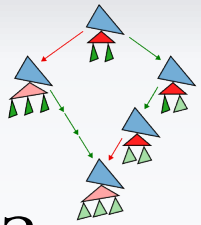
Дополнительное задание на +3

Задание полностью обесценится после 9 октября. 3 балла не входят в стоимость основной л.р. За списанные регулярки в пункте 2 буду нещадно собирать компромат.

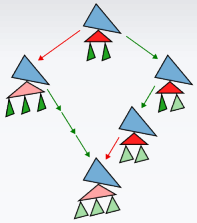
1. Выяснить, какой алгоритм парсинга используется в какой-нибудь regex-библиотеке языка из верхней правой четверти графика RedMonk. Описать его на $\frac{1}{2}$–$\frac{2}{3}$ страницы.
2. Протестировать выбранную библиотеку на следующих регулярных выражениях: `((((a)*a)*)*...a)*` (k раз итерируется операция приписывания буквы `a` к предыдущему выражению и применение к результату итерации Клини), `(a?)(a?)...(a?)aa..a` (k раз `a?`, и затем k раз просто `a`), `(.)*b(.)[n]` (n произвольных букв после буквы `b`); а также двух сериях регулярных выражений на ваш выбор. Построить график зависимости скорости распознавания слова от параметра k (значения параметров лучше брать с шагом, $\geqslant 3$, или хотя бы $\geqslant 2$).



Коллективная часть

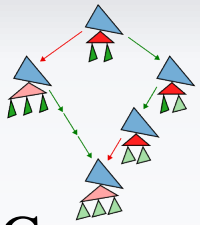
Задания со знаком [+] — для всех.

1. Построение производных и частичных производных, определение длины накачки и семантической детерминированности[+];
2. Построение НКА по классическим алгоритмам (автоматы Глушкова, Томпсона, Антимирова, follow-автомат), методы парсинга [+];
3. Базовые алгоритмы над НКА и ДКА (детерминизация, удаление эпсилон-правил, удаление/добавление ловушки, минимизация, обращение, отрицание, пересечение, ...)[+];
4. Проверка автоматов на равенство, бисимилярность, эквивалентность их языков (последнее — для контроля), вложение языков, мера неоднозначности[+];
5. Интеграция с лабораторными прошлого года (преобразование автомата в regex) и с лабораторными этого года (трансформационный моноид, нормализация по списку правил).
6. Тайпчекер (статический и динамический) + основной разбор[+].



Индивидуальные варианты

- Вариант $= 0 \mod 3$: нормализация regex по пользовательским правилам переписывания.
- Вариант $= 1 \mod 3$: построение регулярных подвыражений для regex.
- Вариант $= 2 \mod 3$: построение трансформационного моноида.



Нормализация regex: синтаксис

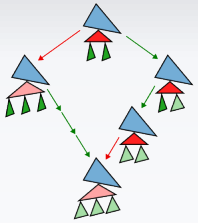
Синтаксис правил переписывания (в нелинейной форме):

<rule> ::= <regex> = <regex>
<regex> ::= (<regex><binary><regex>)
| <symbol><unary> | (<regex>)<unary> | ε
<binary> ::= | | ε
<unary> ::= * | ε

Ассоциативность не предполагается. "Унарный" ε — это отсутствие операции (введено для сокращения записи правил синтаксиса). Пустая бинарная операция — конкатенация.

Вводимое регулярное выражение имеет синтаксис <regex>. Если в правилах переписывания встречаются строчные латинские буквы, которых нет во входном регулярном выражении, их следует понимать как переменные (и только их).

Общая форма входа: <regex> (пустая строка) <rule>*. Либо можно читать выражение и список правил из разных входных файлов (вариант: что-то из потока входа, что-то из файла).

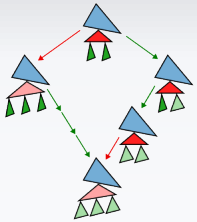


Нормализация: алгоритм

1. Если никакой подтерм выражения не унифицируется с левой частью никакого правила — нормализация завершена.
2. Иначе выбираем самое первое правило в списке, которое унифицируется с некоторым подтермом. Выбираем самый внешний подтерм регулярного выражения, удовлетворяющий условиям унификации, после чего осуществляем перестройку дерева, подставив вместо него правую часть правила, в которой все переменные заменены на их значения, полученные унификацией.

