Пишем собственный язык и компилятор к нему

Или как сделать так, чтобы ИИ не смог вас заменить

Вадим Гончаров

Проект Anti-DDoS
Группа инновационных исследований (R&D Техблок)
ПАО «МТС»

Perl-Conf.Ru/26

Вадим Гончаров

Проект Anti-DDoS
Группа инновационных исследований (RnDT)
ПАО «МТС»

- Соавтор teleperl — MTProto (Telegram) на чистом Perl
- Си (20 лет): ядро FreeBSD, ipfw, netgraph, демоны, Oracle SQL + COBOL
- Perl (15 лет): классические админские задачи, backend
- Сетевые протоколы: замена TCP и IPv6, IETF CBOR WG

О чём расскажу

1. Зачем свой язык, если есть ИИ
2. С чем ИИ может справиться, а с чем нет
3. Теория компиляторостроения и конечные автоматы
4. Классика Lex/Yacc и Perl-вариант Parse::Yapp
5. Примеры: `teleperl` и файрвол на C/eBPF/XDP
6. Какую часть работы нельзя отдать ИИ целиком

Зачем свой язык

- Классика: **DSL** (Domain Specific Language), **DSM**
  - e.g. конфиги, SQL, GraphViz, TeX, AWK, Make, sed
  - tcpdump — хорошо, iptables — плохо, на Си — ужасно
  - даже JSON: матчим в коде вручную или берём jq?
- Качество документации*: «почти всегда» и «кроме случаев»
  - Грамматика — спецификация: принять или отвергнуть
  - Ошибку ловят на сборке, не в три часа ночи
  - Тесты — примеры; грамматика — сам язык

Зачем свой язык

- Классика: **DSL** (Domain Specific Language), **DSM**
  - e.g. конфиги, SQL, GraphViz, TeX, AWK, Make, sed
  - tcpdump — хорошо, iptables — плохо, на Си — ужасно
  - даже JSON: матчим в коде вручную или берём jq?
- Качество документации*: «почти всегда» и «кроме случаев»
  - Грамматика — спецификация: принять или отвергнуть
  - Ошибку ловят на сборке, не в три часа ночи
  - Тесты — примеры; грамматика — сам язык

* документация? какая ещё документация?

Когда модель вывозит

- **Anka, декабрь 2025**
  язык таблиц сделали под ИИ; 98 правил — в запрос
  99,9 % разбора; цепочки 100 % против 60 % у Python

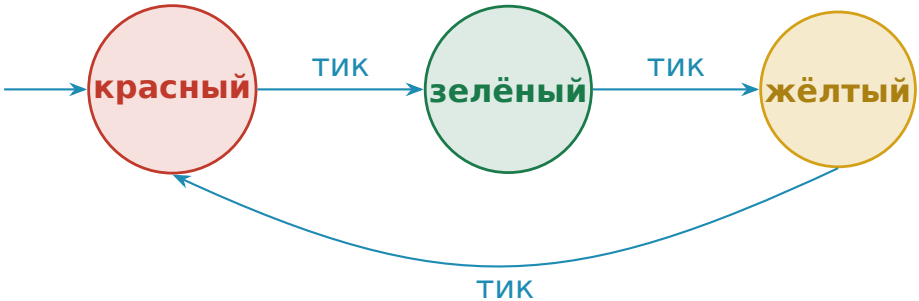
- **Idris + цикл с компилятором, февраль 2026**
  с нуля 22/56; с ошибками компилятора — 54/56
  вывозит не «знание языка», а проверка «да / нет»

Я ВСЁ
А это у нас
жук-невывожук

Когда на незнакомом языке падает

- **EsoLang-Bench, март 2026**
  80 задач: Python 100 %; Brainfuck…Shakespeare 0–11 %
  документацию языка в запрос давали; средние и сложные — ноль
- **PyLang, май 2026**
  учебный язык «C без `for` и либы»: синтаксис выучили — а тесты не проходят в 80 % случаев
- **EAST-ADL, май 2026**
  297 правил в запросе: Claude ∼16 %, не 100 %

Конечный автомат: светофор



- Ровно одно состояние в каждый момент
- Автомат Мура: выход — сам цвет, вход — тик таймера

