Регулярные выражения, парсинг

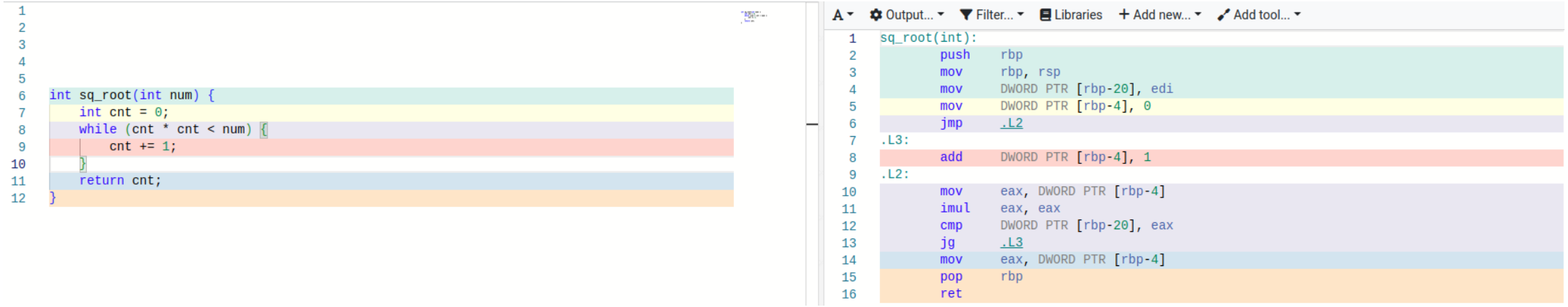
08/04/2023

Проблема

Данные существуют, чтобы ими могли пользоваться люди.

Если у нас есть документ в каком-то человекочитаемом формате, но надо его превращать в данные для компьютера.

Компиляция: C++



```
int sq_root(int num) {
    int cnt = 0;
    while (cnt * cnt < num) {
        cnt += 1;
    }
    return cnt;
}
```

```
sq_root(int):
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-20], edi
        mov     DWORD PTR [rbp-4], 0
        jmp     .L2
.L3:
        add     DWORD PTR [rbp-4], 1
.L2:
        mov     eax, DWORD PTR [rbp-4]
        imul    eax, eax
        cmp     DWORD PTR [rbp-20], eax
        jg      .L3
        mov     eax, DWORD PTR [rbp-4]
        pop     rbp
        ret
```

Интерпретация: python

```
2
3
4
5
6
7  def sq_root(num):
8      cnt = 0
9      while cnt * cnt < num:
10         cnt += 1
11     return cnt
```

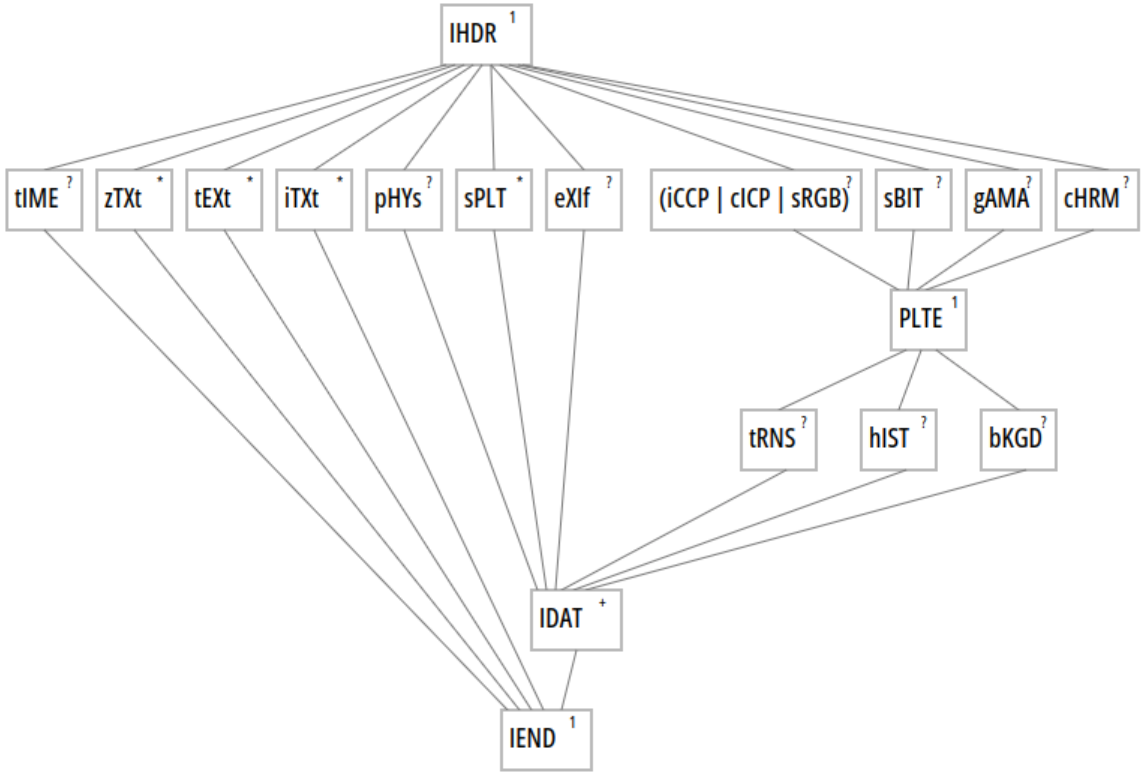
A ▾ | Output... ▾ | Filter... ▾ | Libraries | + Add new... ▾ | Add tool... ▾

```
1    0          0 RESUME                   0
2
3    7          2 LOAD_CONST               0 (<code object sq_root at 0x5564496d3b70, file "e
4               4 MAKE_FUNCTION            0
5               6 STORE_NAME               0 (sq_root)
6               8 LOAD_CONST               1 (None)
7              10 RETURN_VALUE
8
9   Disassembly of <code object sq_root at 0x5564496d3b70, file "example.py", line 7>:
10   7          0 RESUME                   0
11
12   8          2 LOAD_CONST               1 (0)
13              4 STORE_FAST               1 (cnt)
14
15   9          6 LOAD_FAST                1 (cnt)
16              8 LOAD_FAST                1 (cnt)
17             10 BINARY_OP                5 (*)
18             14 LOAD_FAST                0 (num)
19             16 COMPARE_OP               0 (<)
20             22 POP_JUMP_FORWARD_IF_FALSE    14 (to 52)
21
22  10     >>  24 LOAD_FAST                1 (cnt)
23             26 LOAD_CONST               2 (1)
24             28 BINARY_OP               13 (+=)
25             32 STORE_FAST               1 (cnt)
26
27   9         34 LOAD_FAST                1 (cnt)
28             36 LOAD_FAST                1 (cnt)
29             38 BINARY_OP                5 (*)
30             42 LOAD_FAST                0 (num)
31             44 COMPARE_OP               0 (<)
32             50 POP_JUMP_BACKWARD_IF_TRUE    14 (to 24)
33
34  11     >>  52 LOAD_FAST                1 (cnt)
35             54 RETURN_VALUE
```

