

Сопоставление с образцом

Летняя практика, Переславль–Залесский
3–7 июля 2023 г.



Алгоритмы Маркова

Синтаксис

Два вида правил:

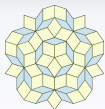
Φ_i	\rightarrow	Ψ_i	(нефинальные)
Φ'_i	\rightarrow_*	Ψ'_i	(финальные)

Семантика

- Данные — единственная строка.
- Правила просматриваются сверху вниз по списку.
- Выбирается самое левое вхождение подстроки Φ_i , после чего в случае финального правила оно просто заменяется на Ψ_i , а в случае нефинального — заменяется на Ψ_i , и операция повторяется над изменённой строкой.

По сути — рекурсивный вызов функции, разбивающей строку по шаблону $(.*)\Phi_i(.*)$. С именованными группами — как-то так:

$(? <x1> .*)\Phi_i(? <x2> .*) \rightarrow \text{group}(x1) ++ \Psi_i ++ \text{group}(x2)$



Алгоритмы Маркова как рекурсивная функция

- С учётом того, что группы всегда соответствуют выражениям $.^*$ (на самом деле — $.^*?$), заменим их просто именами.
- Конкатенацию сделаем неявной не только в регулярках, но и в результате.
- Результат: рекурсивная функция следующего вида.

$$\begin{aligned}
 f(x_1 \Phi_1 x_2) &= f(x_1 \Psi_1 x_2) \\
 &\dots \\
 f(x_1 \Phi'_1 x_2) &= x_1 \Psi'_1 x_2 \\
 f(x_1 \Phi_k x_2) &= f(x_1 \Psi_k x_2) \\
 &\dots
 \end{aligned}$$

Все левые части содержат один и тот же вызов, так что упоминание об f там избыточно. Получился примитивный ЯП. 3 / 20



Над тезисом Чёрча–Тьюринга

От нормальных алгорифмов к машинам Тьюринга

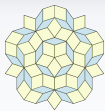
- Введём символ «головка машины» τ и символы состояний ξ_j . Потребуем Φ_i и Ψ_i всегда начинаться с символа τ , за которым следует ξ_j и единственная буква строки.
- Правила становятся коммутирующими; ленивое сопоставление также больше не актуально.

Обратно

- Воспользуемся расширенным тезисом Чёрча–Тьюринга...

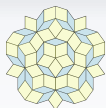
Наблюдение

Переход от алгорифмов к другим моделям вычислений очень прост — почти все преобразования такого типа сводимы к применению системы переписывания термов, которая выражается ~~нормальным алгорифмом~~ Рефал-программой.



Немного терминологии из комбинаторики слов

- Строка w в алфавите Σ ($w \in \Sigma^*$) — слово. ε обозначает пустое слово.
- Подслово v слова $w_1 v w_2$ — делитель.
- Если $w = \underbrace{v v \dots v}_n$ — тогда говорим, что w — n -ая степень v . Пишем $w = v^n$. Например, квадрат — это слово, представимое в виде vv (в стандартном допущении о том, что $v \neq \varepsilon$).
- Простое слово — не являющееся степенью никакого слова, кроме себя самого.
- Длина слова w обозначается $|w|$. Количество вхождений в w буквы t — как $|w|_t$.



Декларативные объявления в HASKELL

`f [] = ...`

`f [x] = ...`

`f (x : _) = ...`

- Частичный разбор терма начиная с внешнего конструктора.
- Сопоставление сверху вниз (аналогично цепочке `if-elif-else`).
- Не перестановочные предложения: третье в том числе описывает и случай, когда список одноэлементный (т.к. алгебраически список определяется конструкторами `:` (т.е. `cons`) и `[]`, т.е. `nil`).



Стандартные ограничения

- Отсутствие повторных переменных в образцах (линейность):
 - `f x x` — нельзя.
 - В языке PROLOG — можно.
- Однозначный разбор по внешнему конструктору (так называемое сопоставление с образцом в свободной алгебре):
 - `f x ++ ['A'] ++ y` — нельзя. Здесь `++` — операция конкатенации списков / строк.
 - `views Wadler'a` — в дальнейшем язык EGISON — можно.



