Визуализация конечных автоматов в/из регулярных выражений

Марко Т. Мораза´н и Тияна Минич
Университет Сетон Холл
{MORAZANM/MINICTIJ3}@SHU.edu

В большинстве курсов по формальным языкам и теории автоматов изучается дуализм между моделями вычислений для распознавания слов в языке и моделями вычислений для генерации слов в языке. Для студентов, не привыкших к формальным формулировкам, эти преобразования редко бывают интуитивно понятными. Чтобы помочь студентам в таких преобразованиях, важную роль могут сыграть средства визуализации. В этой статье представлены средства визуализации, разработанные для FSM - языка, специфического для класса теории автоматов, - для преобразования конечного автомата состояния в регулярное выражение и наоборот. Используя эти инструменты, пользователь может предоставить произвольный конечный автомат или произвольное регулярное выражение и выполнить преобразование в прямом и обратном направлении. На каждом шаге визуализация описывает сделанный шаг. Описаны инструменты, их реализация и проведено сравнение с аналогичными работами. Кроме того, представлены эмпирические данные, собранные в контрольной группе. Эмпирические данные свидетельствуют о том, что инструменты хорошо воспринимаются, эффективны, а обучение их использованию имеет низкую постороннюю когнитивную нагрузку.

1 Введение

В курсах формальных языков и теории автоматов (FLAT) особое внимание уделяется эквивалентности различных моделей вычислений. Например, эквивалентность детерминированных конечных автоматов (dfas) и нетерминированных конечных автоматов (ndfas) устанавливается путем показа студентам, как преобразовать ndfa в dfa. Такое преобразование, хотя и не является тривиальным для начинающего студента FLAT, относительно интуитивно понятно: машина (т.е. ndfa) для распознавания слов на одном языке преобразуется в другую машину (т.е. dfa) для распознавания слов на том же языке. По сути, алгоритм для определения принадлежности к языку преобразуется в другой алгоритм для определения принадлежности к языку.

Менее интуитивно понятными являются преобразования от модели, распознающей слова на языке, к модели, генерирующей слова на языке, и наоборот. То есть алгоритм распознавания слов на языке преобразуется в алгоритм генерации слов на том же языке, а алгоритм генерации слов на языке преобразуется в алгоритм распознавания слов на том же языке. В таких случаях сгенерированный алгоритм не удовлетворяет той же цели. Например, в типичном курсе FLAT студенты учатся преобразовывать автомат pushdown в контекстно-свободную грамматику и наоборот, а также учатся преобразовывать регулярную грамматику в автомат конечного состояния и наоборот.

Преобразования от порождающего алгоритма к распознающему алгоритму и от распознающего алгоритма к порождающему алгоритму могут сбить с толку начинающих студентов FLAT, поскольку формальные утверждения редко бывают для них интуитивно понятными. Главным среди этих преобразований является переход от ndfa к регулярному выражению (regexp) и обратно. Эти преобразования важны, потому что regexp хорошо подходят для выражения шаблонов, а конечные автоматы - для разработки программ [7]. Например, regexps используются в таких инструментах, как awk [25] и emacs [1], а конечные автоматы лежат в основе алгоритмов поиска строк [11] и лексического анализа [23]. Для

облегчения понимания студентами были разработаны инструменты визуализации, такие как JFLAP [26, 27] и OpenFlap [17].

Стивен Чанг (ред.): Тенденции функционального
программирования в образовании (TFPIE)
EPTCS 405, 2024, pp. 36-55, doi:10.4204/EPTCS.405.3

Обычно считается, что визуализации - это мощный педагогический инструмент в классе. Они позволяют учащимся в той или иной степени взаимодействовать со своими проектами. Однако этого может быть недостаточно для создания эффективного инструмента обучения по нескольким причинам. Визуализация требует от пользователей изучения ее интерфейса, а это может создавать дополнительную когнитивную нагрузку на учащихся [8, 29]. Чтобы быть эффективными и снизить такую нагрузку, визуализации должны предоставлять представления, которые ведут себя как сами объекты [10]. Это не означает, что инструмент визуализации не может предлагать более продвинутые функции. Это означает, что важно, чтобы в нем были простые в использовании функции.

В этой статье описываются средства визуализации, разработанные для `FSM` [21], чтобы помочь студентам понять преобразования из `ndfa` в `regexp` и из `regexp` в `ndfa`. `FSM` - это функциональный язык, встроенный в `Racket` [3], разработанный для класса `FLAT` для программирования состояния.
машины, грамматики и регулярные выражения [20]. Эти инструменты преследуют несколько целей: помогают понять алгоритмы построения, снижают лишнюю когнитивную нагрузку и позволяют студентам интерактивно изучать шаги построения как в прямом, так и в обратном направлении. Статья построена следующим образом. В разделе 2 приводится обзор и сравнение работ, связанных с данной тематикой. В разделе 3 представлено краткое введение в синтаксис `FSM`, необходимое для навигации по данной статье. В разделе 4 обсуждается общая идея, лежащая в основе стратегий визуализации. В разделе 5 описана генерация графиков визуализации. В разделе 6 представлены эмпирические данные, собранные в ходе подготовки к применению в классе, на примере контрольной группы. Наконец, в разделе 7 приводятся заключительные замечания и обсуждаются направления дальнейшей работы.

2 Связанные работы

2.1 Алгоритмы построения

2.1.1 `regexp` в `ndfa`

Пусть Σ - алфавит языка. Существует шесть разновидностей регулярных выражений [28]:

1. R = a, где $a \in \Sigma$
2. R = ε, где ε обозначает пустое слово
3. R = $\varnothing$, обозначает пустой язык
4. R = $R_1 \cup R_2$, обозначает объединение двух регулярных выражений
5. R = $R_1 \circ R_2$, обозначает конкатенацию двух регулярных выражений
6. R = R_1^* , обозначает ноль или более конкатенаций регулярного выражения

Создание `ndfa` для разновидностей 1-3 не представляет сложности. Для разновидностей 1-2 каждая соответствующая `ndfa` имеет начальное состояние и конечное состояние. Переход между ними потребляет элемент алфавита для первого сорта и ничего (т. е. пустую строку) для второго сорта. У `ndfa` для третьей разновидности есть только начальное состояние и нет конечного. Преобразования для многообразий 4-6 основаны на свойствах закрытия для регулярных языков. То есть `ндфа` строится с помощью алгоритмов, разработанных в рамках структурных доказательств, устанавливающих, что языки, принимаемые ндфами, замкнуты по объединению, конкатенации и звезде Клейна. Для объединения и конкатенации ndfas M_1 и M_2 рекурсивно строятся для R_1 и R_2 . Для объединения создается новое начальное и новое конечное состояние. Результирующая `ndfa` недетерминированно переходит из нового начального состояния в начальное состояние M_1 или M_2 и из каждого из конечных состояний M_1 и M_2 в новое конечное состояние. При конкатенации добавляются недетерминированные переходы из конечных состояний M_1 в начальное состояние M_2 , а конечными состояниями новой машины являются конечные состояния M_2 . Для звезды Клейна рекурсивно строится

ндфа, M_1 , для R_1 . Генерируется новое начальное состояние, которое также является конечным состоянием.

Кроме того, генерируются недетерминированные переходы из нового начального состояния в начальное состояние M_1 и из конечных состояний M_1 в начальное состояние M_1 . За формальными подробностями этих конструкторов читатель может обратиться к любому учебнику по FLAT (например, [9, 13, 15, 16, 20, 24, 28]).

Преобразование обычно объясняется с помощью обобщенного недетерминированного конечного автомата (GNFA) [28]. GNFA похож на ndfa, но его переходы осуществляются с помощью регулярных выражений. Исходный GNFA имеет два состояния и переход между ними. Переход осуществляется по регулярному выражению, которое преобразуется. Мы выбрали этот подход для наших инструментов визуализации, потому что на каждом шаге одно составное регулярное выражение (т. е. объединение, конкатенация или звезда Клина) может быть разложено для создания необходимых новых под-GNFA. Сосредоточение внимания на одном ребре облегчает генерацию информационного сообщения, объясняющего пройденный шаг, и, таким образом, снижает лишнюю когнитивную нагрузку для студентов.