Бинарные форматы: PNG

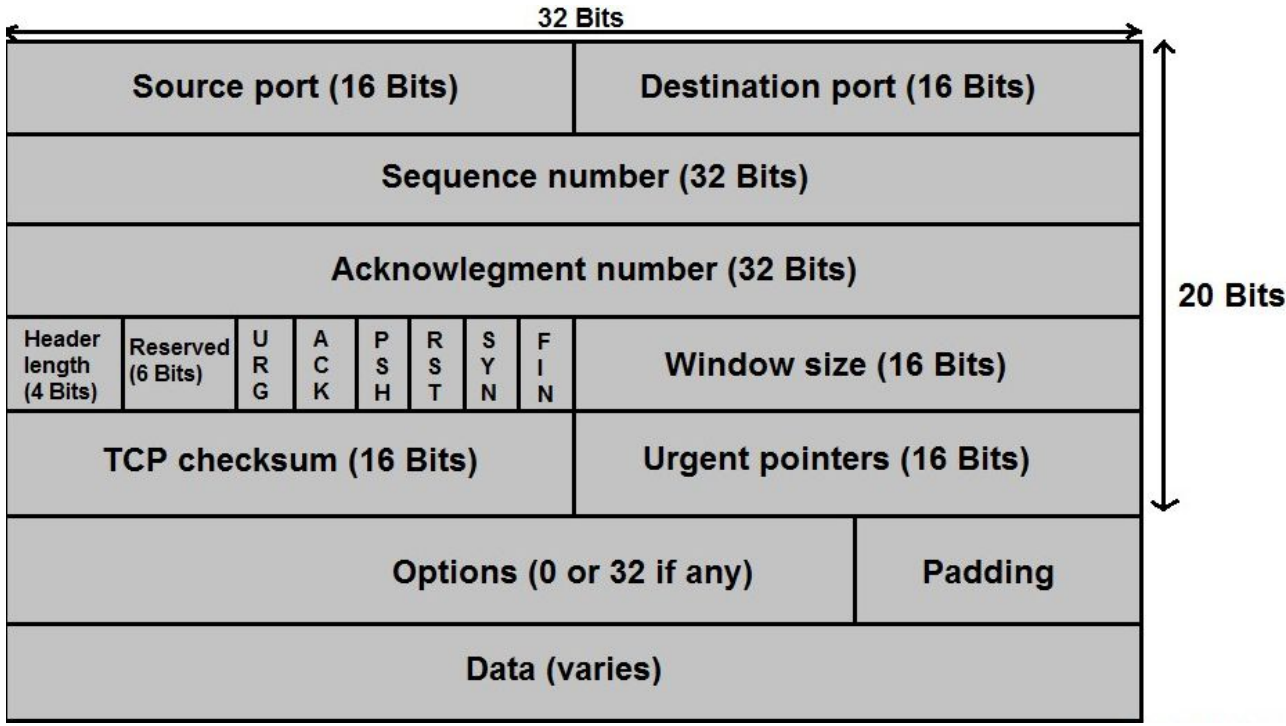
Для файлов формат обычно специфицирован так, что обработка файла может идти последовательно. Например, в png идет последовательная обработка chunk-ов.

Человек не поймет содержимое в png-файле по байтам, нужно, чтобы специальная программа превратила файл из данного формата в картинку.



Бинарные форматы: TCP

Используется при отправке данных по сети. По байт-представлению Data можно что-то понять про сообщение.



