобразования (см.Рис. 2).*

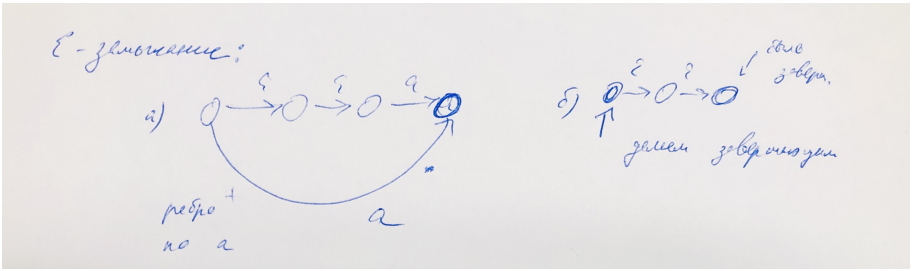


Рис. 2: Основные преобразования при построении ε-замыкания: последовательность переходов $\varepsilon \dots \varepsilon a$ заменить на переход a (а), состояние, из которого существует переход $\varepsilon \dots \varepsilon$ в финальное состояние – обозначить как финальное (б)

Эти утверждения доказываются технически, не будем этим заниматься сейчас (рекомендуется попробовать доказать дома или посмотреть в классических книгах и курсах).

TODO: доказательство Л2.1, алгоритм на базе метода «построение подмножеств»

Утв. 2.4 *(о корректности Л2.1). Для любой строки* $w \in \Sigma^*$*, состояние-подмножество, достигаемое DFA по прочтении строки* w*, содержит элемент* q *тогда и только тогда, когда хотя бы одно из вычислений NFA на* w *заканчивается в состоянии* q*.*

Доказывается индукцией по длине строки w.

Далее из утверждения о правильности выводится, что построенный DFA распознаёт строку $w \in \Sigma^*$ тогда и только тогда, когда распознаёт исходный

NFA. Построение переводит NFA с n состояниями в DFA с 2^n состояниями-подмножествами. На практике, многие из них обычно бывают недостижимы. Поэтому хороший алгоритм должен строить только подмножества, достижимые из уже построенных, начиная с q_0.

2.2 Минимизация ДКА

Говорят, что состояния u, v различаются словом s, если одно из них по s переводит автомат в финальное состояние, а другое нет.

Если состояния не различаются никакой строкой, они называются неразличимыми. На Рис. 3 изображен ДКА, в котором есть такие: действительно, окажемся мы в финальном состоянии или нет, зависит только от количества нулей в строке, следовательно, B и C – неразличимы.

Л. 2.2 *Отношение неразличимости суть отношение эквивалентности.*

Рефлексивность очевидна, симметричность следует из определения (попробуйте заменить u и v местами).

Транзитивность: u и v неразличимы, v и w неразличимы, следовательно, u и w неразличимы, тоже очевидно.

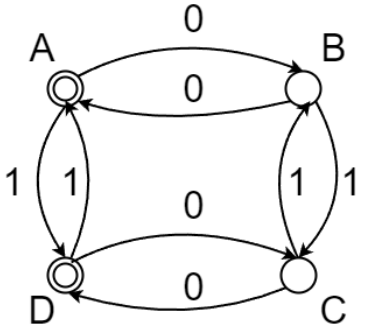


Рис. 3: ДКА, в котором есть неразличимые состояния (найдите их)

По индукции по длине строки доказывается, что модификация автомата как на Рис. 3, если состояния A и B не различимы, не меняет распознаваемый им язык.

Повторяя процедуру модификации для всех классов эквивалентности, оставляя какую-то одну вершину для каждого класса, получим некий автомат с возможно меньшим числом состояний. Можно доказать, что это число состояний – минимально.

Л. 2.3 *Пусть у ДКА M все состояния различимы и любое достижимо из стартового. Тогда M – минимальный автомат для $L(M)$*

Т. 2.1 *Для любого ДКА существует и единственный с точностью до изоморфизма ДКА с минимальным числом состояний.*

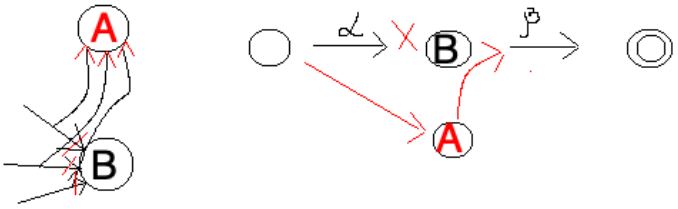


Рис. 4: Вспомогательный рисунок

Интуитивно, для выполнения минимизации нужно выделить:

- Недостижимые состояния – их нужно удалить[7]
- Неразличимые состояния – их можно объединить в одно для каждого класса эквивалентности

Существует, как минимум, 3 способа выделить и объединить неразличимые состояния:

- Наивный алгоритм основан на построении классов эквивалентности и объединении эквивалентных состояний [2], и рассматривается на семинаре. Он работает за $O(n^2)$.
- Алгоритм Хопкрофта, позволяющий решить задачу за $O(nlog(n))$ [3].
- Также существует алгоритм Бржозовского, который строит минимальный ДКА и из НКА [4]

3 Регулярные выражения и языки

3.1 Регулярные выражения

Опр. 3.1 *(Клини [1951]). Регулярные выражения над алфавитом* Σ *определяются так:*

- ε *— регулярное выражение.*
- *Всякий символ* a*, где* $a \in \Sigma$ *— регулярное выражение.*
- *Если* α, β *— регулярные выражения, то тогда* $(\alpha|\beta), (\alpha\beta)$ *и* $(\alpha)^*$ *— тоже регулярные выражения.*

[7] если этого еще не сделали на этапе построения ДКА, то можно обойти его граф из стартового состояния, например, в глубину, и собрать список достижимых состояний, а недостижимые удалить, модифицируя при этом необходимые элементы автомата

Всякое регулярное выражение α определяет язык над алфавитом Σ, обозначаемый через $L(\alpha)$.

Всякий символ из Σ обозначает одноэлементное множество, состоящее из односимвольной строки: $L(a) = \{a\}$

Оператор выбора задает объединение множеств: $L(\alpha|\beta) = L(\alpha) \cup L(\beta)$.

Конкатенация задает конкатенацию языков: $L(\alpha\beta) = L(\alpha)L(\beta)$.

Символ ε определяет пустое множество.

Оператор итерации задает итерацию: $L(\alpha^*) = L(\alpha)^*$.

Приоритеты операций: сперва итерация, затем конкатенация, затем выбор.

Синтаксис регулярных выражений на практике часто расширяется, к примеру:

- повторение один и более раз $(\alpha+)$, $(\alpha+) = \alpha\alpha^*$
- необязательная конструкция ($[\alpha]$, что означает «α или ничего»), $[\alpha] = \alpha|\varepsilon = \alpha|\varepsilon^*$

Л. 3.1 *(«построение Томпсона»). Для всякого регулярного выражения α, существует NFA C_α с одним начальным и одним принимающим состояниями, распознающий язык, задаваемый α.*

Доказательство производится индукцией по структуре регулярного выражения, структурные единицы представлены на Рис. 5.

Схема пошагового перехода от регулярного выражения к ДКА:

$$regexp \to NFA \to NFA_{simplified} \to DFA \to DFA_{min}, \tag{1}$$

была разобрана в разделах 1-3. Данная схема лежит в основе подавляющего большинства библиотек для работы с регулярными выражениями.

3.2 Регулярные языки

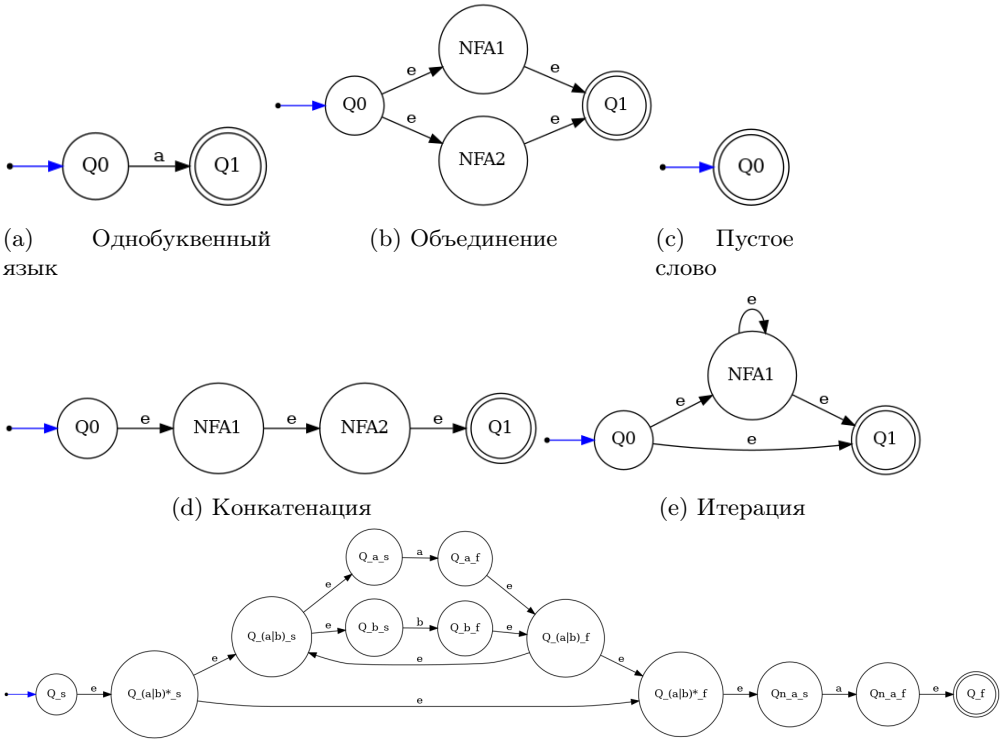
Любое регулярное выражение $reg(\Sigma)$ над алфавитом Σ – формула, задающая регулярный язык L_{reg}.

Утв. 3.1 *Любой регулярный язык задаётся грамматикой $\langle \Sigma, N, S, P \rangle$, где правила из P имеют вид $A \to a, A \to \gamma, A \to \varepsilon$, где γ - либо aB (правая регулярная грамматика), либо Ba (левая регулярная грамматика), $a \in \Sigma$, $A, B, S \in N$.*

3.2.1 Свойства замкнутости регулярных языков

Операции, сохраняющие регулярность (без доказательств): объединение, пересечение, дополнение, разность, обращение, итерация, конкатенация, гомоморфизм, обратный гомоморфизм.

Так как, по определенению, класс регулярных языков замкнут относительно этих операций, то и композиция этих операций даёт и регулярный язык, и НКА, его распознающий.



(a) Однобуквенный язык

(b) Объединение

(c) Пустое слово

(d) Конкатенация

(e) Итерация

(f) Пример: композиция для (a|b)*a

Рис. 5: Базовые автоматы построения Томпсона и пример композиции

3.2.2 Проверка на нерегулярность

Лемма о накачке (разрастании)

3.3 Регулярные выражения на практике

Итак, согласно теории, любое регулярное выражение над алфавитом Σ задает регулярный язык над этим алфавитом. Тем не менее, так ли регулярны языки, задаваемые регулярными выражениями, реализованными в современных ЯП?... Короткий ответ – нет. Как правило, в инженерной практике программирования «регулярными выражениями» называют то или иное вольное расширение выражений из исходного определения, имеющее больше выразительных возможностей, чем то, что мы рассмотрели: в них есть специфические операции, например, нумерованные обратные ссылки и т.д., позволяющие задавать не только регулярные языки.

Пример (из реализации PCRE в Perl)): `/^(a(?1)?b)$/` задаёт язык $a^n b^n, n \in [1 \dots \infty)$

Это регулярное выражение очень простое: `(?1)` ссылается на первую подмаску — `(a(?1)?b)`. Можно заменить `(?1)` подмасками, формируя таким образом рекурсивную зависимость:

```
/^(a(?1)?b)$/
/^(a(a(?1)?b)?b)$/
/^(a(a(a(?1)?b)?b)?b)$/
/^(a(a(a(a(?1)?b)?b)?b)?b)$/
```

...

Очевидно, это выражение способно описать любую строку с одинаковым количеством a и b, но конечный автомат, распознающий язык всех таких строк, построить нельзя.

4 Лексический анализ

Следующее приложение, о котором мы будем говорить – лексический анализ – это выделение во входном тексте характерных подстрок, «значащих» что-то, для дальнейших действий.

Опр. 4.1 *Лексема – последовательность символов, удовлетворяющая некоторому заданному требованию.*

Основная проблема выделения лексем – их может быть много и разных. Давайте работать не с лексемами, а с их «классами», на которые они делятся по смыслу нашей задачи[8].

Опр. 4.2 *Токен – последовательность символов, «осмысленно» описывающая класс некоторой лексемы.*

[8] Здесь считаем такую классификацию однозначной.

Пример: $int \to TYPE$ (int – лексема, $TYPE$ – токен).
Для задания токенов, как правило, используют регулярные выражения.

Опр. 4.3 *Лексер, лексический анализатор, сканер – транслятор, преобразующий входную строку в последовательность токенов.*

4.1 Комментарии к практике

- Примеры работы с генератором лексических анализаторов `flex` были приведены на семинаре.
- В контесте 2 есть задачи, подразумевающие генерацию лексера по спецификации. И еще есть задача, которая демонстрирует, что в частных случаях («найти все вхождения слов в некоторый текст», «найти слово наименьшей длины, содержащее все подслова данного», и т.д.) можно, но не нужно писать регулярки, а лучше строить автомат по известной заранее структуре[9].
- Существует ряд подходов к оптимизации представления регулярных выражений, например, префиксное сжатие [6] и пр. Понятно, что в случае компиляции в минимальный ДКА для дальнейшего использования, этот подход никакого выигрыша в производительности не даст, так как ДКА будет одним и тем же с точностью до изоморфизма. Тем не менее, такой подход может повлиять на производительность промежуточных преобразований автоматов, так как НКА, полученный с оптимизацией, может отличаться от такового без оптимизации.

Итого:

- В практических приложениях обычно используют библиотеки регулярных выражений или встроенные средства ЯП. Эти средства, как правило, реализуются по схеме, описанной выше, за исключением некоторых технических нюансов и ухищрений.
- При этом в лексическом анализе ЯП зачастую используют ручное написание лексеров. По крайней мере, в некоторых промышленных компиляторах (например, в Clang или OpenArkCompiler лексеры написаны вручную). Почему так – рассказано в разделе про компиляторы.
- Тем не менее, лексический анализ – довольно общая задача, и существуют инструменты построения лексеров по спецификациям, например, flex.
- В очень частных случаях («найти все вхождения слов в некоторый текст», «найти слово наименьшей длины, содержащее все подслова данного», и т.д.) можно, но не нужно писать регулярки, а лучше строить автомат по известной заранее структуре.

[9] Например, суффиксный бор в случае с алгоритмом Ахо-Корасик (1975)

5 КС-грамматики и языки

5.1 Грамматики как системы переписывания

Опр. 5.1 *Формальная грамматика – кортеж* $G = (\Sigma, N, R, S)$*:*

- Σ *– терминальный алфавит – алфавит определяемого языка.*
- N *– нетерминальный алфавит*[10] *– алфавит промежуточных символов.*
- *Конечное множество правил* R *вида* $\alpha \to \beta, \alpha \in \{\Sigma \cup N\}^*, \beta \in \{\Sigma \cup N\}^* \cup \{\varepsilon\}$ *– каждое из которых описывает возможную структуру строк* β *со свойством* α*.*
- *Начальный символ* $S \in N$*.*

Грамматика при этом является системой переписывания строк, и системой порождения слов языка, где каждое слово порождается за конечное число шагов. Шаг порождения $w'\alpha w'' \to w'\beta w''$ состоит в замене α на подцепочку β в соответствии с правилом порождения $\alpha \to \beta$. Иначе говоря, если имеется некоторая цепочка и некоторая ее подцепочка является левой частью какого-то правила грамматики, то мы имеем право заменить эту левую часть правила на правую. Конечная последовательность шагов порождений называется порождением. Нуль или более порождений будет обозначать знаком $\to^*$. Обозначение $\alpha \to^* \beta$ говорит о том, что цепочка β получена из цепочки α конечным числом подстановок на основе правил порождения. В этом обозначении может быть так, что подстановка не была применена ни разу, в этом случае цепочка $alpha = beta$.

Язык, задаваемый (порождаемый) грамматикой G – это множество слов, составленных из терминальных символов и порожденных из начального символа грамматики $L = \{w | S \to^* w\}$.

Опр. 5.2 *Грамматики* G_1 *и* G_2 *называются эквивалентными, если они задают один и тот же язык:* $L(G_1) = L(G_2)$

Понятие регулярной грамматики уже вводилось в разделе 3. Ниже будет введено понятие контекстно-свободной грамматики. Эти два типа грамматик являются наиболее исследованными типами иерархии Хомского (типами 3 и 2 соответственно), о которой мы будем говорить позже, и наиболее интересными нам в данном курсе.