Автомат в коде: **FreeBSD init** (PID 1)


```c
typedef long (*state_func_t)(void);
typedef state_func_t (*state_t)(void);

/* Change states in the finite state machine. */
static void
transition(state_t s)
{
    current_state = s;
    for (;;)
        current_state = (state_t) (*current_state)();
}

static state_func_t
runcom(void)
{
    state_func_t next_transition;
    if ((next_transition = run_script(_PATH_RUNCOM)) != NULL)
        return next_transition;
    runcom_mode = AUTOBOOT;
    return (state_func_t) read_ttys;
}
```

сейчас	вход	дальше
runcom	ok	read_ttys
read_ttys	ok	multi_user
multi_user	SIGHUP	clean_ttys
*	SIGTERM	death

Состояние — функция. Переход — `return`.

switch/case: вырезать /* */


```
enum { CODE, SLASH, CMT, STAR,
       STR, SESC, CHR, CESC };
int st = CODE, c;

while ((c = getchar()) != EOF)
switch (st) {
case CODE:
    if (c == '/')       st = SLASH;
    else if (c == '"')  { putchar(c
         ); st = STR; }
    else if (c == '\'') { putchar(c
         ); st = CHR; }
    else                putchar(c);
    break;
case SLASH:            /* / vs /*
     vs divide */
    if (c == '*') st = CMT;
    else {
        putchar('/');   /* operator
              */
        ungetc(c, stdin);
        st = CODE;
    }
    break;
```

```
case CMT:          /* also " and '
     */
    if (c == '*') st = STAR;
    break;
case STAR:
    if (c == '/') st = CODE;
    else if (c != '*')
        st = CMT;   /* ** still in
              */
    break;
// "/* x */" is a string, keep
// /* "x" */ is a comment, drop
// /* * /  still open
```

```
case STR:
    putchar(c);
    if (c == '\\')      st = SESC;
    else if (c == '"') st = CODE;
    break;
case SESC:
    putchar(c); st = STR;
    break;
case CHR:          /* '"' and '\''
     */
    putchar(c);
    if (c == '\\')      st = CESC;
    else if (c == '\'') st = CODE;
    break;
case CESC:
    putchar(c); st = CHR;
    break;
}
/* no //, no \\newline, no
     trigraphs */
```

8 хватает на /* */, строки, символы и escape. Деление — тот же SLASH: / не * → оператор. Десяток состояний и больше — если ещё //, склейка строк через \ и триграфы.

Зачем так сложно, есть же регулярки?

Зачем так сложно, есть же регулярки?

You can't parse [X]HTML with regex. Because HTML can't be parsed by regex. Regex is not a tool that can be used to correctly parse HTML. As I have answered in HTML-and-regex questions here so many times before, the use of regex will not allow you to consume HTML. Regular expressions are a tool that is insufficiently sophisticated to understand the constructs employed by HTML. HTML is not a regular language and hence cannot be parsed by regular expressions. Regex queries are not equipped to break down HTML into its meaningful parts. so many times but it is not getting to me. Even enhanced irregular regular expressions as used by Perl are not up to the task of parsing HTML. You will never make me crack. HTML is a language of sufficient complexity that it cannot be parsed by regular expressions. Even Jon Skeet cannot parse HTML using regular expressions. Every time you attempt to parse HTML with regular expressions, the unholy child weeps the blood of virgins, and Russian hackers pwn your webapp. Parsing HTML with regex summons tainted souls into the realm of the living. HTML and regex go together like love,

Зачем так сложно, есть же регулярки?

marriage, and ritual infanticide. The cannot hold it is too late. The force of regex and HTML together in the same conceptual space will destroy your mind like so much watery putty. If you parse HTML with regex you are giving in to Them and their blasphemous ways which doom us all to inhuman toil for the One whose Name cannot be expressed in the Basic Multilingual Plane, he comes. HTML-plus-regexp will liquify the nerves of the sentient whilst you observe, your psyche withering in the onslaught of horror. Regex-based HTML parsers are the cancer that is killing StackOverflow *it is too late it is too late we cannot be saved* the transgression of a child ensures regex will consume all living tissue (except for HTML which it cannot, as previously prophesied) *dear lord help us how can anyone survive this scourge* using regex to parse HTML has doomed humanity to an eternity of dread torture and security holes *using rege*x as a tool to process HTML establishes a brea*ch between this world* and the dread realm of corrupt entities (like

Зачем так сложно, есть же регулярки?