Парсинг бинарных данных

- Структура данных явно определена стандартом и метаданными
- Можно пройтись по стандарту и сделать так, как в нем написано
- Обычно уже реализовано в какой-нибудь библиотеке

Текстовые форматы: JSON

В отличие от бинарных форматов, текстовые форматы можно читать и делать выводы о содержимом на основе текста.

```
{
        "id": "00000234567894",
        "name": "Jane Doe",
        "birthday": "04/18/1978",
        "gender": "female",
        "type": "user",
        "work": [{
           "employer": {
              "id": "106119876543210",
              "name": "Doe Inc."
           },
           "start_date": "2007 - 08"
         },
        {
          "start_date": "2004",
          "end_date": "2007"
        }]
}
```

Текстовые форматы: C++

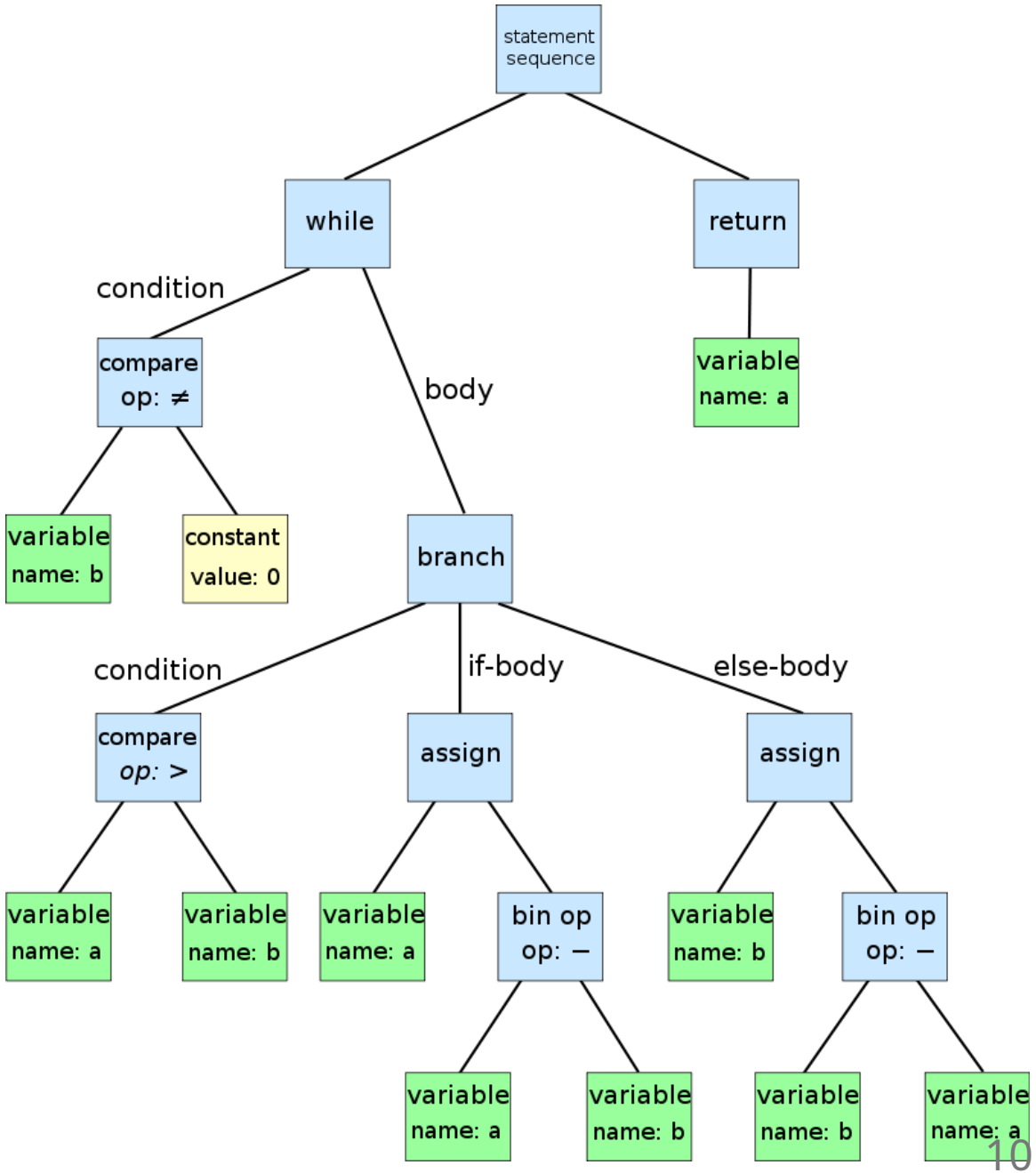
Программа на C++ является текстовым файлом, а компилятор превращает этот текстовый файл в исполняемый, соответствуя правилам стандарта.

Если файл не соответствует стандарту, компилятор может сделать какие-то оптимизации и вызывать undefined behaviour.

Парсинг текстового формата

- Обычно в текстовом формате более случайная структура
- Структура файла определяется специальными разделителями

```
while b != 0:
    if a > b:
        a = a - b
    else:
        b = b - a
return a
```



Рекурсивный парсинг выражения с бинарными операциями

- Формально определяем язык
- Определяем базовую операцию (например, число)
- Выделяем бинарные операции, опираясь на символ
- Рекурсивно парсим части выражения

$$digit := 0|1|2|3|4|5|6|7|8|9$$

$$number := \begin{cases} digit \\ number\ digit \end{cases}$$

$$exp := \begin{cases} number \\ (exp\{+|-|*|/\}\{exp\}) \end{cases}$$

Примеры

- 123
- $123 + 567$
- $(123 + 567) \cdot (123 + 567)$
- $(123 + (567 + 890))$

Вопрос: Что из этих выражений некорректно с точки зрения формального языка?