5.1.1 Выводимость в грамматике

Опр. 5.3 *Отношение непосредственной выводимости. Последовательность терминалов и нетерминалов* $\gamma\alpha\delta$ *непосредственно выводится из* $\gamma\beta\delta$ *при помощи правила* $\alpha \to \beta$ *(*$\gamma\alpha\delta \Rightarrow \gamma\beta\delta$*), если*

[10] В лингвистике нетерминалы называются синтаксическими категориями

- $\alpha \to \beta \in P$
- $\gamma, \delta \in \{\Sigma \cup N\}^* \cup \varepsilon$

Опр. 5.4 *Рефлексивно-транзитивное замыкание отношения — это наименьшее рефлексивное и транзитивное отношение, содержащее исходное.*

Опр. 5.5 *Отношение выводимости является рефлексивно-транзитивным замыканием отношения непосредственной выводимости*

- $\alpha\beta$ *означает* $\exists\gamma_0, \dots \gamma_k : \ \alpha[]\gamma_0[]\gamma_1[] \dots []\gamma_{k-1}[]\gamma_k[]\beta$
- *Транзитивность:* $\forall\alpha, \beta, \gamma \in \{\Sigma \cup N\}^* \cup \varepsilon : \ \alpha\beta, \beta\gamma \Rightarrow \alpha\gamma$
- *Рефлексивность:* $\forall\alpha \in \{\Sigma \cup N\}^* \cup \varepsilon : \ \alpha\alpha$
- $\alpha\beta$ *—* α *выводится из* β
- $\alpha[k]\beta$ *—* α *выводится из* β *за* k *шагов*
- $\alpha[+]\beta$ *— при выводе использовалось хотя бы одно правило грамматики*

Опр. 5.6 (Вывод слова в грамматике) *Слово* $\omega \in \Sigma^*$ *выводимо в грамматике* $\langle \Sigma, N, P, S \rangle$*, если существует некоторый вывод этого слова из начального нетерминала* $S\omega$*.*

Частные случаи вывода:

Опр. 5.7 *Левосторонний вывод. На каждом шаге вывода заменяется самый левый нетерминал.*

Опр. 5.8 *Правосторонний вывод. На каждом шаге вывода заменяется самый правый нетерминал.*

5.2 КС-грамматики

Опр. 5.9 *Контекстно-свободная грамматика – кортеж* $G = (\Sigma, N, R, S)$*:*

- Σ *– терминальный алфавит.*
- N *– нетерминальный алфавит.*
- *Конечное множество правил* R *вида* $N_i \to \alpha, N_i \in N, \alpha \in \{\Sigma \cup N\}^* \cup \{\varepsilon\}$
- *Начальный символ* $S \in N$*.*

То есть, исходя из общего определения[11] формальной грамматики (5.1), КС грамматика – такая грамматика, в которой каждое правило порождения позволяет явно установить свойство подстроки как промежуточный символ, либо вывести подстроку с заданным свойством только из промежуточного символа, вне зависимости от того, что стоит слева или справа в строке в процессе переписывания. Далее будем называть промежуточные символы нетерминальными, и, чтобы не было путаницы, потребуем $\Sigma \cap N = \emptyset$.

При спецификации грамматики часто опускают множества терминалов и нетерминалов, оставляя только множество правил. При этом нетерминалы часто обозначаются прописными латинскими буквами, терминалы — строчными, а стартовый нетерминал обозначается буквой S. Мы будем следовать этим обозначениям, если не указано иное.

Опр. 5.10 *Грамматика называется однозначной, если для любого порождённого по ней слова последовательность порождения – единственна.*

Иными словами, для слова, порождаемого однозначной КС-грамматикой, существует единственное дерево разбора.

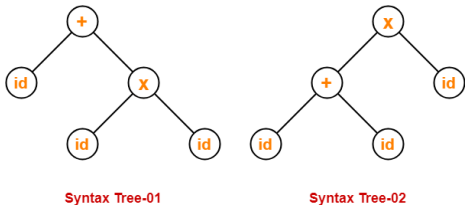


Рис. 6: Неоднозначный разбор арифметического выражения

Соответственно, множество языков, порождаемое КС-грамматиками, называется КС-языками.

Л. 5.1 (Лемма о накачке для КС-языков) *Для каждого КС-языка $L \subseteq \Sigma^*$ существует такая константа $p \geq 1$, что для любой строки $w \in L$, для которой $|w| > p$, существует разложение $w = xuyvz$, где $|uv| > 0$ и $|uyv| \leq p$, для которого $xu^iyv^iz \in L$ при всех $i \geq 0$.*

5.2.1 Формы представления КС-грамматик

В зависимости от вида правил, КС-грамматики подразделяются на формы, свойства и алгоритмы анализа которых зачастую существенно различаются. Опишем некоторые из форм, которыми будем пользоваться.

Опр. 5.11 *Грамматика находится в Нормальной форме Хомского (НФХ, CNF), если любое правило имеет один из трех видов:*

[11] и значения

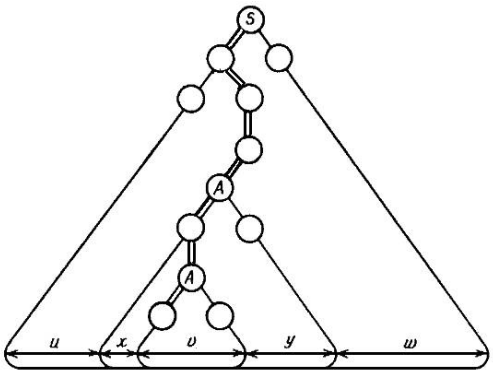


Рис. 7: К лемме о накачке: структура дерева вывода для uvwxy

1. $S \to \varepsilon$
2. $N_i \to N_j N_k, N_i, N_j, N_k \in N$
3. $N_i \to t, N_i \in N, t \in A$

Замечание: в НФХ стартовый нетерминал не встречается в правых частях правил, ε-правила только для стартового нетерминала.

Опр. 5.12 *Грамматика находится в Ослабленной Нормальной форме Хомского (weak-CNF), если...*

Л. 5.2 *Любую КС-грамматику можно привести к НФХ.*

Алгоритм приведения к НФХ был разобран на семинаре.

5.2.2 Об алгоритмах синтаксического анализа КС-языков

КС-языки, наравне с регулярными – наиболее полно исследованный класс формальных языков, для которых существует целое разнообразие алгоритмов разбора различной сложности. На практике, в особенности при анализе языков программирования, основным требованием к алгоритму разбора является его вычислительная эффективность, даже если он не годится для произвольных КС-грамматик. Поэтому зачастую применяются алгоритмы с временной сложностью порядка $O(n)$ на слове длины n, в частности, алгоритмы LL, LR-семейства, которые рассмотрены далее в соответствующих разделах.

Мы же начнём повествование с алгоритма кубичной сложности, позволяющего осуществлять разбор слов, порождаемых КС-грамматиками (даже неоднозначными), заданными в специальной форме, к которой можно привести любую КС-грамматику (размер полученной грамматики – количества правил и нетерминалов – при этом может сильно разрастаться по сравнению с исходной) – нормальной форме Хомского. Алгоритм носит имя создателей – Кока, Янгера и Касами (CYK)[7].

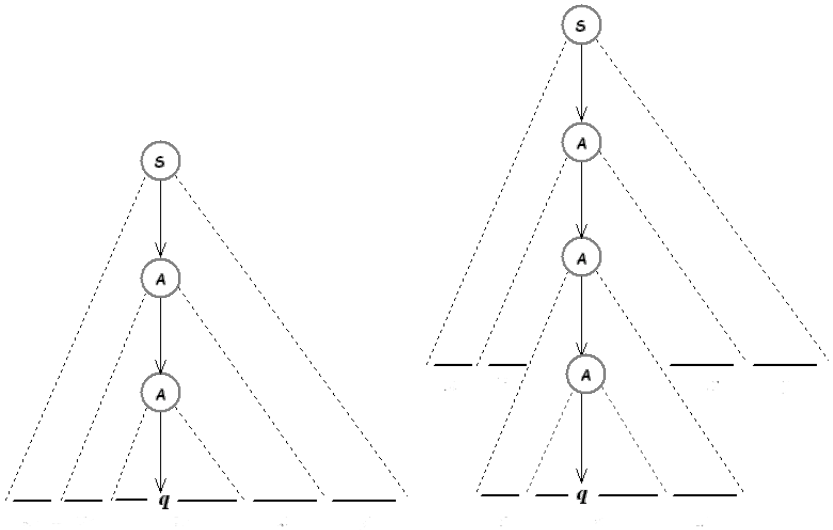


Рис. 8: К доказательству леммы о накачке: «1 шаг накачки».

5.2.3 Алгоритм CYK для строк

CYK (Кока, Янгера и Касами, Cocke-Younger-Kasami) — один из классических алгоритмов синтаксического анализа. Его асимптотическая сложность в худшем случае — $O(n^3 \cdot |N|)$, где n — длина входной строки, а N — количество нетерминалов во входной граммтике [**?**].

Для его применения необходимо, чтобы подаваемая на вход грамматика находилась в Нормальной Форме Хомского (НФХ). Других ограничений нет, следовательно,данный алгоритм применим для работы с произвольными контекстно-своболными языками.

В основе алгоритма лежит принцип динамического программирования. Алгоритм строится из следующих соображений:

1. Для правила вида $A \to a$ очевидно, что из A выводится ω (с применением этого правила) тогда и только тогда, когда $a = \omega$:

$$A\omega \iff \omega = a$$

2. Для правила вида $A \to BC$ понятно, что из A выводится ω (с применением этого правила) тогда и только тогда, когда существуют две цепочки ω_1 и ω_2 такие, что ω_1 выводима из B, ω_2 выводима из C и при этом $\omega = \omega_1\omega_2$:

$$A[]BC\omega \iff \exists\omega_1, \omega_2 : \omega = \omega_1\omega_2, B\omega_1, C\omega_2$$

   Или в терминах позиций в строке:

$$A[]BC\omega \iff \exists k \in [1 \dots |\omega|] : B\omega[1 \dots k], C\omega[k+1 \dots |\omega|]$$

В процессе работы алгоритма заполняется булева трехмерная[12] матрица M размера $n \times n \times |N|$ таким образом, что

$$M[i, j, A] = true \iff A\omega[i \dots j]$$

.

Первым шагом инициализируем матрицу, заполнив значения $M[i, i, A]$:

- $M[i, i, A] = true$, если в грамматике есть правило $A \to \omega[i]$.
- $M[i, i, A] = false$, иначе.

Далее используем динамику: на шаге $m > 1$ предполагаем, что ячейки матрицы $M[i', j', A]$ заполнены для всех нетерминалов A и пар $i', j' : j' - i' < m$. Тогда можно заполнить ячейки матрицы $M[i, j, A]$, где $j - i = m$ следующим образом:

$$M[i, j, A] = \bigvee_{A \to BC} \bigvee_{k=i}^{j-1} M[i, k, B] \wedge M[k, j, C]$$

По итогу работы алгоритма значение в ячейке $M[0, |\omega|, S]$, где S — стартовый нетерминал грамматики, отвечает на вопрос о выводимости цепочки ω в грамматике.

Рассмотрим пример работы алгоритма CYK на грамматике правильных скобочных последовательностей в Нормальной Форме Хомского.

Пример.

Пусть дана грамматика:

$$\begin{array}{lll} S \to AS_2 \mid \varepsilon & S_2 \to b \mid BS_1 \mid S_1S_3 & A \to a \\ S_1 \to AS_2 & S_3 \to b \mid BS_1 & B \to b \end{array}$$

Проверим выводимость строки $\omega = aabbab$ в ней.

Будем иллюстрировать работу алгоритма двумерными матрицами размера $n \times n$, где в ячейках указано множество нетерминалов, выводящих соответствующую подстроку.

Шаг 1. Инициализируем матрицу элементами на главной диагонали:

$$\begin{pmatrix} \{A\} & & & & & \\ & \{A\} & & & & \\ & & \{B, S_2, S_3\} & & & \\ & & & \{B, S_2, S_3\} & & \\ & & & & \{A\} & \\ & & & & & \{B, S_2, S_3\} \end{pmatrix}$$

[12] Можно считать матрицу двухмерной, а не трехмерной булевой, но содержащей в элементах не биты, а списки соответствующих нетерминалов, как показано ниже в примере

Шаг 2. Заполняем диагональ, находящуюся над главной:

$$\begin{pmatrix} \{A\} & & & & & \\ & \{A\} & lightgray\{S_1\} & & & \\ & & \{B,S_2,S_3\} & & & \\ & & & \{B,S_2,S_3\} & & \\ & & & & \{A\} & lightgray\{S_1\} \\ & & & & & \{B,S_2,S_3\} \end{pmatrix}$$

В двух ячейках появились нетерминалы S_1 благодаря присутствию в грамматике правила $S_1 \to AS_2$.

Шаг 3. Заполняем следующую диагональ:

$$\begin{pmatrix} \{A\} & & & & & \\ & \{A\} & \{S_1\} & red\{S_2\} & & \\ & & \{B,S_2,S_3\} & & & \\ & & & \{B,S_2,S_3\} & & lightgray\{S_2,S_3\} \\ & & & & \{A\} & \{S_1\} \\ & & & & & \{B,S_2,S_3\} \end{pmatrix}$$

Шаг 4. И следующую за ней:

$$\begin{pmatrix} \{A\} & & & lightgray\{S_1,S\} & & \\ & \{A\} & \{S_1\} & \{S_2\} & & \\ & & \{B,S_2,S_3\} & & & \\ & & & \{B,S_2,S_3\} & & \{S_2,S_3\} \\ & & & & \{A\} & \{S_1\} \\ & & & & & \{B,S_2,S_3\} \end{pmatrix}$$

Шаг 5 Заполняем предпоследнюю диагональ:

$$\begin{pmatrix} \{A\} & & & \{S_1,S\} & & \\ & \{A\} & \{S_1\} & \{S_2\} & & lightgray\{S_2\} \\ & & \{B,S_2,S_3\} & & & \\ & & & \{B,S_2,S_3\} & & \{S_2,S_3\} \\ & & & & \{A\} & \{S_1\} \\ & & & & & \{B,S_2,S_3\} \end{pmatrix}$$

Шаг 6. Завершаем выполнение алгоритма:

$$\begin{pmatrix} \{A\} & & & \{S_1,S\} & & lightgray\{S_1,S\} \\ & \{A\} & \{S_1\} & \{S_2\} & & \{S_2\} \\ & & \{B,S_2,S_3\} & & & \\ & & & \{B,S_2,S_3\} & & \{S_2,S_3\} \\ & & & & \{A\} & \{S_1\} \\ & & & & & \{B,S_2,S_3\} \end{pmatrix}$$

Стартовый нетерминал находится в верхней правой ячейке, а значит цепочка $aabbab$ выводима в нашей грамматике.

Теперь выполним алгоритм на цепочке $\omega = abaa$.

Шаг 1. Инициализируем таблицу:

$$\begin{pmatrix} \{A\} & & & \\ & \{B, S_2, S_3\} & & \\ & & \{A\} & \\ & & & \{A\} \end{pmatrix}$$

Шаг 2. Заполняем следующую диагональ:

$$\begin{pmatrix} \{A\} & lightgray\{S_1, S\} & & \\ & \{B, S_2, S_3\} & & \\ & & \{A\} & \\ & & & \{A\} \end{pmatrix}$$

Больше ни одну ячейку в таблице заполнить нельзя и при этом стартовый нетерминал отсутствует в правой верхней ячейке, а значит эта строка не выводится в данной грамматике.

5.3 Обобщение синтаксического анализа со строк на графы

5.3.1 CYK для графов

Первым шагом на пути к обобщению CYK для поиска путей, задаваемых языками меток рёбер на графах, является модификация представления входа. Прежде мы сопоставляли каждому символу слова его позицию во входной цепочке, поэтому при инициализации заполняли главную диагональ матрицы. Вместо этого, обозначим числами позиции между символами. В результате, слово можно представить в виде линейного графа следующим образом(в качестве примера рассмотрим слово $aabbab$):

$$0 \xrightarrow{a} 1 \xrightarrow{a} 2 \xrightarrow{b} 3 \xrightarrow{b} 4 \xrightarrow{a} 5 \xrightarrow{b} 6$$

Что нужно изменить в описании алгоритма, чтобы он продолжал работать при подобной нумерации? Каждая буква теперь идентифицируется не одним числом, а парой — номера слева и справа от нее. При этом чисел стало на одно больше, чем при прежнем способе нумерации.

Возьмем матрицу $(n+1) \times (n+1) \times |N|$ и при инициализации будем заполнять не главную диагональ, а диагональ прямо над ней. Таким образом, мы начинаем наш алгоритм с определения значений $M[i, j, A]$, где $j = i+1$. При этом наши дальнейшие действия в рамках алгоритма не изменятся.

Для примера 5.2.3 на шаге инициализации матрица выглядит следующим образом:

$$\begin{pmatrix} \{A\} & & & & & \\ & \{A\} & & & & \\ & & \{B, S_2, S_3\} & & & \\ & & & \{B, S_2, S_3\} & & \\ & & & & \{A\} & \\ & & & & & \{B, S_2, S_3\} \end{pmatrix}$$

А в результате работы алгоритма имеем:

$$\begin{pmatrix} \{A\} & & & \{S_1, S\} & & \{S_1, S\} \\ & \{A\} & \{S_1\} & \{S_2\} & & \{S_2\} \\ & & \{B, S_2, S_3\} & & & \\ & & & \{B, S_2, S_3\} & & \{S_2, S_3\} \\ & & & & \{A\} & \{S_1\} \\ & & & & & \{B, S_2, S_3\} \end{pmatrix}$$

Мы представили входную строку в виде линейного графа, а на шаге инициализации получили его матрицу смежности. Добавление нового нетерминала в язейку матрицы можно рассматривать как нахождение нового пути между соответствующими вершинами, выводимого из добавленного нетерминала. Таким образом, шаги алгоритма напоминают построение транзитивного замыкания графа. Различие заключается в том, что мы добавляем новые ребра только для тех пар нетерминалов, для которых существует соответстующее правило в грамматике.

