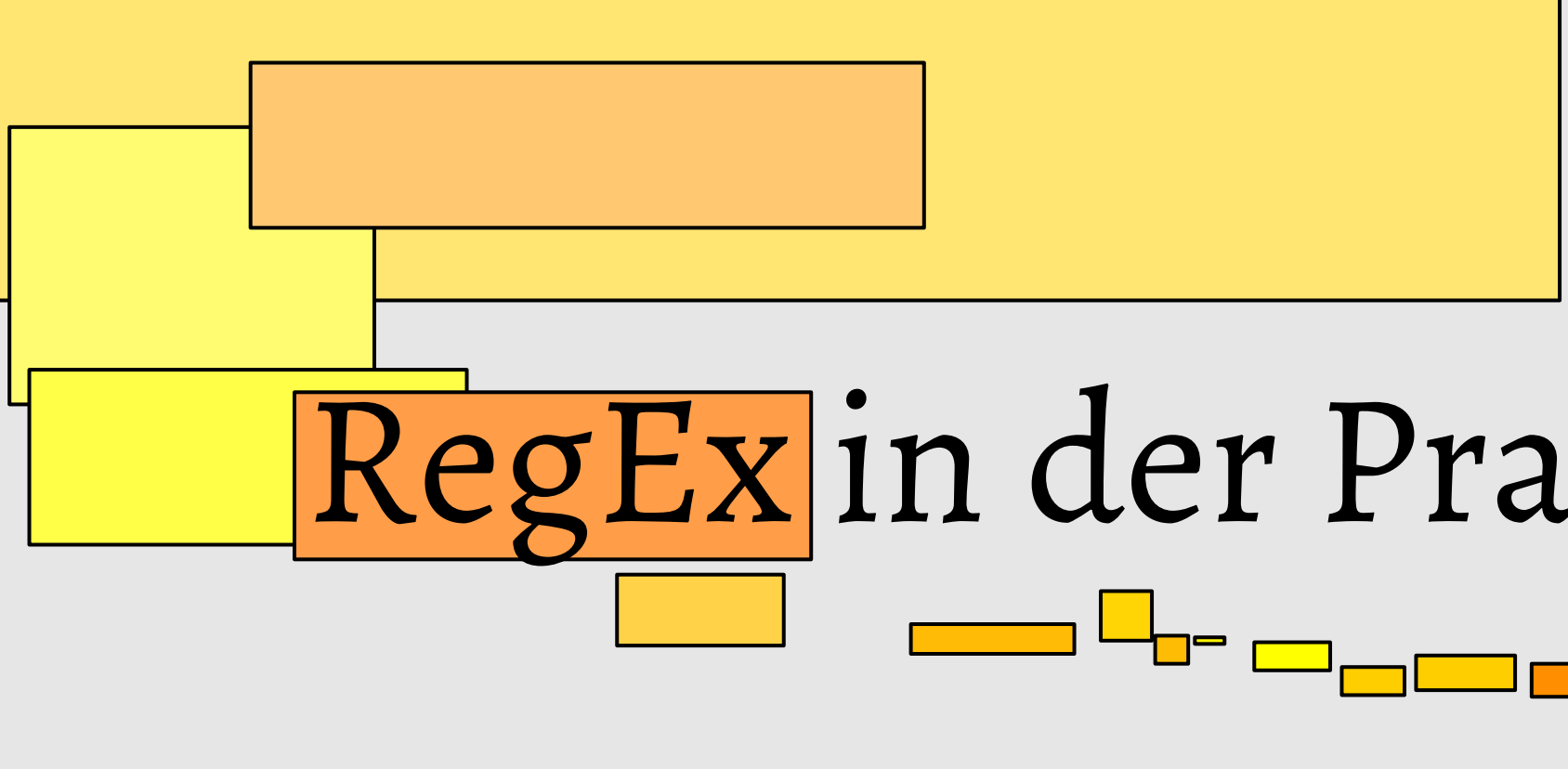


RegEx in der Praxis

Lukas Prokop

BITS Vortragsreihe
27th of Nov 2014

About me

TU Graz student	2010–2016
GDI teaching assistent	2011–2014

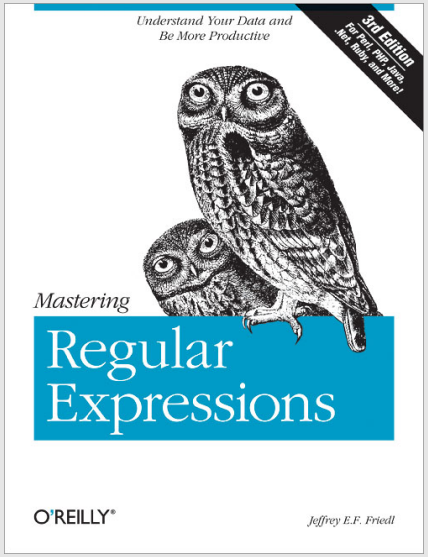
I program a lot.
I ♥ the art of programming.
RegEx is part of it.

Resources

"Mastering Regular Expressions"

O'Reilly, 2nd edition

http://akamaicovers.oreilly.com
/images/9780596528126/lrg.jpg



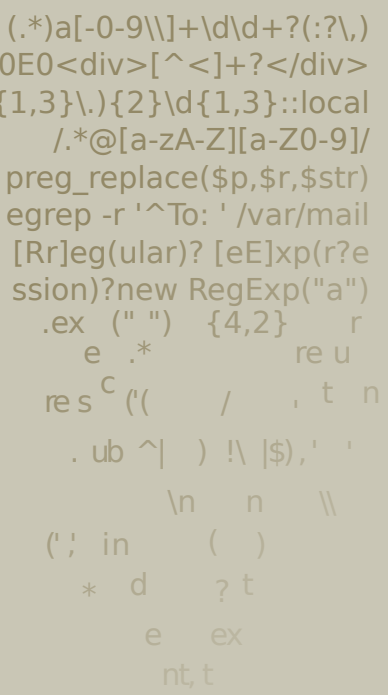
Jeffrey E. F. Friedl

http://regex.info/book.html

classic in the field of
regular expressions

Outline

Outline



RegEx

“Regular Expressions”

“Rational expressions”

abbr. RegEx

abbr. RegExp

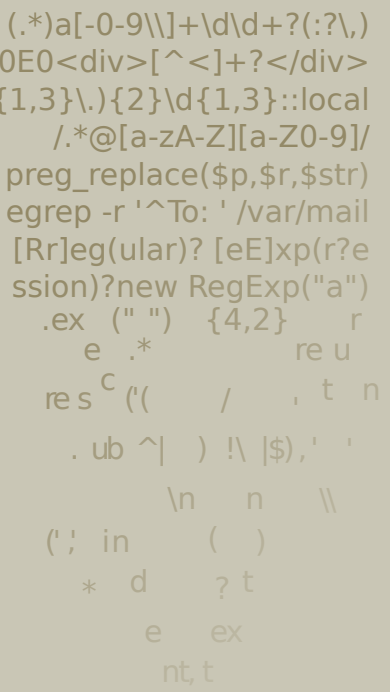
IEEE (“regular expressions”):	1 245 matches
ACM DL (“regular expressions”):	46 381 matches
Google (“regular expressions”):	1 910 000 matches
Google Scholar (“regular expressions”):	2 060 000 matches
Google (“RegEx”):	2 880 000 matches

(.*)a[-0-9\\]+\d\d+?(:?\,)
0E0 [^<]+?
{1,3}\.){2}\d{1,3}::local
/.*@[a-zA-Z][a-Z0-9]/
preg_replace($p,$r,$str)
egrep -r '^To: ' /var/mail
[Rr]eg(ular)? [eE]xp(r?e
ssion)?new RegExp("a")
.ex (" ") {4,2} r
e .* re u
re s c ('(/ , t n
. ub ^|) !\ |$), ' '
\n n \\
(' ; in ( )
* d ? t
e ex
nt, t

RegEx – wording

(.*)a[-0-9\\]+\d\d+?(:?\,)
0E0 [^<]+?
{1,3}\.){2}\d{1,3}::local
/.*@[a-zA-Z][a-Z0-9]/
preg_replace($p,$r,$str)
egrep -r '^To: ' /var/mail
[Rr]eg(ular)? [eE]xp(r?e
ssion)?new RegExp("a")
.ex (" ") {4,2} r
e .* re u
re s c ('(/ , t n
. ub ^|) !\ |$), ' '
\n n \\
(' ; in ( )
* d ? t
e ex
nt, t

RegEx – wording



Goal

- Specify a set of strings
- But name only one

Only works with text. DSL.

(.*)a[-0-9\\]+\d\d+?(:?\,)
0E0 [^<]+?
{1,3}\.){2}\d{1,3}::local
/.*@[a-zA-Z][a-Z0-9]/
preg_replace($p,$r,$str)
egrep -r '^To: ' /var/mail
[Rr]eg(ular)? [eE]xp(r?e
ssion)?new RegExp("a")
.ex (" ") {4,2} r
e .* re u
re s c ('(/ , t n
. ub ^|) !\ |$), ' '
\n n \\
(' ; in ( )
* d ? t
e ex
nt, t

Usecase: Fuzzy string matching

GEDANKEN [from Einstein's term "gedanken-experimenten", such as the standard proof that E=mc2] adj. An AI project which is written up in grand detail without ever being implemented to any great extent. Usually perpetrated by people who aren't very good hackers or find programming distasteful or are just in a hurry. A gedanken thesis is usually marked by an obvious lack of intuition about what is programmable and what is not and about what does and does not constitute a clear specification of a program-related concept such as an algorithm.

— from Hacker's Jargon File

(.*)a[-0-9\\]+\d\d+?(:?\,)
0E0 [^<]+?
{1,3}\.){2}\d{1,3}::local
/.*@[a-zA-Z][a-Z0-9]/
preg_replace($p,$r,$str)
egrep -r '^To: ' /var/mail
[Rr]eg(ular)? [eE]xp(r?e
ssion)?new RegExp("a")
.ex (" ") {4,2} r
e .* re u
re s c ((/ , t n
. ub ^|) !\ |$), ' '
\n n \\
(' ; in ( )
* d ? t
e ex
nt, t

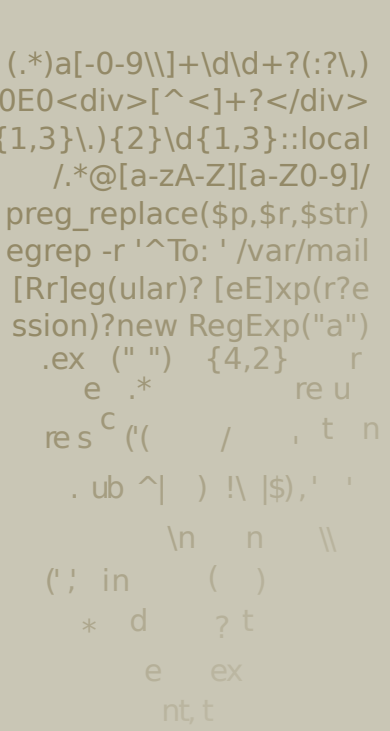
Usecase: Fuzzy string matching

GEDANKEN [from Einstein's term "gedanken-experimenten", such as the standard proof that E=mc2] adj. An AI project which is written up in grand detail without ever being implemented to any great extent. Usually perpetrated by people who aren't very good hackers or find programming distasteful or are just in a hurry. A gedanken thesis is usually marked by an obvious lack of intuition about what is programmable and what is not and about what does and does not constitute a clear specification of a program-related concept such as an algorithm.