Реализация

```
s, ptr = '((123+567)*(123+567))', 0
def parse_digit():
    global ptr
    if s[ptr].isdigit():
        c = ord(s[ptr]) - ord('0')
        ptr += 1
        return c
    return None

def parse_number():
    num = 0
    d = parse_digit()
    while d is not None:
        num = num * 10 + d
        d = parse_digit()
    return num

funcs = {
    '+': lambda a, b: a + b, '-' : lambda a, b: a - b,    '*': lambda a, b: a * b,    '/': lambda a, b: a / b,
}

def parse_expression():
    global ptr
    if s[ptr] == '(':
        ptr += 1 # (
        exp1 = parse_expression()
        op = s[ptr]
        ptr += 1
        exp2 = parse_expression()
        ptr += 1 # )
        return funcs[op](exp1, exp2)
    return parse_number()

parse_expression() # 476100
```

"Парсим" кусочек python

$$code := \begin{cases} loop \\ condition \\ operation \\ function \end{cases}$$

$$function := def\ name(args) : code$$

$$name := \{a|b|c|\dots\}^*$$

$$args := \begin{cases} \emptyset \\ name \\ args, name \end{cases}$$

Recap

- Иногда мы можем просто читать данные, зная нужную нам структуру данных заранее
- Иногда мы можем просто читать данные, потому что у них простая структура
- Иногда мы хотим формально определить, как выглядят корректные данные, и как их правильно парсить
- При рекурсивном переборе мы благдаря стеку рекурсии понимаем структуру данных

Регулярные выражения

Проблема: хочется задавать формальные языки в программе.

Примитивное решение: Сделать рекурсивный перебор с прошлых слайдов.

Хочется более общее решение. Можно заметить, что у нас естественным образом возникает желание обобщать похожие конструкции (например, через * обозначать повторения).

Regexp

Везде ниже `s` - корректное регулярное выражение. Regexp позволяет выделить

- `.` - любой символ
- `[abc]` - один из символов
- `[a-z]` - один из символов в промежутке
- `ss'` - `s'` идет сразу после `s`
- `s*` - повторение `s` сколько-то раз (возможно, ноль)
- `s+` - повторение `s` сколько-то раз (1 и больше)
- `s|s'` - либо `s'`, либо `s`
- `s?` - либо `s`, либо пустая строка
- `s{m;n}` - `s` от `m` до `n` раз подряд
- `(s)` - группировка `s` в одно регулярное выражение.

Пример

```
>>> import re

>>> re.findall(r'[0-9]+', 'abc00123xyz456_0')
['00123', '456', '0']

>>> re.sub(r'[0-9]+', r'*', 'abc00123xyz456_0')
'abc*xyz*_*'
```

Группы символов

- `\d` - цифра
- `\w` - буква (строчная/заглавная) или `\d`
- `\s` - пробельные символы

Если заменить на заглавную букву, то будет "все, кроме". Например, `\D` - все, кроме цифр.

Advanced: группировка с заменой

Скобки позволяют выделить под-выражение, которое можно в будущем использовать для замен.

```
>>> re.findall(r'^(\S+)\s+(\S+)$', 'apple orange')
[('apple', 'orange')]

>>> re.findall(r'[^A-Z]+([A-Z]+)[^A-Z]+\1', 'i have a PEN, i have an APPLE')
[]

>>> re.findall(r'[^A-Z]+([A-Z]+)[^A-Z]+\1', 'i have an APPLE, i have an APPLE')
['APPLE']

>>> re.sub(r'^(\S+)\s+(\S+)$', r'\2 \1', 'apple orange')
'orange apple'
```

Как сделать regexp?

Через автоматы!

Конечные автоматы

Конечным автоматом называется ориентированный граф, где из каждого ребра есть значение из множества Σ, которое называется алфавитом. Такое ребро называется переходом. Для каждой вершины существует переход по каждому символу алфавита.

Иначе говоря, мы из состояния 1 по букве `c` можем перейти в состояние 2 по соответствующему ребру.

То же самое формально

V множество вершин, Σ алфавит

Существует отображение $V \times \Sigma \to V$ (для детерминированного случая)

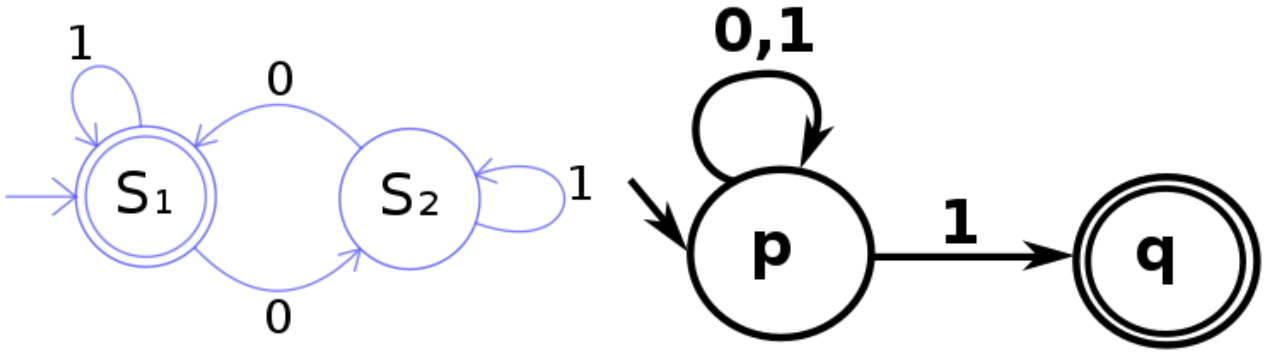
Существует стартовое состояние v_0 и терминальные вершины $v_t \in F$

Автомат "принимает" строку, если ей можно пройти по переходам от старта до терминала.

ДКА/НДКА

В детерминированных автоматах по каждому символу только один переход. В недетерминированных их бывает больше.

Вопрос: Что делают эти автоматы и какой из них детерминированный?