SGML entities, but *more corrupt) a mere glimps*e of the world of reg**ex parsers for HTML will ins**tantly transport a p*rogrammer's consciousness i*nto a w*orl*d of ceaseless screaming, he comes, ~~the pestilent sl~~ithy regex-infection wi**ll devour your HT**ML parser, application and existence for all time like Visual Basic only worse *he comes he comes do not fi*ght h**e comes, his** unholy radiańcé d*estroying all enlightenment, HTML tags **leaking from your eyes like liq**uid pain, the song of r*egular expr*~~ession parsing~~ will ext*inguish the voices of mor**tal man from the sp**here I can see it can you see it it is beautiful t*he f`inal snuf`fing *of the lie**s of Man ALL IS LOST A**LL IS L**OST the *pony he come*s he comes ~~he comes~~ ***the* ich**or permeate*s all* MY FA*CE MY FACE oh god n**o NO NOOOO N**Θ stop t*he an*gles `are` n**ot real ZALGO IS TONY THE PONY HE COMES**

Have you tried using an XML parser instead?

bobince, 2009 — HTML не регулярный, regex не парсер.

Зачем так сложно, есть же регулярки?

bobince, 2009 — HTML не регулярный, regex не парсер.

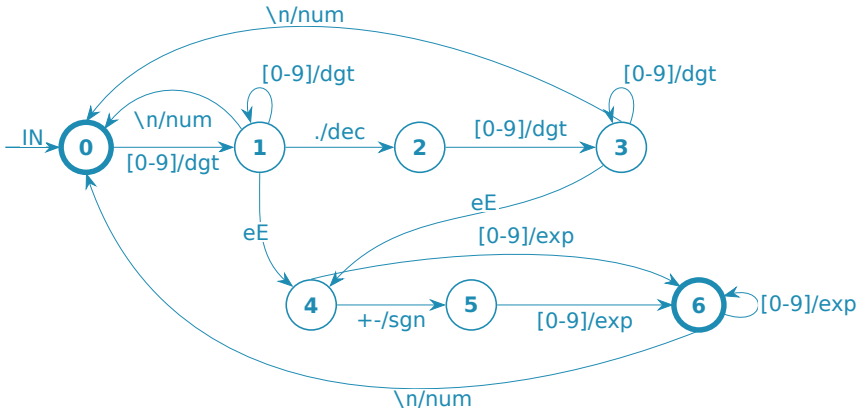
Типы языков по Хомскому

тип	язык	пример
3	регулярный	regex, Ragel, lex
2	контекстно-свободный	C, Python < 3.9
1	контекстно-зависимый	почти нет
0	без ограничений	машина Тьюринга

Регулярка — запись регулярного языка.
HTML — не регулярный. Число — регулярное.

Хомский: тип 3 — автомат, тип 2 — стек. LALR(1) — на yacc.

Ragel: число как автомат



да

```
2
78.1
2e5
69.0e-3
```

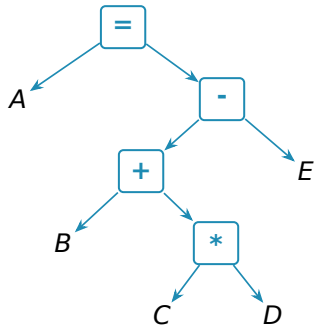
нет

```
.3
46.
-5
3.e2
```

Действия на стрелках (dgt, dec, exp) — ближе к автомату Мили.

AST — Abstract Syntax Tree

$$A = B + C * D - E$$



вызовы функций

```
assign-var(A,
    minus(
        plus(get-var(B),
             mul(get-var(C),
                 get-var(D))),
        get-var(E)))
```

лисп-форма

```
(= A
   (- (+ B
         (* C D))
      E))
```

Парсер строит дерево. Генератор его обходит.