— from Hacker's Jargon File

Find all matches of GEDANKEN, Gedanken and gedanken.

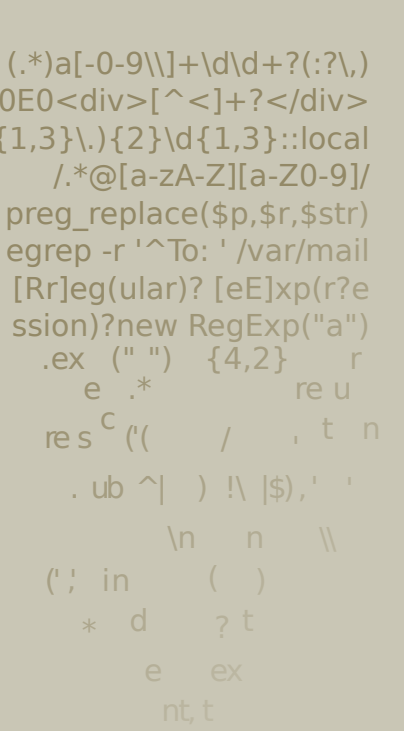
- to find relevant text passage
- to replace occurrences



Usecase: Text extraction

Given

Donald Knuth says, “I define UNIX as 30 definitions of regular expressions living under one roof.”



Usecase: Text extraction

Given

Donald Knuth says, “I define UNIX as 30 definitions of regular expressions living under one roof.”

Extract

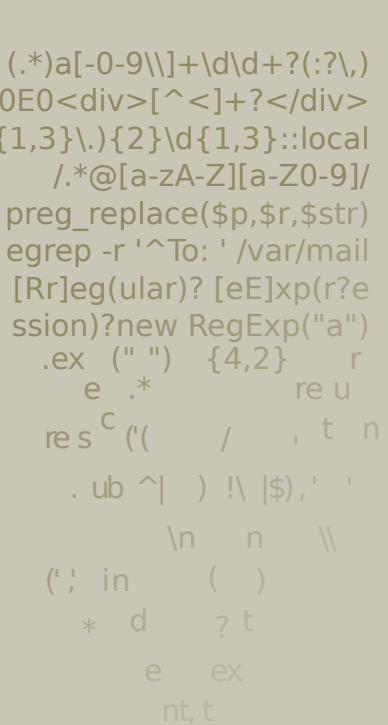
the text between quotation marks

(.*)a[-0-9\\]+\d\d+?(:?\,)
0E0 [^<]+?
{1,3}\.){2}\d{1,3}::local
/.*@[a-zA-Z][a-Z0-9]/
preg_replace($p,$r,$str)
egrep -r '^To: ' /var/mail
[Rr]eg(ular)? [eE]xp(r?e
ssion)?new RegExp("a")
.ex (" ") {4,2} r
e .* re u
re s c ('(/ , t n
. ub ^|) !\ |$), ' '
\n n \\
(' ; in ( )
* d ? t
e ex
nt, t

Usecase: Text splitting

Comma-separated value (CSV):

"103NNN7";"Prokop";"Lukas";"lukas.prokop@stud...at";"11";"50"



Usecase: Text splitting

Comma-separated value (CSV):

"103NNN7";"Prokop";"Lukas";"lukas.prokop@stud...at";"11";"50"

"103NNN7","Prokop","Lukas","lukas.prokop@stud...at","11","50"

(.*)a[-0-9\\]+\d\d+?(:?\,)
0E0 [^<]+?
{1,3}\.){2}\d{1,3}::local
/.*@[a-zA-Z][a-Z0-9]/
preg_replace($p,$r,$str)
egrep -r '^To: ' /var/mail
[Rr]eg(ular)? [eE]xp(r?e
ssion)?new RegExp("a")
.ex (" ") {4,2} r
e .* re u
re s c ('(/ , t n
. ub ^|) !\ |$), ' '
\n n \\
(' ; in ( )
* d ? t
e ex
nt, t

Usecase: Text splitting

Comma-separated value (CSV):

"103NNN7";"Prokop";"Lukas";"lukas.prokop@stud...at";"11";"50"

"103NNN7","Prokop","Lukas","lukas.prokop@stud...at","11","50"

"103NNN7" "Prokop" "Lukas" "lukas.prokop@stud...at" "11" "50"

(.*)a[-0-9\\]+\d\d+?(:?\,)
0E0 [^<]+?
{1,3}\.){2}\d{1,3}::local
/.*@[a-zA-Z][a-Z0-9]/
preg_replace($p,$r,$str)
egrep -r '^To: ' /var/mail
[Rr]eg(ular)? [eE]xp(r?e
ssion)?new RegExp("a")
.ex (" ") {4,2} r
e .* re u
re s c ('(/ , t n
. ub ^|) !\ |$), ' '
\n n \\
(' ; in ( )
* d ? t
e ex
nt, t

Usecase: Text splitting

Comma-separated value (CSV):

"103NNN7";"Prokop";"Lukas";"lukas.prokop@stud...at";"11";"50"

"103NNN7","Prokop","Lukas","lukas.prokop@stud...at","11","50"

"103NNN7" "Prokop" "Lukas" "lukas.prokop@stud...at" "11" "50"

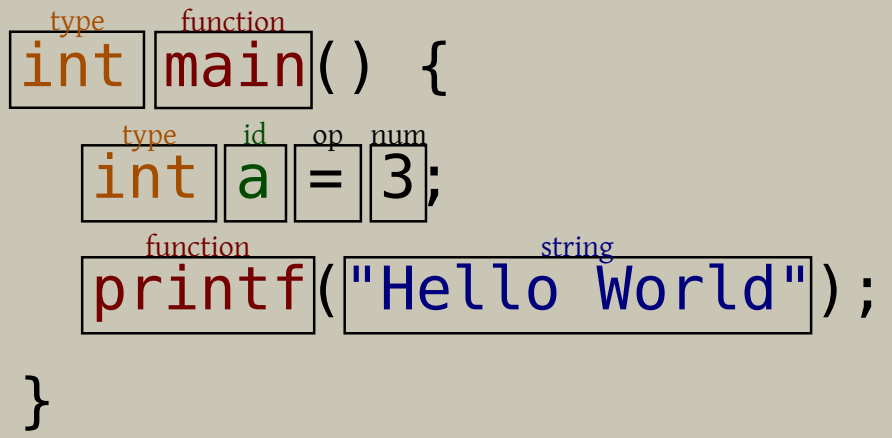
Accept various delimiters.

(.*)a[-0-9\\]+\d\d+?(:?\,)
0E0 [^<]+?
{1,3}\.){2}\d{1,3}::local
/.*@[a-zA-Z][a-Z0-9]/
preg_replace($p,$r,$str)
egrep -r '^To: ' /var/mail
[Rr]eg(ular)? [eE]xp(r?e
ssion)?new RegExp("a")
.ex (" ") {4,2} r
e .* re u
re s c ('(/ , t n
. ub ^|) !\ |$), ' '
\n n \\
(' ; in ( )
* d ? t
e ex
nt, t

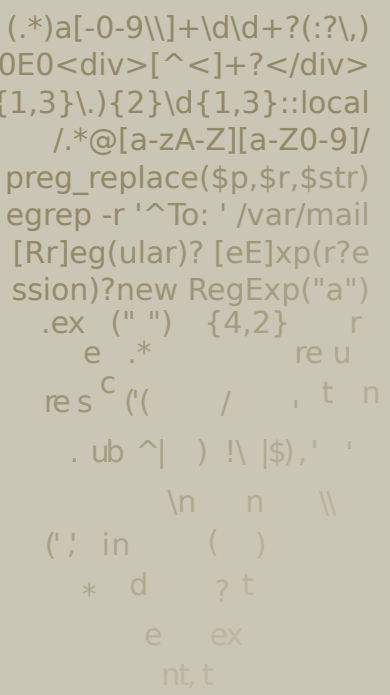
Usecase: Lexing in compilers

```
int main() {
  int a = 3;
  printf("Hello World");
}
```

