6.110 Re-lecture 1

Regular expressions, automata, grammars, parse trees

SMILE!

YOU’RE
ON
CAMERA!

4 lectures in 1

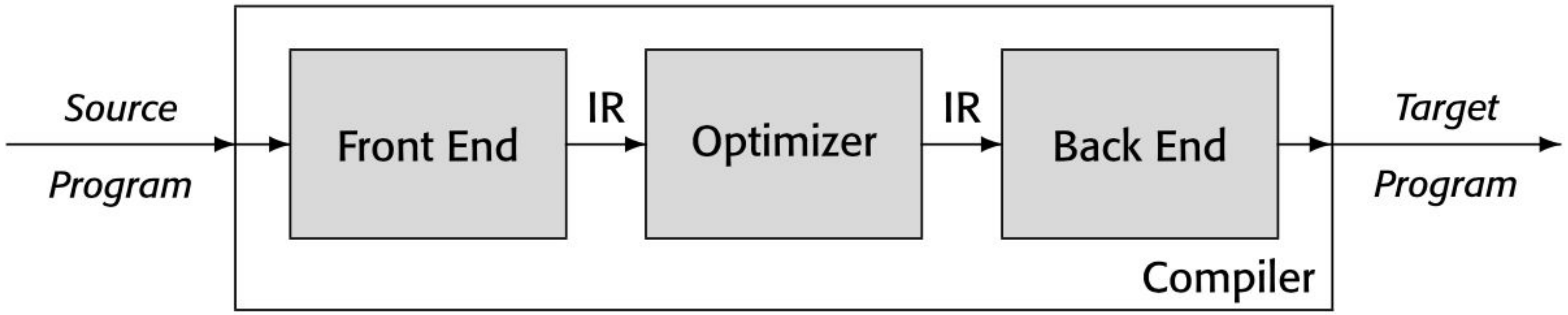
Focus on theory

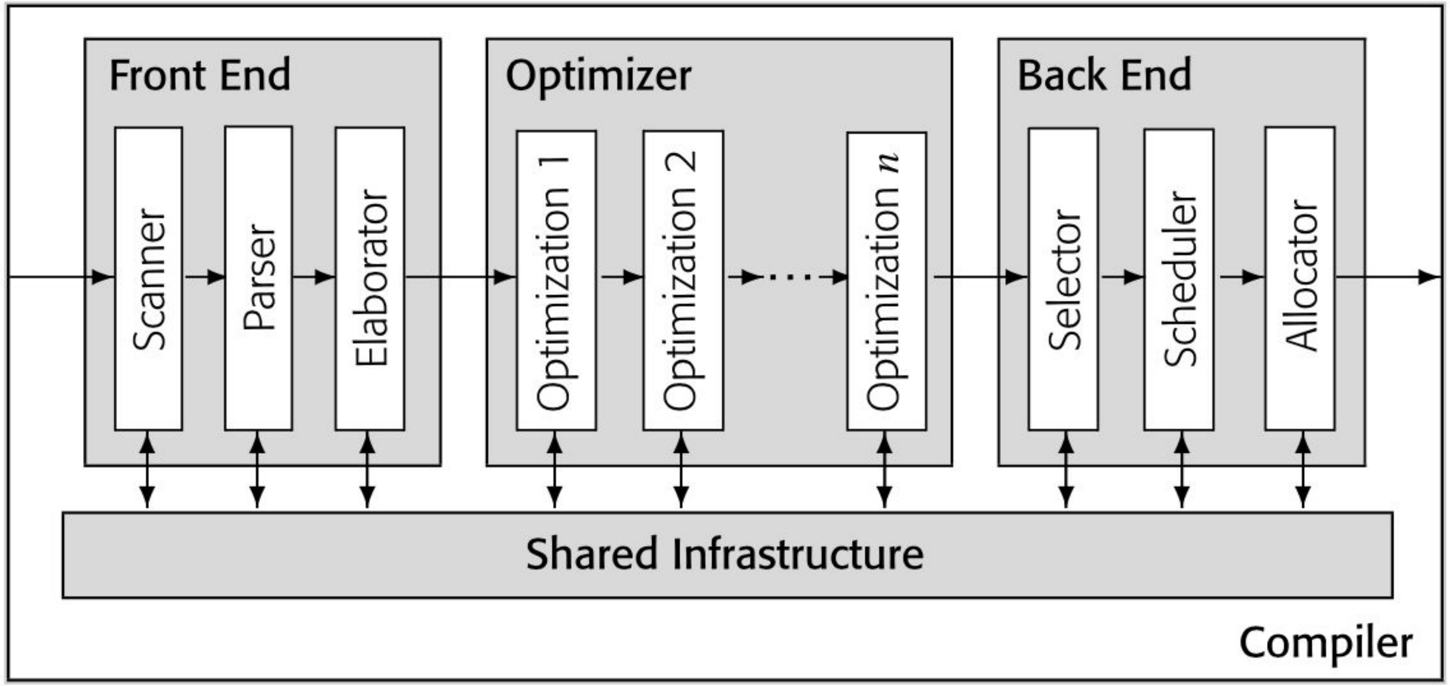
Ideal: 90 minutes

Stop me

Plan

- Overview of compiler components and optimizations
- Regular expressions
- Finite automata
- Duality and constructions
- Tokenization
- Context-free grammar
- Ambiguity
- Precedence


Source Program
Front End
IR
Optimizer
IR
Back End
Target Program
Compiler



Front End
Scanner
Parser
Elaborator
Optimizer
Optimization 1
Optimization 2
Optimization n
Back End
Selector
Scheduler
Allocator
Shared Infrastructure
Compiler

Front end

```
# Comment 1
 # Comment 2

# Factorial:

def fact(  x\
):

  if x == -1:
    return 1.j

  elif x ==0:

    return 1
  else:

        return x* fact(x

- 1)

s = "foo\
\\ \n\'\""
```

```
# Comment 1
 # Comment 2

# Factorial:

def fact(  x\
):

  if x == -1:
    return 1.j

  elif x ==0:

    return 1
  else:

        return x* fact(x

- 1)

s = "foo\
\\ \n\'\""
```

```
(KEYWORD def)
(ID "fact")
(PUNCT "(")
(ID "x")
(PUNCT ")")
(PUNCT ":")
(NEWLINE)
(INDENT)
(KEYWORD if)
(ID "x")
(PUNCT "==")
(PUNCT "-")
(LIT 1)
(PUNCT ":")
(NEWLINE)
(INDENT)
(KEYWORD return)
(LIT +1.i)
(NEWLINE)
(DEDENT)
(KEYWORD elif)
(ID "x")
(PUNCT "==")
(LIT 0)
(PUNCT ":")
(NEWLINE)
(INDENT)
(KEYWORD return)
(LIT 1)
(NEWLINE)
(DEDENT)
(KEYWORD else)
(PUNCT ":")
(NEWLINE)
(INDENT)
(KEYWORD return)
(ID "x")
(PUNCT "*")
(ID "fact")
(PUNCT "(")
(ID "x")
(PUNCT "-")
(LIT 1)
(PUNCT ")")
(NEWLINE)
(DEDENT)
(DEDENT)
(ID "s")
(PUNCT "=")
(LIT "foo\\ \n\'\"")
(NEWLINE)
(ID "fact")
(PUNCT "(")
(LIT 20)
(PUNCT ")")
(NEWLINE)
(ENDMARKER)
```

```
# Comment 1
 # Comment 2

# Factorial:

def fact(  x\
):

  if x == -1:
    return 1.j

  elif x ==0:

    return 1
  else:

        return x* fact(x

- 1)

s = "foo\
\\ \n\'\""
```

```
Module(
  body=[
    FunctionDef(
      name='fact',
      args=arguments(
        posonlyargs=[],
        args=[
          arg(arg='x')],
        kwonlyargs=[],
        kw_defaults=[],
        defaults=[]),
      body=[
        If(
          test=Compare(
            left=Name(id='x', ctx=Load()),
            ops=[
              Eq()],
            comparators=[
              UnaryOp(
                op=USub(),
                operand=Constant(value=1))]),
          body=[
            Return(
              value=Constant(value=1j))],
          orelse=[
       …
```

Intermediate representation

$$a \leftarrow a \times 2 \times b \times c \times d$$

$$a \leftarrow a \times 2 \times b \times c \times d$$

$$\begin{aligned}
t_0 &\leftarrow a \times 2 \\
t_1 &\leftarrow t_0 \times b \\
t_2 &\leftarrow t_1 \times c \\
t_3 &\leftarrow t_2 \times d \\
a &\leftarrow t_3
\end{aligned}$$