Алгоритм можно обобщить и на произвольные графы с метками, рассматриваемые в этом курсе. При этом можно ослабить ограничение на форму входной грамматики: она должна находиться в ослабленной Нормальной Форме Хомского.

5.3.2 Алгоритм (Y. Hellings, 2015) с рабочими множествами для графов

Можно заметить, что CYK производит много избыточных итераций. Можно модифицировать алгоритм, чтобы не просматривались заведомо пустые ячейки. Данная модификация была предложена Хеллингсом [9] в именном алгоритме, но также фигурирует и в более ранних работах [8]. В основе алгоритма лежит обработка двух рабочих множеств: текущего и конечного.

Идеологически, на каждом шаге алгоритма:

- Просматривается какой-то путь, полученный на текущем шаге.
- Нужно попробовать приконкатенировать к нему какую-то из существовавших ранее подцепочек слева, и справа.
- Просмотрев все текущие пути, перейти на новую итерацию цикла.

Процесс повторяется, пока текущее множество не опустеет.

Несмотря на то, что мы храним не матрицу в явном виде, а рабочее множество, можно хранить и матрицу, тогда пути восстанавливаются более естественным способом [8]. Псевдокод алгоритма Хеллингса представлен в листинге 1.

Algorithm 1 Алгоритм Хеллингса

```
1: function HellingsAlgo(G = ⟨Σ, N, P, S⟩, 𝒢 = ⟨V, E, L⟩)
2:     r ← {(N_i, v, v) | v ∈ V ∧ N_i → ε ∈ P} ∪ {(N_i, v, u) | (v, t, u) ∈ E ∧ N_i →
       t ∈ P}
3:     m ← r
4:     while m ≠ do
5:         (N_i, v, u) ← m.pick()
6:         for (N_j, v', v) ∈ r do
7:             for N_k → N_j N_i ∈ P таких что ((N_k, v', u) ∉ r) do
8:                 m ← m ∪ {(N_k, v', u)}
9:                 r ← r ∪ {(N_k, v', u)}
10:            end for
11:        end for
12:        for (N_j, u, v') ∈ r do
13:            for N_k → N_i N_j ∈ P таких что ((N_k, v, v') ∉ r) do
14:                m ← m ∪ {(N_k, v, v')}
15:                r ← r ∪ {(N_k, v, v')}
16:            end for
17:        end for
18:    end while
19:    return r
20: end function
```

Несмотря на то, что теоретически худшие случаи должны при таком подходе давать временную асимптотику, как у CYK для графов, на практике, как правило, данный алгоритм отрабатывает быстрее.

5.3.3 Другие алгоритмы

5.4 КС-достижимость

5.4.1 Постановка задач

Пусть $\boldsymbol{L}(G)$ – язык сконкатенированных меток рёбер графа $G = (V, E, L)$: V, E, L – вершины, рёбра, метки, $E \subseteq V \times L \times V$, $\boldsymbol{L}(G) = \{w\}$,
$w = w(v_0 l_0 v_1, v_1 l_1 v_2, ...)$, $v_i, l_j, v_k \in E$, то классически ставятся следующие задачи:

- Восстановить все пары вершин, служащих началом и концом путей, заданных данной КС-грамматикой.

- Восстановить все пары вершин, служащих началом и концом путей, заданных данной КС-грамматикой, и восстановить само множество путей. Сложности:
  - Пути нужно где-то хранить
  - Путей может оказаться формально бесконечное количество, даже если граф конечен. Решение – пути хранятся в специальной структуре данных, именуемой сжатый лес разбора (Shared packed parsing forest, SPPF).
- Для заданной пары вершин, проверить, есть ли между ними путь, заданный данной КС-грамматикой.
- Проверить пустоту пересечения $L(G)$ и некоторого другого языка.

5.4.2 Классические подходы решения

5.4.3 Вспомогательные структуры данных

Сжатый лес разбора (Shared packed parsing forest, SPPF). Впервые подобная идея была предложена Джоаном Рекерсом в его кандидатской диссертации [**?**]. В дальнейшем она нашла широкое применение в обобщённом синтаксическом анализе и получила серьёзное развитие [12]. Оптимальное асимптотическое поведение достигается при использовании бинаризованного SPPF [**?**] — в этом случае объём леса составляет $O(n^3)$, где n — это длина входной строки.

Рассмотрим способ построения SPPF на примере.

Во-первых, заметим, что в дереве вывода каждая вершина соответствует выводу какой-то подстроки с известными позициями начала и конца. Давайте будем сохранять эту информацию в вершинах дерева. Таким образом, метка любой вершины – это тройка (i, q, j), где i — координата начала подстроки, соответствующей этой вершине, j — координата конца, $q \in \Sigma \cup N$ — метка как в исходном определении. Так как такие вершины содержат символ, терминальный или нетерминальный, их в терминологии лесов разбора принято называть *символьными*.

Во-вторых, заметим, что любая внутренняя вершина со своими непосредственными потомками связаны продукцией в грамматике: вершина появляется благодаря применению конкретной продукции в процессе вывода. Давайте занумеруем все продукции в грамматике и добавим в дерево вывода ещё один тип вершин – *дополнительные*, или *промежуточные* вершины – в которых будем хранить номер применённой продукции. Получим следующую конструкцию: непосредственный предок дополнительной вершины — это левая часть продукции, а непосредственные потомки дополнительной вершины — это правая часть продукции.

Построим модифицированное дерево вывода цепочки ${}_0a_1b_2a_3b_4a_5b_6$ в грамматике

$$G_0 = \langle \{a, b\}, \{S\}, S, \{ \\ (0) S \to a\ S\ b\ S, \\ (1) S \to \varepsilon \\ \}\rangle$$

Сохраняемая нами дополнительная информация позволит переиспользовать вершины в том случае, если деревьев вывода оказалось несколько (в случае неоднозначной грамматики). При этом мы можем не бояться, что переиспользование вершин приведёт к появлению ранее несуществовавших деревьев вывода, так как дополнительная информация позволяет делать только "безопасные" склейки и затем восстанавливать только корректные деревья. Таким образом, мы можем представить лес вывода в виде единой структуры данных без дублирования информации.

Сжатие леса разбора. Построим несколько деревьев вывода цепочки ${}_0a_1b_2a_3b_4a_5b_6$ в грамматике

$$G_1 = \langle \{a, b\}, \{S\}, S, \{ \\ (0) S \to SS, \\ (1) S \to a\ S\ b, \\ (2) S \to \varepsilon \\ \}\rangle$$

Предположим, что мы строим левосторонний вывод. Тогда после первого применения продукции 0 у нас есть два варианта переписывания первого нетерминала: либо с применением продукции 0, либо с применением продукции 1:

$$\mathbf{S} \xrightarrow{0} \mathbf{S}S \xrightarrow{0} \mathbf{S}SS \xrightarrow{1} a\mathbf{S}bSS \xrightarrow{2} ab\mathbf{S}S \xrightarrow{1} aba\mathbf{S}bS \xrightarrow{2} abab\mathbf{S} \xrightarrow{1} ababa\mathbf{S}b \xrightarrow{2} abababa$$

$$\mathbf{S} \xrightarrow{0} \mathbf{S}S \xrightarrow{1} a\mathbf{S}bS \xrightarrow{2} ab\mathbf{S} \xrightarrow{0} ab\mathbf{S}S \xrightarrow{1} aba\mathbf{S}bS \xrightarrow{2} abab\mathbf{S} \xrightarrow{1} ababa\mathbf{S}b \xrightarrow{2} abababa$$

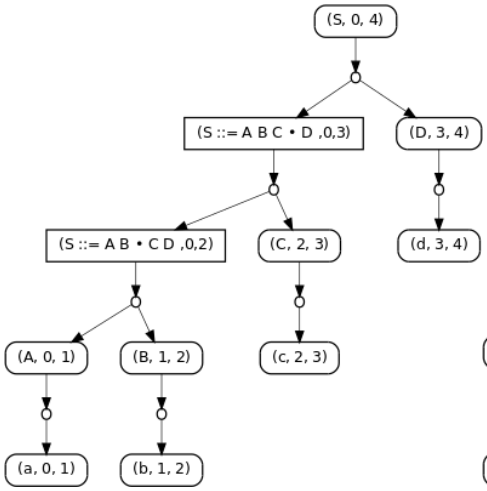
Сначала рассмотрим первый вариант (применили переписывание по продукции 0). Все остальные шаги вывода деретерминированы и в результате мы получим следующее дерево разбора:

Теперь рассмотрим второй вариант — применить продукцию 1. Остальные шаги вывода всё также детерминированы. В результате мы получим следующее дерево вывода:

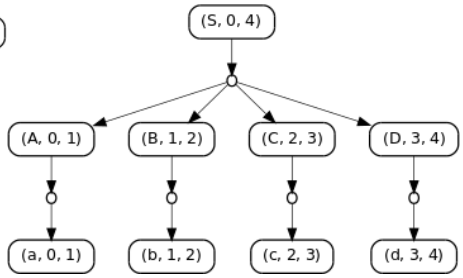
В двух построенных деревьях большое количество одинаковых узлов. Построим структуру, которая содержит оба дерева и при этом никакие нетерминальные и терминальные узлы не встречаются дважды. В результате мы получим следующий граф:

Мы получили очень простой вариант сжатого представления SPPF.

Пример 2 Рассмотрим грамматику:



(a) SPPF



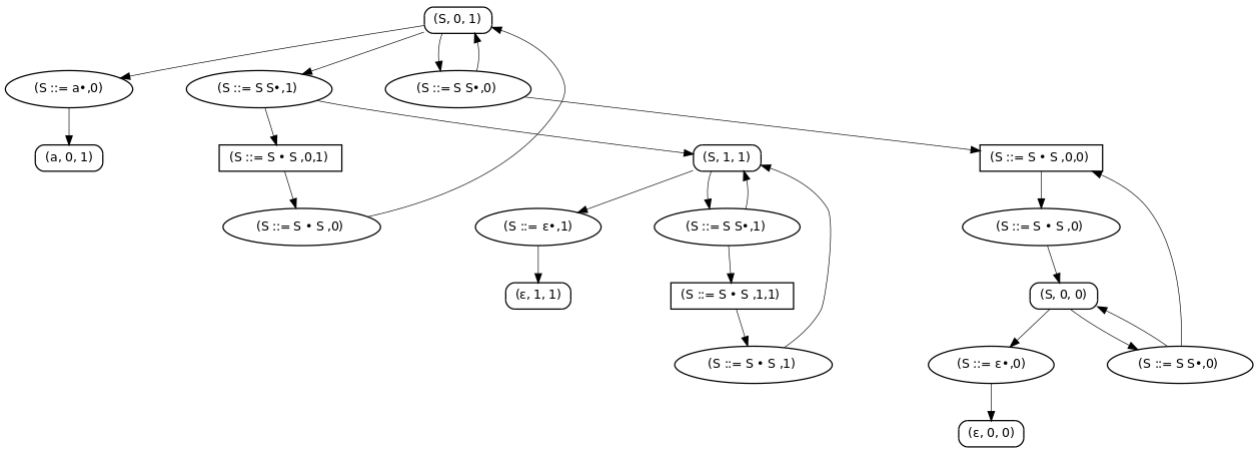
(b) SPPF без промежуточных вершин

$S \to ABCD \quad A \to a \quad B \to b \quad C \to c \quad D \to d.$

и разбор в ней слова *abcd*. Построим лес разбора:

Пример 3 Рассмотрим грамматику:

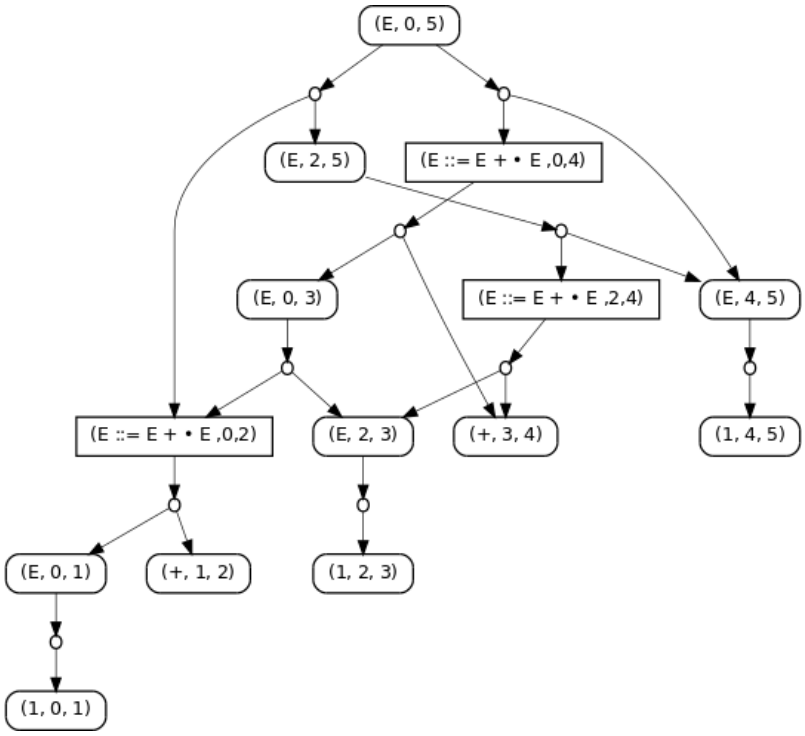
$S \to SS \mid a \mid \varepsilon$. Очевидно, она циклическая. Рассмотрим SPPF для слова a в такой грамматике:



(a) Пример 3: SPPF для a в $S \to SS \mid a \mid \varepsilon$

Пример 4. Рассмотрим неоднозначную грамматику $E \to E + E \mid 1$ и входную строку $1+1+1$. Лес разбора называется неоднозначным, если хранит хотя бы одну неоднозначную конструкцию (несколько деревьев разбора одной строчки). Построим такой лес:

В нем корневая вершина $(E, 0, 5)$ имеет 2 сжатых вершины-потомка. сле-



(a) Пример 4: SPPF для $E \to E + E \mid 1$ и $1 + 1 + 1$

довательно, минимум 2 различных дерева разбора начинается с этой вершины – это деревья, выводящие $(E+(E+E))$ и $((E+E)+E)$ соответственно.

Множество деревьев разбора, содержащихся в SPPF, находится по следующей процедуре: стартуя в корне SPPF, обходим его как дерево, посещая каждую сжатую вершину под текущей, а затем посещая каждую дочернюю вершину под сжатыми рекурсивно.

Структурные свойства SPPF.

- At first note that each symbol node (E, j, i) with $E \in T \cup N \cup \{\varepsilon\}$ is unique, то есть SPPF не может содержать 2 символьные вершины (A, k, l) и (B, m, n) , где $A = B, k = m$ и $l = n$.
- Ноды для терминалов не содержат потомков и являются листовыми в лесе разбора. Ноды для символов-терминалов (A, j, i) имеют сжатые вершины потомки с метками вида $(A ::= \gamma\cdot, k)$, где $j \leq k \leq i$, и возможное число потомков не ограничено двумя.
- Промежуточные вершины (t, j, i) имеют потомками сжатые вершины с метками (t, k), где $j \leq k \leq i$.
- Сжатые вершины (t, k) имеют 1 или 2 потомка. The right child is a symbol node (x, k, i) and the left child (if it exists) is a symbol or intermediate node with label (s, j, k), where $j \leq k \leq i$. Packed nodes have always exactly one parent which is a symbol node or intermediate node.
- It is useful to observe that the SPPF is a bipartite graph, with on the one hand the set of intermediate and symbol nodes and on the other hand the set of packed nodes. Therefore edges always go from a node of the first type to a node of the second type, or the other way round. As a consequence, cyles in the SPPF are always of even length.

Преобразование в абстрактное синтаксическое дерево

Наконец, зачастую, на выходе СА бывает важно получить абстрактное синтаксическое дерево (AST), а не что-либо иное. Причем нас может интересовать только одно AST, и нужно ободнозначнить получение дерева разбора, из которого AST и получается. Конечно, применяются и другие простейшие трансформации, типа удаления пробелов и т.д. Подходы к такому ободнозначниванию бывают различные, например, внедрение специальных фильтров, которые позволяют сделать что-то наподобие PEG, кроме того, существуют подходы, которые (в особенности при проведении обобщенного LL-анализа) позволяют избегать добавления неоднозначных результатов разбора непосредственно при построении леса разбора [14].

5.4.4 КС-достижимость через операции линейной алгебры

Из данных выше материалов следует, что и разбор, и вывод слов языка по грамматике суть исчисление над термами. С другой стороны, алгоритмы

типа CYK демонстрируют процесс разбора как последовательность преобразований специальных матриц. Возникает идея свести разбор к матричному исчислению с хитро заданными операциями сложения и умножения: инструмент матриц намного лучше исследован и оптимизирован человечеством для различных вычислительных задач, существует огромное количество эффективных его реализаций, в конце концов, работа с матрицами более привычна для инженеров, исследователей и студентов, нежели работа с языками и грамматиками.