2.1.2 преобразование ndfa в regexp

В учебниках по FLAT преобразование ndfa в regexp обычно описывается либо с помощью элегантного набора уравнений, либо с помощью графового подхода, использующего диаграмму переходов ndfa. Подход, основанный на уравнениях, представляет язык машины, построенной как объединение конечного числа малых языков [13]. Нумеруя состояния машины $K=\{k_1, k_2, \ldots, k_n\}$, где k_1 - начальное состояние, регулярное выражение для всех слов, которые переводят машину из состояния k_i в состояние k_j без перехода в состояние с номером $m+1$ или больше, обозначается $R(i,j,m)$[1]. Таким образом, мы имеем, что регулярное выражение для языка ндфа N с n состояниями строится следующим образом:

$$L(N) = \bigcup \{R(1,j,n) \mid k_j \in F\}, \text{ где F - множество конечных состояний N.}$$

То есть язык N содержит все слова, которые переводят машину из начального состояния в конечное путем обхода любого состояния. Предполагая, что Δ - это набор правил перехода данной машины, регулярное выражение строится по следующему алгоритму:

$$R(i, j, n) = \begin{cases} \{a \mid (k_i a k_j) \in \Delta\} & \text{если } n = 0 \wedge i \neq j \\ \{\varepsilon\} \cup \{a \mid (k_i a k_j) \in \Delta\} & \text{если } n = 0 \wedge i = j \\ R(i, j, n-1) \cup R(i, n, n-1) R(n, n, n-1)^* R(n, j, n-1) & \text{если } n \neq 0 \end{cases}$$

Это рекурсивное уравнение утверждает, что существует два случая, когда не может быть пройдено ни одного промежуточного состояния (т.е. n=0). Первый случай - это когда $k_i \neq k_j$. В этом случае все синглтоны, потребляемые правилами, которые непосредственно переходят от k_i к k_j , являются необходимыми регулярными выражениями. Второй случай - когда $k_i = k_j$. В этом случае ε добавляется к множеству символов, потребляемых на самоцикле. Если промежуточные состояния могут быть пройдены (т. е. $n \neq 0$), то нужное регулярное выражение представляет собой объединение двух регулярных выражений. Первое генерирует все слова, которые переводят машину из k_i в k_j без перехода в состояние с номером n или больше. Второе объединяет три регулярных выражения: одно генерирует все слова, которые переводят машину из k_i в k_n без прохождения состояния, большего чем n-1, одно генерирует все слова, которые переводят машину из k_n в k_n произвольное количество раз без прохождения состояния, большего чем n-1, и одно генерирует все слова, которые переводят машину из k_n в k_j без прохождения состояния, большего чем n-1.

Графовый подход преобразует ндфа, N, в GNFA, а затем преобразует GNFA в регулярное выражение [28]. Преобразование можно представить как хирургическую операцию на направленном графе. GNFA строится, начиная с диаграммы переходов N и добавляя новое начальное состояние, новое конечное состояние и пустые переходы из нового начального

состояния в начальное состояние `N` и из конечных состояний `N` в новое конечное состояние.

[1]Промежуточное состояние обозначается как k_r , такое, что $0 \le r \le m$. Количество промежуточных состояний не имеет значения.

Рисунок 1 Начальный шаг в визуализации JFLAP.

(a) GNFA для $L=\{ab^* \cup ba^*\}$. **(b)** GNFA после одного шага.

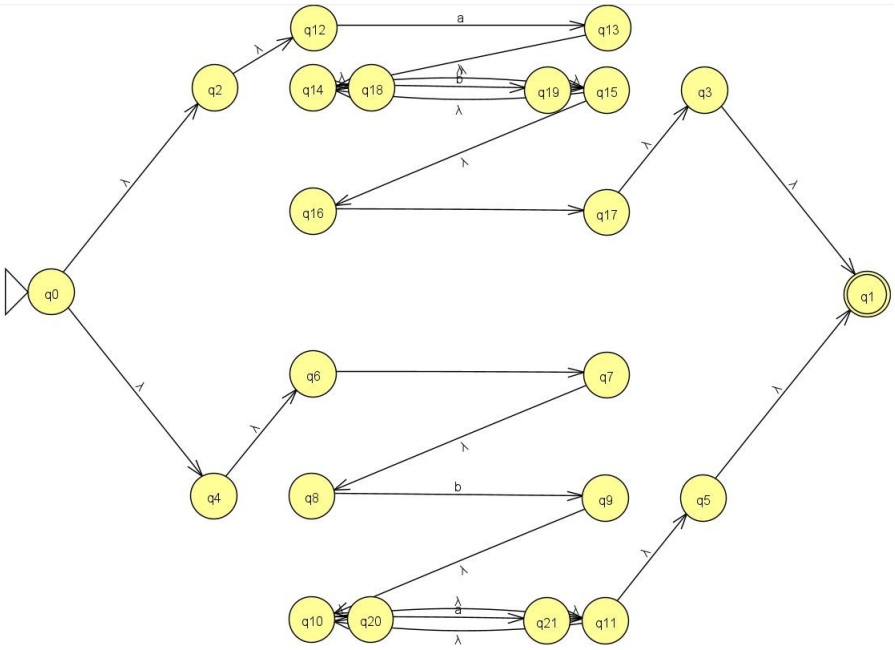
Кроме того, между любой парой состояний k_i и k_j существует единственное ребро в каждом направлении. Если существует один или несколько переходов из k_i в k_j , то меткой ребра в GNFA является union-regexp, содержащий singleton-regexp для каждого из этих переходов. Наконец, если нет ребер из k_i в k_j , то метка стрелки - это null-regexp. Такие добавленные переходы не меняют язык N, поскольку они представляют собой гипотетические переходы и никогда не могут быть использованы. Вычисление регулярного выражения продолжается путем вырывания узлов по частям, пока не останется только новое начальное состояние и новое конечное состояние. В этот момент регулярное выражение на единственном оставшемся ребре предназначено для языка N. При вырывании узла k_r заменяются ребра в k_r , (k_i a k_r), и ребра из k_r , (k_r b k_j). Если в k_r нет самоконтура, то (k_i a k_r) и (k_r b k_j) заменяются на (k_i ab k_j). Если на k_r есть самоконтур, то (k_r b k_j), (k_r c k_r) и (k_r b k_j) заменяются на (k_i ac* b k_j). Наконец, после вырывания k_r несколько ребер между узлами заменяются одним ребром, которое маркируется объединением меток нескольких ребер.

Визуализация FSM использует графовый подход, поскольку он, как правило, проще для понимания студентами, впервые изучающими FLAT. Подход с небольшими локальными шагами, заключающийся в вырывании одного узла за раз, более приемлем, чем, например, вычисление R(i,j,n), которое требует более глобального представления отношения переходов. Используемый алгоритм, как и алгоритм, описанный Сипсером [28], строит GNFA путем добавления нового начального состояния, нового конечного состояния и соответствующих пустых переходов. Однако, в отличие от него, добавление ребер с меткой null-regexp подавляется. Такие ребра не служат никакой реальной цели и для целей визуализации загромождают создаваемую графику. Вместо добавления таких ребер при вырывании узла вычисляются его предшественники и преемники, чтобы правильно подставить ребра в и из вырванного узла. Таким образом, созданные графики легче читаются студентами и снижают лишнюю когнитивную нагрузку.

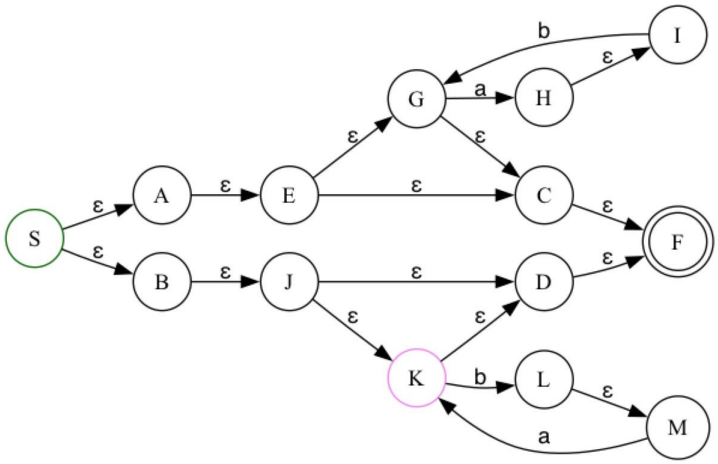
2.2 Визуализация

JFLAP - это инструмент визуализации, который поддерживает преобразование ndfa в regexp и наоборот. Чтобы перейти от ndfa к regexp, пользователь должен вручную построить ndfa. Это включает в себя графическое рисование узлов и ребер, обозначение начального и конечного состояний, а также наложение диаграммы переходов в привлекательной манере. Чтобы преобразовать regexp в ndfa, пользователь должен использовать конкретную грамматику для записи регулярного выражения. В этой грамматике используется + для объединения, * для звезды Клейна, ! для пустого слова и круглые скобки для определения порядка операций. Ни для одного из преобразований пользователю не предоставляется возможность вернуться к вычислениям и просмотреть предыдущие шаги.

Рисунок 2 Результирующие диаграммы переходов для $L=\{ab^* \cup ba^*\}$.



(a) Диаграмма переходов JFLAP.



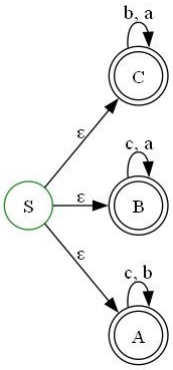
(b) Диаграмма переходов FSM.