Нелинейность и несвобода

- Проблемы с перестановочностью (даже неявной):

$f\ x\ y\ x = \dots$

$f\ x\ x\ y = \dots$

$f\ y\ x\ x = \dots$

- Проблемы с однозначностью ответа:

$f\ x\ ++\ ['A']\ ++\ y = \dots$

\dots

$f\ ['A', 'A', 'A', \dots, 'A']$

(в языке PROLOG возвращаются все результаты)

- Неочевидный алгоритм разбора по образцу:

$f\ y_0\ ++\ ('A' : x)\ ++\ y_1\ ++\ ('A' : x)\ ++\ y_2$

Имея изначальным аргументом f единственный список (строку литералов), мы вынуждены искать представление этого списка как конкатенации пяти списков.



А ещё это сложно...

- P (полиномиальная) сложность (по ресурсу τ) — существует детерминированный алгоритм, вычисляющий $f(x)$, где $|x| = n$, требуя $O(n^k)$ ресурса τ .
- NP (недетерминированная полиномиальная сложность) — существует недетерминированный алгоритм, вычисляющий $f(x)$, где $|x| = n$, требуя $O(n^k)$ ресурса τ .

Практическая разница колоссальная: алгоритм Ежа нахождения минимального решения уравнения в словах — предположительно в классе NP , но до сих пор не известно алгоритма, который бы решал эту задачу детерминированно лучше, чем за $\Theta(2^{2^n})$.



А ещё это сложно...

3SAT — проблема выполнимости булевых формул в КНФ с тремя литералами в дизъюнктах. Первая известная NP-полная задача (теорема Кука–Левина).

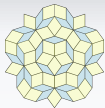
литералы

$$\underbrace{(\bar{L}_{1,1} \vee \bar{L}_{1,2} \vee \bar{L}_{1,3}) \& \dots (L_{n,1} \vee L_{n,2} \vee L_{n,3})}_{\text{Дизъюнкт}}$$

- Положительный литерал: $V_{i,j}$, где $V_{i,j}$ — переменная.
- Отрицательный литерал: $\neg V_{i,j}$, где $V_{i,j}$ — переменная.

SAT для произвольной КНФ сводится к 3SAT методом контролируемых ветвлений:

$$(L_1 \vee L_2 \vee L_3 \vee L_4) \Leftrightarrow (L_1 \vee L_2 \vee \underbrace{V_{L_3 \vee L_4}}_{\text{вхождения свежей переменной}}) \& (L_3 \vee L_4 \vee \underbrace{\neg V_{L_3 \vee L_4}}_{\text{вхождения свежей переменной}})$$



А ещё это сложно...

3SAT — проблема выполнимости булевых формул в КНФ с тремя литералами в дизъюнктах. Первая известная NP-полная задача (теорема Кука–Левина).

Кодировка состояния недетерминированной машины Тьюринга в виде булевых значений:

- $T_{i,j,k}$ — ячейка номер i содержит значение j на шаге k ;
- $H_{i,k,q}$ — головка стоит на ячейке i , в состоянии q , на шаге k .

Правила переписывания кодируются следованием, правила единственности — дизъюнкцией.



А ещё это сложно...

3SAT — проблема выполнимости булевых формул в КНФ с тремя литералами в дизъюнктах. Первая известная NP-полная задача (теорема Кука–Левина).

Эквивалентная проблема — 1-in-3-SAT (существует подстановка такая, что в каждом дизъюнкте ровно один литерал истинен).

$$\text{SAT}(L_1 \vee L_2 \vee L_3) \Leftrightarrow$$

$$\begin{aligned} & \text{1-in-3-SAT} \left((L_1 \vee V_{L_2 \& \neg L_1} \vee V_{\neg(L_1 \vee L_2)}) \& \right. \\ & \quad (L_2 \vee V_{L_1 \& \neg L_2} \vee V_{\neg(L_1 \vee L_2)}) \& \\ & \quad (V_{L_1 \& \neg L_2} \vee V_{L_2 \& \neg L_1} \vee V_{\text{aux1}}) \& \\ & \quad (L_3 \vee V_{\neg L_3} \vee 0) \& \\ & \quad \left. (V_{\neg L_3} \vee V_{\neg(L_1 \vee L_2)} \vee V_{\text{aux2}}) \right) \end{aligned}$$



А ещё это сложно...