Ранее нами был разобран алгоритм для решения задачи КС достижимости на основе CYK. Заметим, что обход матрицы напоминает умножение матриц, в ячейках которых хранятся множества нетерминалов:

$$M_3 = M_1 \times M_2$$
$$M_3[i,j] = \sum_{k=1}^{n} M[i,k] * M[k,j]$$

, где сумма — это объединение множеств:

$$\sum_{k=1}^{n} = \bigcup_{k=1}^{n}$$

, а поэлементное умножение определено следующим образом:

$$S_1 * S_2 = \{N_1^0 ... N_1^m\} * \{N_2^0 ... N_2^l\} = \{N_3 \mid (N_3 \to N_1^i N_2^j) \in P\}.$$

Таким образом, алгоритм решения задачи КС достижимости может быть дан в терминах перемножения матриц над полукольцом с соответствующими операциями.

Для частного случая этой задачи, синтаксического анализа линейного входа, существует алгоритм Валианта [**?**], использующий эту идею. Однако он не обобщается на графы из-за того, что существенно использует возможность упорядочить обход матрицы (см. разницу в CYK для линейного входа и для графа). Поэтому, хотя для линейного случая алгоритм Валианта и является алгоритмом синтаксического анализа для произвольных КС грамматик за субкубическое время, его обобщение до задачи КС достижимости в произвольных графах с сохранением асимптотики является нетривиальной задачей [**?**]. В настоящее время алгоритм с субкубической сложностью получен только для частного случая — языка Дика с одним типом скобок — Филипом Брэдфордом [**?**].

В случае с линейным входом, отдельного внимания заслуживает работа Лиллиан Ли (Lillian Lee) [**?**], где она показывает, что задача перемножения матриц сводима к синтаксическому анализу линейного входа. Аналогичных результатов для графов на текущий момент не известно.

Поэтому рассмотрим более простую идею, изложенную в статье и диссертации Рустама Азимова [**?**]: будем строить транзитивное замыкание графа через наивное (не через возведение в квадрат) умножение матриц.

Пусть $\mathcal{G} = (V, E)$ — входной граф и $G = (N, \Sigma, P)$ — входная грамматика. Тогда алгоритм может быть сформулирован как представлено в листинге 2.

Algorithm 2 Context-free recognizer for graphs

```
1: function contextFreePathQuerying(𝒢, G)
2:     n ← количество узлов в 𝒢
3:     E ← направленные ребра в 𝒢
4:     P ← набор продукций из G
5:     T ← матрица n × n, в которой каждый элемент
6:     for all (i, x, j) ∈ E do                    ▷ Инициализация матрицы
7:         T_{i,j} ← T_{i,j} ∪ {A | (A → x) ∈ P}
8:     end for
9:     for all i ∈ 0 ... n − 1 do      ▷ Добавление петель для нетерминалов,
   порождающих пустую строку
10:        T_{i,i} ← T_{i,i} ∪ {A ∈ N | A → ε}
11:    end for
12:    while матрица T меняется do
13:        T ← T ∪ (T × T)              ▷ Вычисление транзитивного замыкания
14:    end while
15:    return T
16: end function
```

Особенности реализации

Переход к матричным операциям позволяет с минимальными затратими получить эффективную параллельную реализацию алгоритма для решения задачи КС достижимости в графах. Благодаря этому, хотя асимптотически приведенные алгоритмы и имеют большую сложность чем, скажем, алгоритмы CYK и Хеллингса, в результате эффективного распараллеливания на практике они работают быстрее однопоточных алгоритов с лучшей сложностью.

Далее рассмотрим некоторые детали, упрощающие реализацию с использованием современных библиотек и аппаратного обеспечения.

Так как множество нетерминалов и правил конечно, то мы можем свести представленный выше алгоритм к булевым матрицам: для каждого нетерминала заведём матрицу, такую что в ячейке стоит 1 тогда и только тогда, когда в исходной матрице в соответствующей ячейке сожержится этот нетерминал. Тогда перемножение пары таких матриц, соответсвующих нетерминалам A и B, соответствует построению путей, выводимых из нетерминалов, для которых есть правила с правой частью вида AB.

5.4.5 Комментарии к практике

5.4.6 Пример: КС достижимость при анализе программ

Пусть по-прежнему $\boldsymbol{L}(G)$ – язык сконкатенированных меток рёбер графа $G = (V, E, L)$, V, E, L – вершины, рёбра, метки. Если G явдяется некоторым

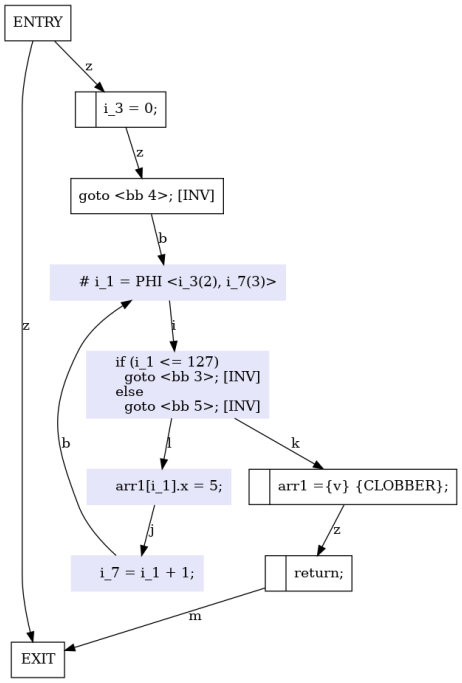


Рис. 12: Пример графа, полученного из программы в промежуточном представлении GCC GIMPLE [13]. Сиреневым подсвечен цикл, детектированный посредством решения задачи КС-достижимости (путь по рёбрам $iljb$).

представлением программы p: $G = (V, E, L) = G(p)$, $E \subseteq V \times L \times V$,

$\boldsymbol{L}(G) = \{w(p)\}$, $w(p) = w(v_0 l_0 v_1, v_1 l_1 v_2, ...)$, $v_i, l_j, v_k \in E$, то можно рассмотреть следующие задачи:

1. Поиск паттерна. Найти все пути в G, содержащие слова из L': $\boldsymbol{L'}(G) \subseteq \boldsymbol{L}(G) : \{P_G^{patterns}\} = \{P_G | w(P_G) \in \boldsymbol{L'}(G)\}$.

2. Проверка на анти-паттерн: пусто ли пересечение языка графа с «языком анти-паттернов» $\boldsymbol{L''}(G)$: $\{P_G \cap \boldsymbol{L''}(G)\}? \equiv \emptyset$.

3. Классическая задача достижимости – найти все пары вершин (состояний программы, точек останова и т.д.), таких, что между ними существует нужный путь?

4. Подзадача классической задачи достижимости – существует ли нужный путь из точки A в B в программе?

Возможна и постановка последовательной проверки на КС-достижимость: сначала выделяется множество путей по (1), а далее проверяется (2). Такие паттерны (2) для (1) назовём «ограничивающими».

5.5 Нисходящий синтаксический анализ

Нисходящий синтаксический анализ идеологически можно рассматривать как задачу поиска левого порождения входной строки, либо, что эквивалентно, как процесс построения дерева разбора добавлением узлов в прямом порядке обхода в глубину, начиная с корня.

Для контекстно-свободной (КС) грамматики $G = \langle \Sigma, N, S, P \rangle$ организация нисходящего анализа выглядит следующим образом: анализатор, разбирающий входную строку w, заканчивающуюся символом конца строки \$, в каждый момент работы содержит в своей памяти пару (α, v), где v – ещё не прочитанная часть входной строки. Алгоритм анализа пытается разобрать v как конкатенацию $\alpha = X_1 \dots X_l$,где $l \geq 0, X_1, \dots, X_l \in \Sigma \cup N$ — последовательность символов, хранящаяся на стеке так, что X_1 лежит на вершине. На каждом шаге ключевым действием является определение правила, применяемого для раскрытия соответствующего нетерминала. Когда правило выбрано, следует произвести проверку соответствия входной строки и терминальных символов правой части правила, и выполнить дальнейшие шаги для её нетерминальных символов. Если в конце разбора $v =$ \$, то есть удалось дойти до конца строки, и при этом все нетерминалы удалось раскрыть, — строка успешно разобрана.

В данной главе мы рассмотрим два подхода к нисходящему анализу: LL-анализ, использующий входной буфер, стек для хранения промежуточных данных, и управляющую таблицу, хранящую правила, применяемые в ходе анализа, а также анализ методом рекурсивного спуска, использующий в качестве стека стек вызовов программных процедур, реализующих применение правил в соответствующих ситуациях.

Дадим необходимые определения.

Определение 1. Говорят, что грамматика содержит *левую рекурсию*, если в ней существует вывод вида $A \vdash^* A\alpha$. Если при этом в грамматике содержится правило $A \vdash A\alpha$, левая рекурсия называется непосредственной, или явной. В противном случае левая рекурсия называется косвенной, или неявной.

Определение 2. Грамматика называется *однозначной*, если у каждого слова имеется не более одного дерева разбора в этой грамматике.

Определение 3. *Левым порождением*, или левосторонним выводом слова ω называется такой вывод ω, в котором каждая последующая строка получена из предыдущей заменой самого левого встречающегося в строке нетерминала по одному из правил. Символически, левое порождение обозначается как $\vdash^*_{lm}$, а любой его шаг — как $\vdash_{lm}$.

Лемма 1 *Пусть $G = \langle \Sigma, N, S, P \rangle$ — КС-грамматика. Предположим, что существует дерево разбора с корнем, отмеченным A, и кроной $\omega \in \Sigma^*$. Тогда в грамматике G существует левое порождение $A \vdash^*_{lm} \omega$.*

Доказательство производится индукцией по высоте дерева (рекомендуем читателям проделать его самим).

Лемма 2 *Для каждой грамматики $G = \langle \Sigma, N, S, P \rangle$ и $\omega \in \Sigma^*$, цепочка ω имеет два разных дерева разбора тогда и только тогда, когда ω имеет два разных левых порождения из P.*

Для описания построения нисходящего анализатора введем два вспомогательных множества, содержащих, соответсвенно, все возможные первые и непосредственно следующие k символов в результирующем выводе.

Определение 4. Пусть $G = \langle N, \Sigma, P, S \rangle$ — КС-грамматика. Множество $FIRST_k$ определяется для сентенциальной формы α как:
$FIRST_k(\alpha) = \{\omega \in \Sigma^* \mid \alpha \to^* \omega$ и $|\omega| < k$ либо $\exists \beta : \alpha \to^* \omega\beta$ и $|\omega| = k\}$, где $\alpha, \beta \in (N \cup \Sigma)^*$.

Определение 5. Пусть $G = \langle N, \Sigma, P, S \rangle$ — КС-грамматика. Множество $FOLLOW_k$ строки формы $\beta \in (\Sigma \cup \Gamma)^*$ как:
$FOLLOW_k(\beta) = \{\omega \in \Sigma^* \mid \exists \gamma, \alpha : S \vdash^* \gamma\beta\alpha$ и $\omega \in FIRST_k(\alpha)\}$

Согласно определениям, $FIRST_k(A)$ и $FOLLOW_k(A)$ содержат, соответсвенно, все возможные первые и непосредственно следующие k символов в результирующем выводе, при использовании нетерминала A

Пусть дана грамматика $\langle \Sigma, N, S, P \rangle$. Алгоритм построения $FIRST_k$ следующий:

$\forall A \in N, FIRST_k(A) \leftarrow$
$\forall a \in \Sigma, FIRST_k(a) \leftarrow \{a\}$
while $FIRST_k(A)|_{a \in N}$ изменяется **do**
  for $A \vdash X_1 \dots X_l \in P$ **do**
    $FIRST_k(A) \leftarrow FIRST_k(FIRST_k(X_1) \cdot \dots \cdot FIRST_k(X_l))$
  end for
end while

Для построения $FOLLOW_k$ нужно выполнить сдедующее:

$FOLLOW_k(S) \leftarrow \{\varepsilon\}$
$\forall A \in N \setminus \{S\}\ FOLLOW_k(A) \leftarrow$
while $FOLLOW_k(A)|_{A \in N}$ изменяется **do**
  for $B \vdash \beta \in P$ **do**
    for $\beta = \mu A \nu$ разбиений, где $A \in N, \mu, \nu \in (\Sigma \cup \{\$\} \cup N)^*$ **do**
      $FOLLOW_k(A) \leftarrow FOLLOW_k(A) \cup FIRST_k(FIRST_k(\nu) \cdot FOLLOW_k(B))$
    end for
  end for
end while

Введём понятие таблицы, управляющей разбором. **Определение 6.** Управляющая таблица для грамматики $G = \langle \Sigma, N, P, S \rangle$ – это частичная фунция $T_k : N \times \Sigma^{\leq k} \vdash P \cup \{-\}$, отображающая пару: нетерминал A и m

терминальных символов — $t_1 \dots t_m$, где $m \leq k$ — в правило, которое нужно применять, если такое правило есть в P.

По строкам в управляющей таблице размещаются все нетерминалы грамматики, по столбцам – всевозможные последовательности терминалов, длиной не более k[13,14], а также столбец для маркера конца строки — \$. В ячейке таблицы указано правило, которое нужно применять, если рассматривается нетерминал A, а следующие m символов строки — $t_1 \dots t_m$, где $m \leq k$, либо прочерк, если такого правила нет.

	...	$t_1 \dots t_m$	...	\$
...	...	...	...	...
A	...	$A \vdash \alpha$	...	...
...	...	...	...	...

Управляющая таблица строится алгоритмически на основании построения для каждого нетерминала A вспомогательных множеств $FIRST_k(A)$ и $FOLLOW_k(A)$.

Приведём алгоритм построения T_k для всех $A \in N$ и $x \in \Sigma^{\leq k} \cup \{\$\}, k > 0$ по $FIRST_k$ и $FOLLOW_k$ (в начале элементы T_k инициализированы '—').

---

for $A \vdash \alpha \in P$ **do**
  for $x \in FIRST_k(FIRST(\alpha) \cdot FOLLOW_k(A))$ **do**
    if $T_k(A, x) =$'—' **then**
      $T_k(A, x) \leftarrow (A \vdash \alpha)$
    else
      Произошел конфликт: нет однозначного правила для A, x
    end if
  end for
end for

Заметим, что в псевдокоде построения таблицы ветвь с сообщением о конфликте нужна для сигнализирования о неоднозначности в заполнении ячейки: не для всех КС-грамматик по множествам $FIRST_k$ и $FOLLOW_k$ возможно выбрать применяемое правило, следовательно, нельзя и построить однозначную управляющую таблицу. Класс грамматик, для которых управляющую таблицу можно построить без конфликтов, называют классом LL(k)-грамматик.

Определение 7. LL(k) грамматика — грамматика, для которой для некоторого $k \geq 1$ существует управляющая таблица T_k, по которой можно однозначно определить, какое правило применять.

[13]На практике таблица может получиться довольно разреженной, поэтому столбцы для последовательностей, не выводимых из нетерминалов грамматики, опускают

[14]Теоретически показательный характер роста количества столбцов от k на практике, как правило, не реализуется, так как реальные языки программирования обычно не задаются грамматиками, дающими теоретически худший случай

Теорема 3. (о построении управляющей таблицы) *Для LL(k)-грамматики $G = \langle N, \Sigma, P, S\rangle$, для любого её правила вида $A \vdash X_1 \dots X_l \in P$, если $x \in FIRST_k(FIRST_k(X_1) \cdot \dots \cdot FIRST_k(X_l) \cdot FOLLOW_k(A))$, то ячейка управляющей таблицы $T_k(A, x)$ содержит единственное правило $A \vdash X_1 \dots X_l$.*.

Теорема 3 утверждает, что для LL(k)-грамматики управляющая таблица может быть построена без возникновения конфликтов. Если же её условие приводит к противоречиям, то грамматика не является LL(k).

Критерий того, что грамматика является LL(k) грамматикой, непосредственно следует из определения:

Лемма 4 (об эквивалентном определении) *$G = \langle N, \Sigma, P, S\rangle$ является $LL(k)$ грамматикой тогда и только тогда, когда $(\forall A \vdash \alpha|\beta \in P) \Rightarrow (FIRST_k(\alpha\gamma) \cap FIRST_k(\beta\gamma) =)$ при всех таких $\omega A\gamma$, что $S \vdash^*_{lm} \omega A\gamma$.*.

Дальнейшие рассуждения и построения будут проводиться для $k = 1$. Важно заметить, что при больших k управляющая таблица сильно разрастается[15], поэтому на практике алгоритм применим для небольших k.