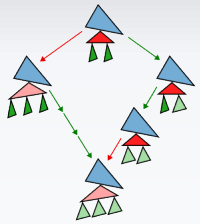
Результатом работы программы является нормализованное регулярное выражение либо зацикливание.

В данном алгоритме используется ограниченная унификация: одно из унифицируемых выражений не содержит переменных.



Нормализация: детали

- Авторы самых быстрых алгоритмов из класса работающих корректно и сданных в срок (причём минимум за 12 часов до конца нештрафного срока) получит от +1 до +3 баллов бонуса.
- Можно подключить проверку правил переписывания на завершаемость с помощью условия Кнута–Бендикса (для этого необходимо доопределить лексикографический порядок на словаре регулярного выражения) и выдавать пользователю соответствующее предупреждение. +1 балл, для сдавших в срок, и +3 балла, если на вашем языке Кнута–Бендикса не сдавали.
- Если вы пишете на языке, на котором нет реализаций Мартелли-Монтанари в первой лабораторной, то получаете ещё +1 балл.



Построение регулярных подвыражений

На вход алгоритму подаётся единственное регулярное выражение R.

Синтаксис входных данных несколько иной, чем в нулевом варианте (есть ассоциативные операторы, допустима позитивная итерация):

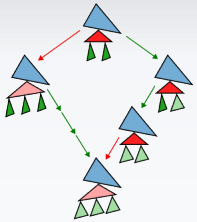
<regex> ::= <regex><binary><regex> | (<regex>)
| <regex><unary> | <symbol> | ε

<binary> ::= | | ε

<unary> ::= * | +

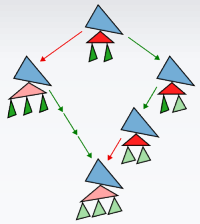
Приоритет операций: * > конкатенация > альтернатива.

Выход алгоритма: список регулярных выражений T_i таких, что $\exists u, v (u\, L(T_i)\, v = L(R))$. При этом среди регулярных выражений T_i могут встречаться выражения, описывающие эквивалентные друг другу языки.



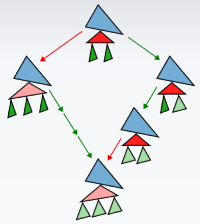
Описание алгоритма

1. Нулевой шаг: построение словаря выражения и задание на нём лексикографического порядка, необходимого для ACI-упрощений.
2. Первый шаг: построение автомата Брзозовски для R. Состояния автомата определят все суффиксные регулярные выражения S_i для R.
3. Второй шаг: построение автоматов Брзозовски для S_i^R. Их состояния $Q_{i,j}$ определят реверсы искомых T_i.
4. Заключительный шаг — обращение $Q_{i,j}$ и удаление из итогового списка регулярных выражений дубликатов.



Детали

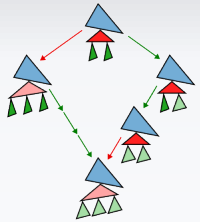
- Чтобы конструкция автомата Брзозовски была конечной, нужно использовать ACI-правила упрощения регулярных выражений, чтобы все состояния всех автоматов были нормализованы. Поскольку | вводится как ассоциативная операция, следует преобразовать входное выражение к неассоциативной (относительно альтернативы) форме.
- В итоговом списке могут быть выражения, описывающие один и тот же язык, и выражения T_i такие, что их язык вкладывается в язык некоторых других T_j (т.е. избыточные). Авторы алгоритмов, реализующих вывод результатов в наиболее лаконичных формах, получат бонусы от +1 до +3 баллов (из работающих корректно и сданных в срок — см. вариант номер 0).



Ещё об интеграции

Упрощение регулярных подвыражений тесно связано с нулевой задачей — переписывание по пользовательским правилам. Если вы используете переписывающий алгоритм вашего коллеги и подберёте удачные упрощающие правила для представления результата, верифицированные с точки зрения завершаемости — +2 балла получат оба (но только один раз, и только сдавшие в срок, см. вариант номер 0).