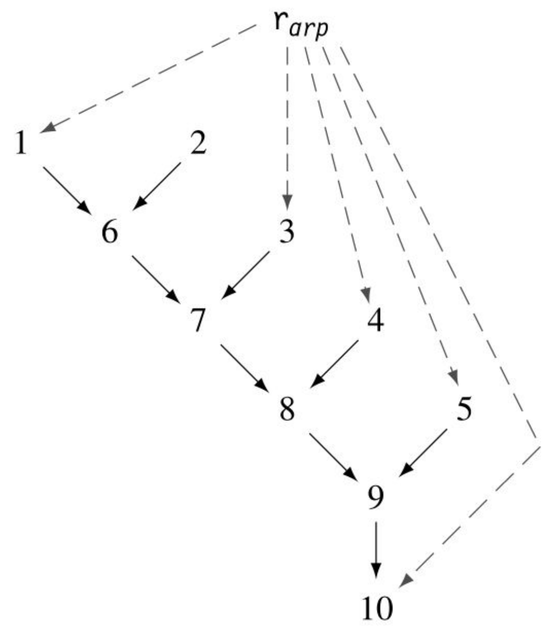
$a \leftarrow a \times 2 \times b \times c \times d$

```
loadAI   r_arp, @a ⇒ r_a         // load 'a'
loadI    2         ⇒ r_2         // constant 2 into r_2
loadAI   r_arp, @b ⇒ r_b         // load 'b'
loadAI   r_arp, @c ⇒ r_c         // load 'c'
loadAI   r_arp, @d ⇒ r_d         // load 'd'
mult     r_a, r_2  ⇒ r_a         // r_a ← a × 2
mult     r_a, r_b  ⇒ r_a         // r_a ← (a × 2) × b
mult     r_a, r_c  ⇒ r_a         // r_a ← (a × 2 × b) × c
mult     r_a, r_d  ⇒ r_a         // r_a ← (a × 2 × b × c) × d
storeAI  r_a       ⇒ r_arp, @a   // write r_a back to 'a'
```

$$a \leftarrow a \times 2 \times b \times c \times d$$

```
1   loadAI  r_arp, @a ⇒ r_a
2   loadI   2         ⇒ r_2
3   loadAI  r_arp, @b ⇒ r_b
4   loadAI  r_arp, @c ⇒ r_c
5   loadAI  r_arp, @d ⇒ r_d
6   mult    r_a, r_2  ⇒ r_a
7   mult    r_a, r_b  ⇒ r_a
8   mult    r_a, r_c  ⇒ r_a
9   mult    r_a, r_d  ⇒ r_a
10  storeAI r_a       ⇒ r_arp, @a
```

(a) Example Code from Chapter 1



(b) Dependence Graph for the Example

`if (x = y)`

  `then` $stmt_1$

  `else` $stmt_2$

$stmt_3$

$$\begin{tikzcd}
 & \texttt{if (x = y)} \arrow[dl] \arrow[dr] & \\
stmt_1 \arrow[dr] & & stmt_2 \arrow[dl] \\
 & stmt_3 &
\end{tikzcd}$$

Back end

$a \leftarrow a \times 2 \times b \times c \times d$

```
loadAI   r_arp, @a ⇒ r_a         // load 'a'
loadI    2         ⇒ r_2         // constant 2 into r_2
loadAI   r_arp,@b  ⇒ r_b         // load 'b'
loadAI   r_arp,@c  ⇒ r_c         // load 'c'
loadAI   r_arp,@d  ⇒ r_d         // load 'd'
mult     r_a, r_2  ⇒ r_a         // r_a ← a × 2
mult     r_a, r_b  ⇒ r_a         // r_a ← (a × 2) × b
mult     r_a, r_c  ⇒ r_a         // r_a ← (a × 2 × b) × c
mult     r_a, r_d  ⇒ r_a         // r_a ← (a × 2 × b × c) × d
storeAI  r_a       ⇒ r_arp, @a   // write r_a back to 'a'
```

```
addi    sp,sp,-32
sw      ra,28(sp)
sw      s0,24(sp)
addi    s0,sp,32
sw      a0,-20(s0)
sw      a1,-24(s0)
sw      a2,-28(s0)
sw      a3,-32(s0)
lw      a4,-20(s0)
lw      a5,-24(s0)
mul     a4,a4,a5
lw      a5,-28(s0)
mul     a4,a4,a5
lw      a5,-32(s0)
mul     a5,a4,a5
slli    a5,a5,1
mv      a0,a5
lw      ra,28(sp)
lw      s0,24(sp)
addi    sp,sp,32
jr      ra
```

Formal languages

Alphabet

$$\Sigma = \{a, b, c, \ldots, z\}$$

$$\Sigma = \{0, 1\}$$

$$\Sigma = \{\texttt{false}, \texttt{true}\}$$

Σ = English words

String

abcdababab

11100011001

3

~~00000…~~

'i' 'like' 'six' 'oh' 'three' 'five'

Language

L = {1, 01, 10, 001, 010, 100, 0001, 0010, 0100, 1000, 00001, 00010, 00100, 01000, 10000, 000001, …}

(assuming $\Sigma = \{0, 1\}$)

L = set of binary strings that contain exactly one 1

(assuming $\Sigma = \{0, 1\}$)

L(s) = whether s contains exactly one 1 (yes or no)

(assuming $\Sigma = \{0, 1\}$)

L = set of decimal numbers that are divisible by 3

(assuming $\Sigma = \{0, 1, 2, \ldots, 9\}$)

L = set of valid hexadecimal numbers

(assuming Σ = ASCII characters)

L = set of syntactically valid Python programs

(assuming Σ = ASCII characters)

L = set of syntactically valid Python programs

(assuming Σ = Python tokens)

L = set of Python source interpretable without error

Regular languages

Regular expression

```
(617|857)-253-(0|1|..|9)(0|1|..|9)(0|1|..|9)(0|1|..|9)
```

0 Empty string

0 Empty string

ε

1 A letter from Σ

0

2 Concatenation

$a \cdot b$

2 Concatenation

ab

2 Concatenation

234324

3 Alternation

$a \mid b$

3 Alternation/Union

$$a \cup b$$

3 Alternation

```
0|1
```

3 Alternation

```
000|001|100|101
```

3 Alternation

```
(0|1)0(0|1)
```

3 Alternation

```
(617|857)-253-(0|1|..|9)(0|1|..|9)(0|1|..|9)(0|1|..|9)
```

4 Kleene star

$a*$

4 Kleene star

```
(0|1)*
```

4 Kleene star

```
0*10*
```

"regex"

[0-9A-Fa-f]

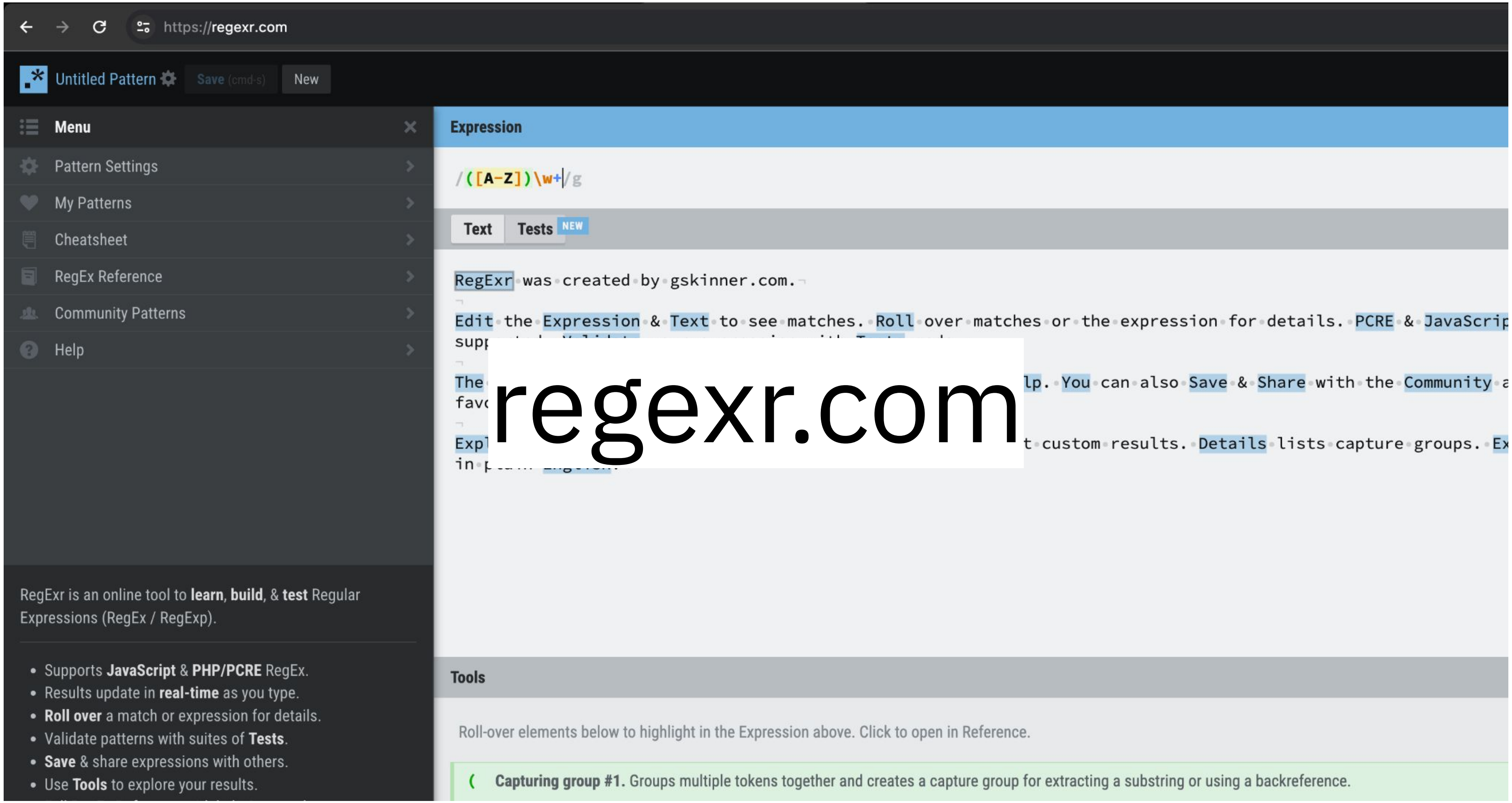
[01]{8}

0|(1[01]*)

```
0x[0-9A-Fa-f]+
0x[0-9A-Fa-f][0-9A-Fa-f]*
```

```
617-253-\d{4}
```

.*



regexr.com

```
/([A-Z])\w+/g
```

```
RegExr was created by gskinner.com.

Edit the Expression & Text to see matches. Roll over matches or the expression for details. PCRE & JavaScrip
supp

The                                                              lp. You can also Save & Share with the Community a
favo

Exp                                                              t custom results. Details lists capture groups. Ex
in p
```

RegExr is an online tool to **learn**, **build**, & **test** Regular Expressions (RegEx / RegExp).