Usecase: Lexing in compilers

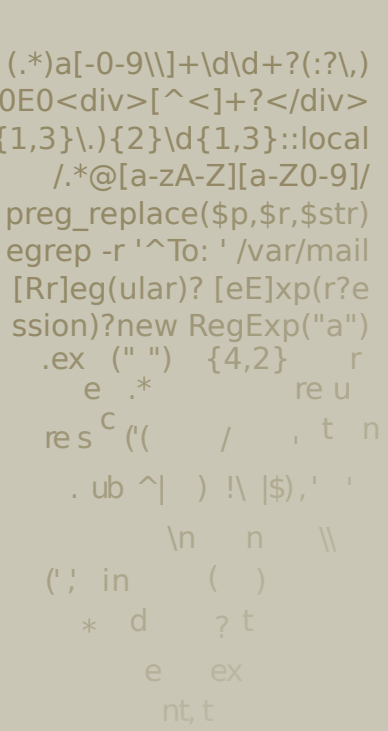


string ⇒ parameterized token stream



Usecases

- Fuzzy string matching
- Text extraction
- Text splitting
- Lexing in compilers



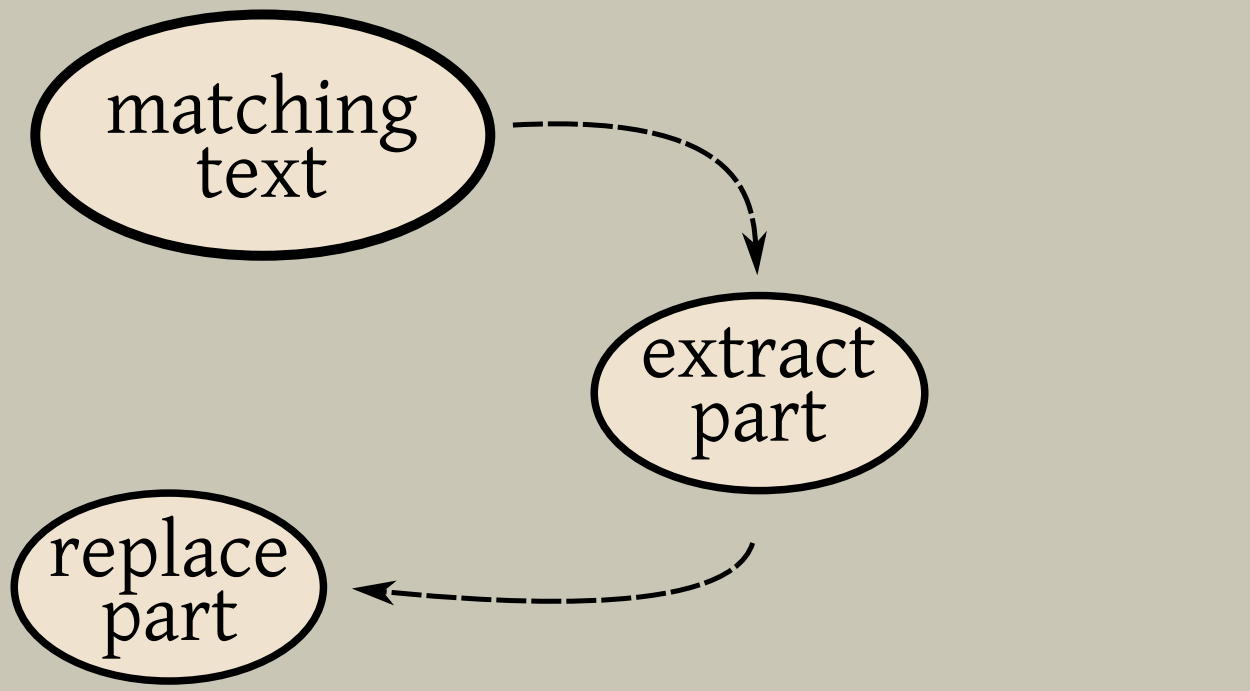
Usecases

- Fuzzy string matching
- Text extraction
- Text splitting
- Lexing in compilers

Remark: Regular expressions are powerful.
But you cannot specify *any* set of string.

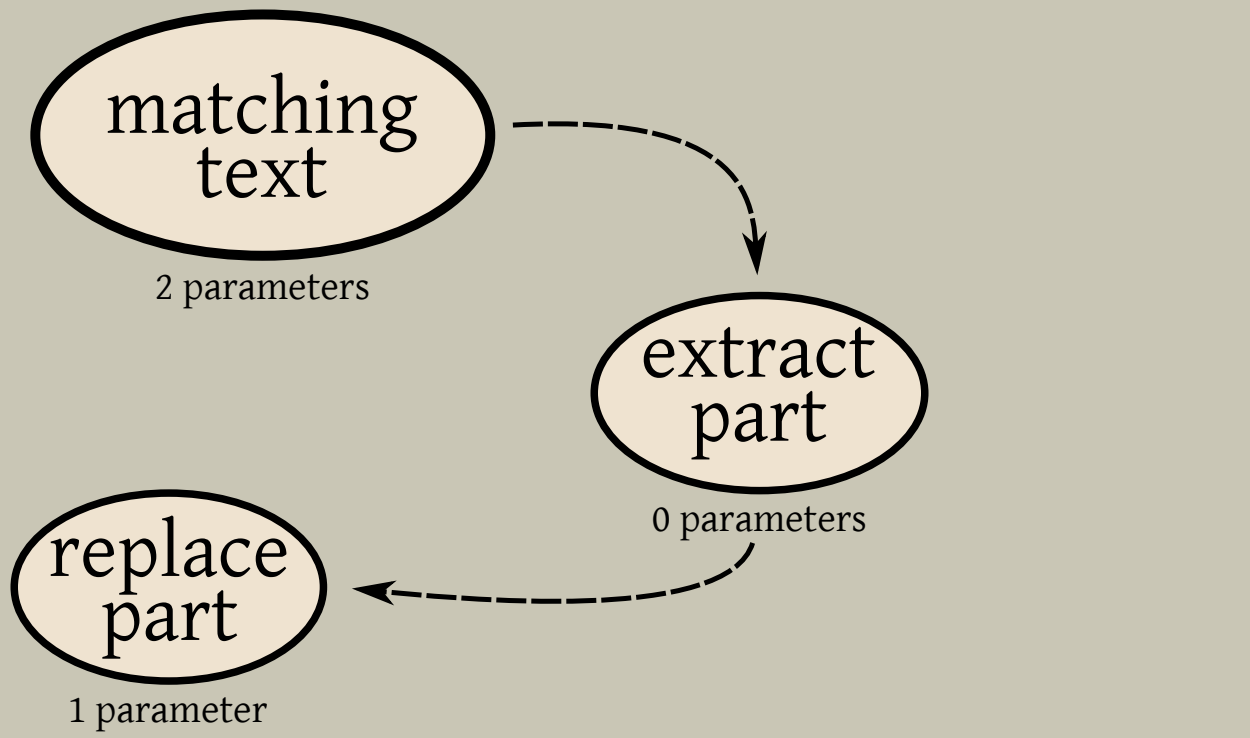
(.*)a[-0-9\\]+\d\d+?(:?\,)
0E0 [^<]+?
{1,3}\.){2}\d{1,3}::local
/.*@[a-zA-Z][a-Z0-9]/
preg_replace($p,$r,$str)
egrep -r '^To: ' /var/mail
[Rr]eg(ular)? [eE]xp(r?e
ssion)?new RegExp("a")
.ex (" ") {4,2} r
e .* re u
re s c ('(/ , t n
. ub ^|) !\ |$), ' '
\n n \\
(' ; in ( )
* d ? t
e ex
nt, t

Boils down to?



(.*)a[-0-9\\]+\d\d+?(:?\,)
0E0 [^<]+?
{1,3}\.){2}\d{1,3}::local
/.*@[a-zA-Z][a-Z0-9]/
preg_replace($p,$r,$str)
egrep -r '^To: ' /var/mail
[Rr]eg(ular)? [eE]xp(r?e
ssion)?new RegExp("a")
.ex (" ") {4,2} r
e .* re u
re s c ('(/ , t n
. ub ^|) !\ |$), ' '
\n n \\
(' ; in ( )
* d ? t
e ex
nt, t

Boils down to?



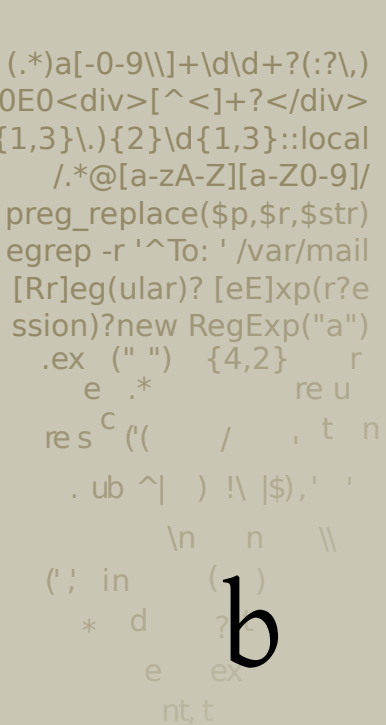
Outline

History

1956	Stephen Kleene (automata theory, TCS)
197_	UNIX guys [ken, dmr] (sed, awk, ...)
1986	Henry Spencer's regex library
1987	Perl
1991	Unicode 1.0.0
1997	PCRE library
today	native programming language support, derivatives with common core

Outline

Outline



Concatenation

b

- ☒ abcd
- ☒ acb
- ☒ abc
- ☒ ab
- ☑ b

ab

- ☒ abcd
- ☒ acb
- ☒ abc
- ☑ ab
- ☒ b

abc

- ☒ abcd
- ☒ acb
- ☑ abc
- ☒ ab
- ☒ b

Alternation

a|b

- [x] a
- [x] b
- [ ] c
- [ ] bc
- [ ] abc
- [ ]

Alternation

a|b

- ☑ a
- ☑ b
- ☒ c
- ☒ bc
- ☒ abc
- ☒

a|b|c

- ☑ a
- ☑ b
- ☑ c
- ☒ bc
- ☒ abc
- ☒

Alternation

a|b

- ☑ a
- ☑ b
- ☒ c
- ☒ bc
- ☒ abc
- ☒

a|b|c

- ☑ a
- ☑ b
- ☑ c
- ☒ bc
- ☒ abc
- ☒

a|b|

- ☑ a
- ☑ b
- ☒ c
- ☒ bc
- ☒ abc
- ☑

Quantifiers

a?

- ☑
- ☑ a
- ☒ aa
- ☒ aaa
- ☒ ab

Quantifiers

a?

- ☑
- ☑ a
- ☒ aa
- ☒ aaa
- ☒ ab

a+

- ☒
- ☑ a
- ☑ aa
- ☑ aaa
- ☒ ab

Quantifiers

a?	a+	a*
☑	☒	☑
☑ a	☑ a	☑ a
☒ aa	☑ aa	☑ aa
☒ aaa	☑ aaa	☑ aaa
☒ ab	☒ ab	☒ ab

Quantifiers

a{1,2}

- [ ]
- [x] a
- [x] aa
- [ ] aaa
- [ ] aaaa

Quantifiers

a{1,2}

- ☒
- ☑ a
- ☑ aa
- ☒ aaa
- ☒ aaaa

a{,2}

- ☑
- ☑ a
- ☑ aa
- ☒ aaa
- ☒ aaaa

Quantifiers

a{1,2}

- ☒
- ☑ a
- ☑ aa
- ☒ aaa
- ☒ aaaa

a{,2}

- ☑
- ☑ a
- ☑ aa
- ☒ aaa
- ☒ aaaa

a{1,}

- ☒
- ☑ a
- ☑ aa
- ☑ aaa
- ☑ aaaa

Quantifiers

a{2}

- [x]
- [x] a
- [x] aa
- [x] aaa
- [x] aaaa

Dot, character lists

.

matches one
arbitrary symbol

- ☒
- ☑ a
- ☑ b
- ☑ c
- ☒ ab

Dot, character lists

in general excludes newlines

.

matches one arbitrary symbol

- [x]
- [x] a
- [x] b
- [x] c
- [x] ab

Dot, character lists

.

matches one
arbitrary symbol

- ☒
- ☑ a
- ☑ b
- ☑ c
- ☒ ab

[abc]

matches one
of *a*, *b* or *c*

- ☒
- ☑ a
- ☑ b
- ☑ c
- ☒ ab

Dot, character lists

.

matches one arbitrary symbol

- ☒
- ☑ a
- ☑ b
- ☑ c
- ☒ ab

[abc]

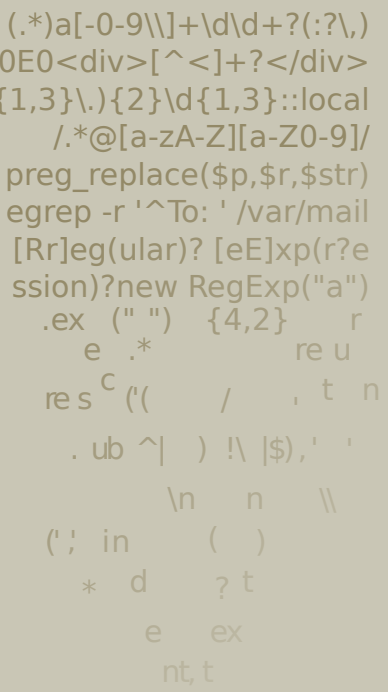
matches one of *a*, *b* or *c*

- ☒
- ☑ a
- ☑ b
- ☑ c
- ☒ ab

[a-d]

matches a range *a* to *d*

- ☑ a
- ☑ b
- ☑ c
- ☑ d
- ☒ e



What if I told you ...