Если при этом упрощающая функция (Simplify) и нормализующий алгоритм (Normalize, с чтением правил из внешнего источника) будут встроены в групповой проект, то капитан группы получит +1 балл, дизайнер тоже +1, интегратор +2 балла, и оба участника индивидуальных проектов — по баллу.



Трансформационный моноид

На вход алгоритма подаётся детерминированный конечный автомат (то, что он детерминированный, необходимо проверить, и в противном случае вывести сообщение о некорректности входных данных).

Первая строка автомата — это перечисление начального состояния (первый элемент кортежа) и множества конечных состояний (второй элемент, т.е. в фигурных скобках). Далее вводятся правила перехода.

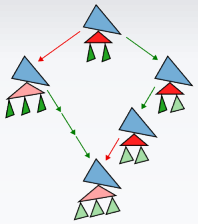
⟨First line⟩ ::= <⟨state⟩,{⟨state⟩(,⟨state⟩)* }>

⟨transition⟩ ::= <⟨state⟩,⟨letter⟩>->⟨state⟩

⟨state⟩ ::= [A–Z][0-9]?

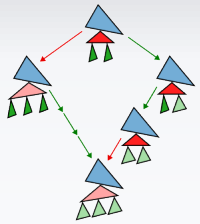
⟨letter⟩ ::= [a–z]

В автомате могут присутствовать состояния-ловушки, и состояния, не достижимые из начального состояния. Такие состояния необходимо удалить.



Т.М.: алгоритм

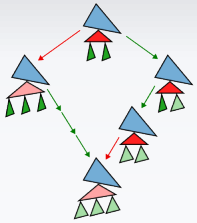
- Определяем лексикографический порядок на входном алфавите автомата.
- Пока хотя бы одно слово внутри итерации вызывает какие-нибудь изменения в моноиде, перебираем все слова длины k, упорядоченные лексикографически, с увеличением k:
  1. если очередное слово w упрощается с помощью хотя бы одного уже построенного правила переписывания, переходим к следующему слову в списке.
  2. иначе строим множество пар $M_w = \{(q_i, q_j)\}$ таких, что $q_i \xrightarrow{w} j$. Если $M_w = M_v$ для какого-то из ранее рассмотренных v, то добавляем правило $w \to v$ в моноид; если M_w уникально, то добавляем w в классы эквивалентности моноида.



Т.М.: анализ результатов

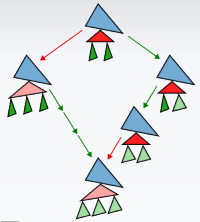
Помимо списка правил моноида и множества его классов эквивалентности $\{w_i\}$ относительно указанных правил, требуется следующее:

- список классов эквивалентности w_i таких, что $w_i \in \mathscr{L}(\mathscr{A})$ (т.е. входят в язык автомата).
- для каждого класса w:
  - список классов эквивалентности v_i таких, что $v_i\, w \in \mathscr{L}(\mathscr{A})$;
  - список классов эквивалентности v_i таких, что $w\, v_i \in \mathscr{L}(\mathscr{A})$;
  - список пар $\langle v_i, v_j \rangle$ таких, что $v_i\, w\, v_j \in \mathscr{L}(\mathscr{A})$;
  - состояние q_i, к которому w синхронизирует $\mathscr{A}$, либо сообщение, что w не синхронизирующее слово.



Т.М.: детали

1. Самая быстрая среди корректных и сданных в срок реализация получит от +1 до +3 бонусных балла.
2. Если дополнительно $\mathscr{A}$ будет тестироваться на минимальность и в случае удачи из трансформационного моноида будет извлекаться информация о классах эквивалентности по Майхиллу–Нероуду (с перечислением представителей этих классов и суффиксов, которые их различают, в форме таблицы), это добавит 2 балла (касается сдающих в срок).
3. Интеграция этой лабораторной работы в группы (функции ClassLength (самое длинное слово в классе эквивалентности), ClassCard (число классов эквивалентности), MyhillNerode — число классов эквивалентности по М.Н.) добавит +2 балла сдающему, +1 балл капитану, +1 балл дизайнеру группы и +2 балла интегратору.