Конвейер компилятора


```
source
  -> lexer   (tokens)
    -> parser  (tree / actions)
      -> codegen (Perl modules, bytecode, ...)
```

- Лексер разобьёт входящий текст на токены
- Парсер проверяет, что токены — фраза языка, собирает из них AST согласно BNF
- Дальше не «магия», а обычная генерация кода

Lex и Yacc — разные работы

Lex / Flex

- регулярки
- числа, имена, строки
- пропускает пробелы и комментарии
- можно задавать состояния автомата в сложных случаях (например, <INSIDE_STR>)

Yacc / Bison

- контекстно-свободная, LALR(1)
- вложенность, приоритеты
- конфликт shift/reduce — «Хьюстон, у нас проблема»

BNF: калькулятор

$$A = B + C * D - E$$

```
<stmt>   ::= <name> "=" <expr>

<expr>   ::= <expr> "+" <term>
           | <expr> "-" <term>
           | <term>

<term>   ::= <term> "*" <factor>
           | <factor>

<factor> ::= <name> | <number> | "(" <expr> ")"
<name>   ::= <letter> | <name> <letter> | <name> <digit>
<number> ::= <digits> | <digits> "." | <digits> "." <digits>
<digits> ::= <digit> | <digits> <digit>
<letter> ::= "A" | ... | "Z" | "a" | ... | "z"
<digit>  ::= "0" | "1" | ... | "9"
```

Нетерминал в < >, терминал в кавычках.

Калькулятор: Lex (calc.l)


```
%{
#include "y.tab.h"
#include <stdlib.h>
#include <string.h>
%}

%option noyywrap

%%
[0-9]+(\.[0-9]*)?       { yylval.num = atof(yytext); return NUMBER; }
[A-Za-z][A-Za-z0-9]*    { yylval.name = strdup(yytext); return NAME; }
[ \t]+                  ;
\n                      { return '\n'; }
.                       { return yytext[0]; }
%%
```

C=4, D=3+5, A=B+C*D-E → 31.

Калькулятор: Yacc — шапка


```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int yylex(void);
void yyerror(const char *s);
double lookup(const char *n);
void store(const char *n, double v);
%}

%union {
    double num;
    char *name;
}

%token <num> NUMBER
%token <name> NAME
%type  <num> stmt expr term factor
```

calc.l
15 строк / 344 B
↓
lex.yy.c
1773 / 45 KB

calc.y
61 строка / 933 B
↓
y.tab.c
537 / 15 KB

Огромный table-driven конечный автомат.

Калькулятор: Yacc — правила


```
%%
input   : /* empty */
        | input line
        ;
line    : '\n'
        | stmt '\n'     { printf("%g\n", $1); }
        ;
stmt    : NAME '=' expr
          { store($1, $3); $$ = $3; free($1); }
        | expr
        ;
expr    : expr '+' term { $$ = $1 + $3; }
        | expr '-' term { $$ = $1 - $3; }
        | term
        ;
term    : term '*' factor { $$ = $1 * $3; }
        | factor
        ;
factor  : NUMBER
        | NAME          { $$ = lookup($1); free($1); }
        | '(' expr ')'  { $$ = $2; }
        ;
```

```
%%

void
yyerror(const char *s)
{
    fprintf(stderr, "%s\n
        ", s);
}

int
main(void)
{
    return (yyparse());
}
```

LALR(1): слева или справа

L — слева направо. A — 1 токен вперёд. R — сворачиваем то, что уже на стеке.

рекурсия слева

```
expr : expr '+' term
     | term
```

1+2+3 → (1+2)+3

рекурсия справа

```
expr : term '+' expr
     | term
```

1+2+3 → 1+(2+3)

- Shift — токен на стек. Reduce — стек в правило
- Yacc любит левую рекурсию: `| term` — база, `expr` растёт слева
- Конфликт shift/reduce означает неоднозначность во входной грамматике

Действия: C против Perl

Yacc / C

```
/* yacc/bison: actions are C, values live in %union */
expr: expr '+' term  { $$ = $1 + $3; }
    | term           { $$ = $1; }
    ;
```

Yapp / Perl

```
# Parse::Yapp: actions are Perl.
# $_[1], $_[3] are the RHS values (like $1, $3).
# $_[0] is the parser object (YYData, YYExpect, ...).

expr: expr '+' term  { $_[1] + $_[3] }
    | term           { $_[1] }
    ;
```

Parse::Yapp

- Грамматика как у yacc: `%token`, `%%`, приоритеты
- Действия — Perl, не C: не нужен `%union`
- `yapp(1)` генерирует модуль-парсер
- Лексер *не* входит в поставку: его пишете сами