2.2.1 Визуализация `regexp` в `ndfa`

Преобразование выполняется из `GNFA` в `regexp`. На каждом шаге разложимое регулярное выражение преобразуется с использованием свойств закрытия `ndfa` над объединением, конкатенацией и звездой Клине. Такой шаг может быть выполнен вручную пользователем или автоматически по нажатию кнопки `Do Step`. Чтобы проиллюстрировать, как выполняется шаг, рассмотрим `GNFA` на рисунке 1a для $L=\{ab^* \cup ba\}$[*2]. В результате выполнения первого шага преобразования получается `GNFA`, показанная на рисунке 1b. Ребро из `q0` в `q1` заменяется

[2]Узлы были вручную перегруппированы, чтобы иллюстрации было удобно читать.

Рис. 3 В ндфа для языка L={w | w отсутствует хотя бы один элемент в {a b c}}.



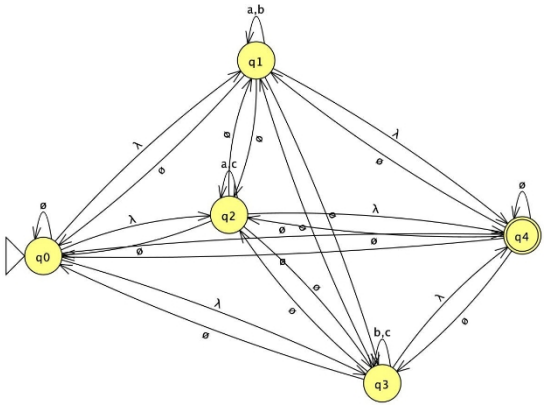
с GNFA, который начинается с q0, недетерминированно переходит к (суб)GNFA для одной из ветвей союза, и из обоих (суб)GNFA недетерминированно переходит к q1. Процесс преобразования может продолжаться по частям или может быть завершен за один шаг нажатием кнопки "Сделать все", в результате чего без ручной перестановки узлов для улучшения читабельности получается ndfa, показанная на рис. 2a. Как может оценить читатель, узлы расположены бессистемно, а некоторые ребра невозможно прочитать, что требует от пользователя перестановки узлов, чтобы сделать диаграмму переходов читаемой.

Визуализация FSM также использует графовый подход для генерации ndfa. Как и в JFLAP, для преобразования де-композируемых регексов используются свойства замыкания регулярных языков: объединение, конкатенация и звезда Клине. В отличие от них, основной целью визуализации FSM является снижение излишней когнитивной нагрузки. Для этого инструмент всегда выбирает следующий разложимый регексп для преобразования и позволяет пользователю вернуться к вычислениям, чтобы просмотреть шаги преобразования. Кроме того, отображается информационное сообщение, выделяющее ребро, которое было преобразовано. Таким образом, пользователю не нужно гадать, что произошло, если какой-то шаг ему не понятен. В отличие от JFLAP, каждая диаграмма переходов отображается с помощью Graphviz [4, 6]. Таким образом, узлы и ребра отображаются в привлекательной манере, что облегчает чтение диаграммы переходов. Например, сравните схему диаграммы переходов, полученную с помощью визуализации FSM, представленную на рисунке 2b, с ее аналогом, полученным с помощью JFLAP на рисунке 2a.

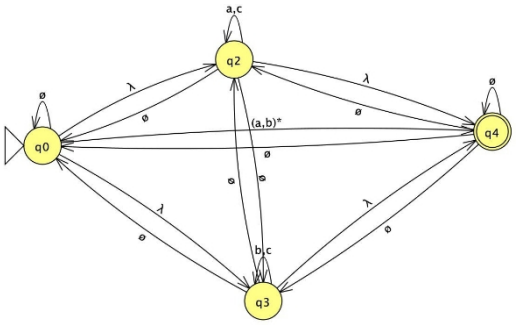
2.2.2 преобразование ndfa в regexp

Визуализация преобразования ndfa в regexp в JFLAP основана на графовом подходе, описанном выше. Пользователь вручную выполняет каждый шаг преобразования, следуя предоставленным инструкциям. В этих инструкциях пользователь должен добавить новое мертвое состояние, создать GNFA, объединив ребра с несколькими метками в объединенный regexp, добавить нуль-меченые направленные ребра между состояниями, между которыми нет ребра, и, наконец, вырвать узлы. Пользователь может выполнить каждый из этих шагов вручную или автоматически. Чтобы проиллюстрировать преобразование, рассмотрим трансформацию ndfa, показанную на рисунке 3. На рисунке 4a показано состояние визуализации после добавления нового конечного состояния и необходимых нулевых переходов между состояниями (узлы перемещены для улучшения читабельности). Несмотря на перемещение узлов для улучшения читаемости, мы можем заметить, что состояние визуализации в лучшем случае трудно читаемо. Основная проблема заключается в том, что визуализация загромождена ребрами с нулевыми метками, что приводит к перекрытию ребер. На рисунке 4b показано состояние визуализации после вырывания первого узла (в данном примере q1). Мы снова видим, что состояние визуализации трудно читаемо. Кроме того, трудно визуально определить эффект от вырывания

Рисунок 4 Состояния визуализации JFLAP.

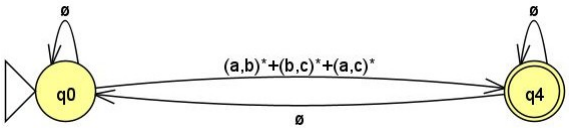


(a) Состояние визуализации после добавления края с нулевой меткой. q1.

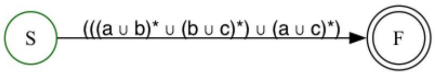


(b) Состояние визуализации после вырывания

Рисунок 5 Окончательное состояние визуализации JFLAP и FSM.



(a) Окончательное состояние визуализации JFLAP.



(b) Окончательное состояние визуализации FSM.

из узла. Наконец, на рисунке 5a показано конечное состояние визуализации JFLAP. В конечном состоянии легко различить результирующее регулярное выражение, несмотря на (бесполезные) переходы с нулевой меткой.

Как и в JFLAP, визуализация FSM также основана на графовом подходе к преобразованию. Однако, в отличие от него, на пользователя ложится гораздо меньшая нагрузка. Пользователю не нужно вручную добавлять конечное состояние, создавать GNFA, добавлять нуль-меченые направленные ребра или выбирать следующий узел для вырывания. Все это автоматически выполняется или опускается по мере того, как пользователь проходит через визуализацию. Таким образом, снижается лишняя когнитивная нагрузка. Пользователю нужно только переходить вперед и назад между состояниями визуализации с помощью клавиш со стрелками. Возможность шагать назад в трансформации позволяет пользователям визуально наблюдать, как объединяются ребра при вырывании узла. Наконец, каждая диаграмма переходов визуализируется с помощью Graphviz [4, 6], чтобы обеспечить привлекательное оформление. Например, для ndfa, представленного на рисунке 3, конечное состояние FSM визуализируется так, как показано на рисунке 5b. Читатель может оценить, что эта графика более привлекательна, чем графика, созданная JFLAP.

3 Краткое введение в FSM

FSM - это специфический язык, встроенный в Racket, для класса FLAT. В FSM машины состояний, грамматики и регулярные выражения являются первоклассными. Недетерминизм - это встроенная функция языка, которую программисты могут использовать так же, как они используют свои любимые функции в любом языке программирования. Типы FSM, актуальные для этой статьи, - регулярные выражения и автоматы конечных состояний.