Входные данные

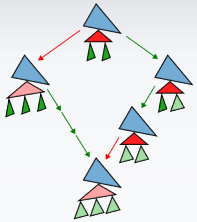
Входные данные — это список операций, которые могут иметь один из трёх видов:

- Объявление:
  `[идентификатор] = ([функция].)*[функция]? [объект]+ (!!)?`
- Специальная форма `test`
- Предикат `[предикат] [объект]+`

Объект — это фиксированное регулярное выражение, натуральное число либо идентификатор. Символ `!!` в конце строки означает, что результаты исполнения функций, указанные в данном объявлении, требуется пошагово отобразить в выходном файле. Результаты и аргументы вычисления предикатов и `test`-функций всегда отображаются в выходном файле.

Пример входа:

```
N1 = Glushkov ((ab)*|a)* !!
N2 = Width N1
N3 = Reverse.MergeBisim.Reverse N1
SemDet N3
```



Функции преобразователя

Преобразования со сменой класса

```
Thompson: RG -> NFA        Antimirov: RG -> NFA        Glushkov: RG -> NFA
IlieYu: RG -> NFA          Arden: NFA -> Regex
```

Преобразования внутри класса

```
Determinize: NFA -> DFA    Reverse: NFA -> NFA         Complement: DFA -> DFA
RemEps: NFA -> NFA         Annote: NFA -> DFA          DeAnnote: NFA -> NFA
Linearize: Regex -> Regex  DeLinearize: NFA ->NFA      DeAnnote: Regex -> Regex
Minimize: NFA -> DFA       DeLinearize: Regex -> Regex MergeBisim: NFA -> NFA
```

Многосортные функции

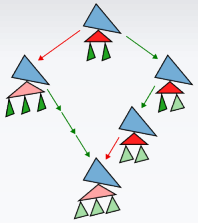
```
PumpLength: Regex -> Int   States: NFA -> Int          Width: NFA -> Int
ClassLength: DFA -> Int    ClassCard: DFA -> Int       MyhillNerode: DFA -> Int
KSubSet: (Int, NFA) -> NFA Ambiguity: NFA -> Value
Normalize: (Regex,FileName) -> Regex                   Simplify: Regex -> Regex
```

Предикаты

```
Bisimilar: (NFA,NFA) -> t/f  Equiv: (NFA,NFA) -> t/f   Equal: (NFA,NFA) -> t/f
Minimal: DFA -> t/f          Minimal: NFA -> t/f/u     SemDet: NFA -> t/f
Subset: (Regex,Regex) -> t/f Subset: (NFA, NFA) -> t/f Equiv: (Regex,Regex) ->
```

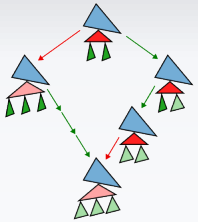
Специальные процедуры

```
Test: (NFA|Regex, Regex,Int) -> IO
```



Методы

- `RemEps` — удаление ε-правил; `MergeBisim` — объединение эквивалентных по бисимуляции состояний (необходимо проверить, чтобы их финальность совпадала).
- `Annote` — это навешивание разметки на все буквы в автомате, стоящие на недетерминированных переходах. Если ветвление содержит ε-переходы, то ε размечаются как буквы (т.е. считаются не пустыми до тех пор, пока разметка не будет снята). Если автомат — ДКА, метод его не меняет.
- `Linearize` размечает буквы в регулярном выражении как в алгоритме Глушкова.
- Метод `Arden` — трансформация НКА в регулярное выражение через решение регулярных уравнений — наследуется от лабораторных работ прошлого года. За встраивание этого метода в систему даётся 4 балла (только интегратору).