Пример: teleperl

- Pure-Perl клиент MTProto/Telegram
- Схема API задана языком TL
- `tl.yp` + `tl-gen.pl` читают схему и генерируют классы
- `yapp -m TL -s tl.yp` → `TL.pm`

github.com/nuclight/teleperl

Вход: схема TL


```
peerUser#59511722    user_id:long = Peer;
peerChat#36c6019a    chat_id:long = Peer;
peerChannel#a2a5371e channel_id:long = Peer;

chatEmpty#29562865     id:long = Chat;
chatForbidden#6592a1a7 id:long title:string = Chat;
channel#1c32b11c flags:# date:int access_hash:flags.13?long photo:ChatPhoto
    usernames:flags2.0?Vector<Username> megagroup:flags.8?true ... = Chat;

---functions---
messages.getFullChat#aeb00b34 chat_id:long = messages.ChatFull;
```

Peer.pm

```
package Telegram::PeerABC;

package Telegram::PeerUser;
use base 'TL::Object';
push @ISA, 'Telegram::PeerABC';
our $hash = 0x59511722;
```

ClifTg.pm

```
if ($ds->isa('Telegram::Messages::DialogsABC')) {
    if ($peer->isa('Telegram::PeerUser')) {..}
    if ($peer->isa('Telegram::PeerChannel')) {..}
    if ($peer->isa('Telegram::PeerChat')) {..}
    ...
}
```

scheme227: один тип справа — *ABC.

Лексер — обычный Perl


```
# Hand-written lexer in tl-gen.pl (not lex/flex).
# Returns (TOKEN, value) pairs for Parse::Yapp.
sub _lex {
    my $input = $_[0]->YYData->{DATA};
    return ('', undef) if pos($$input) >= length($$input);

    $$input =~ m{\G(?:\s+|//[^\n]*|/\*.*?\*/)+}xsgc;

    $$input =~ /\G(---functions---)/gc and return ('FUNCTIONS', $1);
    $$input =~ /\G(---types---)/gc     and return ('TYPES', $1);
    $$input =~ /\G([a-z][a-zA-Z0-9_]*)/gc and return ('LC_ID', $1);
    $$input =~ /\G([A-Z][a-zA-Z0-9_]*)/gc and return ('UC_ID', $1);
    $$input =~ /\G(\#[0-9a-fA-F]{1,8})/gc and return ('ID_HASH', $1);
    $$input =~ /\G(.)/gc or die "Match (.) failed";
    return ($1, $1);
}
```

Грамматика tl.yp


```
%token NUM LC_ID UC_ID ID_HASH
%token FUNCTIONS TYPES
%start program
%left '.'

%%

program: declarations
         { print "done\n"; }

declarations: /* empty */
            | declarations declaration
            ;

declaration: combinator
           | FUNCTIONS
             { $tl_context = TL_FUNCS }
           | TYPES
             { $tl_context = TL_TYPES }
           ;
```

```
combinator: full_id opt_hash opt_args args
            '=' result_type ';'
  {
    my $rec = { id => $_[1], hash => $_[2],
                type => $_[6], args => $_[4] };
    if ($tl_context == TL_TYPES) {
      push @{ $_[0]->YYData->{types} }, $rec;
    } else {
      push @{ $_[0]->YYData->{funcs} }, $rec;
    }
  }
;

args: /* empty */ { [] }
    | args arg { push @{$_[1]}, $_[2]; $_[1] }
    ;
arg: LC_ID ':' type_term
     { { name => $_[1], type => $_[3] } }
   | ...

full_id: LC_ID | UC_ID
       | LC_ID '.' LC_ID { $_[1].".".$_[3] }
       | LC_ID '.' UC_ID { $_[1].".".$_[3] }
```

Списки растут слева, как `expr`. Тело combinator — Perl.