... that lists are a DSL on their own?

(.*)a[-0-9\\]+\d\d+?(:?\,)
0E0 [^<]+?
{1,3}\.){2}\d{1,3}::local
/.*@[a-zA-Z][a-Z0-9]/
preg_replace($p,$r,$str)
egrep -r '^To: ' /var/mail
[Rr]eg(ular)? [eE]xp(r?e
ssion)?new RegExp("a")
.ex (" ") {4,2} r
e .* re u
re s c ('(/ , t n
. ub ^|) !\ |$), ' '
\n n \\
(' ; in ( )
* d ? t
e ex
nt, t

What if I told you ...

... that lists are a DSL on their own?

(.*)a[-0-9\\]+\d\d+?(:?\,)
0E0 [^<]+?
{1,3}\.){2}\d{1,3}::local
/.*@[a-zA-Z][a-Z0-9]/
preg_replace($p,$r,$str)
egrep -r '^To: ' /var/mail
[Rr]eg(ular)? [eE]xp(r?e
ssion)?new RegExp("a")
.ex (" ") {4,2} r
e .* re u
re s c ('(/ , t n
. ub ^|) !\ |$), ' '
\n n \\
(' ; in ( )
* d t
e ex
nt, t

Character lists

... considered harmful

[abc]

matches one
of *a*, *b* or *c*

- ☒
- ☑ a
- ☑ b
- ☑ c
- ☒ d

Character lists

... considered harmful

[abc]

matches one of *a*, *b* or *c*

- ☒
- ☑ a
- ☑ b
- ☑ c
- ☒ d

[^abc]

matches one character else than *a*, *b* or *c*

- ☒
- ☒ a
- ☒ b
- ☒ c
- ☑ d

Character lists

... considered harmful

[abc]

matches one of *a*, *b* or *c*

- ☒
- ☑ a
- ☑ b
- ☑ c
- ☒ d

[^abc]

matches one character else than *a*, *b* or *c*

- ☒
- ☒ a
- ☒ b
- ☒ c
- ☑ d

[-ac]

matches one of *a*, *c* or -

- ☒
- ☑ a
- ☑ c
- ☑ -
- ☒ b

Escaping

\[a\.bc?\]

matches the string with only question mark as a meta character

- ☒ a
- ☑ [a.bc]
- ☑ [a.b]
- ☒ [a_b]
- ☒ \[a\.bc\]

backslash as universal escape character

(in all regex grammars I know)

[0-9.]

matches one digit or a dot character

- ☒
- ☑ 3
- ☑ .
- ☒ -
- ☒ b

Shorthand lists

\d

[0-9]

- [ ]
- [x] 0
- [x] 6
- [ ] 9c
- [ ] 42

Shorthand lists

\d

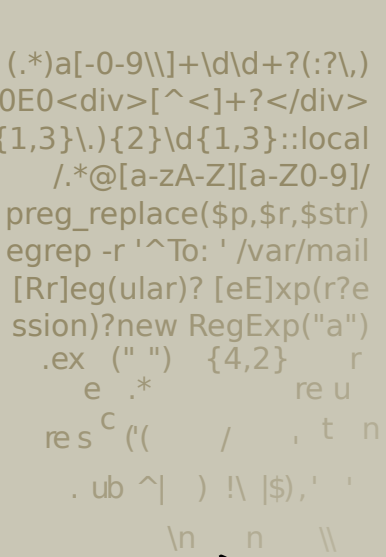
[0-9]

- ☒
- ☑ 0
- ☑ 6
- ☒ 9c
- ☒ 42

\w

[A-Za-z0-9_]

- ☒
- ☑ a
- ☑ b
- ☑ c
- ☒ -



Shorthand lists

\d

[0-9]

- ☒
- ☑ 0
- ☑ 6
- ☒ 9c
- ☒ 42

\w

[A-Za-z0-9_]

- ☒
- ☑ a
- ☑ b
- ☑ c
- ☒ -

\s

one of 25 whitespace characters

- ☒
- ☑ *(tab)*
- ☑ *(newline)*
- ☑ *(space)*
- ☒ _

(.*)a[-0-9\\]+\d\d+?(:?\,)
0E0 [^<]+?
{1,3}\.){2}\d{1,3}::local
/.*@[a-zA-Z][a-Z0-9]/
preg_replace($p,$r,$str)
egrep -r '^To: ' /var/mail
[Rr]eg(ular)? [eE]xp(r?e
ssion)?new RegExp("a")
.ex (" ") {4,2} r
e .* re u
re s c ('(/ , t n
. ub ^|) !\ |$), ' '
\n n \\
(' ; in ( )
* d ? t
e ex
nt, t

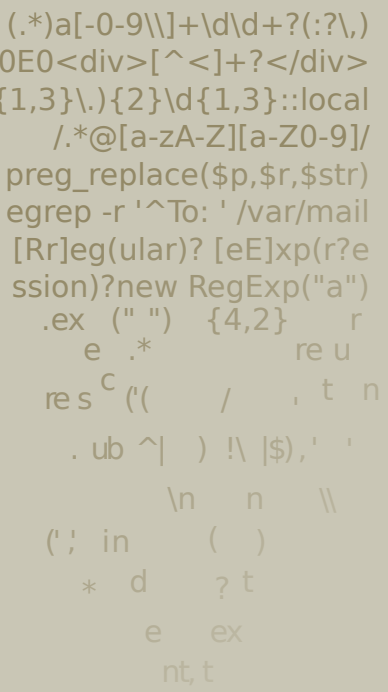
Anchors

... first operators which do not consume anything
... zero-length matches

^bits

starts with

- ☒ a bitsequence
- ☑ bitsequence
- ☑ bits



Anchors

... first operators which do not consume anything
... zero-length matches

^bits

starts with

- ☒ a bitsequence
- ☑ bitsequence
- ☑ bits

bits$

ends with

- ☒ bitsequence
- ☑ habits
- ☑ bits

Grouping

(a|c)

- ☒ abc
- ☒ ab
- ☒ ac
- ☑ c
- ☑ a
- ☒

(ab)(c)

- ☑ abc
- ☒ ab
- ☒ ac
- ☒ c
- ☒ a
- ☒

(ac)?

- ☒ abc
- ☒ ab
- ☑ ac
- ☒ c
- ☒ a
- ☑

Grouping

(a|c)

1

- ☒ abc
- ☒ ab
- ☒ ac
- ☑ c
- ☑ a
- ☒

(ab)(c)

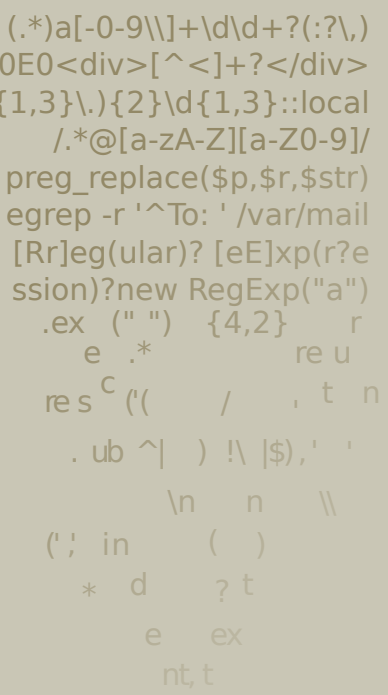
1 2

- ☑ abc
- ☒ ab
- ☒ ac
- ☒ c
- ☒ a
- ☒

(ac)?

1

- ☒ abc
- ☒ ab
- ☑ ac
- ☒ c
- ☒ a
- ☑

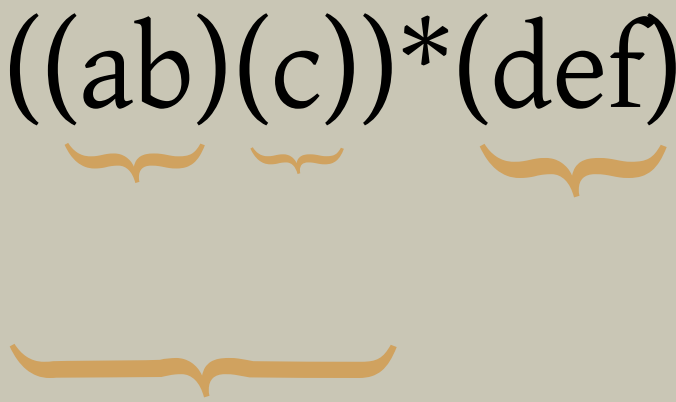


Grouping order

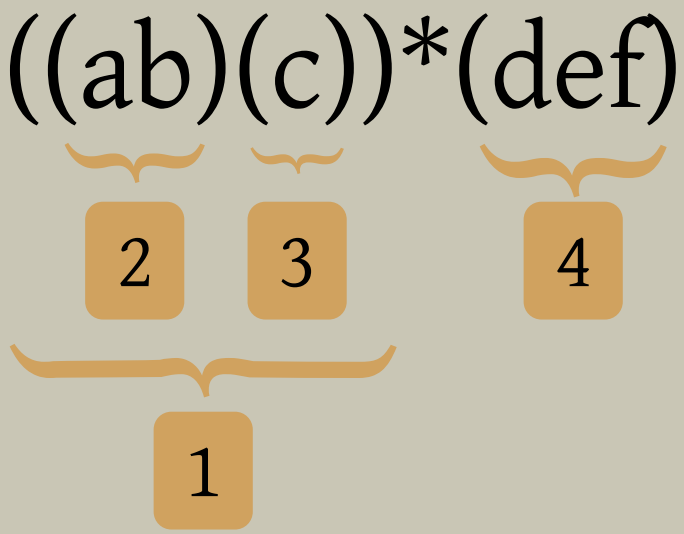
((ab)(c))*(def)

(.*)a[-0-9\\]+\d\d+?(:?\,)
0E0 [^<]+?
{1,3}\.){2}\d{1,3}::local
/.*@[a-zA-Z][a-Z0-9]/
preg_replace($p,$r,$str)
egrep -r '^To: ' /var/mail
[Rr]eg(ular)? [eE]xp(r?e
ssion)?new RegExp("a")
.ex (" ") {4,2} r
e .* re u
re s c ('(/ , t n
. ub ^|) !\ |$), ' '
\n n \\
(' ; in ( )
* d ? t
e ex
nt, t

Grouping order



Grouping order



(.*)a[-0-9\\]+\d\d+?(:?\,)
0E0 [^<]+?
{1,3}\.){2}\d{1,3}::local
/.*@[a-zA-Z][a-Z0-9]/
preg_replace($p,$r,$str)
egrep -r '^To: ' /var/mail
[Rr]eg(ular)? [eE]xp(r?e
ssion)?new RegExp("a")
.ex (" ") {4,2} r
e .* re u
re s c ('(/ , t n
. ub ^|) !\ |$), ' '
\n n \\
(' ; in ( )
* d ? t
e ex
nt, t

Groups as scope

Important!