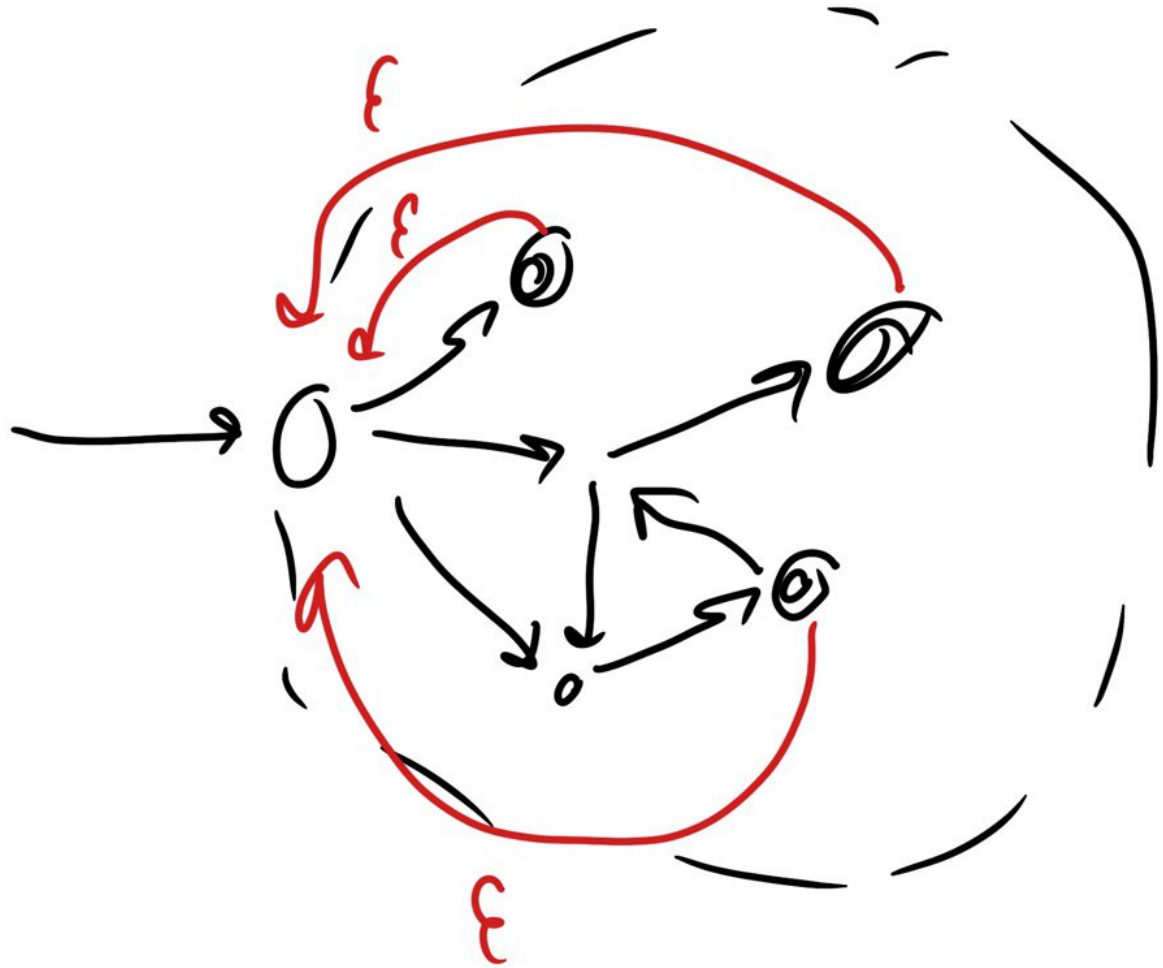


Автомат для regexp'ов

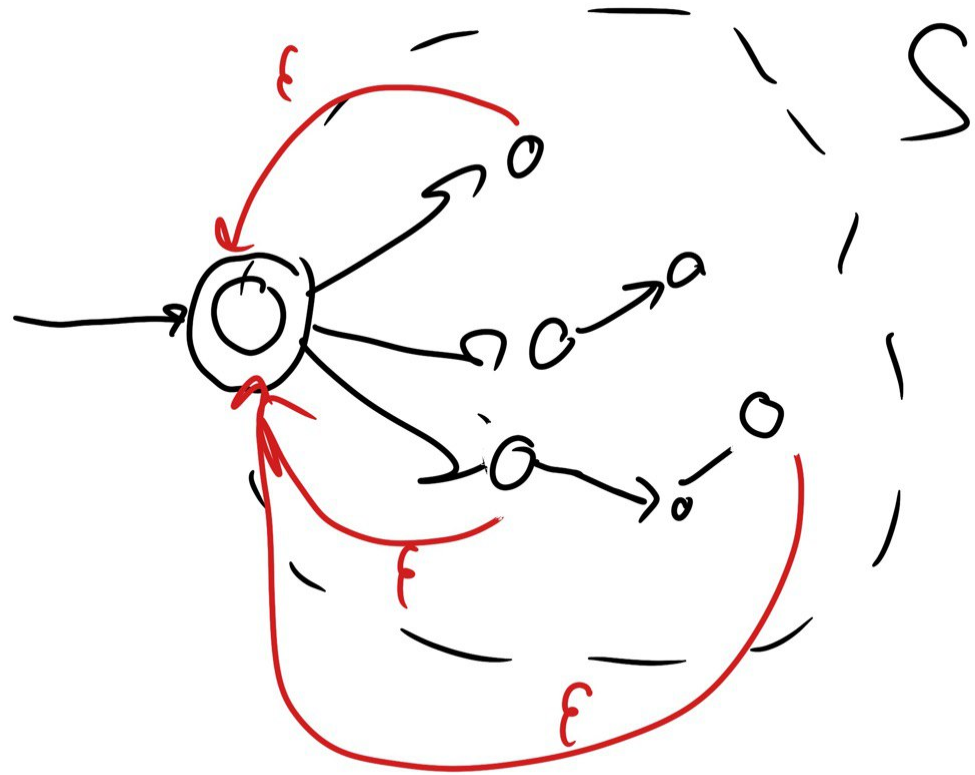
Для каждой операции можно придумать автомат, ей соответствующий, и комбинировать такие автоматы.