Кодогенерация


```
# tl-gen.pl: one type on the right becomes *ABC.
$basepkg .= 'ABC';    # Abstract Base Class

print $f "package $basepkg;\n";
print $f "package $pkg;\nuse base 'TL::Object';\n";
print $f "push \@ISA, '$basepkg';\n";
print $f "our \$hash = 0x$hash;\n";
print $f "sub pack   { pack 'L<', \$hash; ... }\n";
print $f "sub unpack { ... }\n";
```

eBPF/XDP: язык правил

- Правила → Parse::Yapp → JSON
- JSON → Template::Toolkit → Си
- `clang/bpf` → XDP в ядре

```
when proto == udp and pps > 1000 then drop;
when tcp.dport == 53 or tcp.dport == 5353 then reject;
default pass;
```

Yapp → JSON → TT2 → Си в XDP.

eBPF/XDP: дерево в JSON


```
{
  "rules": [
    { "match": ["AND",
        ["byte", ["ip", "proto"], "==", 17],
        ["little-endian32", ["meta", "pps"], ">", 1000]],
      "actions": [["drop"]] },
    { "match": ["OR",
        ["big-endian16", ["tcp", "dport"], "==", 53],
        ["big-endian16", ["tcp", "dport"], "==", 5353]],
      "actions": [["reject"]] },
    { "match": ["TRUE"],
      "actions": [["pass"]] }
  ]
}
```

byte и LE32 — ширина. BE16 — bpf_ntohs.

eBPF/XDP: шаблон и Си


```
[% xdp = { drop => 'XDP_DROP', reject => 'XDP_DROP', pass => 'XDP_PASS' } -%]
[% FOREACH r IN rules -%]
[% IF r.match.0 == 'TRUE' -%]return [% xdp.${r.actions.0.0} %];
[% ELSE -%]if ([% INCLUDE expr n=r.match %])
    return [% xdp.${r.actions.0.0} %];
[% END -%][% END -%]
[% BLOCK expr -%][% SWITCH n.0 -%]
[% CASE 'AND' %][% INCLUDE expr n=n.1 %] && [% INCLUDE expr n=n.2 %]
[% CASE 'OR' %]([% INCLUDE expr n=n.1 %] || [% INCLUDE expr n=n.2 %])
[% CASE 'big-endian16' %]bpf_ntohs(ctx->[% n.1.1 %]) [% n.2 %] [% n.3 %]
[% CASE %]ctx->[% n.1.1 %] [% n.2 %] [% n.3 %]
[% END -%][% END -%]
```

```
if (ctx->proto == 17 && ctx->pps > 1000)
    return XDP_DROP;
if (bpf_ntohs(ctx->dport) == 53 ||
    bpf_ntohs(ctx->dport) == 5353)
    return XDP_DROP;
return XDP_PASS;
```

TT2-шаблон сгенерировал Cursor Grok 4.6 :)

FOREACH печатает `if`; INCLUDE — `&&`, `||`, `bpf_ntohs`.

Что остаётся человеку

- Решить, что один *токен*: `---functions---`, не три минуса, слово и три минуса
- Разрешить конфликт shift/reduce по смыслу, а не «пока уасс замолчит»
- Выбрать, *во что* компилировать — это и есть продукт
- `.yp` можно набросать и моделью, но ответственность строго формального матча «да / нет» ей нельзя отдать

Выводы

- Lex и Yacc — регулярки отдельно, грамматика отдельно
  - этих динозавров достаточно для большинства ваших мини-языков
  - для более новых исследований (e.g. PEG) их всё равно надо знать, понимать, иметь опыт
- Parse::Yapp: те же шаги, действия уже на Perl
- `teleperl` и XDP: разные цели компиляции, но проходим через ту же стадию дерева
- ИИ не телепат, не знает смысл: не заменит выбор языка, цели компиляции, решение конфликтов

Источники

- Anka, 2025 — arXiv:2512.23214
- Idris, 2026 — arXiv:2602.11481
- EsoLang-Bench — arXiv:2603.09678
- PyLang — arXiv:2605.15607
- EAST-ADL — arXiv:2605.21465
- teleperl — github.com/nuclight/teleperl
- TL подробнее — habr.com/ru/post/472970
- HTML + regex — stackoverflow.com/a/1732454
- Хомский — Wikipedia, Chomsky hierarchy
- Ragel — colm.net/open-source/ragel
- FreeBSD init — sbin/init/init.c
- BNF — Wikipedia, Backus–Naur form
- LALR(1) — Wikipedia, LALR parser
- FreeBSD bc — usr.bin/bc
- Parse::Yapp — metacpan.org/dist/Parse-Yapp
- Template::Toolkit — metacpan.org/dist/Template-Toolkit

Спасибо за внимание!

Вопросы?

Perl-Conf.Ru/26