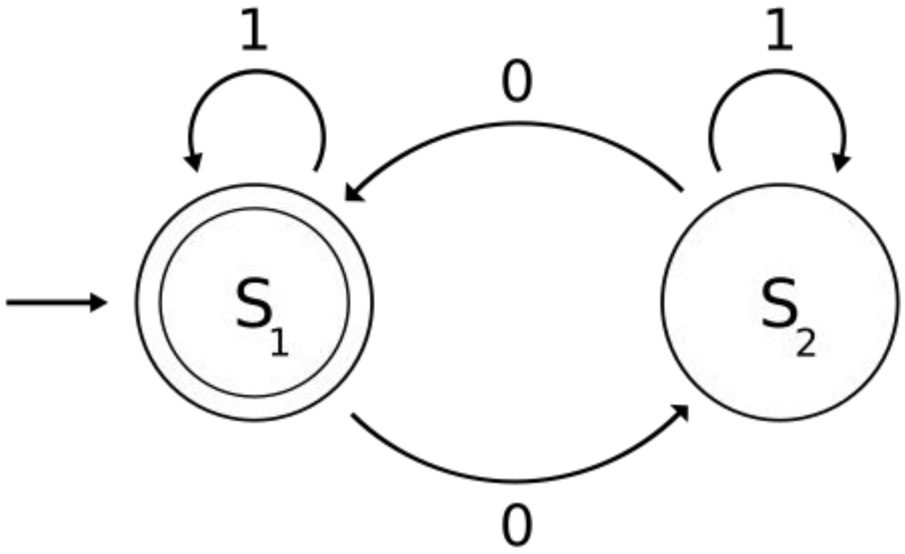
- Supports **JavaScript** & **PHP/PCRE** RegEx.
- Results update in **real-time** as you type.
- **Roll over** a match or expression for details.
- Validate patterns with suites of **Tests**.
- **Save** & share expressions with others.
- Use **Tools** to explore your results.

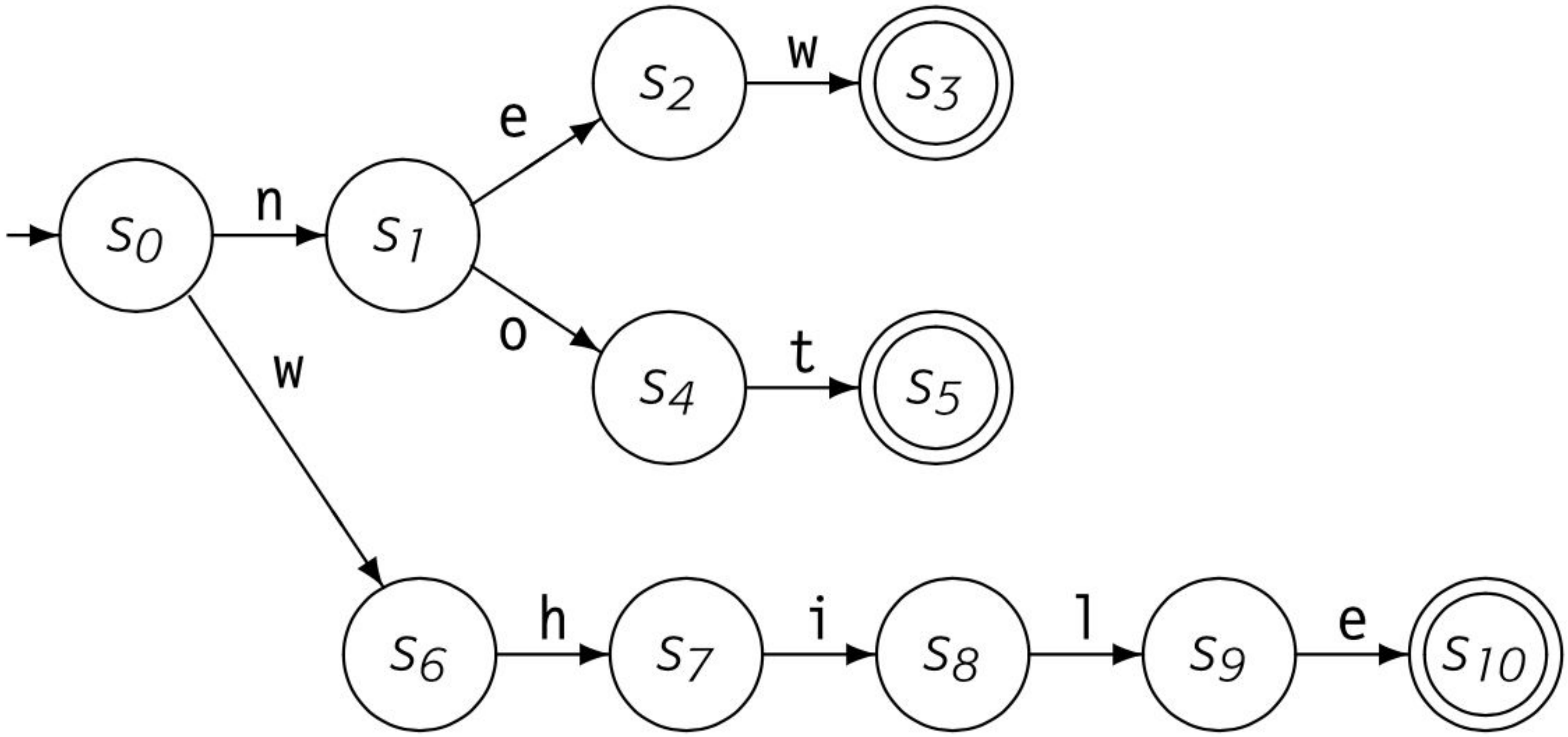
Roll-over elements below to highlight in the Expression above. Click to open in Reference.

(**Capturing group #1.** Groups multiple tokens together and creates a capture group for extracting a substring or using a backreference.

Deterministic Finite Automata

$$M = (Q, \Sigma, \delta, q_0, F)$$

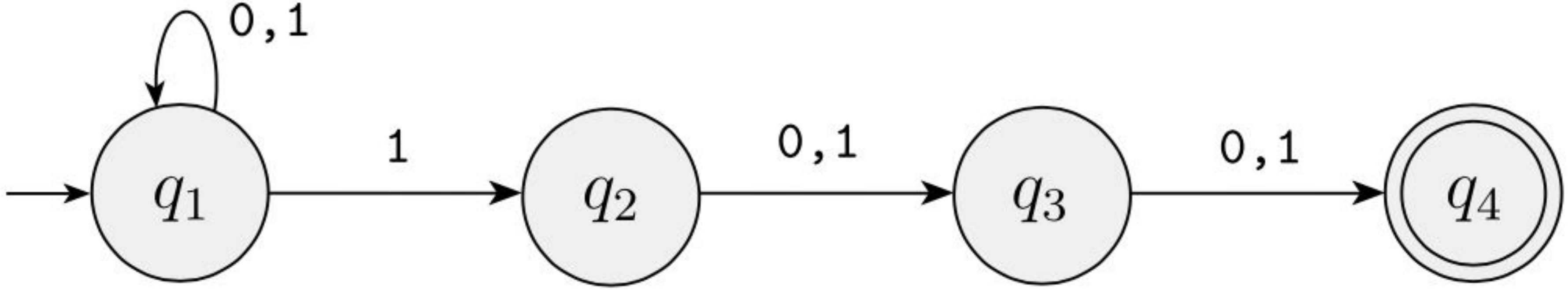

1
1
0
S_1
S_2
0



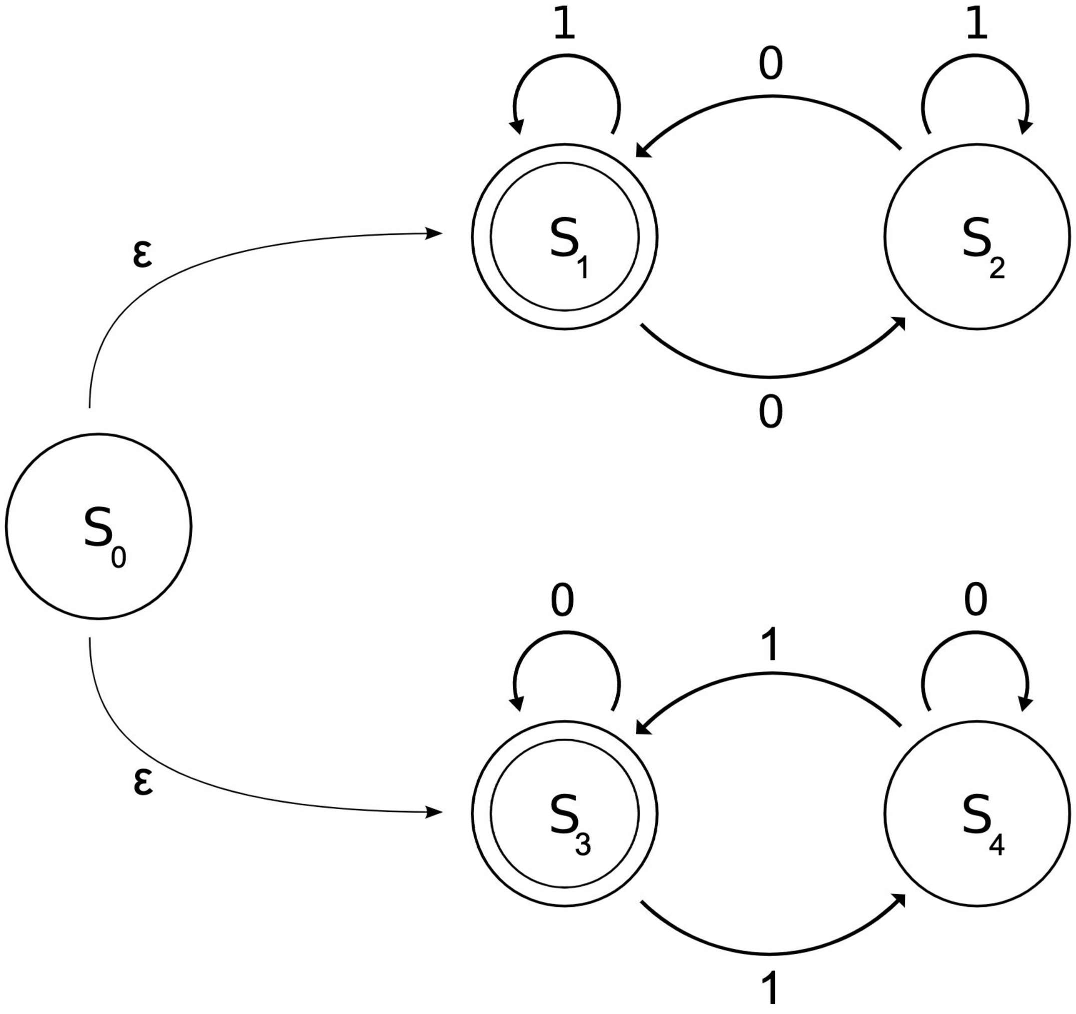
s_0
n
s_1
e
s_2
w
s_3
o
s_4
t
s_5
w
s_6
h
s_7
i
s_8
l
s_9
e
s_{10}