Теорема Англуин

Задача сопоставления строки с образцом NP-полна в алфавите размера 2.

Идея сводящей к 1-in-3-SAT конструкции:

- нужно промоделировать дизъюнкты строками, литералы — их подстроками, при этом литералы должны быть перестановочными (коммутативность дизъюнкции)...
- если $uw = wu$, то что можно сказать про значения u и w ?
- нужно вынудить строки принимать только два различных значения, при этом если некоторая переменная принимает значение u , то «сопряжённая» переменная обязана принимать значение w .
- нужно вынудить дизъюнкт принимать значение «не меньше» одной единицы.



А ещё это сложно...

Теорема Англуин

Задача сопоставления строки с образцом NP-полна в алфавите размера 2.

- Каждой переменной V_i сопоставим переменную образца x_i и двойственную ей y_i .
- Каждому положительному литералу с переменной V_j сопоставим p_j , отрицательному — q_j .
- Дизъюнкты разделим константами **b**.
- Факт двойственности выразим условием: $x_i y_i$ сопоставляется с **a**. Тогда одна из них обязана принять значение ε (FALSE), а вторая — **a** (TRUE).
- Факт 1-in-3-SAT выразим условием: образ дизъюнкта $L_1 \vee L_2 \vee L_3$ сопоставляется с **a**. Тогда образ ровно одного литерала принимает значение **a**, остальные — ε .



А ещё это сложно...

Теорема Англуин

Задача сопоставления строки с образцом NP-полна в алфавите размера 2.

Искомый образец:

$$\mathbf{b}_{x_1 y_1} \mathbf{b}_{x_2 y_2} \dots \mathbf{b}_{x_k y_k}$$

$$\mathbf{b}(L_{1,1}\sigma)(L_{1,2}\sigma)(L_{1,3}\sigma)\mathbf{b}\dots\mathbf{b}(L_{n,1}\sigma)(L_{n,2}\sigma)(L_{n,3}\sigma)$$

Искомая строка: $(\mathbf{ba})^{k+n}$

Например, для 1-in-3-SAT КНФ

$$(\text{P} \vee \text{Q} \vee \text{R}) \ \& \ (\text{P} \vee \neg \text{Q} \vee \neg \text{R}) \ \& \ (\neg \text{P} \vee \text{Q} \vee \neg \text{R})$$

получается задача сопоставления образца:

b $x_P y_P$ **b** $x_Q y_Q$ **b** $x_R y_R$ **b** $x_P x_Q x_R$ **b** $x_P y_Q y_R$ **b** $y_P x_Q y_R$

со строкой **b a b a b a b a b a**.



Регулярные выражения

Проблема неоднозначности разбора строковых данных уже решалась в рамках построения механизма сопоставления с регулярными выражениями.

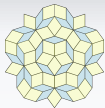
- Жадные квантификаторы — забирают максимально длинную подстроку

```
match = re.search(r'\(.*\)', r'0 ( (12)3)4(56 )7')  
# match[0] = ((12)3)4(56)
```

- Ленивые квантификаторы — забирают максимально короткую подстроку

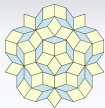
```
match = re.search(r'\(.*?\)', r'0 ( (12 )3)4(56)7')  
# match[0] = ((12)
```

Здесь `.` — произвольная буква, `*` — квантификатор: повторять образец 0 или больше раз, `\(` и `\)` — экранированные представления круглых скобок.



Повторные группы в регулярках

- Элемент стандарта PCRE2 (но не все ЯП следуют этому стандарту).
- Итерация + захват в память \Rightarrow невозможность использовать в правых частях правил какие-либо значения переменных, кроме инициализированных последними.
- На практике — больше 80% регулярок из REGEXLIB с повторными группами — образцы над регулярными языками.