В частном случае для $k = 1$: **Определение 8.** $FIRST(\alpha) = \{a \in \Sigma \mid \exists\gamma \in (N \cup \Sigma)^* : \alpha \vdash^* a\gamma\}$, где $\alpha \in (N \cup \Sigma)^*$ **Определение 9.** $FOLLOW(\beta) = \{a \in \Sigma \mid \exists\gamma, \alpha \in (N \cup \Sigma)^* : S \vdash^* \gamma\beta a\alpha\}$, где $\beta \in (N \cup \Sigma)^*$ Множество $FIRST$ можно вычислить, пользуясь следующими соотношениями:

- $FIRST(a\alpha) = \{a\}, a \in \Sigma, \alpha \in (N \cup \Sigma)^*$
- $FIRST(\varepsilon) = \{\varepsilon\}$
- $FIRST(\alpha\beta) = FIRST(\alpha) \cup (FIRST(\beta)$, если $\varepsilon \in FIRST(\alpha))$
- $FIRST(A) = FIRST(\alpha) \cup FIRST(\beta)$, если в грамматике есть правило $A \vdash \alpha \mid \beta$

Algorithm 3 Алгоритм для вычисления множества $FOLLOW$

Require: Грамматика $G = \langle \Sigma, N, S, P\rangle$
Ensure: $FOLLOW(A)$ для всех $A \in N$
Положим $FOLLOW(X) \leftarrow, \forall X \in N$
$FOLLOW(S) \leftarrow FOLLOW(S) \cup \{\$\}$, где S — стартовый нетерминал
while множества $FOLLOW$ меняются **do**
Для всех правил вида $A \vdash \alpha X\beta : FOLLOW(X) \leftarrow FOLLOW(X) \cup (FIRST(\beta) \setminus \{\varepsilon\})$.
Для всех правил вида $A \vdash \alpha X$ и $A \vdash \alpha X\beta$, где $\varepsilon \in FIRST(\beta) : FOLLOW(X) \leftarrow FOLLOW(X) \cup FOLLOW(A)$
end while

[15]Хоть и не показательно, как в теоретически худшем случае

Алгоритм для вычисления множества $FOLLOW$ представлен в 3.

Задача 1 Рассмотрим грамматику G со следующими правилами:

- $S \vdash aS'$
- $A' \vdash b \mid a$
- $S' \vdash AbBS' \mid \varepsilon$
- $B \vdash c \mid \varepsilon$
- $A \vdash aA' \mid \varepsilon$

Посчитать множества FIRST и FOLLOW.

Решение

$FIRST$ для нетерминалов грамматики G:

$$\begin{aligned} FIRST(S) &= \{a\} & FIRST(B) &= \{c, \varepsilon\} \\ FIRST(A) &= \{a, \varepsilon\} & FIRST(S') &= \{a, b, \varepsilon\} \\ FIRST(A') &= \{a, b\} \end{aligned}$$

$FOLLOW$ для нетерминалов грамматики G:

$$\begin{aligned} FOLLOW(S) &= \{\$\} \\ FOLLOW(S') &= \{\$\} && (S \vdash aS') \\ FOLLOW(A) &= \{b\} && (S' \vdash AbBS') \\ FOLLOW(A') &= \{b\} && (A \vdash aA') \\ FOLLOW(B) &= \{a, b, \$\} && (S' \vdash AbBS', \varepsilon \in FIRST(S')) \end{aligned}$$

Задача решена.

Теперь рассмотрим пример грамматики, не являющейся LL(1).

Задача 2 Построить не-$LL(1)$ грамматику.

Решение

Грамматика $S \vdash aS|a$, согласно теореме 4, не является LL(1)-грамматикой, так как $FIRST(aS) = FIRST(a) = \{a\}$ и $FIRST(aS) \cap FIRST(a) = \{a\}$, но LL(2)-грамматикой, так как $FIRST_2(aS) = \{aa\}$, $FIRST_2(a) = \{a\}$, и $FIRST_2(aS) \cap FIRST_2(a) = —$ является.

Задача решена.

Очевидно, что в случае LL(1)-грамматики управляющая таблица заполняется

следующим образом: правила $A \vdash \alpha, \alpha \neq \varepsilon$ помещаются в ячейки с индексами (A, a), где $a \in FIRST(\alpha)$, правила $A \vdash \alpha$ — в ячейки (A, a), где $a \in FOLLOW(A)$, если $\varepsilon \in FIRST(\alpha)$, а если при этом и $\$ \in FOLLOW(A)$, то и в ячейку $(A, \$)$. Иногда, для небольших грамматик, в целях наглядности в таблицу добавляют 2 столбца с $FIRST, FOLLOW$ множествами для нетерминалов.

Задача 3 Построить таблицу для грамматики $S \vdash aSbS \mid \varepsilon$

Решение

N	$FIRST$	$FOLLOW$	a	b	\$
S	$\{a, \varepsilon\}$	$\{b, \$\}$	$S \vdash aSbS$	$S \vdash \varepsilon$	$S \vdash \varepsilon$

Задача решена.

Теорема о связи LL(1)-грамматики с видом множеств $FIRST$ и $FOLLOW$ приведена ниже:

Лемма 5 (об эквивалентном определении) *Грамматика $G = \langle \Sigma, N, S, P \rangle$ и $\omega \in \Sigma^*$ является LL(1) тогда и только тогда, когда выполнены 2 условия:*

1. $(\forall A \vdash \alpha | \beta \in P) \Rightarrow (FIRST(\alpha) \cap FIRST(\beta) =)$
2. $(\forall A \vdash \alpha | \beta \in P : \varepsilon \in FIRST(\alpha)) \Rightarrow (FOLLOW(A) \cap FIRST(\beta) =)$

Здесь $\alpha, \beta \in (N \cup \Sigma)^*$ — две сентенциальные формы G.

Вернёмся к решению задачи 2 в свете леммы 5.

Задача 3 Проверить, что грамматика, задающая язык строк с равным количеством символов a и b: $S \vdash aSbS|bSaS|\varepsilon$, не является LL(1).

Решение

Но грамматика содержит правило $S \vdash \varepsilon$, и $\varepsilon \in FIRST(\varepsilon)$, следовательно, нужно проверять (2). $FOLLOW(S) = \{a, b, \$\}$ имеет непустое пересечение как с $FIRST(aSbS)$, так и с $FIRST(bSaS)$, поэтому (2) не выполняется, и грамматика не является LL(1).

Но грамматика содержит правило $S \vdash \varepsilon$, и $\varepsilon \in FIRST(\varepsilon)$, следовательно, нужно проверять (2). $FOLLOW(S) = \{a, b, \$\}$ имеет непустое пересечение как с $FIRST(aSbS)$, так и с $FIRST(bSaS)$, поэтому (2) не выполняется, и грамматика не является LL(1).

Задача решена.

Условия критерия накладывают довольно серьёзные ограничения на вид грамматики. В особенности:

1. Грамматика должна быть однозначной:

$$
\begin{aligned}
&G: \\
S &\vdash aA|B|c \\
A &\vdash b|aA \\
B &\vdash aA|a\varepsilon
\end{aligned}
$$

Если анализируемая строка начинается с a, невозможно сделать однозначный выбор между $S \vdash aA$ и $S \vdash B$.

2. Даже вывод ε из двух правил альтернативы невозможен:

$$
\begin{aligned}
&G: \\
S &\vdash aA \\
A &\vdash BC|B \\
C &\vdash b|\varepsilon \\
B &\vdash \varepsilon
\end{aligned}
$$

Рассмотрим два разных левых порождения a в G:

- $S \vdash_{lm} \underline{aA} \vdash_{lm} \underline{aB} \vdash_{lm} a$
- $S \vdash_{lm} \underline{aA} \vdash_{lm} \underline{aBC} \vdash_{lm} a$

В виду того, что из $B \vdash^*_{lm} \varepsilon$ и $BC \vdash^*_{lm} \varepsilon$, нельзя однозначно произвести подчёркнутый шаг левого порождения, a в G имеет два различных дерева вывода, и грамматика не является LL(1).

5.6 LL-алгоритм синтаксического анализа

LL(k) — синтаксический анализ — семейство алгоритмов нисходящего анализа без отката, с предпросмотром. Решение о том, какое правило применять, принимается по управляющей таблице T_k на основании просмотра k символов, непосредственно следующих за текущим во входной строке. Временная сложность алгоритма $O(n)$, где n — длина входной строки.

Для КС грамматики $\langle \Sigma, N, P, T \rangle$ алгоритм использует:

- входной буфер с указателем на позицию текущего символа
- стек с алфавитом $\Gamma = N \cup \Sigma \cup \{\$\}$ для хранения промежуточных данных
- таблицу T_k, управляющую разбором.

При чтении анализируемой строки во входе, алгоритм может заглядывать вперёд на k символов.

Конфигурацией алгоритма назовём пару $\langle x, X\alpha \rangle$ из множества таких пар Q, где x — неразобранная часть входной строки, $X\alpha \in \Gamma^*$ — содержимое стека, $X \in \Gamma$ — символ на вершине. При анализе строки w будем называть конфигурацию $\langle w, S\$ \rangle$ — стартовой, конфигурацию $\langle \$, \$ \rangle$ — конечной. Алгоритм, начиная со стартовой, на каждом шаге анализирует текущую конфигурацию, и выполняет действия, с учётом прочитанной части анализируемой строки: определяется цепочка исследуемых входных символов $u, |u| \leq k$ и символ на вершине стека X, затем, если $X \in N$, рассматривается элемент управляющей таблицы $T_k(X, u)$, и замена содержимого вершины стека правой частью правила из этого элемента; если X – терминальный символ, происходит сравнение с первым символом u, и в случае совпадения – извлечение X и сканирование очередного символа из ввода.

Опишем действия над конфигурациями, $\{f : Q \to Q, f \in \{match, lookup, success, error\}\}$, выполняемые в ходе работы алгоритма:

- **match** — в случае, когда на вершине стека — терминал, и символ на текущей позиции равен этому терминалу, то снять элемент со стека, сдвинуть указатель на 1 позицию вправо. $\langle x, X\alpha \rangle$ переводится в $\langle x', \alpha \rangle$, если $x = ax'$ и $X = a$
- **lookup** — в случае, когда текущая врешина стека — нетерминал X_i, и предпросмотрена подстрока u, найти в управляющей таблице T ячейку с координатами (X_i, u), положить на стек содержимое правой части этой ячейки так, чтобы самый левый символ оказался на вершине. $\langle x, X\alpha \rangle$ переводится в $\langle x, \beta\alpha \rangle$, если $T_k(X, u) = X \vdash \beta$ и $X = a$
- **success** — равершить работу при достижении конфигурации $\langle \$, \$ \rangle$
- **error** — сигнализировать об ошибке и завершить работу.

Если алгоритм оказался в конечной конфигурации — разбор успешно завершён.

5.6.1 Алгоритм LL(1)-анализа

Опишем работу алгоритма LL(1)-анализа, как частного случая LL-анализа с предпросмотром на $k = 1$ символ. Алгоритм по-прежнему использует входную строку, управляющую таблицу, стек, и работает следующим образом:

- На каждом шаге алгоритма его конфигурация — это позиция во входной строке, начиная с которой расположена неразобранная её часть, и стек.
- В начале работы стек пуст, а позиция во входной строке соответствует её началу. На певом шаге в стек добавляются последовательно \$ и стартовый нетерминал S.

```
stack.push($, S)
c ← input.scan()
while stack.top() ≠ $ do
    X ← stack.top()
    if X = c then // match:
        stack.pop()
        c ← input.scan()
    else if X ∈ N then // lookup(X,c):
        if T[X, c] = X ⊢ X_1 ... X_m then
            stack.pop()
            stack.push(X_m, ..., X_1)
        else
            ошибка: пустая ячейка таблицы! //error
        end if
    else
        ошибка! //error
    end if
end while
if c ≠ $ then
    ошибка: не вся строка разобрана! //error
end if//success
```

- На каждом шаге анализируется текущая конфигурация и совершается одно из действий:
  - Действие **success**. Если текущая позиция — конец строки, и вершина стека — символ конца строки, то разбор успешно завершен. Иначе, если стоим на конце строки — **error**.
  - Действие **match**. Если текушая вершина стека — терминал, то анализатор проверяет, что позиция в строке соответствует этому терминалу. Если да, то снимает элемент со стека, сдвигает указатель на 1 позицию вправо, и продолжает разбор. Иначе — завершает разбор с ошибкой — **error**.
  - Действие **lookup**. Если текущая врешина стека — нетерминал X_i, и текущий входной символ c, то ищет в управляющей таблице T ячейку с координатами (X_i, c) и кладёт на стек содержимое правой части этой ячейки так, чтобы самый левый символ оказался на вершине (операция *stack.push* применена к символам правой части справа налево), иначе сигнализирует об ошибке — **error**.

Пример работы LL(1) анализатора. Рассмотрим грамматику $S \vdash aSbS \mid \varepsilon$ и выводимое слово $\omega = abab$.

Расмотрим пошагово работу LL(1)-алгоритма. Используем управляющую таблицу, построенную в предыдущем примере. Символ строки, доступный по указателю позиции в строке, выделен жирным шрифтом.

1. Начало работы.

   Стек: $\boxed{}$

   Входное слово: $\boxed{\mathbf{a}}\boxed{\text{b}}\boxed{\text{a}}\boxed{\text{b}}\boxed{\$}$

   Стек пуст, по указателю доступен первый символ слова.

2. Кладём \$ и стартовый символ S на стек

   Стек: $\boxed{S}\boxed{\$}$

   Входное слово: $\boxed{\mathbf{a}}\boxed{\text{b}}\boxed{\text{a}}\boxed{\text{b}}\boxed{\$}$

3. Ищем ячейку с координатами (S, a), применяем правило из ячейки.

   Стек: $\boxed{a}\boxed{S}\boxed{b}\boxed{S}\boxed{\$}$

   Входное слово: $\boxed{\mathbf{a}}\boxed{\text{b}}\boxed{\text{a}}\boxed{\text{b}}\boxed{\$}$

4. Снимаем терминал a со стека и сдвигаем указатель.

   Стек: $\boxed{S}\boxed{b}\boxed{S}\boxed{\$}$

   Входное слово: $\boxed{\text{a}}\boxed{\mathbf{b}}\boxed{\text{a}}\boxed{\text{b}}\boxed{\$}$

5. Ищем ячейку с координатами (S, b), применяем правило из ячейки.

   Стек: $\boxed{b}\boxed{S}\boxed{\$}$

   Входное слово: $\boxed{\text{a}}\boxed{\mathbf{b}}\boxed{\text{a}}\boxed{\text{b}}\boxed{\$}$

6. Снимаем терминал b со стека и сдвигаем указатель.

   Стек: $\boxed{S}\boxed{\$}$

   Входное слово: $\boxed{\text{a}}\boxed{\text{b}}\boxed{\mathbf{a}}\boxed{\text{b}}\boxed{\$}$

7. Ищем ячейку с координатами (S, a), применяем правило из ячейки.

   Стек: $\boxed{a}\boxed{S}\boxed{b}\boxed{S}\boxed{\$}$

   Входное слово: $\boxed{\text{a}}\boxed{\text{b}}\boxed{\mathbf{a}}\boxed{\text{b}}\boxed{\$}$

8. Снимаем терминал a со стека и сдвигаем указатель.

   Стек: $\boxed{S}\boxed{b}\boxed{S}\boxed{\$}$

   Входное слово: $\boxed{\text{a}}\boxed{\text{b}}\boxed{\text{a}}\boxed{\mathbf{b}}\boxed{\$}$

9. Ищем ячейку с координатами (S, b), применяем правило из ячейки.

   Стек: $\boxed{b}\boxed{S}\boxed{\$}$

   Входное слово: $\boxed{\text{a}}\boxed{\text{b}}\boxed{\text{a}}\boxed{\mathbf{b}}\boxed{\$}$

10. Снимаем терминал b со стека и сдвигаем указатель.

    Стек: $\boxed{S}\boxed{\$}$

    Входное слово: $\boxed{\text{a}}\boxed{\text{b}}\boxed{\text{a}}\boxed{\text{b}}\boxed{\mathbf{\$}}$

11. Ищем ячейку с координатами (S, \$), применяем правило из ячейки.

    Стек: $\boxed{\$}$

    Входное слово: $\boxed{\text{a}}\boxed{\text{b}}\boxed{\text{a}}\boxed{\text{b}}\boxed{\mathbf{\$}}$

12. Оказались в конце строки и на вершине стека символ конца — завершаем разбор.

Шаг	**Стек**	**Остаток строки**	**Текущее действие**
0	|	**abab$**	$stack.push(\$, S)$
1	|$|S	**abab$**	$lookup(S, a)$
2	|$|S|b|S|a	**abab$**	$match$
3	|$|S|b|S	**bab$**	$lookup(S, b)$
4	|$|S|b	**bab$**	$match$
5	|$|S	**ab$**	$lookup(S, a)$
6	|$|S|b|S|a	**ab$**	$match$
7	|$|S|b|S	**b$**	$lookup(S, b)$
8	|$|S|b	**b$**	$match$
9	|$|S	**$**	$lookup(S, \$)$
10	|$	**$**	$match$

Таблица 1: Разбор слова $abab$ в грамматике $S \vdash aSbS \mid \varepsilon$ по LL(1)-алгоритму

Можно расширить данный алгоритм так, чтобы он строил дерево вывода. Дерево будет строиться сверху вниз, от корня к листьям. Для этого необходимо расширить действия:

- В ситуации, когда выполняется **match** (на вершине стека и во входе — одинаковые терминалы), нужно создать листовую вершину с соответствующим терминалом.
- В ситуации, когда нетерминал в стеке заменяется на правую часть правила в ходе выполнения **lookup**, нужно создать нелистовую вершину, соответствующую нетерминалу в левой части применяемого правила.