Non-deterministic Finite Automata

L = set of binary strings containing a 1 in the third position from the end


0,1
1
0,1
0,1
q_1
q_2
q_3
q_4

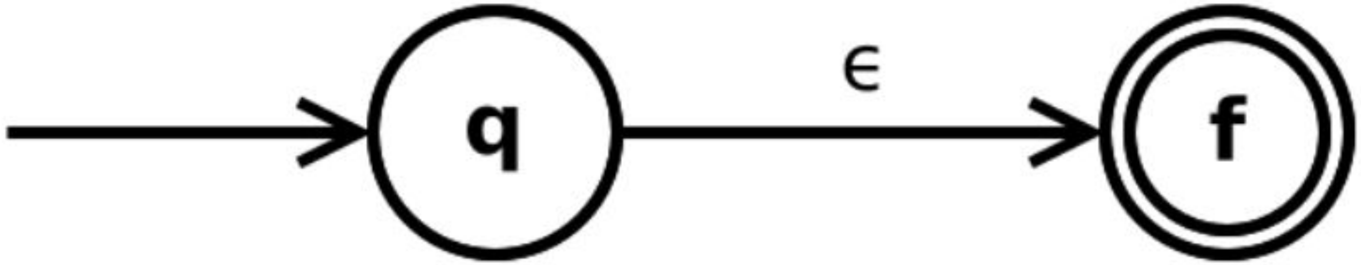
L = set of binary strings with even number of 0s or even number of 1s


S0
ε
ε
S1
S2
S3
S4
1
1
0
0
0
0
1
1

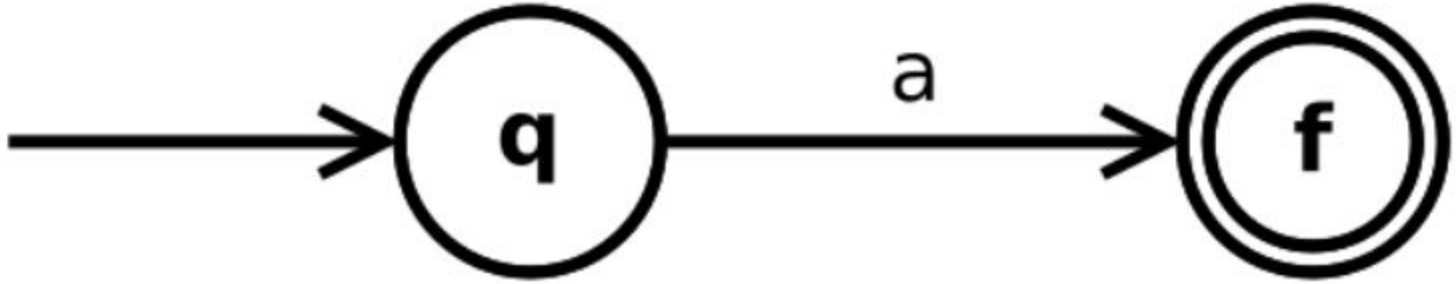
Regex → NFA

Thompson's construction

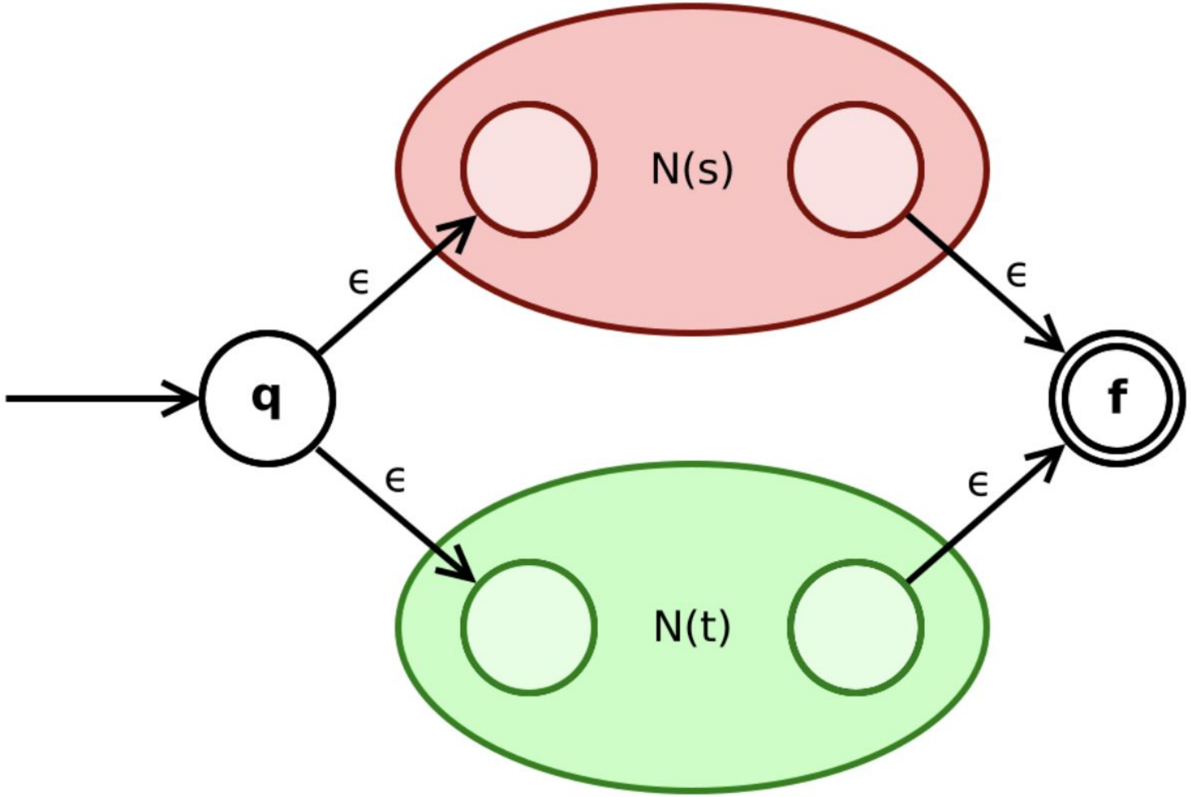
The **empty-expression** ε is converted to