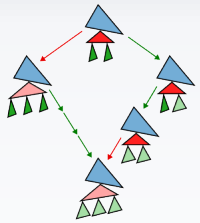


Частичные производные (Antimirov)

$\alpha_c(R)$ — это регулярное выражение R' такое, что если $w \in \mathscr{L}(R')$, то $c\,w \in \mathscr{L}(R)$. Обратное не обязательно выполняется. Вычислить частичные производные можно по следующему рекурсивному алгоритму.

$$
\begin{array}{lcl}
\alpha_c(c) & = & \{\varepsilon\} \\
\alpha_c(c') & = & \varnothing \\
\alpha_c(\varepsilon) & = & \varnothing \\
\alpha_c(r_1\,r_2) & = & \begin{cases} \{r\,r_2 \mid r \in \alpha_c(r_1)\} \cup \alpha_c(r_2) & \text{если } \varepsilon \in \mathscr{L}(r_1) \\ \{r\,r_2 \mid r \in \alpha_c(r_1)\} & \text{иначе} \end{cases} \\
\alpha_c(\bot) & = & \varnothing \\
\alpha_c(r_1|r_2) & = & \alpha_c(r_1) \cup \alpha_c(r_2) \\
\alpha_c(r*) & = & \{r'r* \mid r' \in \alpha_c(r)\}
\end{array}
$$

Автомат Антимирова аналогичен автомату Брзозовски, но состояния представляют собой множества α_w, а не δ_w. Упрощать по ACI состояния не требуется — их множество и так конечно.

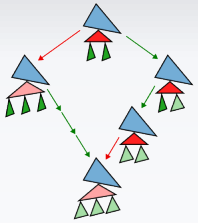


Определение длины накачки (PumpLength)

Дано регулярное выражение R. По возрастанию значения $\mathfrak{n}$:

1. Рассмотреть в R все возможные префиксы w длины $\mathfrak{n}$ и по каждому из них построить производную Брзозовски.
2. Перебрать инфиксы в пределах $\mathfrak{n}$-префиксов на предмет возможности накачки: а именно, если префикс w в выражении $w\,\delta_w(R)$ допускает разбиение на $w_1w_2w_3$, то он накачивается $\Leftrightarrow \mathscr{L}(w_1(w_2)^*w_3\delta_w(R)) \subseteq \mathscr{L}(R)$.
3. Перебор оптимизировать по префиксам: если уже известно, что префикс w накачивается, то рассматривать далее только более длинные префиксы, не начинающиеся с w.

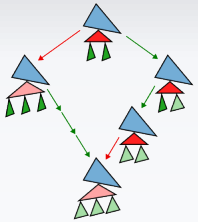
Результирующее $\mathfrak{n}$ и есть искомая длина накачки.



Follow-автомат (IlieYu)

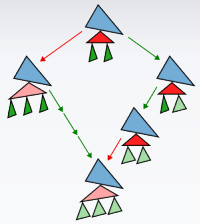
Пусть R — регулярное выражение. Положим $\mathtt{follow}(a_i) = \{a_j \mid \exists w, u (w a_i a_j u \in \mathscr{L}(R))\}$.
Follow-эквивалентность: состояния автомата Глушкова a_i и a_j follow-эквивалентны, если $\mathtt{follow}(a_i) = \mathtt{follow}(a_j)$, и либо a_i, a_j оба финальные, либо они оба не финальные.
Для построения автомата Илия-Ю (или follow-автомата) достаточно:

- Построить автомат Глушкова (Glushkov);
- Объединить follow-эквивалентные состояния.



Мера неоднозначности (Ambiguity)

- Для всех слов w из $\mathscr{L}(\mathscr{A})$ длины от минимальной до $|s|^2$, где s — число состояний НКА $\mathscr{A}$, построить функцию, считающую число путей в $\mathscr{A}$, по которым распознаётся слово w. Делать это лучше, разбивая слово w на меньшие подслова и считая число возможных переходов для них, после чего их комбинации.
- Если число путей растёт быстрее, чем полином степени $|s|$, то объявить автомат экспоненциально неоднозначным. Если число путей растёт медленнее, чем линейная функция, то объявить автомат почти однозначным (либо однозначным, если путь всегда один). Иначе автомат полиномиально неоднозначен.