Рисунок 6 Регулярное выражение FSM для L={ab* U ba*}.

```
#lang fsm

(define a (singleton-regexp "a")) (define b (singleton-regexp "b"))

(define a* (kleenestar-regexp a)) (define b* (kleenestar-regexp b))

(define ab* (concat-regexp a b*)) (define ba* (concat-regexp b a*))

;; L= ab* U ba*
(define ab*Uba* (union-regexp (concat-regexp a b*) (concat-regexp b a*)))

;; word → Boolean
;; Цель: Определить, находится ли данное слово в ab* U ba*
(define (in-ab*Uba*? w)
  (or (and (eq? (first w) 'a) (andmap (λ (s) (eq? s 'b)) (rest w)))
      (and (eq? (first w) 'b) (andmap (λ (s) (eq? s 'a)) (rest w)))))

(check-pred in-ab*Uba*? (gen-regexp-word ab*Uba*))
(check-pred in-ab*Uba*? (gen-regexp-word ab*Uba*))
(check-pred in-ab*Uba*? (gen-regexp-word ab*Uba*))
```

3.1 Регулярные выражения

Конструкторы регулярных выражений над алфавитом Σ - это конструкторы:

1. (null-regexp)
2. (empty-regexp)
3. (singleton-regexp "a"), где a∈Σ
4. (union-regexp r1 r2), где r1 и r2 - регулярные выражения
5. (concat-regexp r1 r2), где r1 и r2 - регулярные выражения
6. (kleenestar-regexp r1), где r - регулярное

выражение Функции селектора FSM для регулярных

выражений:

singleton-regexp-a: Извлекает встроенную строку
kleenestar-regexp-r1: Извлекает встроенное регулярное выражение **union-regexp-r1:** Извлекает первое встроенное регулярное выражение **union-regexp-r2:** Извлекает второе встроенное регулярное выражение **concat-regexp-r1:** Извлекает первое встроенное регулярное выражение **concat-regexp-r2:** Извлекает второе встроенное регулярное выражение

Для различения подтипов регулярных выражений определены следующие предикаты:

```
empty-regexp?     singleton-regexp?     kleenestar-regexp?
union-regexp?     concat-regexp?        null-regexp?
```

Каждый из них потребляет значение любого типа и возвращает булево значение. Наконец, наблюдатель `gen-regexp-word` принимает на вход регулярное выражение и возвращает слово на языке данного регулярного выражения. Этот наблюдатель недетерминированно решает, сколько повторений `kleenestar-regexp` нужно сгенерировать, и недетерминированно решает, какую ветвь `union-regexp` использовать при генерации.

В качестве примера программирования рассмотрим регулярное выражение `FSM`, представленное на рисунке 6 для `L={ab* U ba* }`. Код стал более читаемым благодаря независимому определению каждого необходимого `подрегламента`. Читатель может оценить, что это делает реализацию доступной практически для любого студента. В модульных тестах используется вспомогательный предикат `in-ab*Uba*?` для определения того, находится ли сгенерированное слово в `L`. Тесты выглядят одинаково, но это не так (по всей вероятности), учитывая, что при генерации каждого слова, как описано выше, недетерминистически решается количество повторений для звезды Клейна и ветвь объединения, используемая для генерации слова.

3.2 Автоматы конечного состояния

Конструкторы FSM-машин, которые представляют интерес для этой статьи, - это конструкторы автоматов с конечным состоянием:

```
make-dfa: K Σ s F δ→ dfa      make-ndfa: K Σ s F δ→ ndfa
```

`K` - список состояний, `Σ` - список символов алфавита, `s∈K` - начальное состояние, `F⊆K` - список конечных состояний, а δ - отношение перехода (для `dfa` оно должно быть функцией). Отношение перехода представляется в виде списка правил перехода. Правило перехода `dfa` - это тройка `(K Σ K)`, содержащая состояние источника, элемент для чтения и состояние назначения. Для перехода `ndfa` элемент для чтения может быть `EMP` (т. е. ничего не читается).

Наблюдателями являются:

```
(sm-states M) (sm-sigma M)   (sm-startM) (sm-finals M) (sm-rules)
(sm-type M)       (sm-apply M w) (sm-showtransitions M w)
```

Первые 5 наблюдателей извлекают компонент из данной машины состояний, `sm-type` возвращает тип данной машины состояний, `sm-apply` применяет данную машину к данному слову и возвращает `'accept` или `'reject`, а `sm-showtransitions` возвращает трассировку конфигураций, пройденных при применении данной машины к данному слову, заканчивающуюся результатом. Однако трассировка возвращается только в том случае, если машина является `dfa` или если слово принимается `ndfa`.

Наконец, `FSM` обеспечивает визуализацию машинного рендеринга и машинного исполнения. Примитивами визуализации являются:

```
(sm-граф M)      (sm-визуализировать M [(s p)* ])
```

Первая возвращает изображение диаграммы переходов данной машины. Второй запускает инструмент визуализации `FSM`. Необязательные два списка, `(s p)`, содержат состояние данной машины и инвариантный предикат для этого состояния. Выполнение машины всегда может быть визуализировано, если машина является `dfa`. Аналогично `sm-showtransitions`, выполнение машины `ndfa` может быть визуализировано только в том случае, если данное слово есть в языке машины. За более подробной информацией о визуализации выполнения машины в `FSM` читатель может обратиться к предыдущей публикации [22].

Чтобы проиллюстрировать программирование конечных автоматов в `FSM`, рассмотрим `ndfa` для решения `L={ab* U ba* }`, показанный на рисунке 7. В начале машина недетерминированно решает, находится ли данное слово в ab^* или в ba^* , и переходит, соответственно, в `A` или `D`. Юнит-тесты иллюстрируют слова, которые есть и которых нет в `L`. Обратите внимание, что программист задает только недетерминированное поведение (переходы из `S`) и не обременен

реализацией недетерминированного поведения.

Рисунок 7 FSM ndfa для L={ab* ∪ ba*}.

```
#lang fsm

;; L= ab* U ba*
(define ab*Uba*-ndfa (make-ndfa '(S A B D E)
                                '(a b)
                                'S
                                ' (B C E F)
                                `((S ,EMP A) (S ,EMP D)
                                  (A a B) (B b B)
                                  (D b E) (E a E))))

(check-equal? (sm-apply ab*Uba*-ndfa '(b b)) 'reject)
(check-equal? (sm-apply ab*Uba*-ndfa '(a a b))
'reject) (check-equal? (sm-apply ab*Uba*-ndfa '(a))
'accept) (check-equal? (sm-apply ab*Uba*-ndfa '(b))
'accept) (check-equal? (sm-apply ab*Uba*-ndfa '(a b
b)) 'accept) (check-equal? (sm-apply ab*Uba*-ndfa '(b
a)) 'accept)
```

4 Общий дизайн визуализации

Чтобы снизить лишнюю когнитивную нагрузку, визуализации FSM генерируют и собирают изображения для каждого шага преобразования. В каждом изображении есть краткое информационное сообщение, поясняющее пройденный этап. Пользователю достаточно использовать клавиши со стрелками, чтобы пройти через преобразование. Использование этих клавиш определяется следующим образом:

→ Переход к следующему шагу визуализации← Переход к предыдущему шагу визуализации
↓ Перемещение в конец визуализации *к содержанию* ↑ Переход к началу визуализации

Визуализация всегда отображает инструкции по использованию клавиш со стрелками.

Изображения хранятся в структуре viz-state, которая определяется следующим образом:

```
;; Структура, (viz-state (listof images) (listof images)),
;; содержит обработанные и необработанные изображения.
(struct viz-state (pimgs upimgs))
```

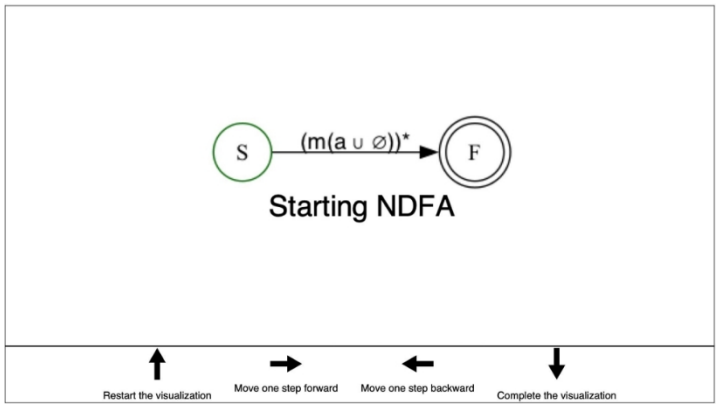
Первый список, pimgs, обозначает изображения, отображавшиеся ранее. Второй список, upimgs, обозначает изображения, которые будут отображаться. Первое изображение в upimgs - это текущее отображаемое изображение. Изначально все изображения находятся в списке upimgs. Стрелка вправо перемещает первое изображение из upimgs в pimgs, а стрелка влево - наоборот. Стрелка вниз перемещает все изображения из upimgs, кроме последнего, в pimgs. Стрелка вверх перемещает все изображения к пимгам. Каждый раз, когда делается шаг вперед или назад, внизу каждого графика размещается информационное сообщение, а также инструкции по использованию стрелки. Когда это уместно, цвет используется для выделения изменений в трансформации.

4.1 Иллюстративный пример: преобразование regexp в ndfa

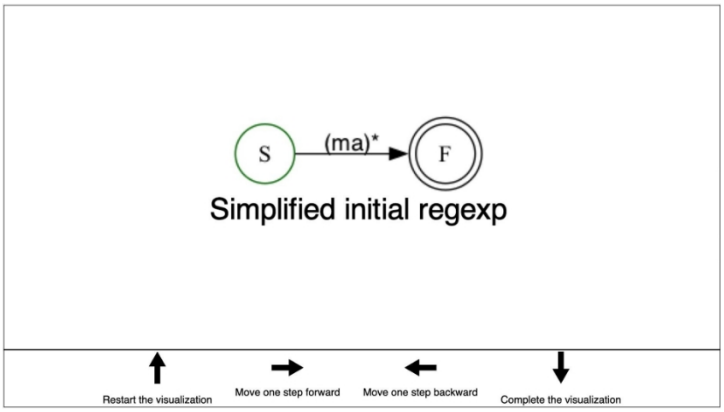
Для наглядности рассмотрим преобразование (m(a ∪ ∅))* в ndfa. Исходный GNFA визуализации представлен на рисунке 8a. Она содержит единственный переход из начального

состояния в конечное

Рисунок 8 Первые состояния визуализации при преобразовании `regexp` в `ndfa`.