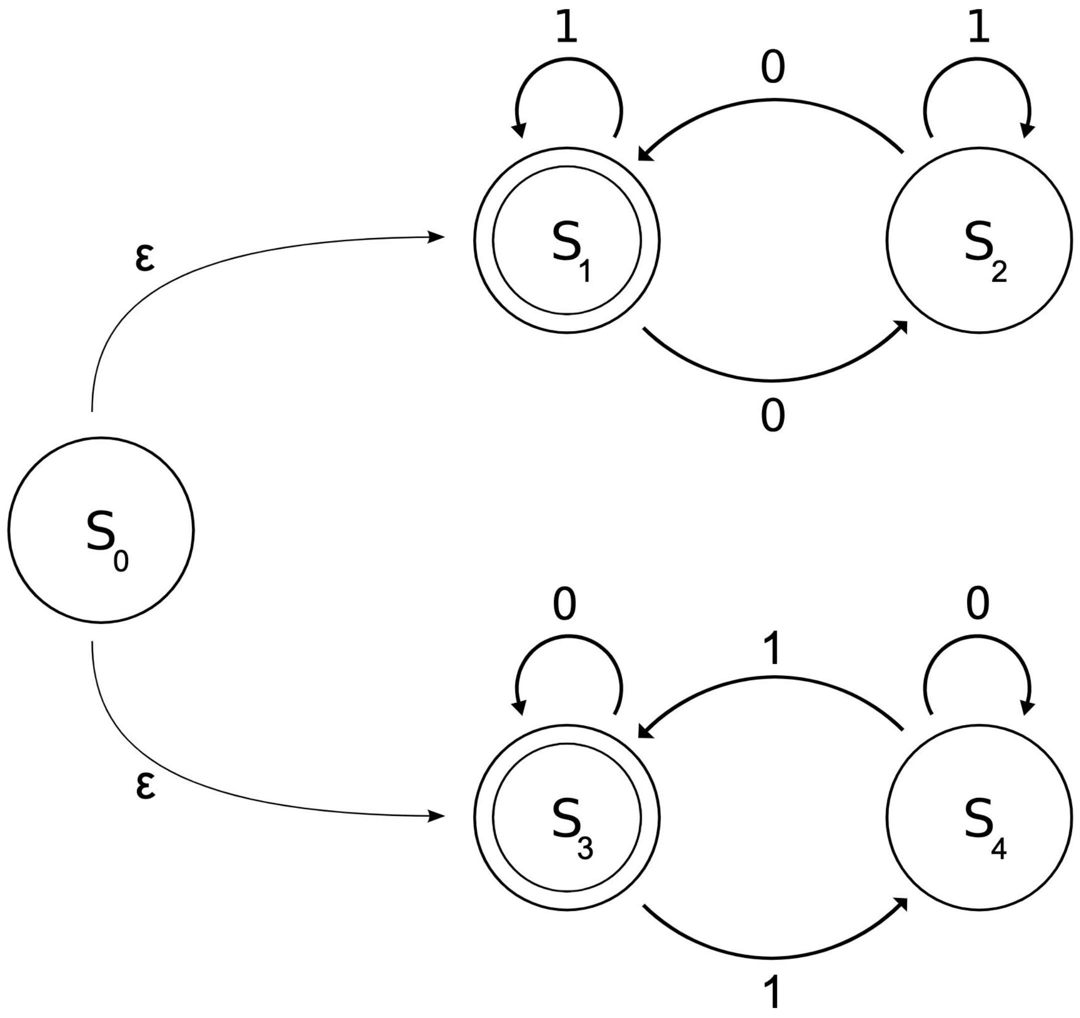


A **symbol** *a* of the input alphabet is converted to

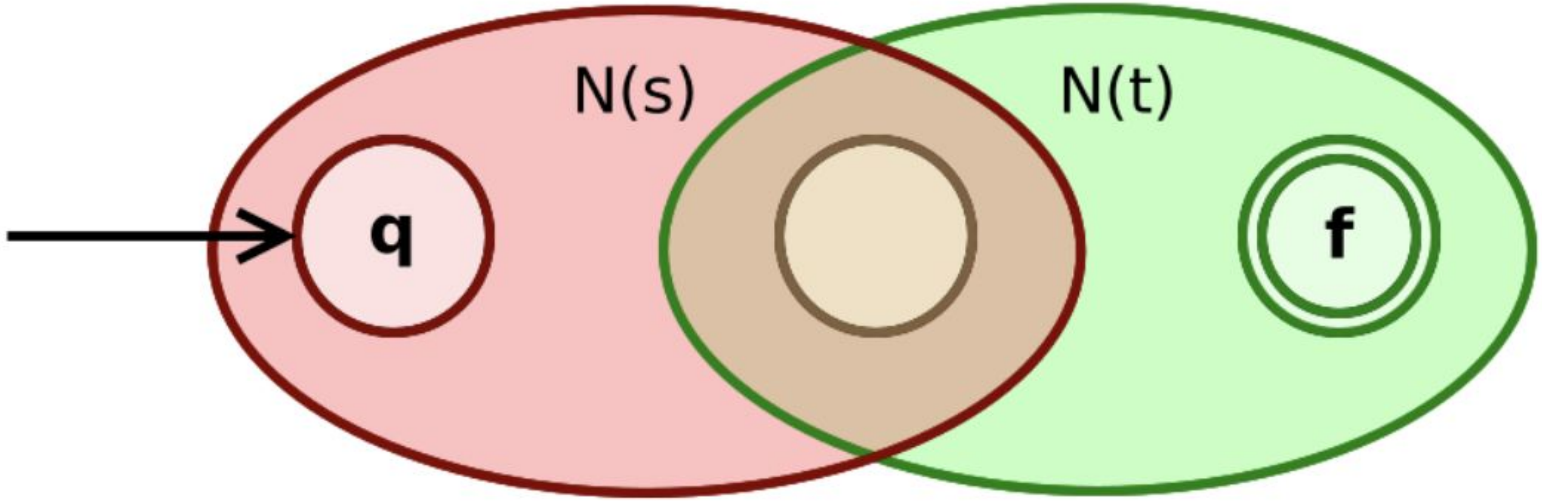


The **union expression** *s*|*t* is converted to

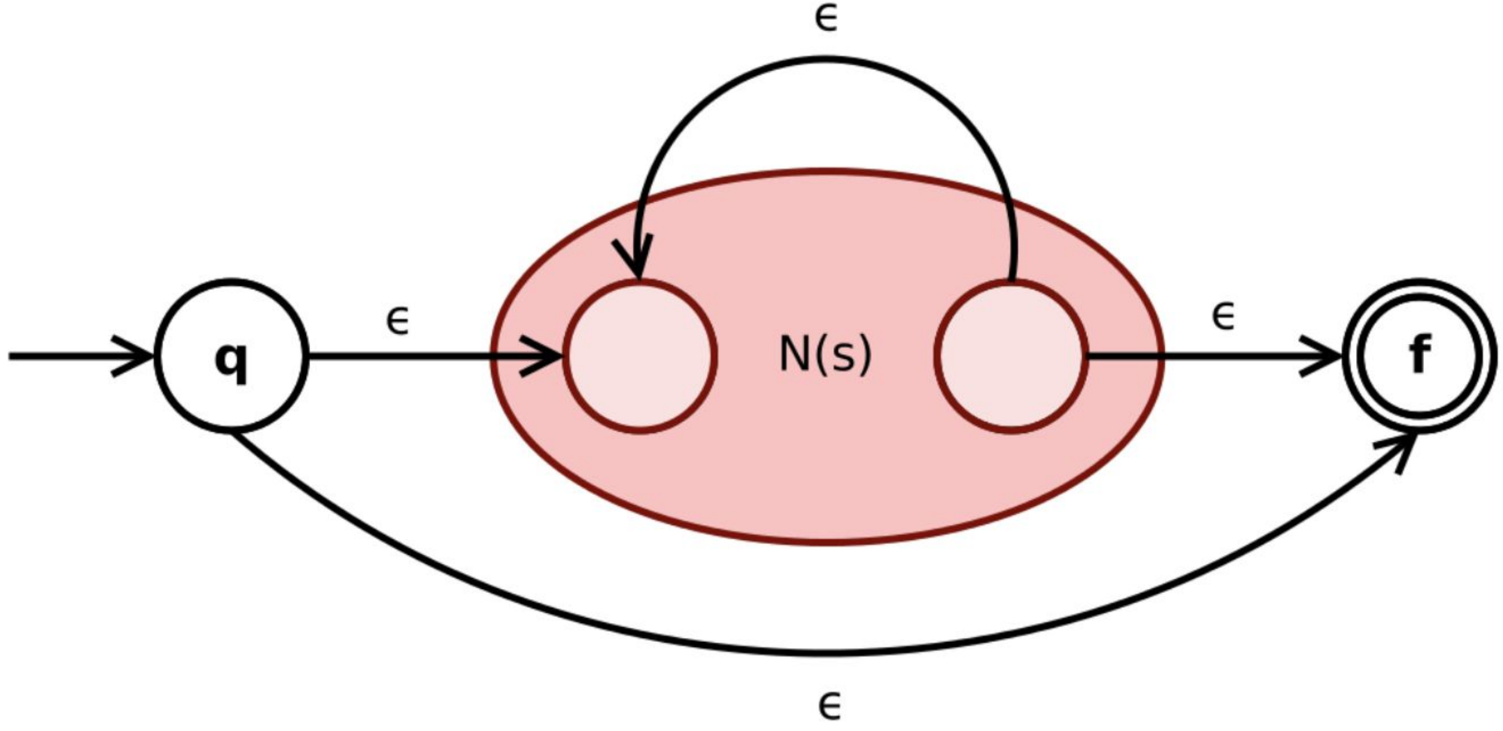



1
1
0
S1
S2
ε
0
S0
0
0
1
S3
S4
ε
1

The **concatenation expression** *st* is converted to

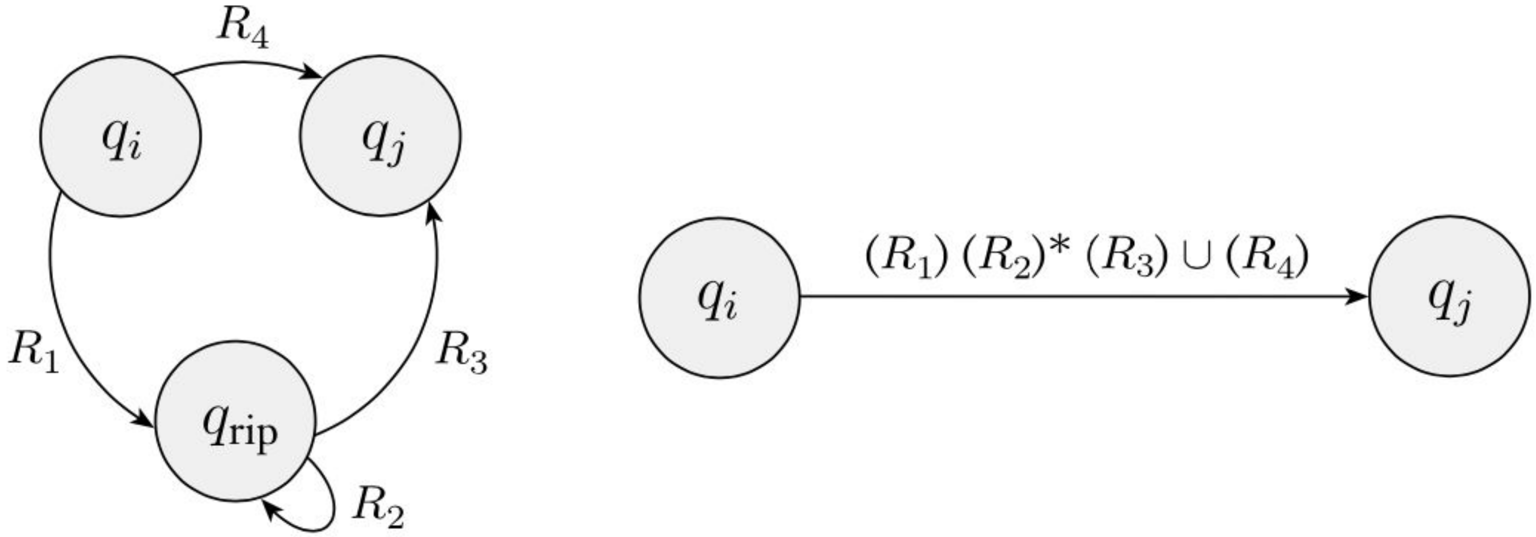


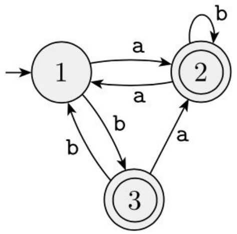
The **Kleene star expression** *s^** is converted to