Quantifiers always apply to the last regex element.

abcd?ef

Quantifier applies only to “d”; the last letter.

a(bcd)?ef

Quantifier applies to “bcd” group.

Outline

(.*)a[-0-9\\]+\d\d+?(:?\,)
0E0 [^<]+?
{1,3}\.){2}\d{1,3}::local
/.*@[a-zA-Z][a-Z0-9]/
preg_replace($p,$r,$str)
egrep -r '^To: ' /var/mail
[Rr]eg(ular)? [eE]xp(r?e
ssion)?new RegExp("a")
.ex (" ") {4,2} r
e .* re u
re s c ('(/ , t n
. ub ^|) !\ |$), ' '
\n n \\
(' ; in ( )
* d ? t
e ex
nt, t

Equivalence quiz

(a|b|c)

Equivalence quiz

(a|b|c)

([abc])

Equivalence quiz

a+

Equivalence quiz

$a+$

$\cong$

aa^*

Equivalence quiz

a{1,3}

Equivalence quiz

a{1,3}

aa?a?

Equivalence quiz

(a|b)?

(.*)a[-0-9\\]+\d\d+?(:?\,)
0E0 [^<]+?
{1,3}\.){2}\d{1,3}::local
/.*@[a-zA-Z][a-Z0-9]/
preg_replace($p,$r,$str)
egrep -r '^To: ' /var/mail
[Rr]eg(ular)? [eE]xp(r?e
ssion)?new RegExp("a")
.ex (" ") {4,2} r
e .* re u
re s c ('(/ , t n
. ub ^|) !\ |$), ' '
\n n \\
(' ; in ( )
* d ? t
e ex
nt, t

Equivalence quiz

(a|b)?

(a|b|)

(.*)a[-0-9\\]+\d\d+?(:?\,)
0E0 [^<]+?
{1,3}\.){2}\d{1,3}::local
/.*@[a-zA-Z][a-Z0-9]/
preg_replace($p,$r,$str)
egrep -r '^To: ' /var/mail
[Rr]eg(ular)? [eE]xp(r?e
ssion)?new RegExp("a")
.ex (" ") {4,2} r
e .* re u
re s c ('(/ , t n
. ub ^|) !\ |$), ' '
\n n \\
(' ; in ( )
* d ? t
e ex
nt, t

Equivalence quiz

a(b(c)?)?

Equivalence quiz

a(b(c)?)?

a(|b|b(c))

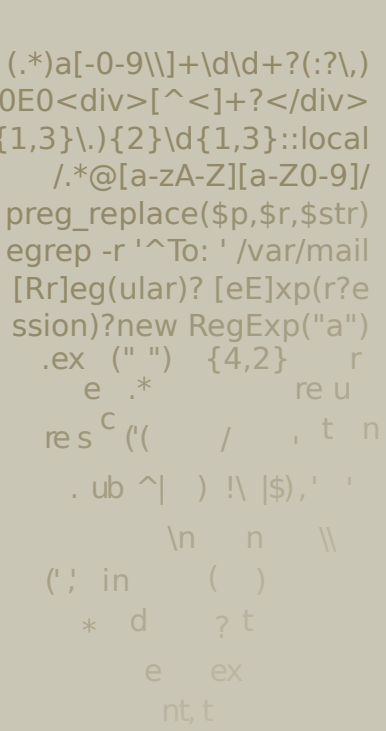
Equivalence quiz

a

Equivalence quiz

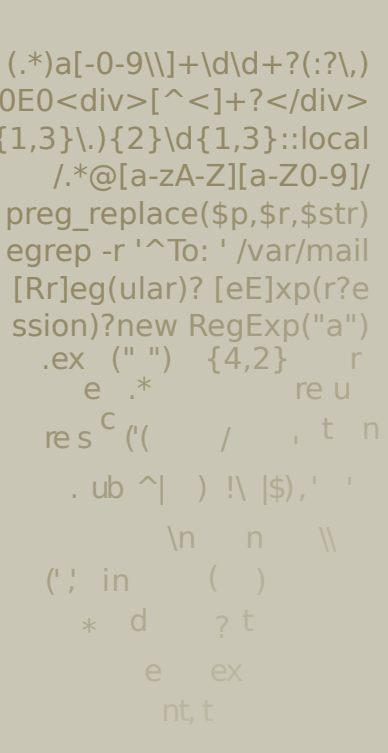
a

a{1}



Ready for some advanced stuff?

Outline



Word boundaries

Special meta character to denote
“beginning of word” or “end of word”.

Semantically is followed / preceeded by
whitespace or punctuation.

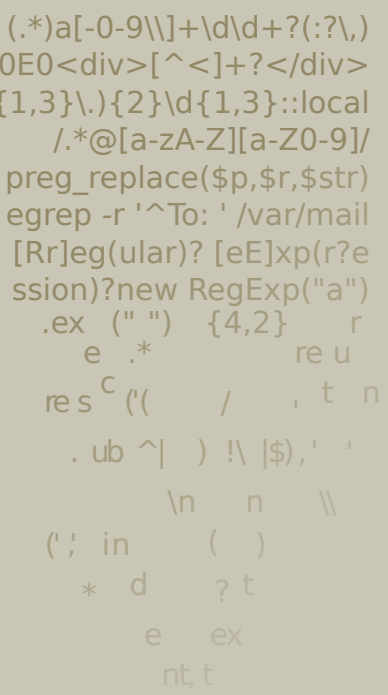
Variant 1 (GNU POSIX extension):

\<bits\>

Variant 2 (PCRE):

\bbits\b

- ☒ bits_and_bytes
- ☑ bits and bytes
- ☑ … without bits. Hence …
- ☑ “Keep those bits!”, he said.
- ☑ :bits are the solution!



Character classes

Problem. Character lists are tedious.
Shorthand lists are not flexible.

Solution.
POSIX-only character “classes”.

[[:alpha:]]

alphabetic characters
(depends on locale)

[:alnum:]	[:lower:]
[:alpha:]	[:print:]
[:ascii:]	[:punct:]
[:blank:]	[:space:]
[:cntrl:]	[:upper:]
[:digit:]	[:word:]
[:graph:]	[:xdigit:]

☒
☑ a
☑ b
☑ D
☒ ab

http://www.regular-expressions.info/posixbrackets.html

(.*)a[-0-9\\]+\d\d+?(:?\,)
0E0 [^<]+?
{1,3}\.){2}\d{1,3}::local
/.*@[a-zA-Z][a-Z0-9]/
preg_replace($p,$r,$str)
egrep -r '^To: ' /var/mail
[Rr]eg(ular)? [eE]xp(r?e
ssion)?new RegExp("a")
.ex (" ") {4,2} r
e .* re u
re s c ('(/ , t n
. ub ^|) !\ |$), ' '
\n n \\
(' ; in ( )
* d ? t
e ex
nt, t

Character classes

Why are character classes more flexible than shorthands?

[^a[:digit:]]

one character which is not an a or a digit

- ☒ a
- ☑ b
- ☑ #
- ☑ *(newline)*
- ☒ 4

Scoping

Problem. I want to match “bits” or “bats”.

bi|ats	matches “bi” or “ats”
b(i|a)ts	matches “bits” or “bats”

Solution. We use groups for scoping.
Okay... fine. But we introduced a new group!

b(i|a)ts

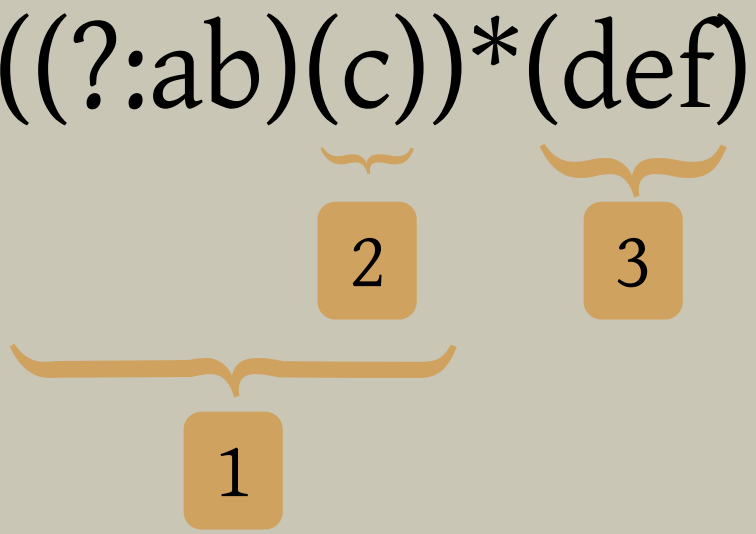
1

Non-grouping matches

I want to group something, I don't want as a group... finally.

(?:regex)

Hence...

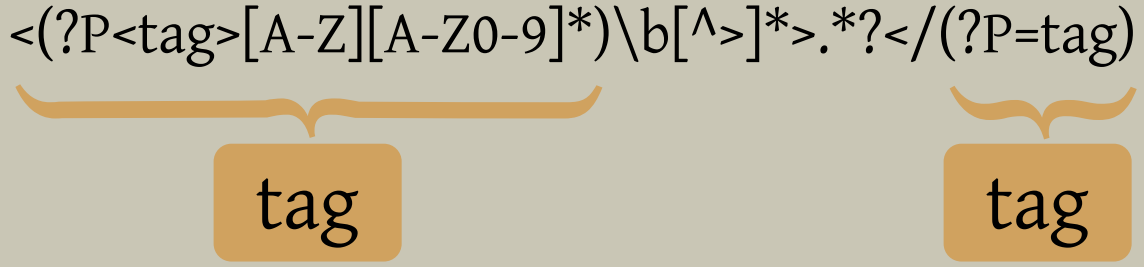


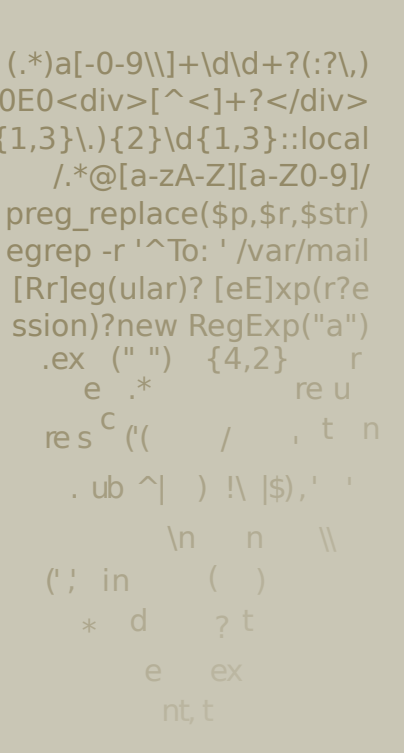
Named groups

Group something and give it a name.