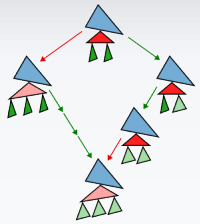


Семантический детерминизм (SemDet)

Язык состояния q — это $\{w \mid q \xrightarrow{w} q_f\}$, где q_f — какое-нибудь финальное состояние. Языки состояний удобно строить с помощью производных: в качестве аргумента производной достаточно взять произвольный префикс ν, соответствующий переходу $q_0 \xrightarrow{\nu} q$.

Скажем, что НКА $\mathscr{A}$ семантически детерминирован, если для всякой неоднозначности $q_i \xrightarrow{a} q_{j_1}, \ldots, q_i \xrightarrow{a} q_{j_k}$ существует такое состояние q_{j_s}, что языки всех q_{j_t} $(1 \leqslant t \leqslant k)$ вкладываются в его язык (это означает, что переход $q_i \xrightarrow{a} q_{j_s}$ надёжен: если слово распознаётся автоматом, оно обязательно будет распознано после такого перехода).

Данное определение фактически определяет алгоритм, выясняющий, какие состояния НКА являются семантически не детерминированными.



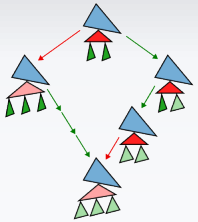
Ширина НКА (Width, а также KSubSet)

Определим конструкцию автомата k-подмножеств для НКА $\mathscr{A}$ следующим образом:

- Если существует не более k переходов по a из q_i в состояния $q_{s_1}, \ldots, q_{s_r}$, то объединяем состояния q_{s_i} в одно (и далее рассматриваем все переходы из всех этих состояний).
- В противном случае порождаем состояния, объединяющие все возможные комбинации из k состояний.

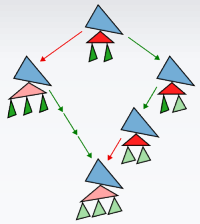
Скажем, что $\mathscr{A}$ имеет ширину k, если автомат k-подмножеств для $\mathscr{A}$ семантически детерминирован[1].

[1] Это понятие немного отличается от понятия ширины автомата из статей Куперберга, но для автоматов на конечных словах разница не существенная.



Парсинг слова по НКА и regex

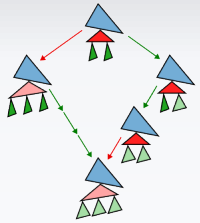
- Стандартный алгоритм парсинга по НКА: во всех позициях недетерминированности порождается точка возврата, к которой алгоритм возвращается при неудаче (в случае, если НКА — не автомат Томпсона, т.е. могут быть больше 2 недетерминированных переходов из одного узла, требуется сохранять также номер последнего посещённого перехода).
- Алгоритм Томпсона: при чтении очередного символа строки осуществляются переходы во все достижимые состояния, а пути, приведшие к неудаче, отсекаются. Моделирует переход к ДКА.
- Алгоритм Роба Пайка (для regex): читаем здесь его описание, сделанное Керниганом.



Метод Test

Аргументы: НКА или регулярное выражение; регулярное выражение без альтернатив (только с итерацией Клини) — тестовый сет; натуральное число — шаг итерации в сете.

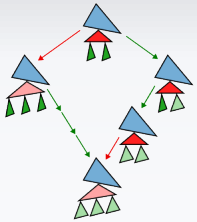
- В первом тесте все итерации раскрываются в ε. С очередным тестом все итерации в тестовой строке возрастают на величину шага. Например, вызов `test((a|ab)*,((ab*)a)*,3)` построит сначала пустое слово, а потом тест `abababaababababaababababa`. Для построенной строки замеряется время её разбора и заносится в таблицу.
- Завершается тестирование, когда сделано более 12 шагов, либо парсинг входной строки занимает больше 3 минут (таймаут).
- Для НКА строятся таблицы для двух алгоритмов: с возвратами и параллельного; для regex — одна таблица.



Тайпчекер

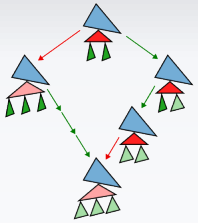
Проверка типов может выполняться в двух режимах: статически и динамически. Статически — до выполнения преобразований, и динамически — в процессе.