Дерево вывода для LL(1), как и в целом для LL(k), будет строиться однозначно, что следует из однозначности грамматик.

Также отметим, что LL-анализ, как и безоткатный рекурсивный спуск, не работает с леворекурсивными грамматиками: алгоритм может зациклиться. Таким образом, по некоторым грамматикам можно построить LL(k)-анализатор (для LL(k) грамматик), но не по всем. Методы борьбы с левой рекурсией даны в следующих разделах, а вот с неоднозначностями ничего не поделаешь.

5.7 Рекурсивный спуск

Идея рекурсивного спуска основана на использовании стека вызовов программы в качестве стека анализатора следующим образом:

- Для каждого нетерминала программируется функция, принимающая необработанный остаток строки s и возвращающая пару: результат вывода префикса данной строки из соответствующего нетерминала (выводится/не выводится) и новый необработанный остаток строки.
- Каждая функция реализует обработку цепочки согласно правым частям правил для соответствующих нетерминалов: считывание символа ввода при обработке терминального символа, вызов соответствующей функции при обработке нетерминального.

У данного подхода есть два ограничения:

1. Неприменим для грамматик, содержащих левую рекурсию. Иначе анализатор может зациклиться.
2. Шаги должны быть однозначными. Иначе нет возможности детерминированно выбрать конкретную функцию для вызова в некоторых ситуациях.

Если в грамматике, для которой разрабатывается рекурсивный спуск, есть альтернатива $A \vdash u_1|\dots|u_z$, то однозначный выбор применяемой функции обработки нетерминала A (либо применяемого правила в вызываемой функции, если для каждого правила в альтернативе не реализована отдельная функция) может быть автоматизированно осуществлён по проверке условия наличия префикса ещё не обработанной части строки s длины не более k в $FIRST_k(u_j), j \in [1,z]$, причём условие должно выполняться не более чем для одного j, иначе грамматика неоднозначна. Если такой j не найден, но существует $\hat{j} \in [1,z]$, такой, что, $u_{\hat{j}} \vdash^* \varepsilon$, и данный префикс принадлежит $FOLLOW_k(A)$, то можно положить $j = \hat{j}$. В данных и только в данных случаях правило $A \vdash u_j$ может быть выбрано для применения. Следовательно, для однозначного выбора правила требуется проанализировать $FIRST_k(A)$ и $FOLLOW_k(A)$, и, классически, рекурсивному спуску подлежат языки, задаваемые классом LL(k) грамматик[16].

Приведём алгоритм выбора правила из альтернативы для $k = 1$.

Рассматривается альтернатива: $A \vdash u_1|\dots|u_z$
$inSym$ – первый символ необработанной части строки
if $(\exists j \in [1,z]) : inSym \in FIRST(u_j))$ **then**
  Выбрать правило $A \vdash u_j$
else if $(\exists \hat{j} \in [1,z]) : u_{\hat{j}} \vdash^* \varepsilon \ \&\ inSym \in FOLLOW(A))$ **then**
  Выбрать правило $A \vdash u_{\hat{j}}$
else
  Ошибка!
end if

---

[16] на практике это ограничение может быть ослаблено различными ухищрениями, вроде откатов и пр.

Приведём общий вид функции обработки $funcA$ нетерминала A, символически обозначая считывание символа из входного потока s, моделируемого объектом класса строки, реализующего методы $s.current$, возвращающий символ в текущей позиции, и $s.scan$, который возвращает терминальный символ и модифицирует s так, что в нём после вызова $c = s.scan()$ остаток строки, расположенный за c. Если возвращаеммое значение самой первой в стеке вызовов функции — пара вида *(True, [])*, то разбор завершился успехом. Временная сложность алгоритма от длины строки n — $O(n)$, так как строка сканируется только один раз.

---

if $len(s) = 0$ **then**
  $return(True, w)$
end if
Текущее правило: $A \vdash X_1 X_2 \dots X_k$
for $i \in [1, k]$ **do**
  if $X_i \in N$ **then**
    $res, s \leftarrow funcX_i(s)$ // *call*, $C()$
    if $res = False$ **then**
      $return\ (False, s)$ // *return*, $R()$
    end if
  else if $(X_i \in \Sigma \cup \{\varepsilon\})$ & $((X_i = \varepsilon) || X_i = s.current())$ **then**
    if $X_i \neq \varepsilon$: $s.scan()$ // *match_terminal*, $M_\Sigma()$
    if $i = k$ **then**
      $return\ (True, s)$ // *return*, $R()$
    end if
  else
    $return\ (False, s)$ // *return*, $R()$
  end if
end for

Заметим, что алгоритм совершает 3 типа действий[17]:

1. **call**, $C()$: если символ на текущей позиции рассматриваемого правила — нетерминал, совершить вызов функции его обработки.
2. **return**, $R()$: возврат из вызова. Производится при попытке сдвига с крайней правой позиции в рассматриваемом правиле, либо в случае пустого слова во вводе, либо в случае ошибки.
3. **match terminal**, $M_\Sigma()$: если символ на текущей позиции в правиле — ε, просто сдвинуть позицию на 1. Если терминал — проверить соответствие его текущему входному символу, и, если они равны, то сдвинуть позицию в правиле на 1 и считать следующий символ.

Заметим, что действия **call** и **return** реализуют логику **lookup** из алгоритма анализа по таблице, **match terminal** — логику **match**.

[17]Как правило, на практике эти действия не формализуют

Рассмотрим работу рекурсивного спуска реализующего разбор слова $aabb$ по грамматике $S \vdash aSbS \mid \varepsilon$

Шаг	**Стек вызовов**	**Остаток строки**	**Текущее действие**
0	|main|**S(aabb$)**	**aabb$**	$M_\Sigma(S \vdash \mathbf{a}SbS, aabb\$)$
1	|main|**S(aabb$)**	**abb$**	$C(S \vdash a\mathbf{S}bS, S)$
2	|main|S|**S(abb$)**	**abb$**	$M_\Sigma(S \vdash \mathbf{a}SbS, abb\$)$
3	|main|S|**S(abb$)**	**bb$**	$C(S \vdash a\mathbf{S}bS, S)$
4	|main|S|S|**S(bb$)**	**bb$**	$M_\Sigma(S \vdash \varepsilon, bb\$), R()$
5	|main|S|**S(abb$)**	**bb$**	$M_\Sigma(S \vdash aS\mathbf{b}S, bb\$)$
6	|main|S|**S(abb$)**	**b$**	$C(S \vdash aSb\mathbf{S}, S)$
7	|main|S|S|**S(b$)**	**b$**	$M_\Sigma(S \vdash \varepsilon, b\$), R()$
8	|main|S|**S(abb$)**	**b$**	$R()$
9	|main|**S(aabb$)**	**b$**	$M_\Sigma(S \vdash aS\mathbf{b}S, b\$)$
10	|main|**S(aabb$)**	**$**	$C(S \vdash aSb\mathbf{S}, S)$
11	|main|S|**S($)**	**$**	$M_\Sigma(S \vdash \varepsilon, \$), R()$
12	|main|**S(aabb$)**	**$**	$R()$

Таблица 2: Разбор слова $aabb$ в грамматике $S \vdash aSbS \mid \varepsilon$ рекурсивным спуском.

Данный подход применяется как для ручного написания синтаксических анализаторов, так и при генерации анализаторов по грамматике, например средствами ANTLR.

5.8 Преобразования грамматики к LL(1)

Иногда грамматику $G = \langle \Sigma, N, S, P \rangle$, не являющуюся LL(1), можно привести к LL(1) грамматике. В первую очередь, можно применить методы устранения левой рекурсии и левую факторизацию. Следует отметить, что в ходе преобразований не всякая грамматика становится LL(1), а также то, что грамматика может стать менее понятной. Также доказано, что существование LL грамматики, эквивалентной G, является алгоритмически неразрешимой задачей.

5.8.1 Устранение левой рекурсии

Непосредственная левая рекурсия, то есть правила вида $A \vdash A\alpha$, можно устранить следующим образом.

1. Группируем правила с A в левой части: $A \vdash A\alpha_1 | \ldots | A\alpha_m | \beta_1 | \ldots | \beta_n$, где никакая из сентенциальных форм β_i не начинается с A.
2. Добавляем новый нетерминал A'

3. Заменяем этот набор правил на

$$A \vdash \beta_1 A' | \ldots | \beta_n A'$$
$$A' \vdash \alpha_1 A' | \ldots | \alpha_m A' | \varepsilon$$

Теперь из A можно вывести те же строчки, что и раньше, но без левой рекурсии. Заметим, что в ходе данного преобразования появляются новые ε-правила, по одному на каждый добавленный нетерминал. Метод выше устраняет только непосредственную левую рекурсию.

Пусть дана грамматика $G = \langle \Sigma, N, S, P \rangle$, не содержащая ε-правил. Для удаления из G скрытой левой рекурсии, включающей два и более шага, применяется следующий алгоритм:

Нетерминалы пронумерованы в произвольном порядке, $n \leftarrow |N|$
for $i \in [1, n]$ **do**
  for $j \in [1, i-1]$ **do**
    $A_j \vdash \beta_1 | \ldots | \beta_k$ — все текущие правила для A_j
    Заменить все $A_i \vdash A_j \alpha$ на $A_i \vdash \beta_1 \alpha | \ldots | \beta_k \alpha$
  end for
  удалить правила $A_i \vdash A_i$
  устранить непосредственную левую рекурсию для A_i.
end for

Полученная грамматика не содержит левой рекурсии. В ходе преобразования могут появиться ε-правила.

5.8.2 Левая факторизация

Идея левой факторизации лежит в том, чтобы в случае, когда неясно, какую из альтернатив применять для раскрытия нетерминала A, изменить правила для A так, чтобы отложить решение до тех пор, пока не будет достаточно информации для принятия однозначного решения.

Преобразование: для правил $A \vdash \alpha\beta_1 | \alpha\beta_2$ грамматики $G = \langle \Sigma, N, S, P \rangle$ и непустой строчки с префиксом, выводимым из α, когда неизвестно, какое правило применять, можно добавить новое правило $A \vdash \alpha A'$, и после анализа того, что выводимо из α, попробовать применить новое правило $A' \vdash \beta_1$ либо $A' \vdash \beta_2$.

После преобразования грамматика может стать не LL(1) (см. задачу 3).
$FIRST(\alpha\beta) = FIRST(\alpha) \cup (FIRST(\beta) if \varepsilon \in FIRST(\alpha))$
Доказательство.
(Достаточность). Предположим, что грамматика G не является $LL(1)$, тогда для некоторого слова w, выводимого в G, существует 2 левосторонних вывода.

while В грамматике есть альтернативы с общим префиксом **do**
Для каждого $A \in N$ найти самый длинный префикс α для альтернатив в P с A в левой части.
if $\alpha \neq \varepsilon$ **then**
Заменить все $A \vdash \alpha\beta_1 | \ldots | \alpha\beta_m | \gamma$ на:
$A \vdash \alpha A' | \gamma$
$A' \vdash \beta_1 | \ldots | \beta_m$
end if
end while

5.8.3 Комментарии к практике

На семинаре (`fltp/p10/`) были разобраны:

- Рекурсивный спуск
- LL(1)-анализ:
  - $First_1/Follow_1$ - построение
  - Построение таблицы разбора
  - Построение и тестирование анализатора
- Устранение левой рекурсии (`fltp/p10/left_recursion_elimination`)

5.9 Восходящий разбор: LR

5.9.1 LR(0)

5.9.2 SLR

Автомат – такой же, как в LR(0). Таблица отличается только тем, что reduce выполняется только там, где это имеет смысл.

5.9.3 (C)LR(1)

Канонический LR.

5.9.4 LALR

Наиболее часто реализуемый на практике подход.

https://github.com/meyerd/flex-bison-example

Пусть есть грамматика, не разбираемая из-за конфликтов сдвиг-свертка или свертка-свертка по алгоритму SLR.

В этом случае грамматика преобразуется следующим образом:

- ищется нетерминал, на котором возникла вызвавшая конфликт свертка. Обозначим его A.

- вводятся новые нетерминалы $A_1, A_2, \ldots, A_n$, по одному на каждое появление A в правых частях правил.
- везде в правых частях правил A заменяется на соответствующее A_k.
- набор правил с A в левой части повторяется n раз по разу для каждого A_k.
- правила с A в левой части удаляются, тем самым полностью удаляя A из грамматики. Для преобразованной грамматики (она порождает такой же язык, что и исходная) повторяется попытка построения SLR(1) таблицы разбора.

Действие основано на том, что Follow(A) есть объединение всех Follow(A_k). В каждом конкретном состоянии новая грамматика имеет уже не A, а одно из A_k, то есть множество Follow для данного состояния имеет меньше элементов, чем для A в исходной грамматике.

Это приводит к тому, что для LALR(1) совершается меньше попыток поставить «приведение» в клеточку таблицы разбора, что уменьшает риск возникновения конфликтов с приведениями, иногда вовсе избавляет от них и делает грамматику, не разбираемую по SLR(1), разбираемой после преобразования.

Множество Follow(A_k) называется lookahead set для A и k-той встречи в правилах, отсюда название алгоритма.

5.9.5 Комментарии к практике

На семинаре (`p11/`) было разобрано несколько примеров работы с Flex/Bison для лексического и синтаксического анализа с вычислениями по ходу рабора соответственно.

5.10 О применении синтаксического анализа на практике

Как правило, в ходе синтаксического анализа мы не желаем просто узнавать, что это программа – синтаксически корректная программа на ЯП / строка какого-то языка; мы хотим что-то скомпилировать / извлечь и тд. То есть получить ее синтаксическую структуру, и с ней уже работать.

Тем не менее, бывает интересна и сама процедура вывода, если требуется что-то делать по ходу этой процедуры. Механизм выполнения действий во время разбора называется синтаксически управляемой трансляцией, и рассматривается в следующем разделе.

Пример: напишем грамматику арифметических выражений с $+, *, (,)$

```
S -> S + S
S -> S * S
S -> (S)
S -> n
```

Данная грамматика действительно задаёт указанные выражения. Но чем она плоха с точки зрения их вычислений?[18] И чем грамматика, выписанная ниже, лучше на практике?

```
E -> T
E -> E + T
T -> F
T -> T * F
F -> n
F ->(E)
```

По данной грамматике уже можно однозначно выполнить арифметические действия на основании полученной структуры и того, что записано в терминалах. Записи можно считать «значениями» или «атрибутами».

Но можно пойти дальше и считать, что у нетерминалов тоже есть атрибуты... С одним, частным вариантом их обработки мы уже познакомились, когда разрабатывали парсер на Bison – в нём можно было производить вычисления атрибутов и выполнять действия по-восходящей по мере разбора. О том, обстоят дела в общем случае, будет рассказано в главе «Синтаксически управляемая трансляция».

6 Синтаксически управляемая трансляция

6.1 Введение

Сначала мы работали с задачей разпознавания – принадлежит ли исследуемое слово языку – да / нет. Потом нам понадобилось строить дерево разбора – извлекать синтаксическую структуру из слова в языке. Теперь нам и этого станет мало.

Заметим, что дерево разбора – это тоже цепочка в некотором языке (любое дерево кодируется как $root[child_1[...], clild_2[...], ...]$).

Опр. 6.1 *Трансляция - преобразование некоторой входной строчки в выходную.* $\tau : L_i \Rightarrow L_o$, $L_i \in \Sigma_i^*, L_o \in \Sigma_o^*$

Примеры:

- Вычисление арифметического выражения
- Преобразование арифметического выражения
- Любое преобразование программы в компиляторе
- Восстановление дерева по коду Прюфера

[18] Для ответа на этот вопрос нарисуйте дерево разбора в данной грамматике какого-нибудь выражения

То есть, фактически, синтаксический анализ – это трансляция[19].

Зачем же урезать модели трансляции, если у нас есть ЯП общего назначения (Тьюринг-полный)? В теории, чтобы можно было гарантировать некоторые свойства транслятора.

Опр. 6.2 (Нестрогое) *Синтаксически управляемая трансляция (англ. Syntax-directed translation, SDT, СУТ) — преобразование текста в последовательность команд через добавление таких команд в правила грамматики*

В этом месте может возникнуть резонный вопрос – почему бы просто не разобрать слово, а потом обойти полученное дерево разбора, и выполнить необходимые вычисления? Действительно, зачастую в алгоритмах преобразования различных графоструктурированных данных (например, в преобразованиях компилятора) именно так и поступают. Однако, существует минимум две причины так не делать:

- Экономия памяти – как минимум, можно не хранить всё дерево разбора в памяти. Проблема – больше историческая.
- Актуальная проблема: есть логика выражений, в которой мы что-то делаем с атрибутами; если мы запишем дерево, а потом сделаем visitor по дереву, нам снова придется описать всю логику работы внутри обходчика еще раз – получается дублирование функциональности.

В то же время, СУТ позволяет и логику действий, и синтаксис описать в одном месте.

6.2 Атрибутные грамматики

Расширим понятие грамматики атрибутами и семантическими действиями.