Ассоциативные образцы

Для краткости ++ опускается. Например: $x'A'u$ — сокращённая запись для $x \text{ ++ } ['A'] \text{ ++ } u$.

- Сопоставление ленивое, сверху вниз.
- Эффективный доступ к строке с начала, с конца или с середины (например, при использовании суффиксного массива). Условный "шаг" — вызов сопоставления. С точки зрения реализации, "шаг" имеет сложность, большую, чем $O(1)$.
- Строки — очевидно. Как добавить леса с доступом "с середины"?

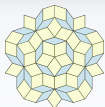


Ассоциативные образцы

- Сопоставление ленивое, сверху вниз.
- Эффективный доступ к строке с начала, с конца или с середины (например, при использовании суффиксного массива). Условный "шаг" — вызов сопоставления. С точки зрения реализации, "шаг" имеет сложность, большую, чем $O(1)$.
- Строки — очевидно. Как добавить леса с доступом "с середины"? Решение: дополнительный конструктор, добавляющий глубину вложенности.

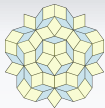
Дан список списков. Выделить список, содержащий букву 'А'.

```
(list x1 (list z1 'A' z2) x2) — с тегами (АТД);  
(x1 (z1 'A' z2) x2) — без тегов (ака РЕФАЛ);
```



Базисный Рефал: в образцах можно всё

- Свободные образцы: вместо примитивного образца — всё множество возможных образцов над свободным моноидом, в том числе с повторными переменными.
- «Много функций»: введение идентификатора функции, стоящего в образце самым первым и определяющего выбор подмножества правил переписывания.
- Вложенные структуры: использование лесов строк (дополнительного конструктора в моноиде) в качестве данных программы.



Примеры образцов

- Ленивый образец, находящий две одинаковые заковыченные последовательности?
- Образец, проверяющий, есть ли в слове квадрат (т.е. дважды повторяющееся подслово)?



Примеры образцов

- Ленивый образец, находящий две одинаковые заковыченные последовательности?

x1 " x2 " x3 " x2 " x4

Заметим, что если внутри значения x2 окажутся хотя бы одни кавычки, то существует более «ленивое» сопоставление (с той же длиной значения x1, но меньшей — x2).

- Образец, проверяющий, есть ли в слове квадрат (т.е. дважды повторяющееся подслово)?



Примеры образцов

- Ленивый образец, находящий две одинаковые заковыченные последовательности?

x1 " x2 " x3 " x2 " x4

- Образец, проверяющий, есть ли в слове квадрат (т.е. дважды повторяющееся подслово)?

Первая проба: x1 x2 x2 x3



Примеры образцов

- Ленивый образец, находящий две одинаковые заковыченные последовательности?

x1 " x2 " x3 " x2 " x4

- Образец, проверяющий, есть ли в слове квадрат (т.е. дважды повторяющееся подслово)?

Первая проба: x1 x2 x2 x3

Fail! Тривиально, из-за наличия в моноиде единицы — пустого слова. Требуется подчеркнуть непустоту — сказать, что в квадрат входит хотя бы один символ.

Разделим переменные образца на два класса:

- `e.name` — `expression`, сопоставляется с чем угодно;
- `t.name` — `term`, сопоставляется только с символом либо скобочным выражением.



Примеры образцов

- Ленивый образец, находящий две одинаковые заковыченные последовательности?

x1 " x2 " x3 " x2 " x4

- Образец, проверяющий, есть ли в слове квадрат (т.е. дважды повторяющееся подслово)?

Разделим переменные образца на два класса:

- `e.name` — `expression`, сопоставляется с чем угодно;
- `t.name` — `term`, сопоставляется только с символом либо скобочным выражением.

Тогда решение задачи выглядит так:

e.x1 t.y e.x2 t.y e.x2 e.x3

Далее сокращаем `e.name` просто до `name` (начинается не с `t`), а `t.name` — до `tname`.