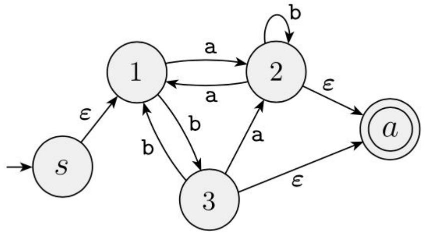
DFA/NFA → Regex

Generalized NFA

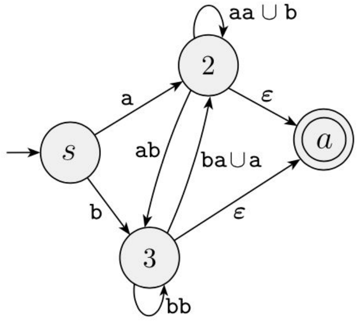

R_4
q_i
q_j
R_1
R_3
q_{rip}
R_2
q_i
$(R_1)\,(R_2)^*\,(R_3) \cup (R_4)$
q_j


1
2
3
a
a
b
b
a
b

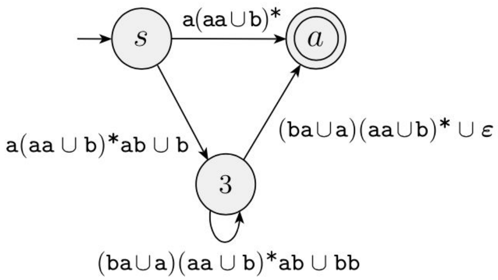
(a)


s
1
2
3
a
ε
a
a
b
ε
b
a
b
ε

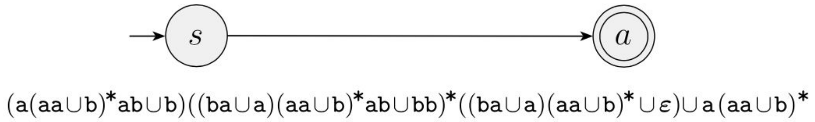
(b)


aa ∪ b
2
a
ε
s
ab
ba∪a
a
b
ε
3
bb

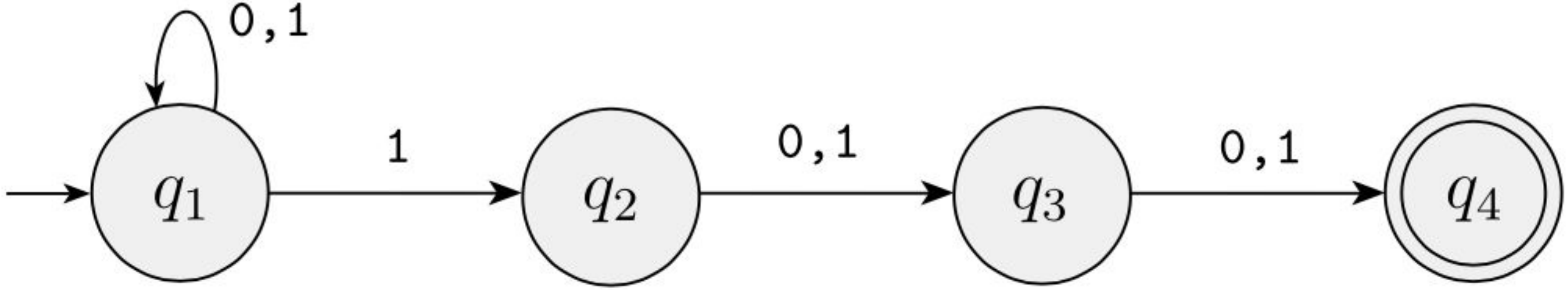
(c)

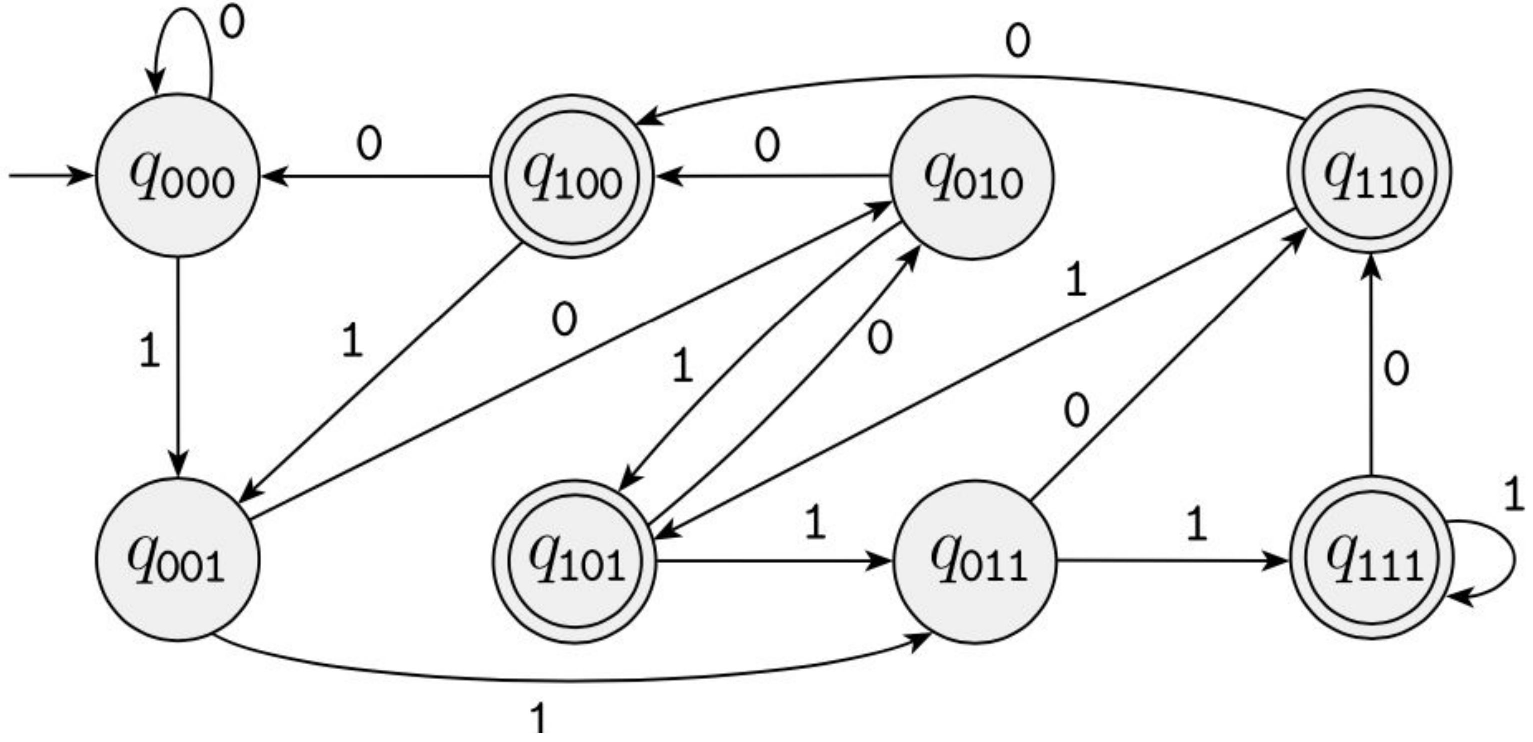

a(aa∪b)*
s
a
(ba∪a)(aa∪b)* ∪ ε
a(aa ∪ b)*ab ∪ b
3
(ba∪a)(aa ∪ b)*ab ∪ bb

(d)