- Пусть каждый символ в $X \in \Sigma \cup N$ в грамматике может иметь атрибуты, которые содержат данные[20]. Это может быть $key : value$ словарь, структура или union, не принципиально. Пусть, для определённости, для X с атрибутом t обращение к атрибуту может выглядеть как $X.t$, а ко всему атрибутам $X.attr$. Грамматика, содержащая такие «расширенные» символы, называется атрибутной грамматикой.
- Дополним атрибутную грамматику $G = (\Sigma, N, P, S)$ сематническими действиями – множеством функций A – $G = (\Sigma, N, P, S, A)$, где $\forall a \in A \exists p \in P : a(\{l.attr : l \in L\}, \{r.attr : r \in R\})$, l, r – всевозможные символы в соответственно левой и правой частях правила p, вызывается тогда и только тогда, когда применяется правило p. Говорят, что такая грамматика задаёт схему трансляции. Далее будем рассматривать только КС-грамматики, поэтому $|L| = 1$.

[19] В задачах обобщения на графы это не всегда так – нас могут интересовать пересечения, пустота, etc

[20] Обычно такие атрибуты могут включать в себя тип переменной, значение выражения, и т.п.

6.2.1 Типы атрибутов

Типы атрибутов вводятся с точки зрения действия над ними семантических операций в ходе разбора.

Опр. 6.3 *Синтезированные атрибуты – атрибуты, высчитываемые из правых частей правил.*

Синтезированные атрибуты содержат информацию, подтягиваемую вверх по ходу восходящего разбора (либо возврата из рекурсивного спуска, etc), в общем, вычисляются по мере восхождения от терминалов к корню дерева разбора: в момент сворачивания по некоторому правилу, мы знаем атрибуты правой части, но ещё не знаем атрибуты левой. Они-то и «синтезируются» на основе атрибутов правой части[21].

Пример: вычисления на синтезируемых атрибутах:

```
E -> E+T  { E.val = E.val + T.val then print (E.val)}
E -> T   { E.val = T.val}
T -> T*F  { T.val = T.val * F.val}
T -> F   { T.val = F.val}
F -> Id   {F.val = id}
```

Другие примеры с синтезируемыми атрибутами были рассмотрены на паре про Flex/Bison.

Опр. 6.4 *Наследуемые атрибуты – атрибуты, высчитываемые из соседних либо родительских вершин дерева разбора.*

Пример: присвоение типа переменным при создании (`int a,b,c;`). Пример грамматики составить самостоятельно.

6.3 Более общая формулировка

Возьмем понятие трансляции из прошлого подраздела. Введем СУ схему как:

Опр. 6.5 *СУТ – это пятерка* (Σ, N, P, S, Π)*, где*

- Π– *выходной алфавит*
- P *– конечное множество правил вида* $A \to \alpha, \beta, \alpha \in (N \cup \Sigma)*, \beta \in (N \cup \Pi)*$,
- *вхождения нетерминалов в цепочку* β *образуют перестановку нетерминалов их цепочки* α
- *Если нетерминалы повторяются более одного раза, их различают по индексам*

[21] В этом месте становится понятно, почему Bison работает именно на синтезированных атрибутах, и вычисления происходят именно так

В таком виде мы можем задавать, как преобразовывать цепочку. Получается, СУТ-схема задает синхронный вывод 2 цепочек.

- Если $A \to (\alpha, \beta) \in P$, то $(\gamma A^i \delta, \gamma' A^i \delta') \Rightarrow (\gamma \alpha^i \delta, \gamma' \beta^i \delta')$
- Рефлексивно-транзитивное замыкание отношения $\Rightarrow$ называется отношением выводимости $\Rightarrow *$
- Трансляцией называется множество пар $\{(\alpha, \beta) | (S, S) \Rightarrow *(\alpha, \beta), \alpha \in \Sigma^*, \beta \in \Pi^*\}$
- Схема называется простой, если в любых правилах вида $A \to (x, y)$ нетерминалы x, y встречаются в одном и том же порядке.
- Схема называется однозначной, если не существует двух правил $A \to a, b, A \to a, c$, таких, что b, c – разные символы.

Т. 6.1 *Выходная цепочка однозначной СУТ-схемы может быть сгенерирована при одностороннем выводе.*

Также существует понятие обобщенной СУТ-схемы.

Там, фактически, параллельно строятся два дерева разбора:

Для каждой внутренней вершины дерева, соответсующей нетерминалу A, с каждым A_j связвывается цепочка (трансляция) символа A_j

TODO: дописать

$$
\begin{array}{lcll}
E & \to & E + T & , \; E_1 = E_1 + T_1 \\
 & & & , \; E_2 = E_2 + T_2 \\
 & | & T & , \; E_1 = T_1, \; E_2 = T_2 \\
T & \to & T * F & , \; T_1 = T_1 * F_1 \\
 & & & , \; T_2 = T_1 * F_2 + T_2 * F_1 \\
 & | & F & , \; T_1 = F_1, \; T_2 = F_2 \\
F & \to & (E) & , \; F_1 = (E_1) \\
 & & & , \; F_2 = (E_2) \\
 & | & sin(E) & , \; F_1 = sin(E_1) \\
 & & & , \; F_2 = cos(E_1) * E_2 \\
 & | & cos(E) & , \; F_1 = cos(E_1) \\
 & & & , \; F_2 = -sin(E_1) * E_2 \\
 & | & x & , \; F_1 = x, \; F_2 = 1 \\
 & | & n & , \; F_1 = n, \; F_2 = 0
\end{array}
$$

Рис. 13: Обобщенная СУТ, позволяющая описать простейшее дифференцирование

6.4 Магазинный преобразователь

Под механизмом СУТ лежит (или может лежать) формальный вычислитель – магазинный преобразователь, представляющий собою выходную ленту + МП автомат, который на каждый шаг что-нибудь на выходную ленту печатает.

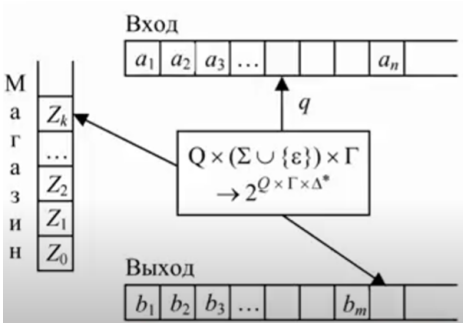


Рис. 14: МП-преобразователь

Доказывается, что, так как МП-преобразователь не может что-нибудь переставить на своем стеке, то класс трансляций МП-автомата не шире класса простых СУ-схем.

Также доказывается, что по любой простой СУ-схеме можно простроить МП-преобразователь, то есть классы простых СУ-схем и МП-преобразователей совпадают.

7 Компиляторные технологии

7.1 Представление кода в виде дерева

Дерево разбора (именуемое ещё «concrete syntax tree» в книгах по компиляторам [10]) – подробно разобранная нами в разделе о грамматиках структура представления синтаксиса. В компиляторах его использование, вернее, использование его как явного представления программы, избыточно, так как некоторые синтаксические конструкции могут быть удалены или слиты воедино после синтаксического разбора.

Опр. 7.1 *Abstract syntax tree (AST) – упрощённое представление синтаксической структуры программы – помеченное ориентированное дерево, в котором внутренние вершины помечены операторами языка программирования, а листья – соответствующими операндами.*

Таким образом, листья *AST* являются пустыми операторами и представляют только переменные и константы.

AST отличается от дерева разбора тем, что в нём отсутствуют узлы и рёбра для тех синтаксических правил, которые не влияют на семантику программы. Например:

- отсутствует информация о скобках – она задается структурой дерева
- вышеупомянутое упрощение $numterm - -leafnode \to leafnode : val$

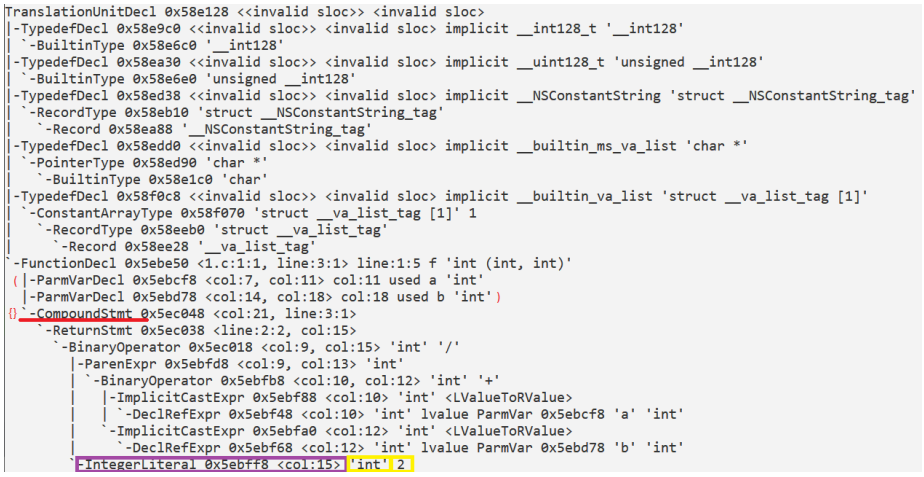


Рис. 15: Clang AST для функции целочисленного осреднения 2 целых чисел

Обычно всё незначимая подцепочка просто заменяется на значение(я) из терминала(ов).

Понятно, что структура элементов дерева укладывается в иерархии. Здесь следует отметить 2 момента по программированию:

- У 2 разных корней (ноды разных категорий) может не быть общего предка, и на практике они наследованы от разных базовых классов. То есть иерархически по классам дерево получается не деревом, а лесом. И методы для каждого дерева из леса могут быть различными.

- Представим, мы находимся в вершине дерева A, и нам нужно сделать кодогенерацию для дочерней вершины B, которая может быть типов $C_1, C_2, C_3, .., C_n$. Следовательно, в функцию $CG :: GenerateCode A$ придётся вставить switch-case на n элементов для каждой из альтернатив C_i. Но так придётся делать для каждой из функций!

Напрашивается способ, как решить вышеуказанные моменты изящно. Для этого служит паттерн ООП «Visitor»[11], который мы рассматривать не будем. Любой модуль, использующий AST для своих целей ($ASTConsumer$) реализует в себе такой «Visitor».

7.2 Синтаксический разбор

Как правило, в компиляторах на данный момент доминируют три способа построения AST по входной программе:

- LALR(1)-парсинг + модификации парсера для специфических операций типа «составление таблицы символов», etc. Использовался ранее в GCC до версии 3.X.X, затем был переработан во вручную написанный рекурсивный спуск.

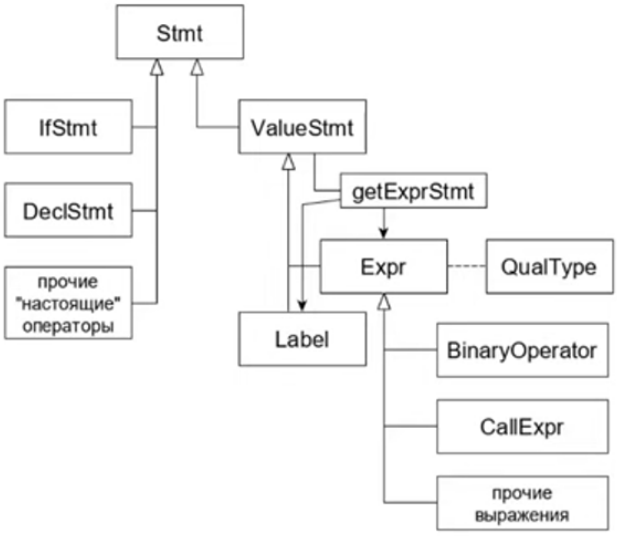


Рис. 16: Clang FE: Иерархия Stmt в Clang AST

- Рекурсивный спуск, написанный вручную. Используется в Clang, современном GCC, Rust C и др[22].
- GLR-анализ – обобщенный LR-разбор, как правило, использующий GLR-парсеры общего назначения, частично доработанные. Пример – Elsa C++ Parser.

Если смотреть по соотношению в индустрии, подход №2 с рекурсивным спуском существенно доминирует. Почему так? Этому есть, как минимум, 4 причины:

1. Языки C и C++ на самом деле не контекстно-свободные
2. Но большинство конструкций при этом вообще регулярные – язык «почти регулярный», и следует ожидать длинные цепочки вывода
3. Стандарт – довольно строгий, и содержит много частных случаев
4. Частные случаи и правила для «почти регулярных» цепочек легче прописывать и отлаживать вручную, чем городить LR-грамматику.

7.3 Лексический анализ C-подобных языков

Проблемы:

[22]На момент проведения занятия весной 2022 г. было выяснено, что MSVC тоже использует рекурсивный спуск, о других проприетарных компиляторах автору ничего не известно.

```
int 'int'        [StartOfLine]  Loc=<1.c:1:1>
identifier 'f'   [LeadingSpace] Loc=<1.c:1:5>
l_paren '('             Loc=<1.c:1:6>
int 'int'               Loc=<1.c:1:7>
identifier 'a'   [LeadingSpace] Loc=<1.c:1:11>
comma ','               Loc=<1.c:1:12>
int 'int'        [LeadingSpace] Loc=<1.c:1:14>
identifier 'b'   [LeadingSpace] Loc=<1.c:1:18>
r_paren ')'             Loc=<1.c:1:19>
l_brace '{'      [LeadingSpace] Loc=<1.c:1:21>
return 'return'  [StartOfLine] [LeadingSpace]   Loc=<1.c:2:2>
l_paren '('      [LeadingSpace] Loc=<1.c:2:9>
identifier 'a'          Loc=<1.c:2:10>
plus '+'                Loc=<1.c:2:11>
identifier 'b'          Loc=<1.c:2:12>
r_paren ')'             Loc=<1.c:2:13>
slash '/'               Loc=<1.c:2:14>
numeric_constant '2'            Loc=<1.c:2:15>
semi ';'                Loc=<1.c:2:16>
r_brace '}'      [StartOfLine]  Loc=<1.c:3:1>
eof ''          Loc=<1.c:3:2>
```

Рис. 17: Токены для функции целочисленного осреднения 2 целых чисел (получены командой clang -Xclang -dump-tokens main.c)

- Как было сказано ранее для C и C++, реальные C-подобные языки синтаксически сложны, наделены большим количеством корнеркейсов, и, как правило, контекстно зависимы. Интуитивно, если представить бесконтекстный парсинг таких языков рекурсивным спуском, в лексере будут откаты – мы что-то считали из потока лексем, интерпретировали это как-то, потом подчитали ещё что-то, поняли, что ошиблись, и вернули подчитанное обратно во входной поток с целью дальнейшего анализа.
- Есть 2 типа токенов (или даже больше! В Clang существуют аннотирующие токены, которые парсер внедряет во входную последовательность с целью указать, что некоторая подпоследовательность им уже проанализирована) – Token и PreprocessingToken (для макроопределений)

Интуитивно бы сделать 1 лексер на 2 класса токенов. Но в некоторых компиляторах, например в Clang, все с точностью до наоборот – 2 лексера (Lexer и TokenLexer) и один класс токенов (Token)! Как результат, при обработке, например, `#include`, нужно поддерживать целый стек лексеров, какие-то из которых просто лексеры, а какие-то TokenLexer.

7.4 Взаимодействие компонент фронтенда

В учебниках по компиляторам [10] и различных курсах часто пишут, что взаимодействие компонент фронтенда выглядит как конвейер:

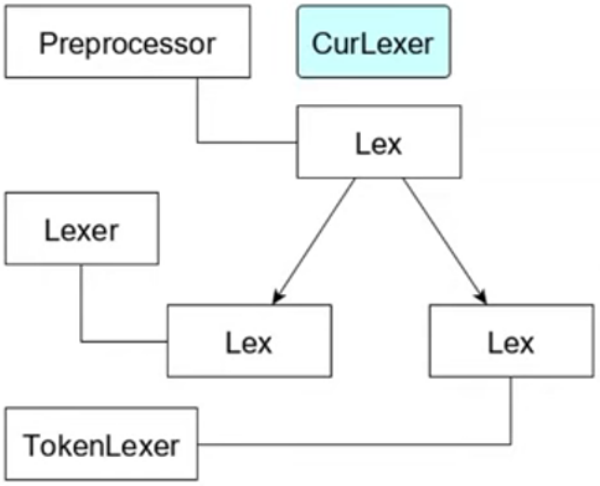


Рис. 18: Два лексера в Clang

лексический –> синтаксический –> семантический анализ

Это крайне грубое представление о работе современных компиляторных фронтендов. В следующем подразделе мы покажем, что в деталях это совсем не так.

7.5 Clang как фронтенд

При вызове `clang -cc1` создаётся экземпляр класса `Clang::CompilerInstance` в методе `cc1_main`, в нём выставляется базовое действие, которое должен сделать фронтенд[23]. Действие активируется `Act`, после чего Clang его выполняет.

7.5.1 Иерархия базовых действий

Как правило, мы хотим что-то выводить – у нас в качестве действий используются вызовы методов `Emit<actionname>Action`. Стоит отметить, что:

- Все такие действия наследуют от `CodeGenAction`
- CodeGenAction делает CodeGen консьюмером для AST
- ASTFrontendAction добавляет использование семантического анализа
- Точка входа при таких действиях: `ParseAST`.