(?P<name>regex)

So we can give meaningful names
instead of integers...





Lookahead

We want to match only if the following text matches, but we don't want to consume it.

(?=regex)...(..[0-1])

For input “regex3”, group 1 will be “ex3”.

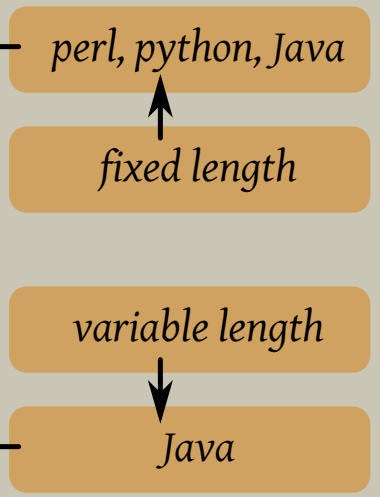
(?=reg(ex)?)...(..[0-1])

The RegEx engine forgets about the lookahead. So still only 1 group defined.

Lookahead

We want to match only if the following text matches, but we don't want to consume it.

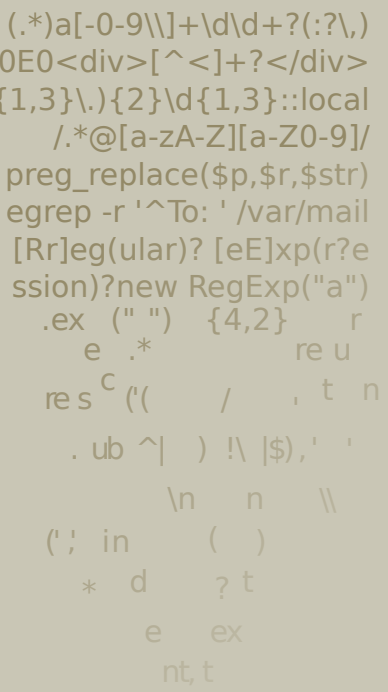
(?=regex)...(..[0-1])



For input “regex3”, group 1 will be “ex3”.

(?=reg(ex)?)...(..[0-1])

The RegEx engine forgets about the lookahead.
So still only 1 group defined.



Lookbehind

Same for the text before the current position.

(?<=bits)and bytes

Lookarounds summary:

positive lookahead	*if Y matching X follows*	(?=X)Y
negative lookahead	*if Y not matching X follows*	(?!X)Y
positive lookbehind	*if Y is preceded by X*	(?<=X)Y
negative lookbehind	*if Y is not preceded by X*	(?<!X)Y

If-then-else

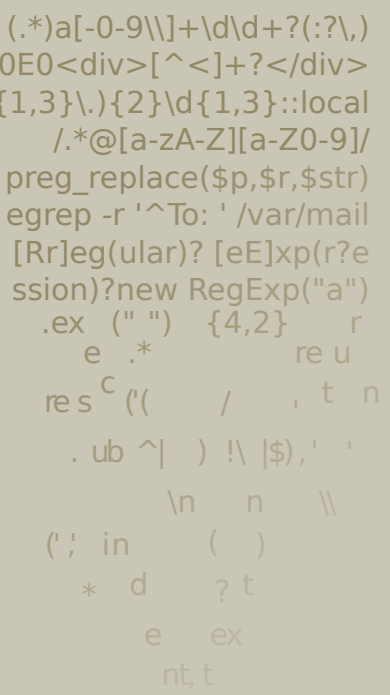
Define regex as conditional. If true, proceed with regexA, otherwise proceed with regexB.

(?(id or name)regexA|regexB)

id or name? Pfft, let's use lookaheads 😊

(?(?=%PDF-1\.3)oldspec|newspec)

```
>>> re.search("^([A-Z]\w{1,3}) = (?(1)[^'\"]|.)", "Var = 'hello'")
>>> re.search("^([A-Z]\w{1,3}) = (?(1)[^'\"]|.)", "Var = 3")
<_sre.SRE_Match object; span=(0, 7), match='Var = 3'>
```



Backreferences

We matched previously some substring.
We now want the same substring.

(.*)a[-0-9\\]+\d\d+?(:?\,)
0E0 [^<]+?
{1,3}\.){2}\d{1,3}::local
/.*@[a-zA-Z][a-Z0-9]/
preg_replace($p,$r,$str)
egrep -r '^To: ' /var/mail
[Rr]eg(ular)? [eE]xp(r?e
ssion)?new RegExp("a")
.ex (" ") {4,2} r
e .* re u
re s c ('(/ , t n
. ub ^|) !\ |$), ' '
\n n \\
(' ; in ( )
* d ? t
e ex
nt, t

Backreferences

We matched previously some substring.
We now want the same substring.

XML:

```
<tag attr1="value1" attr2="value2">
  content
</tag>
```

Backreferences

We matched previously some substring.
We now want the same substring.

XML:

```
<tag attr1="value1" attr2="value2">
  content
</tag>
```

RegEx:

```
<tag( \w+="[^"]+")*>
  (.*)
</tag>
```

Backreferences

We matched previously some substring.
We now want the same substring.

XML:

```
<tag attr1="value1" attr2="value2">
  content
</tag>
```

RegEx:

```
<(\w+)( \w+="[^"]+")*>
  (.*)
</(\w+)>
```

Backreferences

We matched previously some substring.
We now want the same substring.

XML:

```
<tag attr1="value1" attr2="value2">
  content
</tag>
```

RegEx:

```
<(\w+)( \w+="[^"]+")*>
  (.*)
</(\w+)>
```

matches:

```
<tag>content</tag>
```

matches:

```
<tag>content</tagged>
```

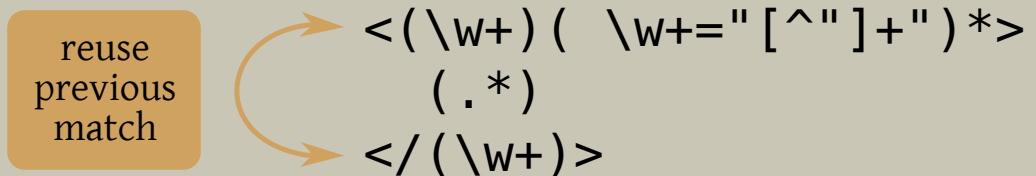
Backreferences

We matched previously some substring.
We now want the same substring.

XML:

```
<tag attr1="value1" attr2="value2">
  content
</tag>
```

RegEx:



matches:

```
<tag>content</tag>
```

matches:

```
<tag>content</tagged>
```

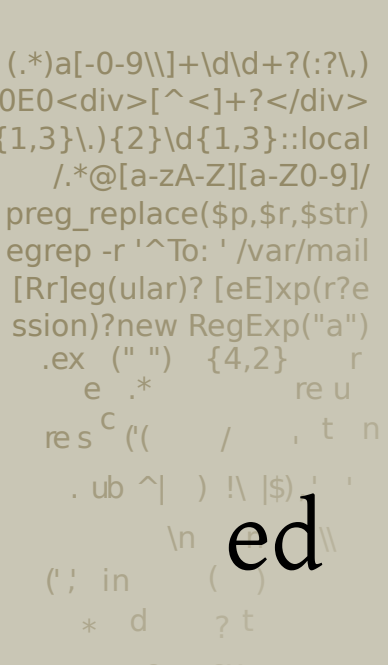
Backreferences

RegEx:

```
<([a-z]\w+)( \w+="[^"]+")*>
  (.*)
</\1>
```

Matches if and only if the opening and closing tag have the same name 😎

Outline



Tools relying on RegEx

... and also contributing to regex research in some way

ed

ex

vi

Java

awk

sed

egrep

python

sam

grep

emacs/elisp

perl

Tcl

go

lex

flex

Outline

RegEx confusions 😕

- You are lying. My regex looks different!
- How do I match arbitrary text between two marks?
- How long will the match be?
- Does it match a line or the whole string?
- How do I reference a repeated match in a group?
- What about language theory?
- What about performance?
- What about Unicode?

(.*)a[-0-9\\]+\d\d+?(:?\,)
0E0 [^<]+?
{1,3}\.){2}\d{1,3}::local
/.*@[a-zA-Z][a-Z0-9]/
preg_replace($p,$r,$str)
egrep -r '^To: ' /var/mail
[Rr]eg(ular)? [eE]xp(r?e
ssion)?new RegExp("a")
.ex (" ") {4,2} r
e .* re u
re s c ('(/ , t n
. ub ^|) !\ |$), ' '
\n n \\
(' ; in ( )
* d ? t
e ex
nt, t

Confusion #1: same same

... but different

```
$ grep -rn 'basic strategy' Main/*.txt
```

Confusion #1: same same

... but different

```
$ grep -rn 'basic strategy' Main/*.txt
```

no regex param

globbing

Before running the executable “grep”, the asterisk gets expanded for all matches where asterisk stands for some arbitrary string.

grep will never know of the existence of asterisk.

The POSIX standard defines globbing as shell builtin feature. Did you know “?” stands for one character?

Confusion #1: MS Word

- You can use parentheses to group the wildcard characters and text and to indicate the order of evaluation. For example, type **<(pre)*(ed)>** to find "presorted" and "prevented".
- You can use the \n wildcard to search for an expression and then replace it with the rearranged expression. For example, type **(Ashton) (Chris)** in the **Find what** box and **\2 \1** in the **Replace with** box. Word will find **Ashton Chris** and replace it with **Chris Ashton**.

To find	Type	Example
Any single character	**?**	s?t finds sat and set.
Any string of characters	*****	s*d finds sad and started.
The beginning of a word	**<**	<(inter) finds interesting and intercept, but not splintered.
The end of a word	**>**	(in)> finds in and within, but not interesting.
One of the specified characters	**[ ]**	w[io]n finds win and won.
Any single character in this range	**[-]**	[r-t]ight finds right and sight. Ranges must be in ascending order.
Any single character except the characters in the range inside the brackets	**[!x-z]**	t[!a-m]ck finds tock and tuck, but not tack or tick.
Exactly n occurrences of the previous character or expression	**{n}**	fe{2}d finds feed but not fed.
At least n occurrences of the previous character or	**{n,}**	fe{1,}d finds fed and feed.