s
a
(a(aa∪b)*ab∪b)((ba∪a)(aa∪b)*ab∪bb)*((ba∪a)(aa∪b)*∪ε)∪a(aa∪b)*

NFA → DFA


0,1
1
0,1
0,1
q_1
q_2
q_3
q_4


q_{000}
q_{100}
q_{010}
q_{110}
q_{001}
q_{101}
q_{011}
q_{111}
0
1

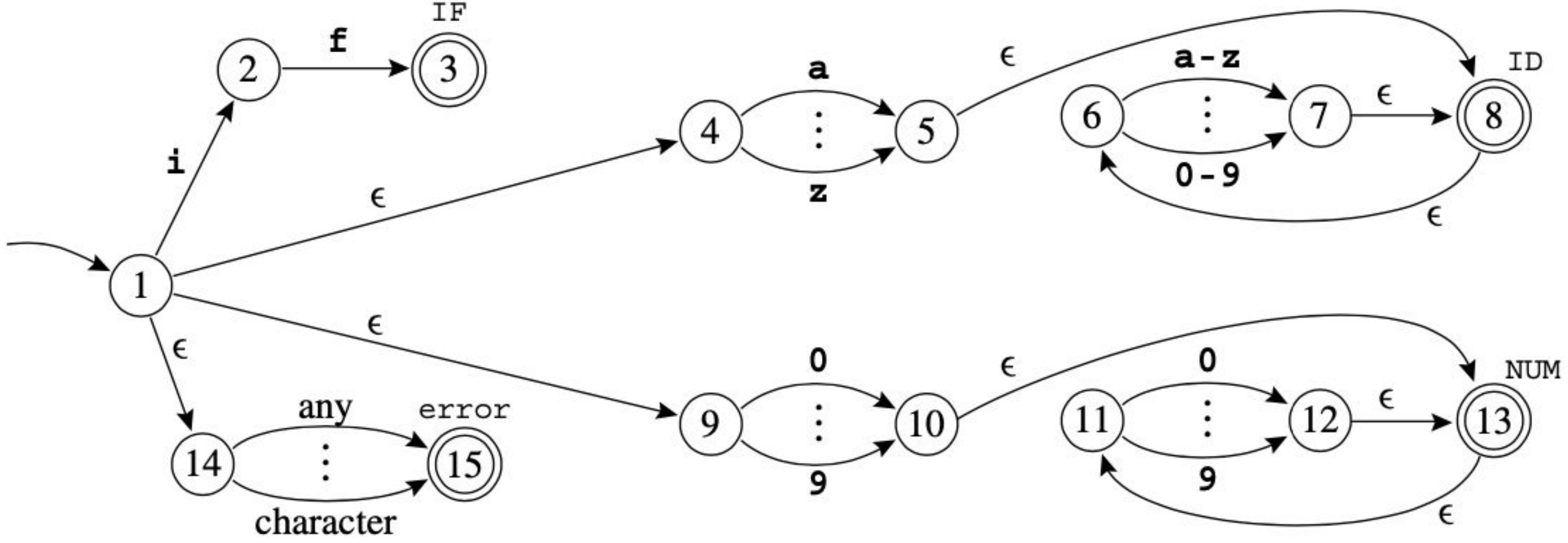
DFA → NFA

Trivial

DFA Minimization

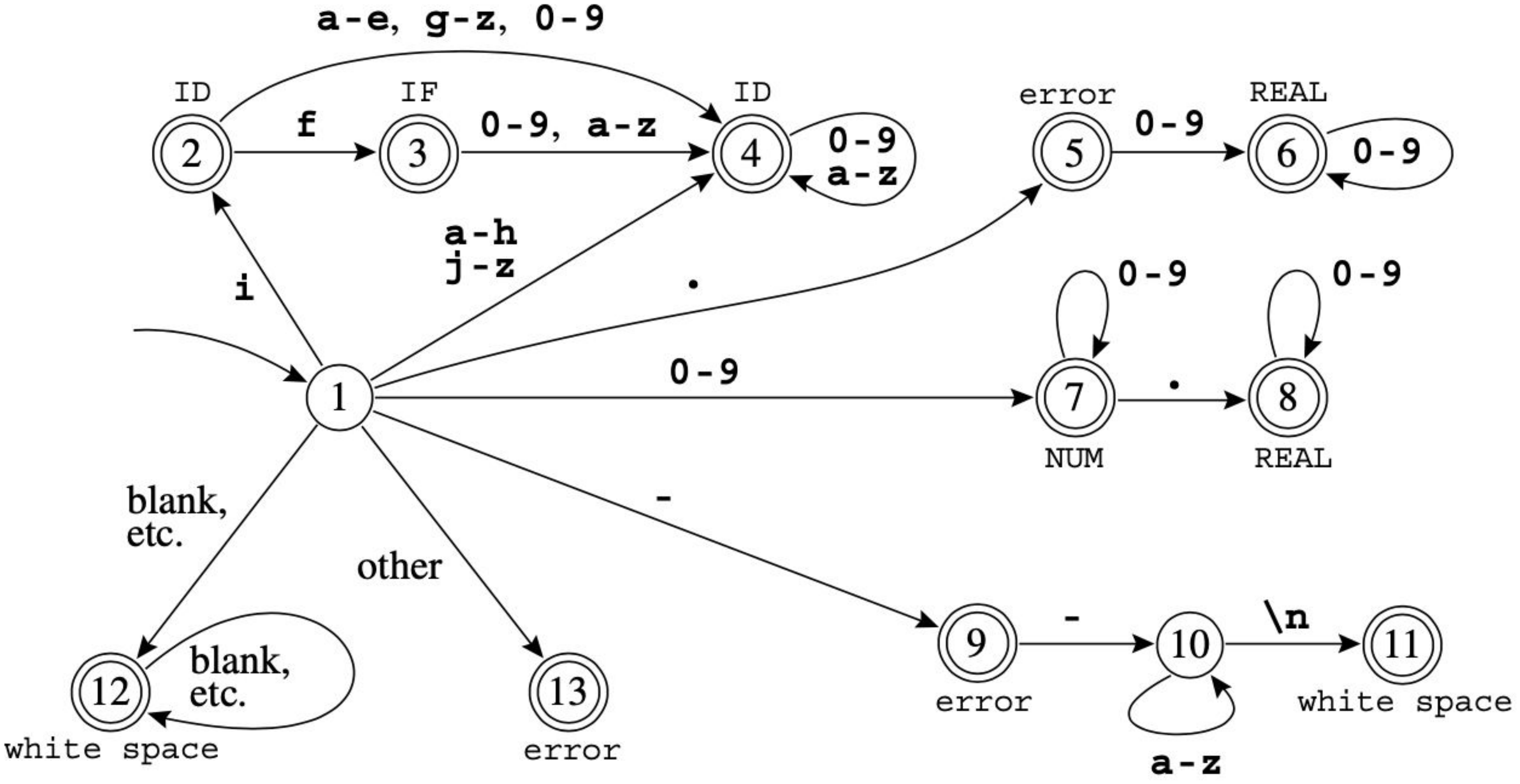
Hard

Why?



IF
f
2
3
i
1
ϵ
4
a
z
5
ϵ
a-z
0-9
6
7
8
ID
ϵ
ϵ
ϵ
14
any
character
15
error
9
0
9
10
ϵ
11
0
9
12
13
NUM
ϵ
ϵ

Greedy



a-e, g-z, 0-9
ID
IF
ID
error
REAL
f
0-9, a-z
0-9
a-z
0-9
0-9
2
3
4
5
6
a-h
j-z
i
.
0-9
0-9
0-9
1
7
.
8
NUM
REAL
blank, etc.
other
-
-
\n
9
10
11
blank, etc.
12
13
white space
error
error
white space
a-z

L = set of binary strings that start with 0s, followed by an equal number of 1s

Pumping lemma

Take 6.045 or 6.840.

36 Answers Sorted by: Highest score (default)

1 2 Next

4407

Locked. There are disputes about this answer's content being resolved at this time. It is not currently accepting new interactions.

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, marriage, and ritual infanticide. The <center> 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 regex* as a tool to process HTML establishes a brea*ch between this world* and the dread realm of c͒orrupt entities (like SGML entities, but *more corrupt) a mere glimp*se of the world of reg**ex parsers for HTML will ins**tantly transport a p*rogrammer's consciousness i*nto a wor*l*d of ceaseless screaming, he comes~~, the pestilent sl~~ithy regex-infection wil**l 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, ~~hi~~**s unholy radiancé dest*roying all enlightenment, HTML tags* ***leaking from your eyes like liq****uid* pain, the song of re*gular expre~~ssion 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* the f`inal snuf`fing *of the lie**s of Man ALL IS LOST ALL IS*** **L**OST the *pony he comes* he c̶omes~~he comes~~ ***the ich***or permeates *al*l MY FACE *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**

For the quiz, you should know how to:

- Simulate regex/DFA/NFA
- Design a regex/DFA/NFA
- Convert regex to NFA
- Convert NFA to DFA

How to practice: Do textbook exercises!

Context-free grammar

$$
\begin{aligned}
\langle \text{SENTENCE} \rangle &\to \langle \text{NOUN-PHRASE} \rangle \langle \text{VERB-PHRASE} \rangle \\
\langle \text{NOUN-PHRASE} \rangle &\to \langle \text{CMPLX-NOUN} \rangle \mid \langle \text{CMPLX-NOUN} \rangle \langle \text{PREP-PHRASE} \rangle \\
\langle \text{VERB-PHRASE} \rangle &\to \langle \text{CMPLX-VERB} \rangle \mid \langle \text{CMPLX-VERB} \rangle \langle \text{PREP-PHRASE} \rangle \\
\langle \text{PREP-PHRASE} \rangle &\to \langle \text{PREP} \rangle \langle \text{CMPLX-NOUN} \rangle \\
\langle \text{CMPLX-NOUN} \rangle &\to \langle \text{ARTICLE} \rangle \langle \text{NOUN} \rangle \\
\langle \text{CMPLX-VERB} \rangle &\to \langle \text{VERB} \rangle \mid \langle \text{VERB} \rangle \langle \text{NOUN-PHRASE} \rangle \\
\langle \text{ARTICLE} \rangle &\to \texttt{a} \mid \texttt{the} \\
\langle \text{NOUN} \rangle &\to \texttt{boy} \mid \texttt{girl} \mid \texttt{flower} \\
\langle \text{VERB} \rangle &\to \texttt{touches} \mid \texttt{likes} \mid \texttt{sees} \\
\langle \text{PREP} \rangle &\to \texttt{with}
\end{aligned}
$$

1	*Expr*	$\rightarrow$	$\underline{(}$ *Expr* $\underline{)}$
2		|	*Expr* *Op* *Expr*
3		|	`name`

4	*Op*	$\rightarrow$	+
5		|	-
6		|	×
7		|	÷

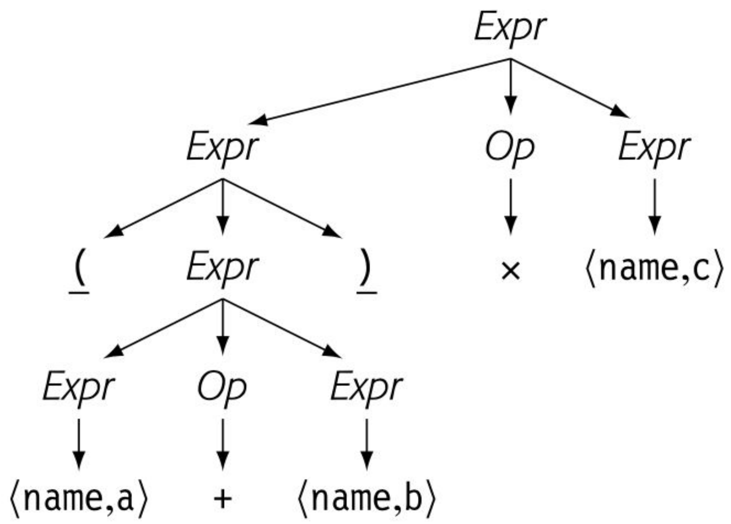
1	*Expr*	→	$\underline{(}$ *Expr* $\underline{)}$
2		|	*Expr* *Op* *Expr*
3		|	`name`