[23]только одно, поэтому clang не может одновременно, например, скомпилировать программу (-emit-obj) и сдампить AST (-ast-dump)

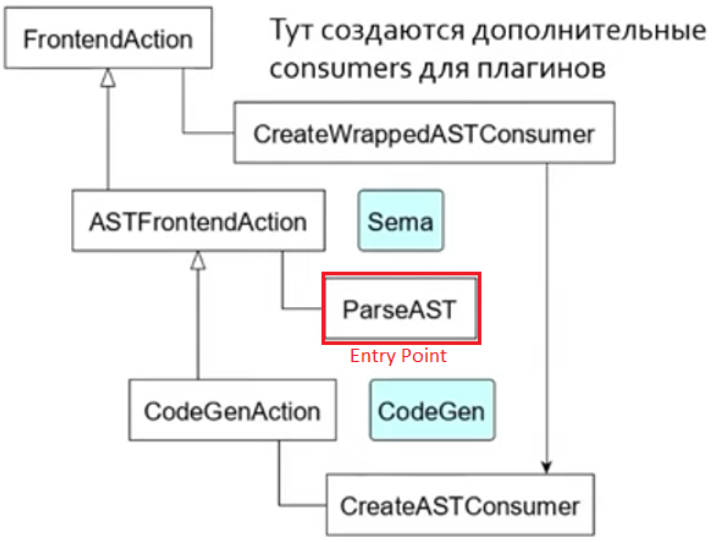


Рис. 19: Clang FE: Иерархия действий при парсинге

7.5.2 Парсинг в Clang

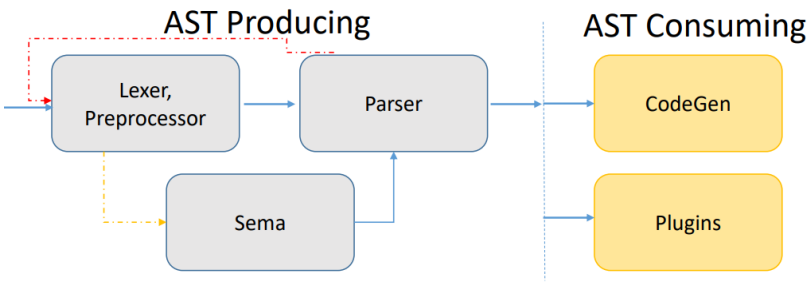


Рис. 20: Clang FE: Диаграмма зависимостей при парсинге / консьюминге AST

Главный модуль в парсинге – Parser. Его задача – подготовить AST, далее включаются все продьюсеры, потребляющие AST. Лексер – однопроходный и по умолчанию не зависит от парсера[24]. Это 2 лексера, описанных выше. Стек лексеров хранит объект класса Preprocess, он и является настоящим лексером в Clang. Семантический модуль Sema, по теории, не должен зависеть от лексера, но он от него зависит! (Ужас!).

[24]Это не совсем так, в виду возможности махинаций с токенами и возможностью бектрекинга

Парсер – это рукописный рекурсивный спуск, как было сказано ранее. То есть написан набор методов `Parser:Parse<XYZ>`, по функции для каждого нетерминала. Если в ходе парсинга происходит ошибка, в принципе, предпостроенная часть AST имеет право на существование, а в месте, в котором возник затык, вставляется вершина с записью о возможной ошибке (ошибки вставляются по сопоставлению с большим enum'ом). Также есть опция `-fixit`, позволяющая исправлять простейшие синтаксические ошибки.

LALR(1) не используется, потому что C/C++ языки, для которых:

- Грамматика «ну почти» регулярная – довольно простая[25]
- При этом язык (на самом деле) контекстно-зависимый
- Потенциально мало бектрекинга
- Довольно строгий стандарт
- Много особых случаев, которые гораздо проще прописывать вручную

Проблемы:

- Невозможность раннего определения идентификаторам категории (лексер даже не пытается). Поток токенов ну оооочень простой. Парсер должен по грамматике догадаться по грамматике, что это. А сам язык сложный.
- Бектрекинг может быть необходим при таком подходе!

Бектрекинг в лексере: интерфейс (завёрнутый в `TentativeParsingAction`-объект)

- EnableBacktrackAtThisPos[26] – запомнить точку отката
- CommitBacktrackedTokens – забыть
- Backtrack – откатиться

Следовательно: лексер поддерживает бектрекинг, после которого подпоследовательность снова считывается, и снова разбирается парсером. Это довольно накладно по производительности, поэтому придумали ещё один тип токенов – аннотирующие. Как правило, их используют для `typename`, `scope_identifiers`. Парсер внедряет этот токен в последовательность токенов для указания, что уже понял, что это за тип и т.д. (проверка: `if TryAnnotateTypeOrScopeToken()`, установка: `setTypeAnnotation(tok,ty)`).

Мало того, парсер способен внедрять не только аннотирующие, а и вообще любые токены (см. метод `ExpectAndConsume`). Иногда это используется для обработки ошибок.

[25] По крайней мере, большинство правил

[26] Данный вызов укладывает позиции в стек: откатываясь к n-й контрольной точке, далее можно откатиться к $n-1$-й и так далее

7.5.3 Семантический анализ

Утверждение Языки C/C++ (и многие другие) КС-языками не являются. Наиболее известным примером не-контекстно-свободности ЯП является конструкция

if cond then stmt1 else stmt2.

В виду того, что парсер осуществляет анализ в КС-приближении, а ЯП по сути КС-языками не являются, в компиляторы включается семантический модуль, позволяющий производить обработку контекстно-зависимых правил. Подхода к разбору бывает два:

- Непосредственно "налету по ходу разбора, парсер вызывает семантический модуль, который преобразует AST в ходе построения. Это позволяет существенно сэкономить память и не реализовывать логику проходов дважды.
- После получения результатов КС-разбора, семантический модуль трансформирует их в AST.

Clang использует первый подход, выполняя и синтаксический и семантический анализ в один проход. Семантический анализ выполняется модулем Sema по вызову из модуля Parser.

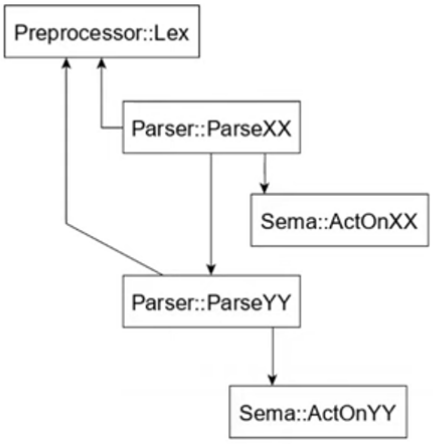


Рис. 21: Clang FE: Parser + Sema

Семантический анализ происходит по схеме:

Parse(XX) –> Sema::ActOn(XX) –> Ok? –> change AST / No? –> Error, причём AST строит именно семантический анализатор. То есть модуль Sema по сути решает 2 задачи:

- Ищет ошибки
- Строит AST

Заметим, что схема вызова семантического модуля похожа на вызов сематнического правила на атрибутах в Bison, которое строило бы AST "снизу–вверх".

7.5.4 Выводы

Кратко сформулируем выводы о внутреннем устройстве Clang-фронтенда[27]:

- Парсинг осуществляется рекурсивным спуском
- AST строит именно семантический модуль, на основании того, что разобрал парсер
- В лексере присутствует бектрекинг, лексеров – 2 типа
- Лексер, парсер и семантический модуль имеют куда более хитрые созависимости, нежели описано в классической схеме построения компиляторов.

7.6 Обработка AST

Положим, у нас есть AST, построенное компилятором. Какие дальнейшие действия нужно предпринять, чтобы превратить его в исполняемый код?

7.6.1 Операции над AST

Рассмотрим представление программы в виде AST, сгенерированное компилятором. Над данными деревьями компилятор может выполнять операции статического и семантического анализа – проверять некоторые свойства программы выполнены на этапе компиляции – например, проверить, что все переменные определены, или в данной единице трансляции любому вывову free соответствует вызов выделения динамической памяти, и нигде нет повторных free. А также над AST будут выполняться трансформации в промежуточное представление среднего уровня (middle-end IR), либо из него будет выполняться непосредственная генерация кода. К тому же, обработка AST может производиться для визуализации, реструктуризации кода, вычисления различных метрик.

7.6.2 Паттерн Visitor

Во второй части данного курса нам нужно рассмотреть несколько паттернов, то есть тактических приёмов решения программистских задач отдельных типов, упрощающих реализацию.

В большинстве операций над построенным AST различные типы узлов дерева следует рассматривать по-разному (выше была дана иерархия наследования классов AST-вершин в Clang, из которой было очевидно, что в

[27] Для Clang версии 12.0.0

иерархии содержатся довольно семантически разные программные единицы – операторы, утверждения, переменные ...). Логично, что если разработка компилятора производится с использованием ООП, подобная иерархия будет повторена, вне зависимости от компилятора и языка программирования – по классу для каждого типа вершины, объединенные в некоторую иерархию для минимизации размера кода компилятора и переноса отношения иерархичности из предметной области. Над конкретными объектами отдельных классов будут совершаться отдельные операции, и эти объекты будут, возможно, содержать различные поля. Мало того, существующие в нескольких классах иерархии общие операции будут, скорее всего, выполнять различные действия. Но если разбросать все операции по классам различных узлов, то получится система, которую трудно писать, сопровождать, поддерживать [28]. К примеру, код, отвечающий за проверку типа, будет перемешан с кодом, реализующим дамп содержимого вершины. А добавление новой операции, возможно, потребует перекомпиляции всей иерархии[29].

Решением вышеописанной проблемы, которое довольно очевидно напрашивается из схемы работы с AST – посетить все его вершины, и выполнить некоторые действия в них, с учетом того, что вершины могут быть разных типов, является размещение взаимосвязанных операций из каждого класса в отдельный объект, называемый посетителем (visitor), и передавать его элементам AST по мере обхода. Принимая посетителя, элемент дерева отправляет ему запрос (вызывает метод), в котором содержится класс элемента и сам элемент. Посетитель должен выполнить операцию над элементом, причем ту же самую, которая находилась бы в классе элемента, если б визитора не было.

Для C++. Класс visitor, как правило, создается за счёт использования подхода CRTP – класс наследуется от класса-шаблона, параметризованного этим же классом.

```
class ScalarExprEmitter :
public StmtVisitor<ScalarExprEmitter,
                   Value*> {
  CodeGenFunction &CGF;
  CGBuilderTy &Builder;
  bool IgnoreResultAssign;
  llvm::LLVMContext &VMContext;
public:
  ScalarExprEmitter(CodeGenFunction &cgf,
                    bool ira=false)
    : CGF(cgf), Builder(CGF.Builder), IgnoreResultAssign(ira),
      VMContext(cgf.getLLVMContext()) {}
```

[28] Вспомним также, что, например, в Clang AST не является древо-структурированным в смысле выполнения существования единственного общего класса – предка у классов двух произвольных вершин

[29] Если это общая операция, содержащаяся в базовом классе

}

8 О выразительности языков и грамматик

8.1 Иерархия Хомского

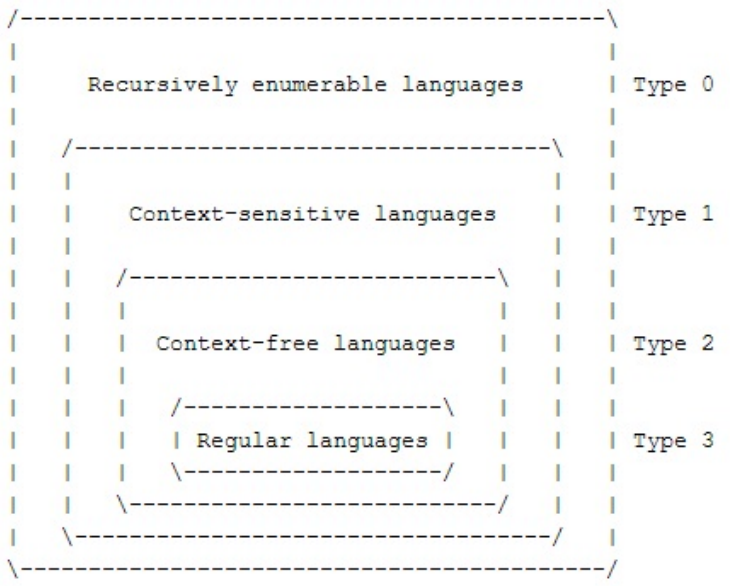


Рис. 22: Иерархия Хомского

8.2 О некоторых грамматиках промежуточных типов

8.2.1 Грамматики с контекстами

Разбор строки *aabbcc* в грамматике с контекстами языка $a^n b^n c^n$ изображен на Рис. . Обратим внимание, что в терминалы, выводимые из правил, применяемых с учётом проверки контекстов, ведёт более одной стрелки.

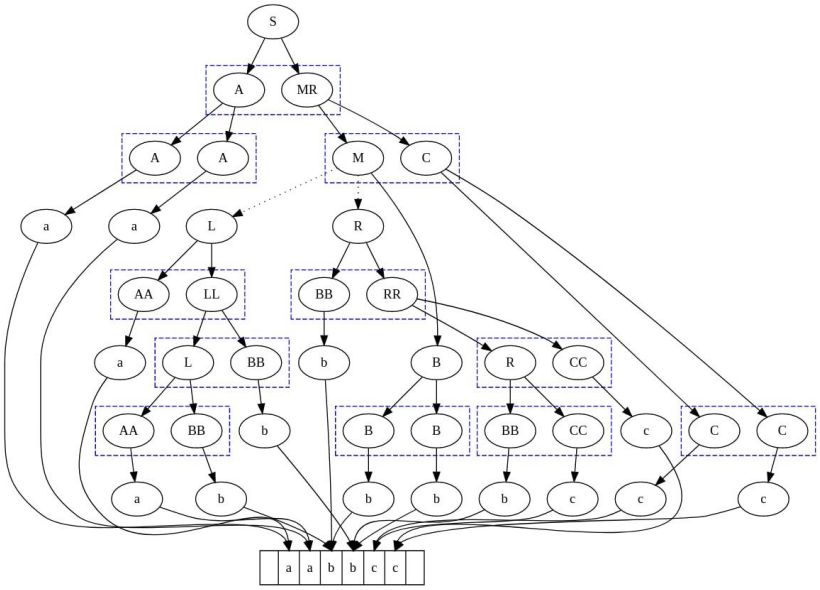


Рис. 23: Мультидерево разбора строки *aabbcc* в грамматике с контекстами языка $a^n b^n c^n$

8.2.2 Грамматики надстройки деревьев

9 Алгоритмы КС-достижимости в терминах линейной алгебры

9.1 Матричный подход

9.2 Тензорный подход

10 О некоторых обобщённых алгоритмах КС-разбора

10.1 О GLR-алгоритме

10.2 О GLL-алгоритме на примере инструмента Iguana parser

11 Приложение

11.1 Необходимые определения из близких областей

11.1.1 Графы

В данном курсе мы будем рассматривать только конечные ориентированные помеченные графы, подразумевая под «графами» именно такие графы,

если не указано противное.

Опр. 11.1 *Граф $G = (V, E, L)$, где V — конечное множество вершин, E — конечное множество рёбер, L — множество меток.*

Опр. 11.2 *Отношением достижимости на графе в смысле нашего определения называется двухместное, транзитивно-рефлексивное,*

Опр. 11.3 *Транзитивным замыканием графа называется транзитивное замыкание отношения достижимости по всему графу.*

11.2 Ссылки на контесты и дополнительные материалы

Контест 1: http://judge2.vdi.mipt.ru/cgi-bin/new-register?contest_id=220221
Контест 2: http://judge2.vdi.mipt.ru/cgi-bin/new-register?contest_id=220222

Список литературы

[1] Хопкрофт Д., Мотвани Р., Ульман Д. Введение в теорию автоматов, языков и вычислений. Санкт-Петербург : Вильямс, 2008.

[2] `http://neerc.ifmo.ru/wiki/index.php?title=Минимизация_ДКА,_алгоритм_за_O(n%5E2)_с_построением_пар_различимых_состояний`

[3] `http://neerc.ifmo.ru/wiki/index.php?title=Минимизация_ДКА,_алгоритм_Хопкрофта_(сложность_O(n_log_n))`

[4] `http://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_Бржозовского`

[5] Истинное могущество регулярных выражений. Хабр. https://habr.com/ru/post/171667, 2013 (перевод, оригинал также доступен в Интернете).

[6] Префиксное сжатие регулярных выражений. Хабр. https://habr.com/ru/post/117177

[7] Younger, Daniel H. Recognition and parsing of context-free languages in time n3 (англ.) // Information and Computation. — Vol. 10, no. 2. — P. 189–208. — doi:10.1016/s0019-9958(67)80007-x

[8] Melski, David and T. Reps. “Interconvertibility of a class of set constraints and context-free-language reachability.” Theor. Comput. Sci. 248 (2000): 29-98.

[9] Hellings, «Path Results for Context-free Grammar Queries on Graphs», 2015.

[10] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Compilers: Principles, Techniques, and Tools.

[11] Fluentcpp: Design Patterns vs Design Principless: Visitor. https://www.fluentcpp.com/2022/02/09/design-patterns-vs-design-principles-visitor

[12] https://grammarware.net/text/2016/sppf.pdf

[13] https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html

[14] van der Sanden, L.J. Parse Forest Disambiguation, 2014. https://pure.tue.nl/ws/portalfiles/portal/46998704/784691-1.pdf

[15] А. Лаздин. Компиляторы не только для программирования . CPPConf, 2024. url: https://youtu.be/1H63tUXGh7o