(a) Начальное состояние для `(m(a ∪ ∅)) .*`

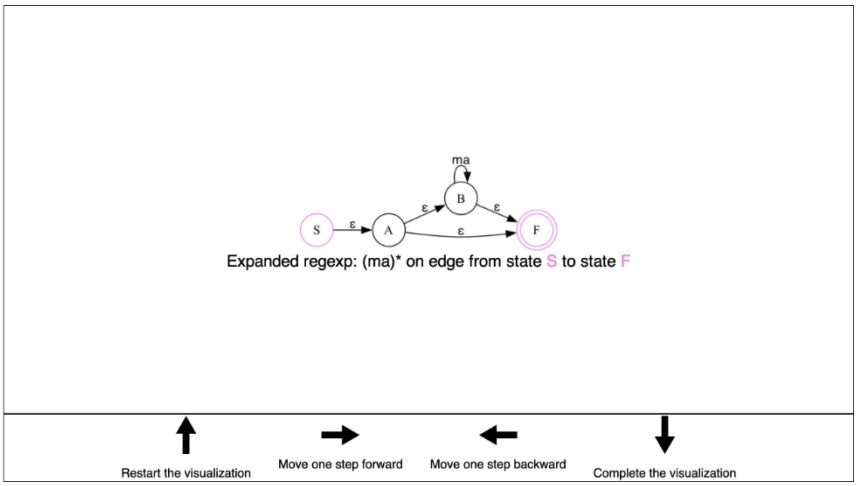


(b) Состояние после упрощения `regexp`.

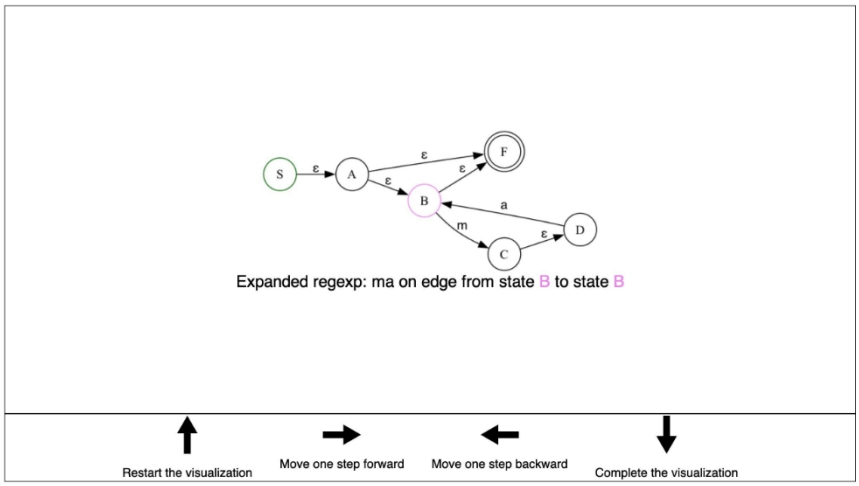
состояние, помеченное `regexp` для преобразования. Сообщение указывает на то, что это начальное (приближенное) `ndfa`. На первом шаге заданное регулярное выражение упрощается до `(ma)*` . Состояние визуализации после этого шага показано на рисунке 8b. Сообщение информирует пользователя о том, что исходное регулярное выражение было упрощено. Технически этот шаг не является необходимым, но полезен для того, чтобы сделать визуализацию более понятной для студентов, которые склонны писать слишком сложные `регекспы`. Далее преобразуется `регулярное выражение` Kleene star, которое переводит машину из состояния `S` в состояние `F`. Состояние визуализации после этого шага показано на рисунке 9a. Обратите внимание, что в сообщении указано расширенное регулярное выражение, а также исходное и конечное состояния. Как в сообщении, так и на графике эти состояния выделены фиолетовым цветом. На последнем шаге преобразования расширяется `ma`. Учитывая, что это регулярное выражение находится на самопереходе `B`, исходное и конечное состояния совпадают. Состояние визуализации после этого расширения показано на рис. 9b. Обратите внимание, что сообщение указывает на расширенный `regexp` и выделяет фиолетовым цветом одно состояние.

В любой момент трансформации пользователь может переместиться назад по трансформации, чтобы просмотреть состояния до и после визуализации. Эта функция, а также предоставляемые сообщения позволяют пользователю внимательно изучить, как продвигается трансформация на каждом шаге.

Рисунок 9 Конечные состояния визуализации при преобразовании `regexp` в `ndfa`.



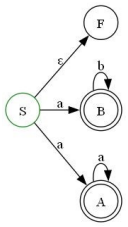
(a) Состояние визуализации после расширения `(ma) .*`



(b) Состояние визуализации после расширения ма.

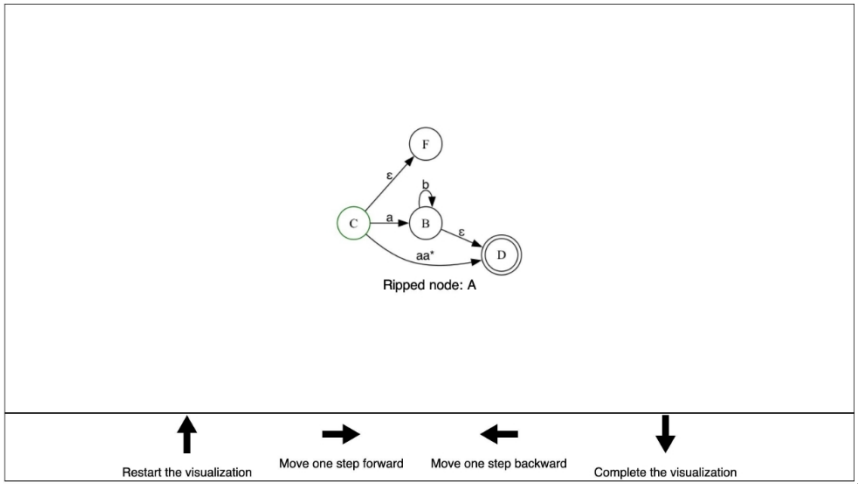
4.2 Иллюстративный пример: преобразование ndfa в regexp

Для примера рассмотрим преобразование следующего `ndfa`:

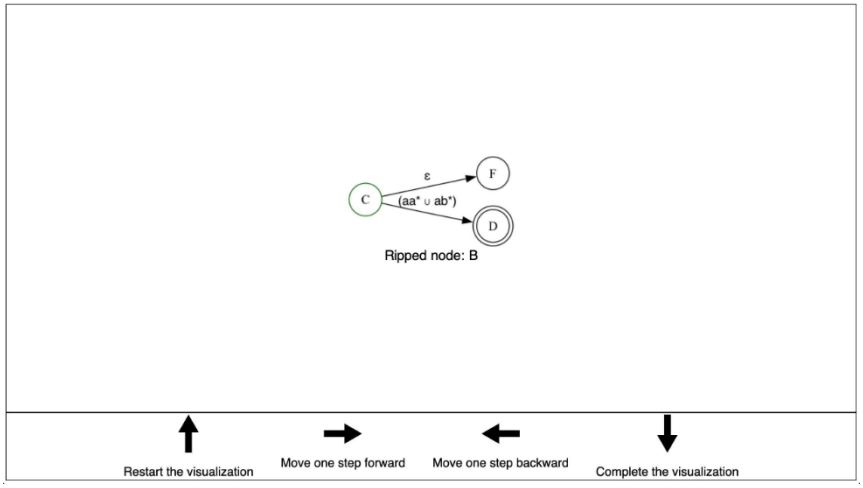


Программист без необходимости включил нефинальное состояние `F`, которое достижимо только пустым переходом из начального состояния и не имеет ни одного исходящего перехода. Шаги визуализации вырывают узел за узлом. На рисунке 10a показано состояние визуализации после вырывания `S` и `A`. Обратите внимание, что в результате вырывания этих двух узлов на aa^* есть переход из `C` в `D`. Вырывание `B`

Рисунок 10 Этапы визуализации разрывов узлов при преобразовании `ndfa` в `regexp`.



(a) Состояние визуализации после вырывания `S` и `A`.



(b) Визуализация состояния после вырывания `B`.