Функции над образцами

- Функция, находящая в аргументе две одинаковые заковыченные последовательности, и возвращающая одну из них?
- Функция, находящая в слове квадрат (т.е. дважды повторяющееся подслово)?

Далее сокращаем `e.name` просто до `name` (начинается не с `t`), а `t.name` — до `tname`.



Функции над образцами

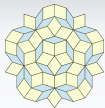
- Функция, находящая в аргументе две одинаковые заковыченные последовательности, и возвращающая одну из них?

```
F {x1 " x2 " x3 " x2 " x4 = x2;}
```

- Функция, находящая в слове квадрат (т.е. дважды повторяющееся подслово)?

```
F {e.x1 t.y e.x2 t.y e.x2 e.x3 = t.y e.x2;}
```

Далее сокращаем `e.name` просто до `name` (начинается не с `t`), а `t.name` — до `tname`.



Задача Корлюкова ++

Дано двоичное число длины 3. Записать функцию, возвращающую в двоичной форме (с лидирующими нулями) число единиц в нём.



Задача Корлюкова ++

Дано двоичное число длины 3. Записать функцию, возвращающую в двоичной форме (с лидирующими нулями) число единиц в нём.

Решение «в лоб»:

```
F { 0 0 0 = 0 0;  
    0 0 1 = 0 1;  
    0 1 0 = 0 1;  
    0 1 1 = 1 0;  
    1 0 0 = 0 1;  
    1 0 1 = 1 0;  
    1 1 0 = 1 0;  
    1 1 1 = 1 1; }
```



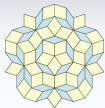
Задача Корлюкова ++

Дано двоичное число длины 3. Записать функцию, возвращающую в двоичной форме (с лидирующими нулями) число единиц в нём.

Решение «в лоб»:

$$\begin{aligned} F \{ & 0\ 0\ 0 = 0\ 0; \\ & 0\ 0\ 1 = 0\ 1; \\ & 0\ 1\ 0 = 0\ 1; \\ & 0\ 1\ 1 = 1\ 0; \\ & 1\ 0\ 0 = 0\ 1; \\ & 1\ 0\ 1 = 1\ 0; \\ & 1\ 1\ 0 = 1\ 0; \\ & 1\ 1\ 1 = 1\ 1; \} \end{aligned}$$

- Два нуля влекут результат 0 1;
- Две единицы влекут результат 1 0;
- Осталось разобраться с $t\ t\ t$. Но такие данные порождают всегда $t\ t$.



Задача Корлюкова ++

Дано двоичное число длины 3. Записать функцию, возвращающую в двоичной форме (с лидирующими нулями) число единиц в нём.

Второе приближение:

```
F { t t t = t t;  
    x1 0 x2 0 x3 = 0 1;  
    x1 1 x2 1 x3 = 1 0;  
}
```



Задача Корлюкова ++

Дано двоичное число длины 3. Записать функцию, возвращающую в двоичной форме (с лидирующими нулями) число единиц в нём.

Второе приближение:

```
F { t t t = t t;
```

```
    x1 0 x2 0 x3 = 0 1; — другое;
```

```
    x1 1 x2 1 x3 = 1 0;
```

```
}
```

- Видно, что вторая и третья строчки почти одинаковы — на первом месте в результате стоит выделенное значение, на втором — другое; Длина строки $x1\ x2\ x3$ всегда равна 1, и это именно то другое значение!



Задача Корлюкова ++

Дано двоичное число длины 3. Записать функцию, возвращающую в двоичной форме (с лидирующими нулями) число единиц в нём.

Почти решение:

```
F { t t t = t t;  
    x1 t x2 t x3  
    = t x1 x2 x3;  
}
```



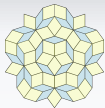
Задача Корлюкова ++

Дано двоичное число длины 3. Записать функцию, возвращающую в двоичной форме (с лидирующими нулями) число единиц в нём.