- Для каждой операции необходимо проверить корректность типов в композиции. В том числе убедиться, что алгоритму построения дополнения всегда подаётся на вход ДКА (если эта проверка делается статически, то полагаться можно только на то, что этот ДКА является результатом преобразования, заведомо возвращающего ДКА).
- При статическом анализе, если происходит поглощение операций в композиции (например, `Determinize.Minimize` — очевидно, минимальный автомат всегда будет детерминированным), то удалить лишнее действие ещё до его исполнения и сообщить об этом.
- При динамическом анализе сообщать о лишних операциях постфактум: если операция не поменяла входные данные, вывести сообщение об этом.



Тестирование

1. Написать генератор случайных регулярных выражений, параметризованный их длиной, совокупным числом итераций Клини и звёздной высотой (максимальной вложенностью итераций).
2. Написать генератор случайных заданий для фреймворка, в том числе некорректных и содержащих бессмысленные действия (поглощаемые).
3. Написать экспериментальный верификатор гипотез: по набору действий над единственным регулярным выражением, содержащих единственный предикат, строится сет тестов (параметризованный размером), в которых это выражение меняется на случайное. После чего в результатах тестирования выделяется доля тестов с положительным значением предиката, и кейсы, когда гипотеза не выполнилась (если таких кейсов больше 10% от общего числа, то достаточно вывести 10%).

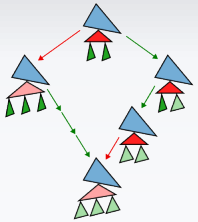


Реализация предикатов

- НКА $\mathscr{A}_1$ и $\mathscr{A}_2$ бисимилярны, если их стартовые состояния находятся в отношении бисимуляции, и каждое состояние $\mathscr{A}_i$ состоит в бисимуляции с некоторым состоянием $\mathscr{A}_j$, причём финальность бисимилярных состояний совпадает. Удобно делать через представление НКА грамматикой и л.р.1.
- Эквивалентность двух regex или НКА можно проверить композицией:

```
\\   Случай регулярок рассматривается аналогично,
\\   но сначала переводим в НКА
E1 = Minimize.Determinize A1
E2 = Minimize.Determinize A2
Equal A1 A2 \\  это и будет значение Equivalent A1 A2
```

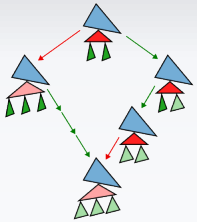
Без минимизации придётся строить дополнения и пересечения языков, что не всегда удобно. Минимальный автомат для regex — важный индикатор свойств языка, поэтому имеет смысл не строить эту конструкцию каждый раз заново, если она была построена хотя бы однажды, а брать, например, из таблицы.



Предикаты: продолжение

- Равенство (буквальное) двух НКА проверяется с точностью до переименовки состояний, поэтому нужно разработать способ стандартно именовать их состояния (проблемы могут возникнуть с состояниями, имеющие недетерминированные переходы). Кроме имён, для состояний нужны идентификаторы (например, соответствующие производным Антимирова либо размеченным буквам Глушкова)[2]. Второй способ проверки равенства НКА — через отношение биективной бисимуляции грамматик — более медленный.
- Минимальность ДКА можно проверять, просто сверяя число состояний с числом состояний в минимальном ДКА для языка. `IsMinimal A` возвращает `f`, если число состояний `A` больше числа символов в исходной регулярке (см. свойство автомата Глушкова), либо если число состояний `A` больше, чем минимальное число состояний для ранее построенного НКА по той же регулярке. Если число состояний в `A`, из которых есть хотя бы один переход, равно длине накачки, то `IsMinimal A = t` признаётся минимальным. В противном случае `IsMinimal A = u`.

[2]Красиво (но не обязательно, и это задача интегратора), если метками состояний в минимальном ДКА выступят к.э. Майхилла–Нероуда.



Баллы

- Метод определения ширины автомата и автомата k-подмножеств опциональный: за него добавляется +2 балла.
- В варианте lua ещё не было бисимуляции — за её реализацию будет добавлено +3 балла.