означает, что необходим новый переход от `C`, единственного предшественника, к `D`, единственному преемнику. Это приводит к появлению двух ребер между `C` и `D`, и поэтому они объединяются с помощью регулярного выражения union, что приводит к состоянию визуализации, показанному на рис. 10b. Наконец, вырывание `F` не влияет на ребро между `C` и `D`, которое помечено результирующим регулярным выражением.

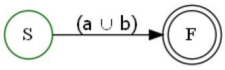
В любой момент преобразования пользователь может переместиться назад, чтобы просмотреть состояния до и после ви- суализации. Таким образом, студент может внимательно изучить, как вырываются узлы и создаются новые переходы.

5 Реализация

5.1 Построение графиков для преобразования regexp в ndfa

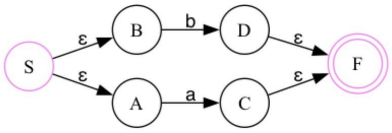
В процессе преобразования существует GNFA, переходы которой помечены произвольными регекспами. Задача состоит в том, чтобы преобразовать GNFA так, чтобы его переходы были помечены только одиночными и пустыми регекспами. На каждом шаге для преобразования выбирается переход, помеченный объединением, конкатенацией или регэкспом "звезда Клейна". Эти преобразования основаны на известных конструкторах для свойств закрытия регулярных языков [13, 20, 24, 28]. Когда регексп преобразуется, выбранный переход удаляется из GNFA и добавляются новые состояния и ребра. Для создания нового графика используется программа Graphviz.

Союзный regexp, (union-regexp r_1 r_2), маркирующий переход между двумя состояниями S и F, преобразуется путем создания четырех новых состояний: скажем, A, B, C и D. Каждая ветвь союза использует исключительно два из этих состояний, и они соединены переходом, маркированным соответствующим регулярным выражением для этой ветви. Например, A и C соединены с помощью r_1, а B и D - с помощью r_2. S соединен с A и B



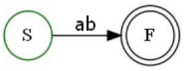
пустыми переходами. C и D соединены с F пустыми переходами. Визуально преобразование:

результаты:

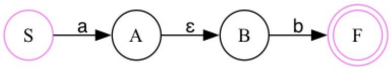


Наконец, S и F выделены фиолетовым цветом, указывая на голову и хвост заменяемого края.

Конкатенационный regexp (concat-regexp r_1 r_2), обозначающий переход между двумя состояниями S и F, преобразуется путем создания двух новых состояний: скажем, A и B. В GNFA добавляются переход из S в A, обозначенный r_1, переход из A в B, обозначенный пустым regexp, и переход из B в F, обозначенный r_2. Визуально преобразование:

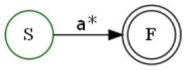


результаты:



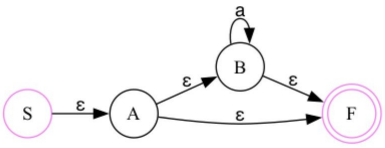
Наконец, S и F выделены фиолетовым цветом, указывая на голову и хвост заменяемого края.

Звездный регексп Клина (kleenestar-regexp r_1), обозначающий переход между двумя состояниями S и F, преобразуется путем создания двух новых состояний: скажем, A и B. S соединен с A, A соединен с B, A соединен с F, а B соединен с F переходами, помеченными пустым регекспом. Наконец, на B есть циклический переход, помеченный r_1. Визуально,



преобразование:

результаты:

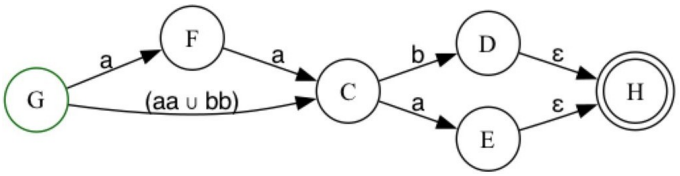


Наконец, S и F выделены фиолетовым цветом, указывая на голову и хвост заменяемого края.

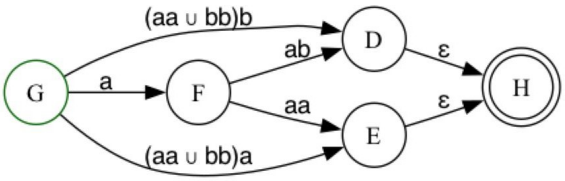
5.2 Построение графики для преобразования ndfa в regexp

Основная часть графики создается путем вырывания узлов. Вырывание узла `A` требует удаления переходов в `A` и из `A` и генерации новых переходов, соединяющих каждого предшественника `A` с каждым наследником `A`. Необходимо различать два случая: либо в `A` есть циклический переход, либо его нет.

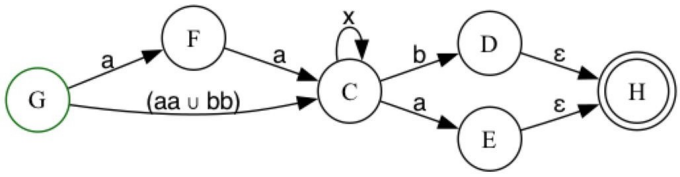
Если на `A` нет цикла, то каждый предшественник `A` соединяется с каждым преемником `A` переходом, помеченным конкатенационным `regexp`, который содержит регулярное выражение от предшественника к `A` и регулярное выражение от `A` к преемнику. Например, если `(M` r_1 `A)` и `(A` r_2 `N)` являются переходами в текущем `GNFA`, то эти два перехода удаляются и заменяются на `(M r` r_{12} `N)`. Визуально, если текущий `GNFA` имеет вид:



Вырывание `C` означает, что `G` и `F` должны быть соединены с `E` и `D`. Новые ребра помечаются конкатенацией каждого ребра в `C` и каждого ребра из `C`. В результате получается `GNFA`:



Если `A` имеет цикл на самом себе, то каждый предшественник `A` соединяется с каждым преемником `A` переходом, помеченным конкатенацией регулярного выражения от предшественника к `A`, регулярного выражения звезды Клейна для регулярного выражения цикла и регулярного выражения к преемнику `A`. Например, если `(M` r_1 `A)`, `(A` r_2 `A)` и `(A` r_3 `N)` являются переходами в текущем `GNFA`, то при вырывании `A` эти три перехода удаляются и заменяются на `(M r r` $r_1^*{}_3$ `N)`. Визуально, если текущий `GNFA` имеет вид:



Вырывание `C` означает, что `F` и `G` должны иметь переходы в `E` и `D`. Новые ребра маркируются конкатенацией каждого ребра в `C`, регулярным выражением Kleene star, содержащим метку на цикле `C`, и ребрами из `C`. Результирующая `GNFA` имеет вид:

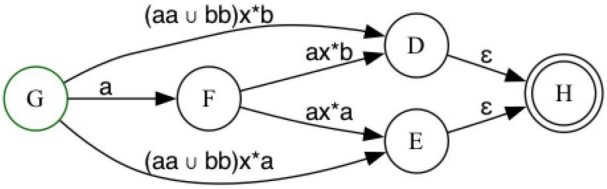
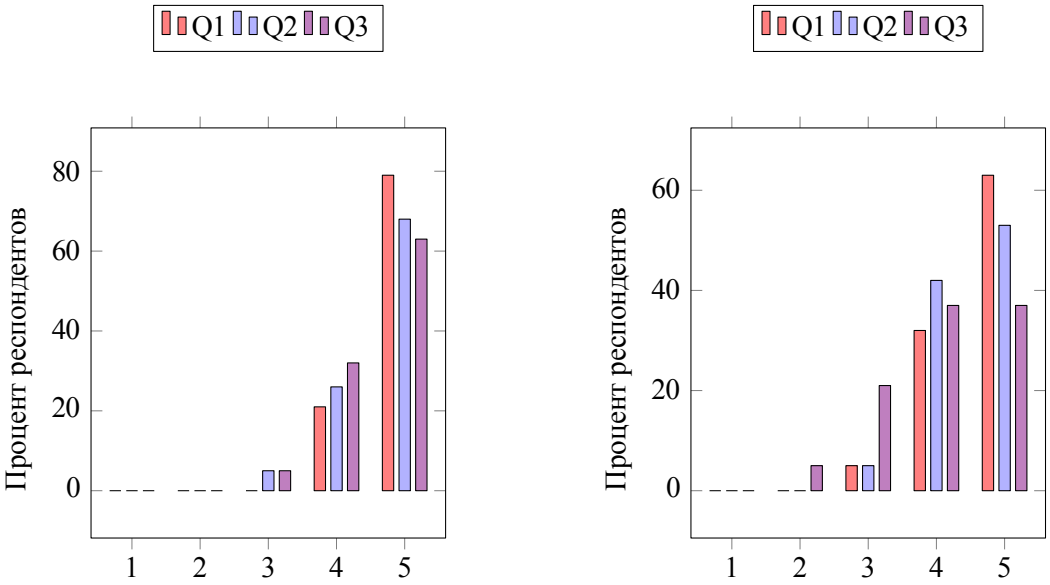


Рисунок 11 Распределение ответов в контрольной группе.