Почти решение:

```
F { t t t = t t;  
    x1 t x2 t x3  
    = t x1 x2 x3;  
}
```

- А теперь видно, что и первая строчка не нужна — в ней делается то же, что и во второй.
- Итоговая функция содержит всего одну строчку:
 $x_1 t x_2 t x_3 = t x_1 x_2 x_3$.
Причём x_i справа от знака $=$ можно располагать как угодно относительно друг друга, ответ всё равно будет правильным.



Задача Корлюкова ++

Чуть-чуть усложним задачу.

Дано двоичное число длины 4. Записать в двоичной форме (с лидирующими нулями) число единиц в нём.

Видно, что за исключением случая четырёх единиц, сгодилось бы предыдущее решение, если бы мы умели выкидывать из образца один ноль и собирать всё остальное в новый образец.



Задача Корлюкова ++

Чуть-чуть усложним задачу.

Дано двоичное число длины 4. Записать в двоичной форме (с лидирующими нулями) число единиц в нём.

Видно, что за исключением случая четырёх единиц, сгодилось бы предыдущее решение, если бы мы умели выкидывать из образца один ноль и собирать всё остальное в новый образец.

Как описать образцы для элементов образцов?

$[Образец](, [Выражение] : [Образец])^*$



Задача Корлюкова ++

Чуть-чуть усложним задачу.

Дано двоичное число длины 4. Записать в двоичной форме (с лидирующими нулями) число единиц в нём.

Видно, что за исключением случая четырёх единиц, сгодилось бы предыдущее решение, если бы мы умели выкидывать из образца один ноль и собирать всё остальное в новый образец.

Как описать образцы для элементов образцов?

$[Образец](, [Выражение] : [Образец])^*$

Решение задачи Корлюкова++ в этих терминах:

$F \{1 \ 1 \ 1 \ 1 = 1 \ 0 \ 0;$

$x1 \ 0 \ x2$

$, x1 \ x2 : z1 \ t \ z2 \ t \ z3 = 0 \ t \ z1 \ z2 \ z3; \}$



Другие образцы

- Слово содержит как букву 'A', так и букву 'B'.
- Два слова являются циклическими перестановками (т.е. $w_1 = uv$, $w_2 = vu$).
- Слово не содержит ничего, кроме (произвольного числа) букв 'A'.
- Два слова являются степенями одного и того же слова ($w_1 = v^i$, $w_2 = v^j$).



Другие образцы

- Слово содержит как букву 'A', так и букву 'B'.
Ответ: образец `x1 'A' x2, x1 x2: z1 'B' z2`
(поскольку 'A' точно не содержит ничего от 'B')
- Два слова являются циклическими перестановками (т.е. $w_1 = uv$, $w_2 = vu$).
- Слово не содержит ничего, кроме (произвольного числа) букв 'A'.
- Два слова являются степенями одного и того же слова ($w_1 = v^i$, $w_2 = v^j$).



Другие образцы

- Слово содержит как букву 'A', так и букву 'B'.

Ответ: образец $x_1 \text{ 'A' } x_2, x_1 x_2: z_1 \text{ 'B' } z_2$
(поскольку 'A' точно не содержит ничего от 'B')

- Два слова являются циклическими перестановками (т.е. $w_1 = uv, w_2 = vu$).

Тут решение тривиальное: $(x_1 x_2) (x_2 x_1)$.

- Слово не содержит ничего, кроме (произвольного числа) букв 'A'.
- Два слова являются степенями одного и того же слова ($w_1 = v^i, w_2 = v^j$).



Другие образцы

- Слово содержит как букву 'A', так и букву 'B'.

Ответ: образец $x_1 \text{ 'A' } x_2, x_1 \text{ } x_2 : z_1 \text{ 'B' } z_2$
(поскольку 'A' точно не содержит ничего от 'B')

- Два слова являются циклическими перестановками (т.е. $w_1 = uv, w_2 = vu$).

Тут решение тривиальное: $(x_1 \text{ } x_2) (x_2 \text{ } x_1)$.

- Слово не содержит ничего, кроме (произвольного числа) букв 'A'.

Решение нетривиальное: $x, \text{ 'A' } x : x \text{ 'A' }$.

- Два слова являются степенями одного и того же слова ($w_1 = v^i, w_2 = v^j$).