Нам могут понадобиться ϵ-переходы, которые соответствуют переходу без символов. В таких случаях regexp соответствует НДКА.

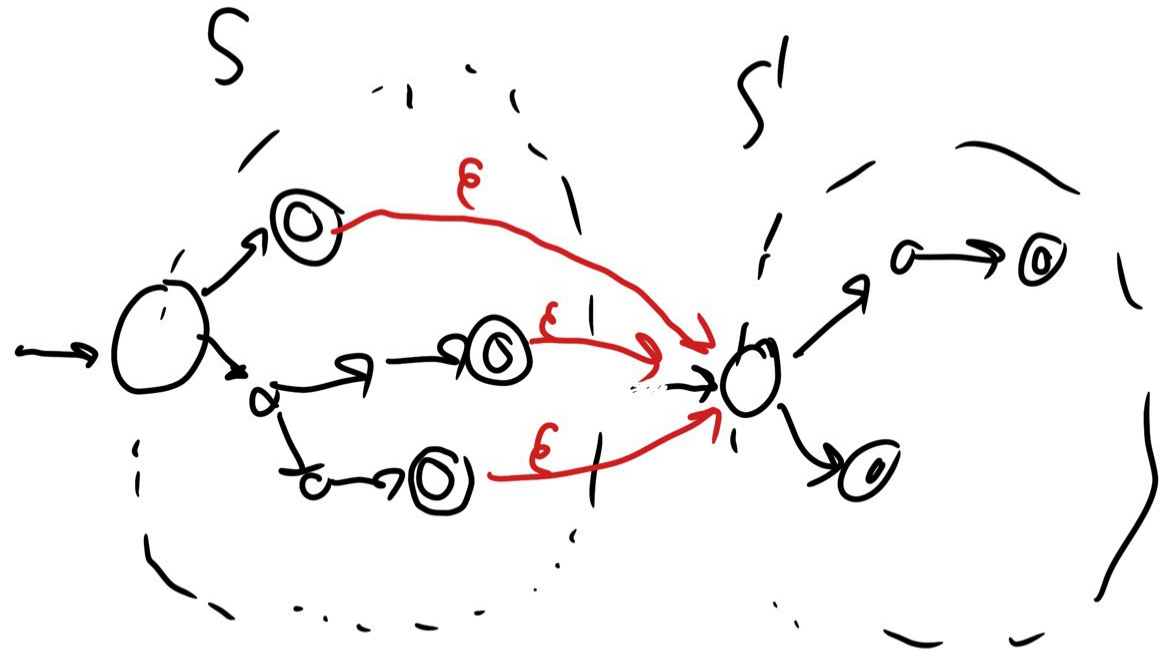
s+



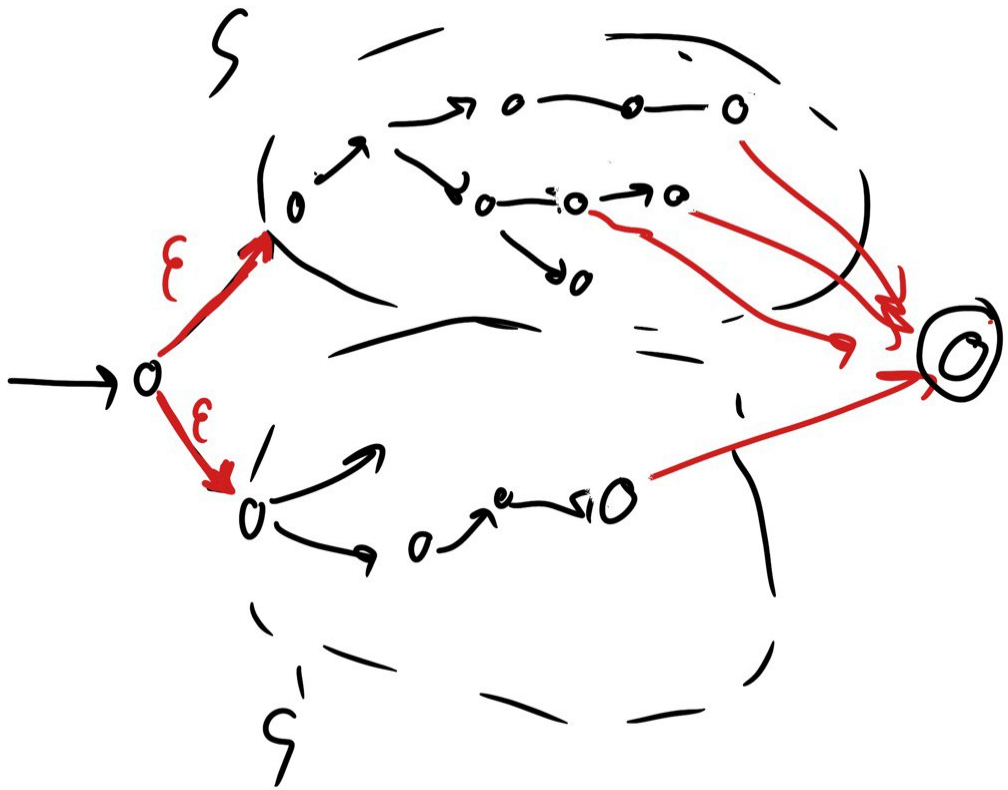
s*



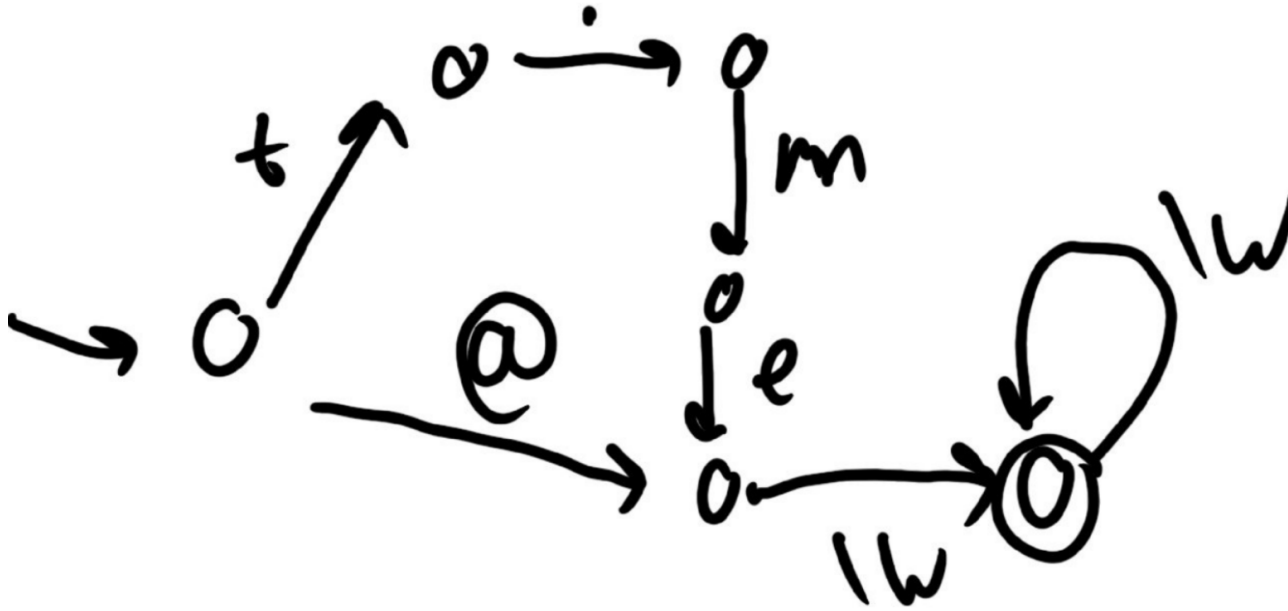
ss'