(a) преобразование `ndfa` в `regexp`

(b) `regexp` в `ndfa`

6 Эмпирические данные

Для первоначальной оценки полезности наших новых учебных инструментов, прежде чем внедрять их в учебный процесс, мы собрали эмпирические данные в фокус-группе с помощью добровольного анонимного опроса[3]. В опросе приняли участие 19 студентов (5 студентов-старшекурсников Сетон-Холла, которые еще не посещали занятия по `FLAT`; 13 студентов-старшекурсников Instituto Universita´rio de Lisboa, которые в настоящее время посещают занятия по `FLAT`, и 1 аспирант, который работает ассистентом преподавателя `FLAT` в Instituto Universita´rio de Lisboa).[4]. Все они были ознакомлены с преобразованиями `ndfa` в `regexp` и `regexp` в `ndfa` с помощью инструментов `FSM`, описанных в этой статье. Перед изучением преобразований студенты получили краткое введение в `FSM` с упором на программирование ndfas и `regexp`. После изучения каждого из преобразований и использования инструментов визуализации студенты прошли опрос, в котором были заданы следующие вопросы об инструменте визуализации `ndfa` to `regexp`:

> Q1: В целом, насколько полезна визуализация для понимания преобразования ndfa в regexp? Q2: Насколько сложно использовать визуализацию?
> Вопрос 3: Насколько сложно понять визуализированное преобразование?

Респонденты отвечали по шкале Лайкерта [14]. В вопросе 1 используется шкала от [1] Совсем не полезно до
[5] Чрезвычайно полезно. В вопросах 2 и 3 используется шкала от [1] Чрезвычайно трудно до [5] Чрезвычайно легко. Распределение ответов представлено на рисунке 11a.

Ответы на вопрос `1` показывают, что все респонденты считают инструмент визуализации полезным для понимания преобразования из `ndfa` в `regexp` (ответы 4 и 5). Такие результаты не ожидались, учитывая, что большинство респондентов практически не имеют опыта работы с формальными языками и теорией автоматов. Это говорит о том, что визуализация полезна даже для новичков в `FLAT`.

[3] Никто из добровольцев не получил никаких льгот за свое участие.

[4] Только 5 добровольцев из Университета Сетон Холл знакомы с `Racket`-подобными языками (в частности, со студенческими языками `Racket`, используемыми в [2, 18, 19]).

Ответы на вопрос Q2 показывают, что большинство респондентов, 94%, считают, что инструмент визуализации прост в использовании (ответы 4 и 5). Это говорит о том, что усилия, направленные на снижение посторонней когнитивной нагрузки, связанной с обучением использованию визуализации, оказались успешными.

Ответы на вопрос Q3 показывают, что большинство респондентов, 95%, считают, что визуализированная трансформа-.
преобразования легко понять. Это также неожиданный результат, учитывая, что большинство респондентов не были знакомы с алгоритмами преобразования. Это говорит о том, что размер каждого шага в визуализации делает преобразование доступным для новичков.

Вторая часть опроса была посвящена преобразованию `regexp` в `ndfa`. Опрос включал на следующие вопросы:

Q1: В целом, насколько полезна визуализация для понимания преобразования regexp в ndfa? B2: Насколько сложно использовать визуализацию?
Вопрос 3: Насколько сложно понять визуализированное преобразование?

Ответы на эти вопросы также даются по шкале Лайкерта [14]. В вопросе 1 используется шкала от [1] Совсем не полезно до [5] Чрезвычайно полезно. В вопросах 2 и 3 используется шкала от [1] Чрезвычайно трудно до [5] Чрезвычайно легко. Распределение ответов представлено на рисунке 11b.

Что касается первого вопроса, то мы видим, что респонденты твердо уверены (95% ответили 4 или 5), что визуализация
инструмент полезен для понимания преобразования от `regexp` к `ndfa`. Эти результаты неожиданны, поскольку большинство респондентов, как было замечено ранее, не имеют опыта работы с формальными языками и теорией автоматов. Наряду с результатами, полученными для Q1 для предыдущего преобразования, это говорит о том, что предоставление визуальной трассировки алгоритмов построения полезно для студентов всех уровней подготовки.

Для Q2 мы видим, что большинство респондентов, 95%, считают, что визуализация проста в использовании. Это говорит о том, что наши усилия по снижению посторонней когнитивной нагрузки увенчались успехом. Мы связываем это с простым в использовании интерфейсом со стрелками и информативными сообщениями на каждом шаге.

Для Q3 мы отмечаем, что большинство респондентов, 74%, считают, что трансформация не требует больших усилий.
понимают (ответы 4 и 5). Значительное меньшинство респондентов, 21 %, считают, что понимают меньше (ответ 3). Такое распределение вполне ожидаемо для студентов, начинающих изучать FLAT, поскольку для полного понимания этого преобразования респонденты должны быть знакомы со свойствами закрытия регулярных языков и соответствующими алгоритмами построения. Тем не менее, эти результаты очень обнадеживают, учитывая, что даже новички считают, что понимают это преобразование.

Кроме того, респондентам были заданы качественные вопросы. На вопрос о том, какие характеристики инструментов визуализации им больше всего нравятся, были получены следующие ответы:

"Возможность пошагового прохождения каждого шага в визуализации очень полезна. Я думаю о том, как мои ученики могут не понять тот или иной шаг, и я могу просто прокручивать его туда-сюда столько, сколько мне нужно :) Мне также очень понравились сообщения, которые объясняют, что было сделано на каждом шаге, мне кажется, что они действительно помогают следить за тем, что происходит!"

"Так гораздо проще жонглировать разными состояниями в моей голове".

"Мне нравится фиолетовое выделение узла, который должен быть разрушен".

"Они довольно просты и понятны.

Цвета приятные, и следовать им тоже легко".

"Очень легко понять, что происходит. Я обязательно буду использовать его для обучения".

Эти отзывы свидетельствуют о том, что, благодаря воспринимаемой ясности и удобочитаемости, студенты в рамках курса будут охотно пользоваться инструментами визуализации.

На вопрос о том, что им меньше всего нравится в инструментах визуализации, были получены следующие ответы:

"Длинные названия государств I-xyzw... могут, на мой взгляд, сделать визуализацию несколько чрезмерной".

"Может быть, названия новых штатов должны иметь лучшую схему наименования вместо случайных имен".

"Движение узлов вместо статичных точек и растущая рамка".

Первые два комментария относятся к предыдущим названиям, которые случайным образом генерировались для новых штатов. Прежние названия включали случайное 6-значное натуральное число (например, I-872431). В свете вышеупомянутых отзывов и перед публикацией случайная генерация названий штатов была обновлена и теперь включает только случайное число, если это необходимо. Новый метод генерации позволяет получить кратчайшее возможное имя штата, не используемое при составлении ндфа. Использование нового метода генерации случайных имен штатов отражено в предыдущих разделах этой статьи (т. е. случайное имя штата с 6-значным натуральным числом не генерируется ни для одного из использованных примеров).

Вторая проблема связана с размещением узлов в сгенерированных графах. Учитывая сложность рисования графов, основным стимулом для современных исследований в области автоматизированного рисования графов является облегчение визуального анализа различных видов сложных сетевых или связанных систем [12]. Было создано и успешно внедрено несколько библиотек для рисования графов. Среди наиболее широко используемых - Graphviz [5], а FSM использует Graphviz для создания своих диаграмм. Graphviz, однако, не обеспечивает контроля над узлами
размещение, и мы должны принять движение государства по мере роста диаграмм.

7 Заключительные замечания

В этой статье представлены новые средства визуализации FSM для преобразования ndfa в regexp и regexp в ndfa. Визуализации одновременно отображают изображения диаграмм переходов и выводят информационные сообщения, чтобы помочь пользователю ориентироваться в преобразовании. Это улучшает предыдущие подходы, отображая диаграммы переходов в привлекательной манере. Кроме того, инструмент визуализации regexp to ndfa имеет соответствующее цветовое кодирование состояний, чтобы наглядно показать, какое ребро было расширено. Средства визуализации FSM, в отличие от других средств визуализации для этих преобразований, могут продвигаться как вперед, так и назад. Наконец, оба преобразования могут быть завершены за один шаг или перезапущены одним щелчком мыши из любой точки вычислений, не мешая пользователю перемещать симуляцию вперед и назад. Все эти действия выполняются нажатием клавиш со стрелками. Таким образом, снижается дополнительная когнитивная нагрузка, связанная с обучением работе с инструментами.