Другие образцы

- Слово содержит как букву 'А', так и букву 'В'.

Ответ: образец $x_1 \text{ 'A' } x_2, x_1 x_2 : z_1 \text{ 'B' } z_2$
(поскольку 'А' точно не содержит ничего от 'В')

- Два слова являются циклическими перестановками (т.е. $w_1 = uv, w_2 = vu$).

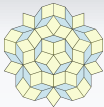
Тут решение тривиальное: $(x_1 x_2) (x_2 x_1)$.

- Слово не содержит ничего, кроме (произвольного числа) букв 'А'.

Решение нетривиальное: $x, \text{ 'A' } x : x \text{ 'A' }$.

- Два слова являются степенями одного и того же слова ($w_1 = v^i, w_2 = v^j$).

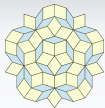
По аналогии с предыдущим: $(x_1) (x_2), x_1 x_2 : x_2 x_1$.



Сложение n-ичных чисел

Данные: списки из ограниченных контейнеров (чтобы использовать коммутативность).

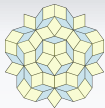
```
SumBase {  
    (e.Base)(e.N1 (e.D1))(e.N2 (e.D2))  
    , e.D1 e.D2 : e.Base e.Remainder = ... ;  
        /* Как дополнительно проверить,  
           что Remainder < Base? */  
    (e.Base)(e.N1 (e.D1))(e.N2 (e.D2)) = ... ;  
    (e.Base) e.MaybeN1 () e.MaybeN2 = ... ;  
        /* Почему переменные называются так? */  
}
```



Ещё несколько задач

Какие языки описываются следующими образцами?

- $x_1, x_1 A B : B A x_1, x_1 : x_2 x_2$
- $x_1, x_1 x_1: t_1 x_2 x_1 x_3, x_1: t_1 x_2 t_2 x_4$



Ещё несколько задач

Какие языки описываются следующими образцами?

- $x1, x1 A B : B A x1, x1 : x2 x2$

Пустой язык. Поскольку уравнение (а это именно уравнение) $x1 A B = B A x1$ имеет решения вида $B (A B)^*$, а они все — нечётной длины.

- $x1, x1 x1: t1 x2 x1 x3, x1: t1 x2 t2 x4$



Ещё несколько задач

Какие языки описываются следующими образцами?

- $x_1, x_1 A B : B A x_1, x_1 : x_2 x_2$

Пустой язык. Поскольку уравнение (а это именно уравнение) $x_1 A B = B A x_1$ имеет решения вида $B (A B)^*$, а они все — нечётной длины.

- $x_1, x_1 x_1 : t_1 x_2 x_1 x_3, x_1 : t_1 x_2 t_2 x_4$

Язык слов вида w^s , где $s \geq 2$. Почему они подходят, понять легко, а вот почему не подходят другие слова, проще понять, изучив основы комбинаторики слов.



Библиотека pygments

Возможность генерировать разметку синтаксиса в \LaTeX или в любых других пакетах, поддерживающих pygmentize.

```
MAIN ::= tokens = {STATE : [ <DEF>+ ]  
                  (, [STATE : [ <DEF>+ ])*  
                  }
```

```
DEF ::= (REGEX, TOKEN_NAME(, STATE_ACTION?))
```

```
STATE_ACTION ::= '#push' | '#pop' | STATE
```

```
REGEX ::= r("|')PYTHON_REGEX\1
```

Можно определять сразу много групп токенов вместо TOKEN_NAME, посредством вызова bygroups.

Определение переменных Рефала:

```
(r'([ets]\.)([a-zA-Z0-9_-]+)',  
  bygroups(Keyword.Reserved, Name.Variable))
```



Двумерный случай

- Определим отношение соседства как отношение примыкания в матрице.
- С помощью образцов зададим правила переписывания.

Применение двумерных систем переписывания по образцу:

- Эволюционные алгоритмы (например, игра «жизнь» Конвея) (классические клеточные автоматы).
- Генерация случайных объектов (стохастические клеточные автоматы).