(.*)a[-0-9\\]+\d\d+?(:?\,)
0E0 [^<]+?
{1,3}\.){2}\d{1,3}::local
/.*@[a-zA-Z][a-Z0-9]/
preg_replace($p,$r,$str)
egrep -r '^To: ' /var/mail
[Rr]eg(ular)? [eE]xp(r?e
ssion)?new RegExp("a")
.ex (" ") {4,2} r
e .* re u
re s c ('(/ , t n
. ub ^|) !\ |$), ' '
\n n \\
(' ; in ( )
* d ? t
e ex
nt, t

Who has programmed more than 10 LOCs Lua?

Lua string matching

“Lua patterns can match sequences of characters, ... If you're used to other languages that have regular expressions to match text, remember that Lua's pattern matching is not the same: it's more limited, and has different syntax.” http://lua-users.org/wiki/PatternsTutorial

```
> = string.find("abcdefg", 'b..')
2 4
> = string.match("foo 123 bar", '%d%d%d')
123
> = string.match("text with an Uppercase letter", '%u')
U
> = string.match("abcd", '[bc][bc]')
bc
> = string.match("abcd", '[^ad]')
b
> = string.match("123", '[0-9]')
1
```

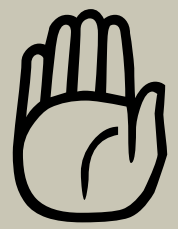
Lua string matching

`*` `+` `?` as usual. But as many times as possible.

`-` matches zero or more times, but as *few* times as possible.

```
> = string.match("abc", 'a.*')
abc
> = string.match("abc", 'a.-')
a
> = string.match("abc", 'a.-$')
abc
> = string.match("abc", '^.-b')
ab
```

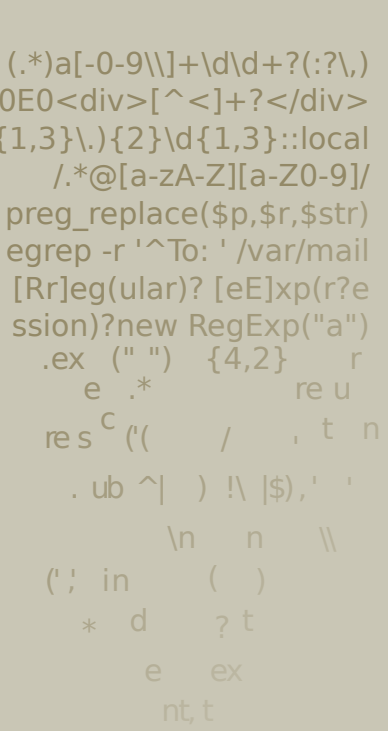
Who thinks CSS has RegEx?



CSS regex selectors

`a[href]`	*a* tag with href element
`a[href$=.svg]`	href **ends** with .svg
`a[href*=tugraz]`	href **contains** tugraz
`a[href^=https]`	href **starts** with https

Just selection / matching. No extraction.
No references. No replacements.



Confusion #1: conclusion

Not everything looking like RegEx, is RegEx.

Do not invent your own text matching standard!
Performance is not an excuse.

Still pointing out differences?
There are two major RegEx standards:

POSIX: the original UNIX definition
PCRE: additional features by perl (most of what we talked about in the Advanced chapter)

Confusion #2: delimiters

How do I match arbitrary text between two marks?

Student asks: How to extract everything between two marks?

Donald Knuth says, “I define UNIX as 30 definitions of regular expressions living under one roof.”

Student's approach: Well... everything between “ and ”

“(.*)”

... if you search for a substring.

Confusion #2: delimiters

How do I match arbitrary text between two marks?

Student asks: How to extract everything between two marks?

Donald Knuth says, “I define UNIX as 30 definitions of regular expressions living under one roof.”

RegEx-master's approach: Everything until the next quotation mark

“([^”]*)”

... if you search for a substring.

Confusion #2: delimiters

How do I match arbitrary text between two marks?

Student asks: How to extract everything between two marks?

> Donald Knuth says, “I define UNIX as 30 definitions of regular expressions living under one roof.”

Rationale:

Every regex engine returns longest, leftmost match.
So the engine will pass the second quotation mark and search for the longest match → performance problem and will probably match more

Confusion #2: delimiters

How do I match arbitrary text between two marks?

Student asks: How to extract everything between two marks?

Donald Knuth says, “I define UNIX as 30 definitions of regular expressions living under one roof.”

As python code:

```
>>> import re
>>> text = 'As "John Doe" said recently, "No one knows"'
>>> re.search('"(.*)"', text).group(1)
'John Doe" said recently, "No one knows'
>>> re.search('"([^"]*)"', text).group(1)
'John Doe'
```

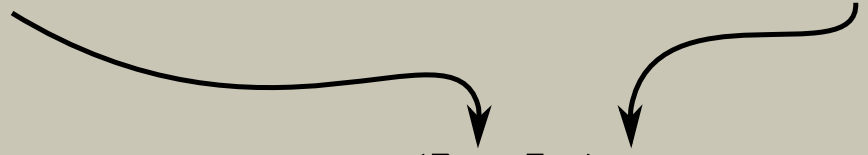
(.*)a[-0-9\\]+\d\d+?(:?\,)
0E0 [^<]+?
{1,3}\.){2}\d{1,3}::local
/.*@[a-zA-Z][a-Z0-9]/
preg_replace($p,$r,$str)
egrep -r '^To: ' /var/mail
[Rr]eg(ular)? [eE]xp(r?e
ssion)?new RegExp("a")
.ex (" ") {4,2} r
e .* re u
re s c ('(/ , t n
. ub ^|) !\ |$), ' '
\n n \\
(' ; in ( )
* d ? t
e ex
nt, t

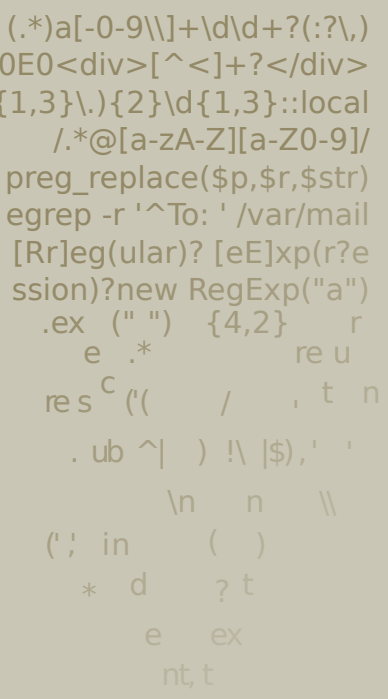
Confusion #2: delimiters

conclusion

Conjecture: .* is almost always wrong.

Nice approach: Which characters terminate the string? Ask for it *not* to occur. Then ask for it.





Confusion #3: greediness

How long will the match be?
Longest, leftmost match...
Can we change that?

“Longest” is called “greediness”.

PCRE:

??	match 0-1 times, shortest possible
***?**	match 0-infinity times, shortest possible
+?	match 1-infinity times, shortest possible

Confusion #3: greediness

```
>>> import re
>>> text = 'As "John Doe" said recently, "No one knows"'
>>> re.search('"(.*?)"', text).group(1)
'John Doe'
>>> re.search('"([^"]*)"', text).group(1)
'John Doe'
```

(.*)a[-0-9\\]+\d\d+?(:?\,)
0E0 [^<]+?
{1,3}\.){2}\d{1,3}::local
/.*@[a-zA-Z][a-Z0-9]/
preg_replace($p,$r,$str)
egrep -r '^To: ' /var/mail
[Rr]eg(ular)? [eE]xp(r?e
ssion)?new RegExp("a")
.ex (" ") {4,2} r
e .* re u
re s c ('(/ , t n
. ub ^|) !\ |$), ' '
\n n \\
(' ; in ( )
* d ? t
e ex
nt, t

Confusion #3: greediness

conclusion

Greediness control is important!

via special operators ← python, perl, ...

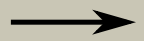
via modifiers ← PHP

via special flags within regex

(.*)a[-0-9\\]+\d\d+?(:?\,)
0E0 [^<]+?
{1,3}\.){2}\d{1,3}::local
/.*@[a-zA-Z][a-Z0-9]/
preg_replace($p,$r,$str)
egrep -r '^To: ' /var/mail
[Rr]eg(ular)? [eE]xp(r?e
ssion)?new RegExp("a")
.ex (" ") {4,2} r
e .* re u
re s c ('(/ , t n
. ub ^|) !\ |$), ' '
\n n \\
(' ; in ( )
* d ? t
e ex
nt, t

Confusion #4

- grep is only meant for line-wise mode.
- perl 6: `^^` and `$$` instead of `/m`
- Everything else is multiline capable and has a modifier for it.



- `/./` - Any character except a newline.
- `/./m` - Any character (the `m` modifier enables multiline mode)
- `/\w/` - A word character (`[a-zA-Z0-9_]`)
- `/\W/` - A non-word character (`[^a-zA-Z0-9_]`). Please take a look at Bug #4044 if using `/\W/` with the `/i` modifier.
- `/\d/` - A digit character (`[0-9]`)
- `/\D/` - A non-digit character (`[^0-9]`)
- `/\h/` - A hexdigit character (`[0-9a-fA-F]`)
- `/\H/` - A non-hexdigit character (`[^0-9a-fA-F]`)
- `/\s/` - A whitespace character: `/[ \t\r\n\f]/`
- `/\S/` - A non-whitespace character: `/[^ \t\r\n\f]/`

Confusion #5: repetition

How do I reference a repeated match in a group?

Input: abc

RegEx: ^(.*)$
Output: 'abc'

RegEx: ^(.)*$
Output: 'c'

You cannot extract a variadic number of matches.
You need a programming language outside and individually match parts.

Start search from an incremental offset and match always the start of the string.

python: **`search`**`(pattern, string, flags=0)`
search for substring
`match``(pattern, string, flags=0)`
search with implicit ^

Confusion #5: repetition

How do I reference a repeated match in a group?