`s|s'`



```
((t\.me/)|@)
(\w+)
```

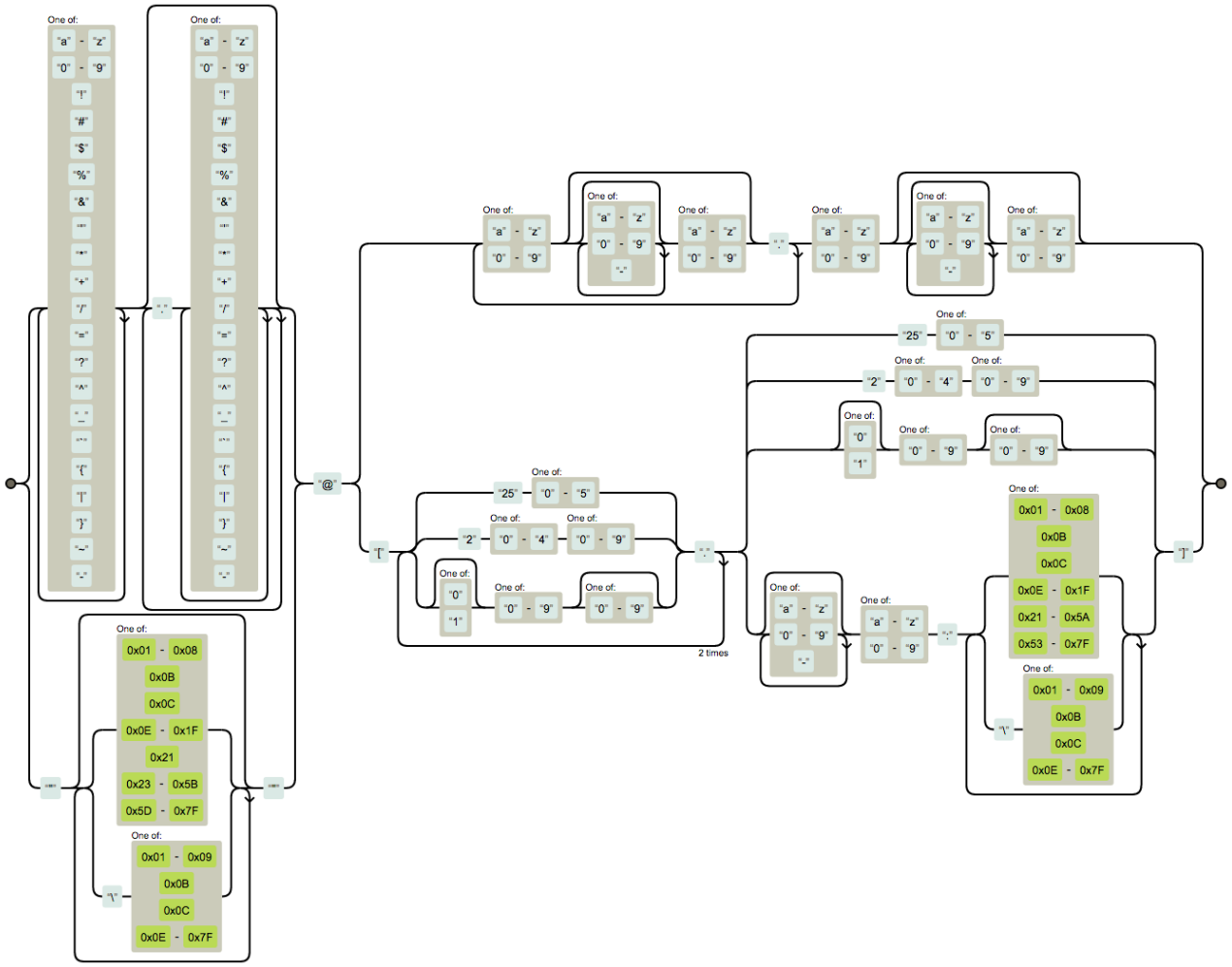


Вопрос

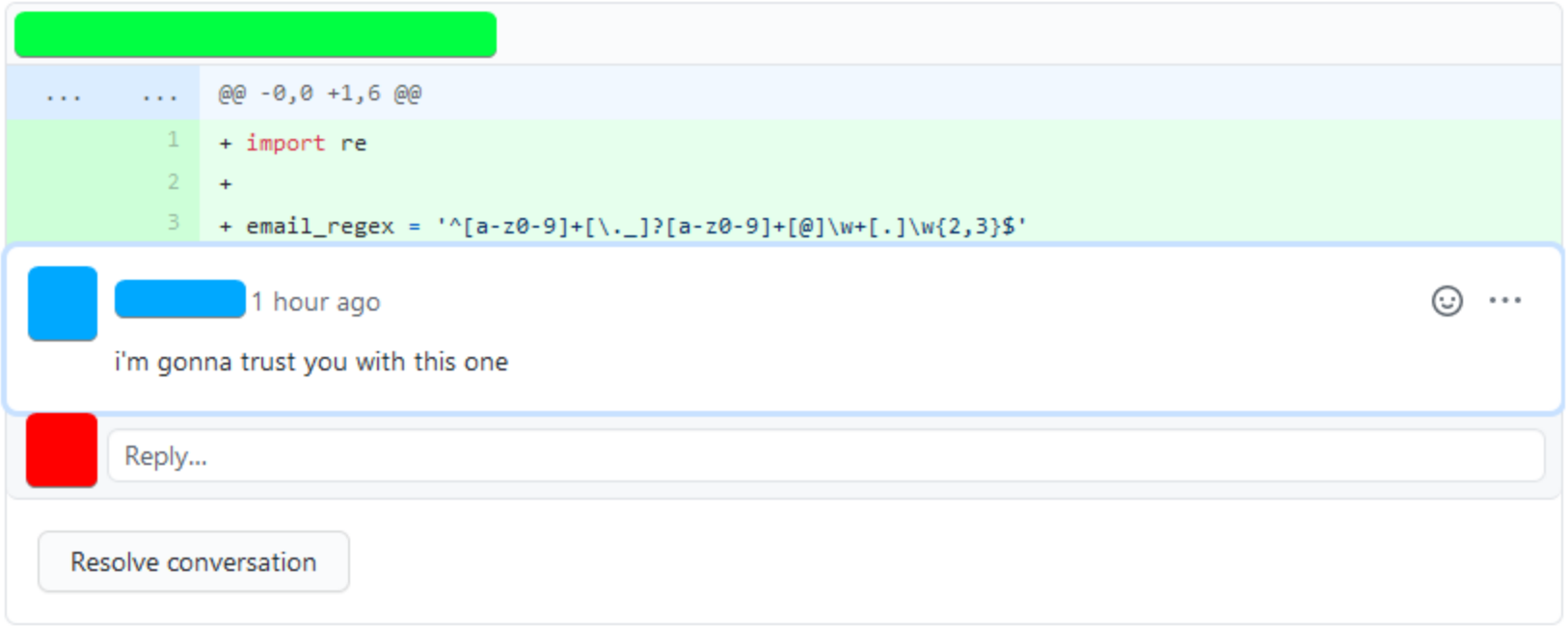
Как сделать `s?`

Емэйл

(?:[a-z0-9!#%&'+/=?^_`{|}~-]+)|"(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]|\[\x01-\x09\x0b\x0c\x0e-\x7f])")@(?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?|\[(?:(?:(2(5[0-5]|[0-4][0-9])|1[0-9][0-9]|[1-9]?[0-9]))\.){3}(?:(2(5[0-5]|[0-4][0-9])|1[0-9][0-9]|[1-9]?[0-9])|[a-z0-9-]*[a-z0-9]:(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f]|\[\x01-\x09\x0b\x0c\x0e-\x7f])+)])



Мем


@@ -0,0 +1,6 @@
1 + import re
2 +
3 + email_regex = '^[a-z0-9]+[\._]?[a-z0-9]+[@]\w+[.]\w{2,3}$'
1 hour ago
i'm gonna trust you with this one
Reply...
Resolve conversation

Идея пет-проекта

Можно пойти в опенсорс-проекты, которые делают всякие парсеры/сериализаторы для разных форматов и посмотреть, есть ли там какие-то issue, по которым авторы просят помощь.

Или можно написать свой язык программирования, который можно парсить, строить ast, и конвертировать в Python/C :)

Для дальнейшего изучения

- НДКА $\rightarrow$ ДКА
- Минимизация автомата
- Машина Тьюринга
- https://regexone.com/ и подобные