4	*Op*	→	+
5		|	-
6		|	×
7		|	÷

(a + b) x c

1	*Expr*	→	(*Expr*)
2		|	*Expr* *Op* *Expr*
3		|	name

4	*Op*	→	+
5		|	-
6		|	×
7		|	÷



Ambiguity

1	*Stmt*	→	`if` *Expr* `then` *Stmt*
2		|	`if` *Expr* `then` *Stmt* `else` *Stmt*
3		|	*Other*

Left factoring

1	*Stmt*	→	`if` *Expr* `then` *Stmt*
2		|	`if` *Expr* `then` *Stmt* `else` *Stmt*
3		|	*Other*

1	*Stmt*	→	`if` *Expr* `then` *Stmt*
2		|	`if` *Expr* `then` *WithElse* `else` *Stmt*
3		|	*Other*
4	*WithElse*	→	`if` *Expr* `then` *WithElse* `else` *WithElse*
5		|	*Other*

Precedence climbing

1	*Expr*	$\rightarrow$	(*Expr*)
2		|	*Expr* *Op* *Expr*
3		|	`name`

4	*Op*	$\rightarrow$	+
5		|	-
6		|	×
7		|	÷

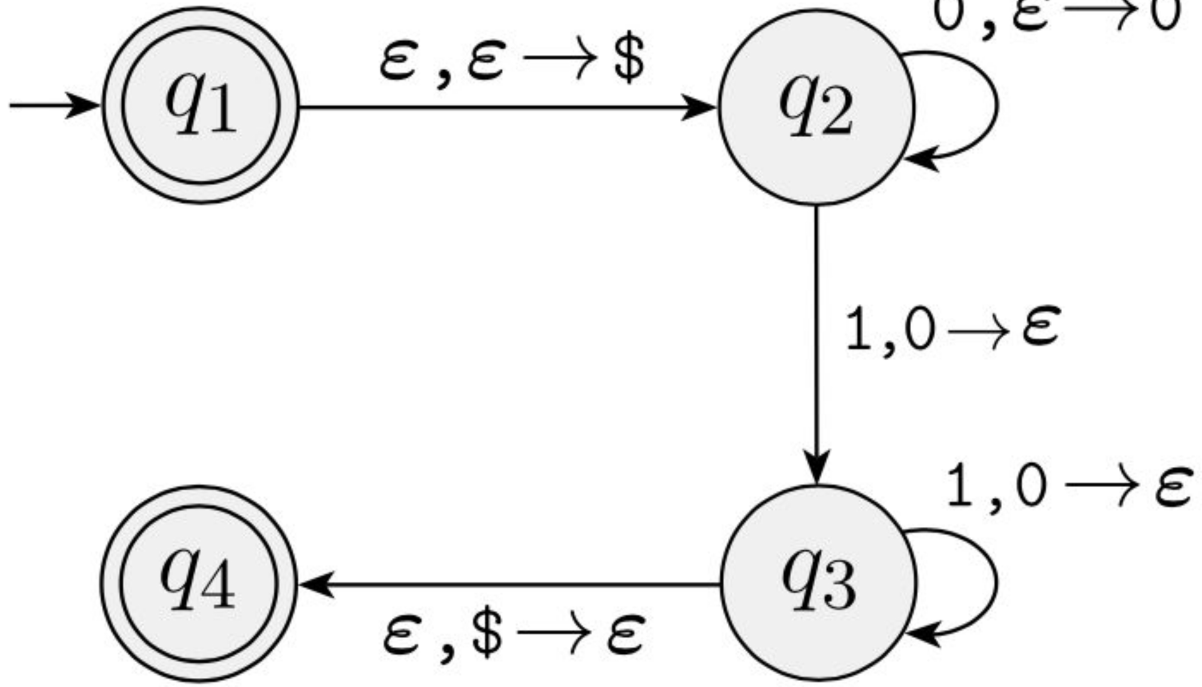
1	*Expr*	→	$\underline{(}$ *Expr* $\underline{)}$
2		|	*Expr* *Op* *Expr*
3		|	`name`

4	*Op*	→	+
5		|	-
6		|	×
7		|	÷

0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	→	*Term* × *Factor*
5		|	*Term* ÷ *Factor*
6		|	*Factor*
7	*Factor*	→	$\underline{(}$ *Expr* $\underline{)}$
8		|	`num`
9		|	`name`

Pushdown Automata

L = set of binary strings that start with 0s, followed by an equal number of 1s


q_1
$\varepsilon, \varepsilon \rightarrow \$$
q_2
$0, \varepsilon \rightarrow 0$
$1, 0 \rightarrow \varepsilon$
q_3
$1, 0 \rightarrow \varepsilon$
$\varepsilon, \$ \rightarrow \varepsilon$
q_4

Remember this?

```
expression ::= resize ('|' resize)*;
resize     ::= primitive ('@' size)*;
size       ::= (number 'x' number);
primitive  ::= filename | '(' expression ')';

topToBottomOperator ::= '---' '-'*;
filename            ::= [A-Za-z0-9.][A-Za-z0-9._-]*;
number              ::= [0-9]+;
whitespace          ::= [ \\t\\r\\n]+;
```

Extended Backus–Naur form*

(* in spirit)

A ::= B* C

```
A   ::= A' C
A'  ::= ε | BA'
```

For the quiz, you should know how to:

- Parse a string using a given grammar (draw parse trees)
- Eliminate ambiguity
- Fix precedence issues
    - Make sure you understand the arithmetic examples.
    - Reminder: You can collaborate/ask for help on miniquiz.

How to practice: Do textbook exercises!

Top-down parsing

Recursive descent parser

<rant>

Use first principles

Ask TAs

</rant>

Project 1!

Left factoring (again)

$$
\begin{array}{lcl}
\mathit{Factor} & \rightarrow & \texttt{name} \\
 & | & \texttt{name}\ \underline{[}\ \mathit{ArgList}\ \underline{]} \\
 & | & \texttt{name}\ \underline{(}\ \mathit{ArgList}\ \underline{)} \\
\mathit{ArgList} & \rightarrow & \mathit{Expr}\ \ \mathit{MoreArgs} \\
\mathit{MoreArgs} & \rightarrow & \texttt{,}\ \ \mathit{Expr}\ \ \mathit{MoreArgs} \\
 & | & \epsilon
\end{array}
$$

$$
\begin{array}{lcl}
\textit{Factor} & \rightarrow & \texttt{name}\ \textit{Arguments} \\
\textit{Arguments} & \rightarrow & \underline{[}\ \textit{ArgList}\ \underline{]} \\
 & | & \underline{(}\ \textit{ArgList}\ \underline{)} \\
 & | & \epsilon \\
\textit{ArgList} & \rightarrow & \textit{Expr}\ \textit{MoreArgs} \\
\textit{MoreArgs} & \rightarrow & \texttt{,}\ \textit{Expr}\ \textit{MoreArgs} \\
 & | & \epsilon
\end{array}
$$

Left recursion

```
Expr   ::= Expr + Term
           | Expr - Term
           | Term;
Term   ::= Term × Factor
           | Term ÷ Factor
           | Factor;
Factor ::= ( Expr )
           | num
           | name;
```

$$\begin{aligned} Fee &\rightarrow Fee\ \alpha \\ &\mid\ \beta \end{aligned}$$

$$\begin{aligned} Fee &\rightarrow \beta\ Fee' \\ Fee' &\rightarrow \alpha\ Fee' \\ &\mid\ \epsilon \end{aligned}$$

```
Expr    ::= Term Expr';
Expr'   ::= + Term Expr'
          | - Term Expr'
          | ε;
Term    ::= Factor Term'
Term'   ::= × Factor Term'
          | ÷ Factor Term'
          | ε;
Factor  ::= ( Expr )
          | num
          | name
```

```
Expr   ::= Term ((+|-) Term)*

Term   ::= Factor ((×|÷) Factor)*

Factor ::= ( Expr )

         | num

         | name;
```

Indirect left recursion

Constraint propagation

$$NT \rightarrow \varepsilon$$
$$\Rightarrow$$
$$NT \rightarrow^{*} \varepsilon$$

$$NT_0 \to NT_1 NT_2 \ldots \text{ and } NT_i \to^* \varepsilon$$
$$\Rightarrow$$
$$NT_0 \to^* \varepsilon$$

$$\mathbf{T} \in \mathrm{First}(\mathbf{T})$$

$$x \in \text{First}(S) \Rightarrow x \in \text{First}(S\ S_1\ S_2\ S_3 \ldots)$$

$$x \in \mathrm{First}(S)$$
$$\Rightarrow$$
$$x \in \mathrm{First}(S\beta)$$

$$\text{First}(S) \subseteq \text{First}(S\beta)$$

$$x \in \text{First}(\beta) \quad \text{and} \quad NT \rightarrow^{*} \varepsilon$$

$$\Rightarrow$$

$$x \in \text{First}(NT\,\beta)$$

$$\mathrm{x} \in \mathrm{First}(\mathrm{S}\beta) \quad \text{and} \quad (NT \rightarrow S\beta)$$
$$\Rightarrow$$
$$\mathrm{x} \in \mathrm{First}(NT)$$