Input: abc

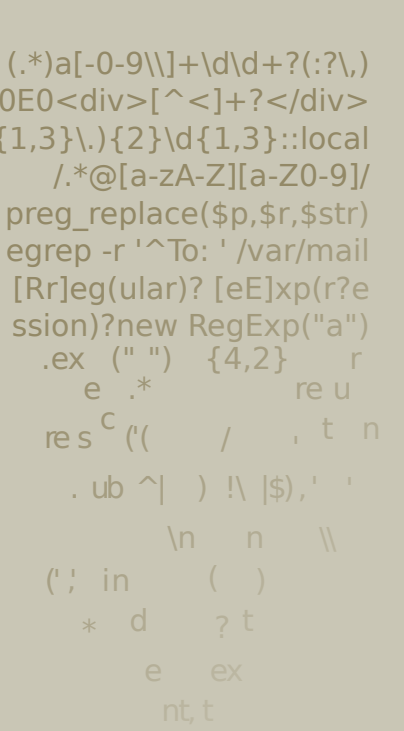
RegEx: ^(.*)$
Output: 'abc'

RegEx: ^(.)*$
Output: 'c'

You cannot extract a variadic number of matches.
You need a programming language outside and individually match parts.

Start search from an incremental offset and match always the start of the string.

javascript: modifier y



Confusion #6: lang theory

Regular expressions specify regular languages.

→ originally, yes

→ nowadays, no

(.*)a[-0-9\\]+\d\d+?(:?\,)
0E0 [^<]+?
{1,3}\.){2}\d{1,3}::local
/.*@[a-zA-Z][a-Z0-9]/
preg_replace($p,$r,$str)
egrep -r '^To: ' /var/mail
[Rr]eg(ular)? [eE]xp(r?e
ssion)?new RegExp("a")
.ex (" ") {4,2} r
e .* re u
re s c ('(/ , t n
. ub ^|) !\ |$), ' '
\n n \\
(' ; in ( )
* d ? t
e ex
nt, t

Confusion #6: lang theory

Regular expressions specify regular languages.

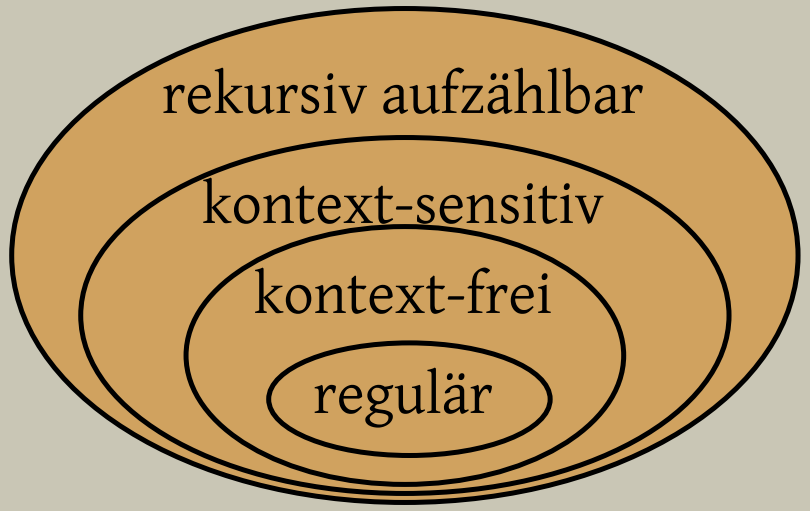
→ originally, yes

→ nowadays, no

(.*)a[-0-9\\]+\d\d+?(:?\,)
0E0 [^<]+?
{1,3}\.){2}\d{1,3}::local
/.*@[a-zA-Z][a-Z0-9]/
preg_replace($p,$r,$str)
egrep -r '^To: ' /var/mail
[Rr]eg(ular)? [eE]xp(r?e
ssion)?new RegExp("a")
.ex (" ") {4,2} r
e .* re u
re s c ('(/ , t n
. ub ^|) !\ |$), ' '
\n n \\
(' ; in ( )
* d ? t
e ex
nt, t

Confusion #6: lang theory

Backreferences are not regular.
Omit backreferences and you can compute regular expressions efficiently.

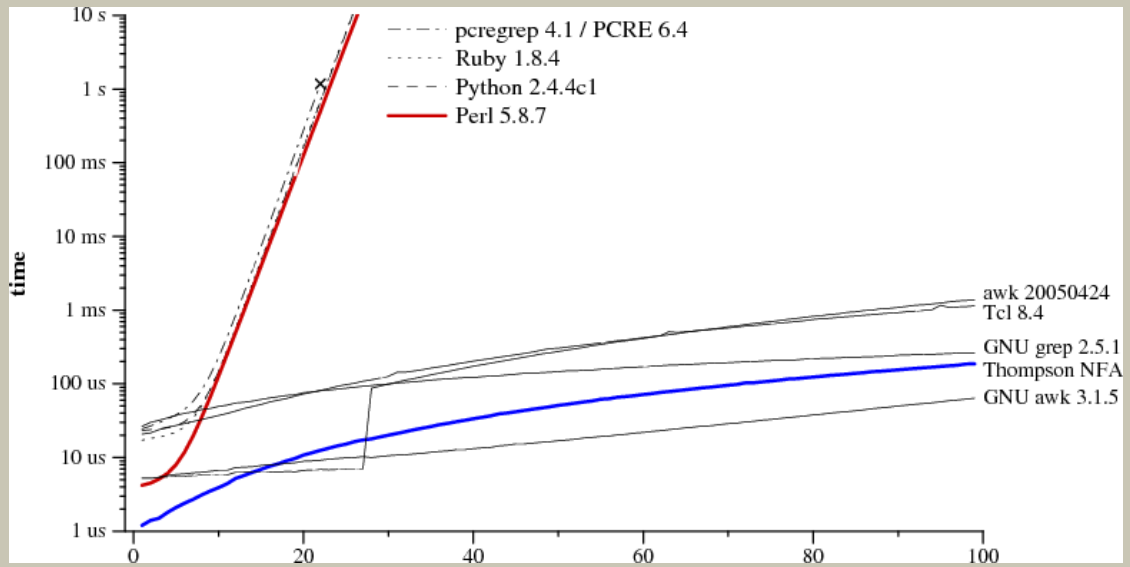


most state-of-the-art
regex engine:
re2 used by golang
linear time and no backrefs

Confusion #7: Performance

POSIX engines had wrong approaches.
Linear time besides backreferences desirable.
Can be achieved, but pathological regexes exist:

`x?x?x?x?x?x?x?x?` for input `xxxxxxx`
`awk "/X(.+)*X/{print}"` for `echo =XX==================`
`^(a+)+$` for `aaaaaaaaaaaaaaaX`
`(x*y*)*` for `xyxyxyxxxyxyxyyyxxxyx`



(.*)a[-0-9\\]+\d\d+?(:?\,)
0E0 [^<]+?
{1,3}\.){2}\d{1,3}::local
/.*@[a-zA-Z][a-Z0-9]/
preg_replace($p,$r,$str)
egrep -r '^To: ' /var/mail
[Rr]eg(ular)? [eE]xp(r?e
ssion)?new RegExp("a")
.ex (" ") {4,2} r
e .* re u
re s c ('(/ , t n
. ub ^|) !\ |$), ' '
\n n \\
(' ; in ( )
* d ? t
e ex
nt, t

Confusion #7: Performance

conclusion

- Avoid optional element following optional element
- Especially if they share structure
- Things are getting better. Faster backref-less engines coming!
- In the meanwhile: Don't let user specify regular expression!

Confusion #8: Unicode

POSIX regex → one char = one byte
Unicode? → one char = one unicode point

Done. Right?

We decode string and encode it to charset required by engine. Engine computes, returns result and we decode & encode it back. Normalization (etc.) is not part of regex discussion, right?

Confusion #8: Unicode

POSIX regex → one char = one byte
Unicode? → one char = one unicode point

Right, but the point are character classes. Which characters should we be able to denote in character classes? From which writing systems? We need convenient classes. Much research to do.

```
\p{Hiragana}
\p{Katakana}
```

selection by keyword

```
\x{1F4A9}
```

by explicit unicode point

(.*)a[-0-9\\]+\d\d+?(:?\,)
0E0 [^<]+?
{1,3}\.){2}\d{1,3}::local
/.*@[a-zA-Z][a-Z0-9]/
preg_replace($p,$r,$str)
egrep -r '^To: ' /var/mail
[Rr]eg(ular)? [eE]xp(r?e
ssion)?new RegExp("a")
.ex (" ") {4,2} r
e .* re u
re s c ('(/ , t n
. ub ^|) !\ |$), ' '
\n n \\
(' ; in ( )
* d ? t
e ex
nt, t

Confusion #8: Unicode

conclusion

POSIX regex → one char = one byte

Unicode? → one char = one unicode point

This is not only a technical issue. This is linguistically interesting.

Performance should not be a problem as far as I can see. Character classes can be implemented by membership tests and this can be done efficiently.

Further reading - state of the art

Russ Cox, Google Code Search, Go prog. lang.

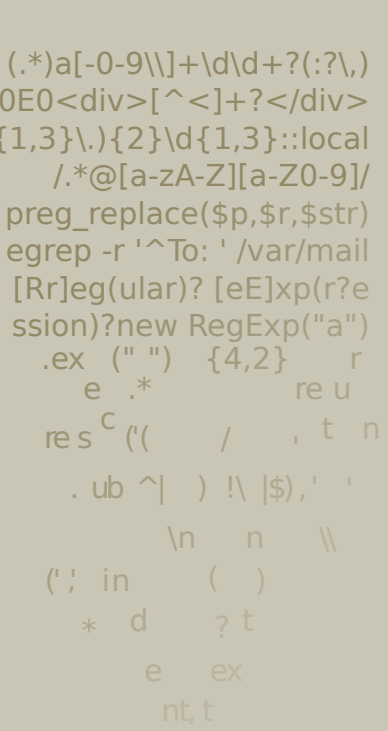
http://swtch.com/~rsc/regexp/
“Implementing regular expressions”

Unicode Consortium, TR 18

http://www.unicode.org/reports/tr18/
“Unicode Regular Expressions”

Nick Patch, unicode & regex

https://speakerdeck.com/patch/unicode-regular-expression-engines
“Unicode Regular Expression Engines”



Finally... ?

RegEx is a nice tool for text processing. RegEx needs a little bit of theory and practice and you can handle it.

What's your opinion on RegEx?

Difficult question: Should we use RegEx to show user which input in a textfield is accepted?

(.*)a[-0-9\\]+\d\d+?(:?\,)
0E0 [^<]+?
{1,3}\.){2}\d{1,3}::local
/.*@[a-zA-Z][a-Z0-9]/
preg_replace($p,$r,$str)
egrep -r '^To: ' /var/mail
[Rr]eg(ular)? [eE]xp(r?e
ssion)?new RegExp("a")
.ex (" ") {4,2} r
e .* re u
re s c ('(/ , t n
. ub ^|) !\ |$) , ' '
\n n \\
(' ; in ( )
* d ? t
e ex
nt, t

Thanks

Please stand back, we know RegEx!

Thanks to you and the BITS!

http://lukas-prokop.at/talks/regex_in_practice/