Будущая работа включает использование описанных инструментов в классе и измерение впечатлений студентов. Мы предполагаем использовать визуализации, чтобы помочь

студентам понять формальные утверждения. А именно,

Планируется знакомить студентов с алгоритмами преобразования, используя формальную нотацию (учитывая, что студентам важно понимать формальные утверждения), а затем использовать средства визуализации, чтобы помочь студентам понять формальную нотацию и реализовать алгоритмы в FSM. Будущая работа также включает разработку средств визуализации для алгоритмов построения, основанных на закрывающих свойствах регулярных языков, включая объединение, конкатенацию, звезду Клине, дополнение и пересечение. Кроме того, мы расширяем сферу применения наших средств визуализации на деривации для регулярных, контекстно-свободных и контекстно-чувствительных грамматик. Цель - помочь студентам понять, почему то или иное слово является членом языка, с помощью создания деревьев разбора.

Благодарности.

Авторы благодарят Филипе Александра Азиньиса дос Сантоса и Альфонсо Мануэля Баррала Каникё из Университета Лиссабоны за приглашение в их класс для проведения исследования в контрольной группе. Кроме того, авторы благодарят Оливию Кемпински, Андреса Мальдонадо, Жози Дез Розьер и Шамиля Джатдоева за их отзывы о предыдущих версиях этой рукописи.

Ссылки

[1] Ричард Столлман и др. (2023): *Руководство по GNU Emacs*, издание версии 29.1. Free Software Foundation, Inc. Последнее обращение: Ноябрь 2023.

[2] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt & Shriram Krishnamurthi (2018): *Как разрабатывать программы: Введение в программирование и вычисления*, второе издание. MIT Press, Cambridge, MA, USA.

[3] Мэтью Флэтт, Роберт Брюс Финдлер и PLT: *Руководство по рэкету*. Доступно по адресу https://docs.racket-lang. org/guide/. Последнее обращение 2023-07-07.

[4] Эмден Р. Ганснер и Стивен С. North (2000): *Система визуализации открытых графиков и ее применение в программной инженерии*. *Softw. Pract. Exper.* 30(11), p. 1203-1233, doi:10.1002/1097-024X(200009)30:*11<1203*::*AID-SPE338>3*.0.CO;2-N.

[5] Эмден Р. Ганснер и Стивен С. North (2000): *Система визуализации открытых графиков и ее применение в программной инженерии*. *Softw. Pract. Exper.* 30(11), p. 1203-1233, doi:10.1002/1097-024X(200009)30:*11<1203*::*AID-SPE338>3*.0.CO;2-N.

[6] E.R. Gansner, E. Koutsofios, S.C. North & K.-P. Vo (1993): *Техника рисования направленных графов*. *IEEE Transactions on Software Engineering* 19(3), pp. 214-230, doi:10.1109/32.221135.

[7] Герман Грубер и Маркус Хольцер (2014): *From Finite Automata to Regular Expressions and Back-A Sum- mary on Descriptional Complexity*. In Zolta´n E´ sik & Zolta´n Fu¨lo¨p, editors: *Proceedings 14th International Conference on Automata and Formal Languages, AFL 2014, Szeged, Hungary, May 27-29, 2014*, *EPTCS* 151, pp. 25-48, doi:10.4204/EPTCS.151.2.

[8] Мэри Хегарти (2004): *Динамические визуализации и обучение: Getting to the Difficult Questions*. *Learning and Instruction* 14(3), pp. 343-351, doi:10.1016/j.learninstruc.2004.06.007. Динамические визуализации и обучение.

[9] Джон Э. Хопкрофт, Раджив Мотвани и Джеффри Д. Ульман (2006): *Введение в теорию автоматов, языков и вычислений (3-е издание)*. Addison-Wesley Longman Publishing Co., Inc., USA.

[10] Эдвин Л. Хатчинс, Джеймс Д. Холан и Дональд А. Норман (1985): *Интерфейсы прямого манипулирования*. *Hum.- Comput. Interact.* 1(4), p. 311-338, doi:10.1207/s15327051hci0104 2.

[11] Дональд Е. Кнут, Джеймс Х. Моррис-младший и Вон Р. Пратт (1977): *Быстрое сопоставление шаблонов в строках*. *SIAM Journal on Computing* 6(2), pp. 323-350, doi:10.1137/0206024.

[12] Е. Круя, Дж. Маркс, А. Блэр и Р.К. Уотерс (2001): *Краткая заметка об истории рисования графиков*. In
P. Mutzel, M. Junger & S. Leipert, editors: *International Symposium on Graph Drawing (GD)*, Lecture Notes in Computer Science, Springer, pp. 272-286, doi:10.1007/3-540-45848-4 22. Доступно на https://www.merl.com/publications/TR2001-49.

[13] Гарри Р. Льюис и Христос Х. Пападимитриу (1997): *Элементы теории вычислений*, 2-е издание. Prentice Hall PTR, Upper Saddle River, NJ, USA, doi:10.1145/300307.1040360.

[14] Ренсис Лайкерт (1932): *Техника измерения установок*. *Archives of Psychology* 140, pp. 1-55.

[15] Питер Линц (2011): *Введение в формальные языки и автоматы*, 5-е издание. Jones and Bartlett Publishers, Inc., USA.

[16] Джон К. Мартин (2003): *Введение в языки и теорию вычислений*, 3 издание. McGraw-Hill, Inc., New York, NY, USA.

[17] Мостафа Камель Осман Мохаммед (2020): *Преподавание формальных языков с помощью визуализаций, симуляторов, упражнений с автоматической оценкой и программированных инструкций*. In Jian Zhang, Mark Sherriff, Sarah Heckman, Pamela A. Cutter & Alvaro E. Monge, editors: *Труды 51-го Технического симпозиума ACM по образованию в области компьютерных наук, SIGCSE 2020, Портленд, ИР, США, 11-14 марта 2020 г.*, ACM, с. 1429, doi:10.1145/3328778.3372711.

[18] Марко Т. Моразайн (2022): *Animated Problem Solving - An Introduction to Program Design Using Video Game Development*. Texts in Computer Science, Springer, doi:10.1007/978-3-030-85091-3.

[19] Марко Т. Моразайн (2022): *Анимированный дизайн программ - промежуточный дизайн программ с использованием разработки видеоигр*. Texts in Computer Science, Springer, doi:10.1007/978-3-031-04317-8.

[20] Марко Т. Моразайн (2024): *Формальные языки и теория автоматов на основе программирования - проектирование, реализация, проверка и доказательство*. Texts in Computer Science, Springer, doi:10.1007/978-3-031-43973-5.

[21] Marco T. Moraza´n & Rosario Antunez (2014): *Функциональные автоматы - формальные языки для студентов, изучающих компьютерные науки*. In James Caldwell, Philip K. F. Ho¨lzenspies & Peter Achten, editors: *Proceedings 3rd International Workshop on Trends in Functional Programming in Education*, *EPTCS* 170, pp. 19-32, doi:10.4204/EPTCS.170.2.

[22] Марко Т. Моразайн, Джошуа М. Шаппель и Сачин Махашабде (2020): *Визуальное проектирование и отладка детерминированных автоматов конечных состояний в FSM*. *Electronic Proceedings in Theoretical Computer Science* 321,
pp. 55–77, doi:10.4204/eptcs.321.4.

[23] Доминик Перрен (1990): *Глава 1 - Конечные автоматы*. In Jan Van Leeuwen, editor: *Formal Models and Semantics*, Handbook of Theoretical Computer Science, Elsevier, Amsterdam, pp. 1-57, doi:10.1016/B978- 0-444-88074-1.50006-8.

[24] Элейн Рич (2019): *Automata, Computability and Complexity: Theory and Applications*. Pearson Prentice Hall.

[25] Арнольд Роббинс (2015): *Эффективное программирование на Awk (4-е изд.)*. O'Reilly Media, Inc., США.

[26] Сьюзан Х. Роджер (2006): *JFLAP: Интерактивный пакет формальных языков и автоматов*. Jones and Bartlett Publishers, Inc., USA.

[27] Сьюзан Х. Роджер, Барт Бресслер, Томас Финли и Стивен Рединг (2006): *Превращение теории автоматов в практический курс*. Даг Болдуин, Пол Т. Тайманн, Сьюзан М. Халлер и Ингрид Рассел, редакторы: *Труды 37-го Технического симпозиума SIGCSE по образованию в области компьютерных наук, SIGCSE 2006, Хьюстон, Техас, США, 3-5 марта 2006 г.*, ACM, с. 379-383, doi:10.1145/1121341.1121459.

[28] Майкл Сипсер (2013): *Введение в теорию вычислений*, 3-е издание. Cengage Learning.

[29] Джон Свеллер, Йерун Дж. Г. ван Мерриенбоер и Фред Паас (1998): *Cognitive Architecture and Instructional Design*. *Educational Psychology Review* 10, pp. 251-296, doi:10.1023/A:1022193728205.