`reg(ular )?exp((?<=g...)|(?<= ...)ression)s`

Julian Bradfield

Regular Expressions *per* Inf1/2

Input symbols: $a, b, c, \ldots$

Empty: ϵ

Concatenation: $e_1 e_2$

Union/alternation: $e_1 \cup e_2$

Kleene star: $e*$

Parentheses as usual.

And (I presume) we tell them that you can add intersection and complementation, but it's harder to implement.

Regular Expressions in real (Unix) life

for decades were defined by grep and egrep. Give or take some backslashes, we had:

Literal characters: `a`

Concatenation: `ab`

Union: `a|b`

Kleene star: `a*`

Parentheses

Plus a bunch of convenient but obviously sugary abbreviations:

Character sets and classes: `.`, `[abc]`, `[^A-Z]`, `\w`

Optionality: `(xyx)?`

More quantifiers: `a+`, `(bc){3,7}`

and slightly less obviously:

Zero-width assertions matching ϵ at:

beginning/end of string: `^In the`, `Omega$`

edge of word: `\bthe\b`

and so on.

Still obviously sugar, but your Inf1 students might have to think about producing the FSM.

Zero-width assertions matching ϵ at:

beginning/end of string: `^In the`, `Omega$`

edge of word: `\bthe\b`

and so on.

Still obviously sugar, but your Inf1 students might have to think about producing the FSM.

Larry Wall begat Perl ...

Perl

Perl

Practical Extraction and Report Language

Perl

Practical Extraction and Report Language

Pathologically Eclectic Rubbish Lister

Perl is the antithesis of everything the Programming Language group does.

Perl

Practical Extraction and Report Language

Pathologically Eclectic Rubbish Lister

Perl is the antithesis of everything the Programming Language group does. Most of my programming is in Perl.

General philosophy of Perl:

Perl

Practical Extraction and Report Language

Pathologically Eclectic Rubbish Lister

Perl is the antithesis of everything the Programming Language group does. Most of my programming is in Perl.

General philosophy of Perl:

If you see a useful feature in another language, put it in Perl: associative arrays, references, symbolic (call-by-name) references, dynamic binding, static binding, dynamic code construction, object orientation, ...

Perl

Practical Extraction and Report Language

Pathologically Eclectic Rubbish Lister

Perl is the antithesis of everything the Programming Language group does. Most of my programming is in Perl.

General philosophy of Perl:

If you see a useful feature in another language, put it in Perl: associative arrays, references, symbolic (call-by-name) references, dynamic binding, static binding, dynamic code construction, object orientation, ...

and you never need to throw anything out

Perl

Practical Extraction and Report Language

Pathologically Eclectic Rubbish Lister

Perl is the antithesis of everything the Programming Language group does. Most of my programming is in Perl.

General philosophy of Perl:

If you see a useful feature in another language, put it in Perl: associative arrays, references, symbolic (call-by-name) references, dynamic binding, static binding, dynamic code construction, object orientation, ...

and you never need to throw anything out

and leave behind the stuff that just gets in the way: types, encapsulation

Perl

Practical Extraction and Report Language

Pathologically Eclectic Rubbish Lister

Perl is the antithesis of everything the Programming Language group does. Most of my programming is in Perl.

General philosophy of Perl:

If you see a useful feature in another language, put it in Perl: associative arrays, references, symbolic (call-by-name) references, dynamic binding, static binding, dynamic code construction, object orientation, ...

and you never need to throw anything out

and leave behind the stuff that just gets in the way: types, encapsulation

Perl was not the only language pushing more complex regexps, but probably the most influential.

PCREs

PCREs

Perl-Compatible Regular Expressions

PCREs

Perl-Compatible Regular Expressions

Preposterously Convoluted Regular Expressions

have spread into many other places (thanks, Henry Spencer and Phil Hazel) – even back into GNU grep.

Quantifier greediness

In real life, we don't just match regexps, we look for substrings matching them. Which substring?

Quantifier greediness

In real life, we don't just match regexps, we look for substrings matching them. Which substring?

Real-life regexps are non-deterministic, but with deterministic non-determinism.

Quantifier greediness

In real life, we don't just match regexps, we look for substrings matching them. Which substring?

Real-life regexps are non-deterministic, but with deterministic non-determinism.

Consider `<.*>` to find an XML tag in

```
<irritating>spurious</irritating>angle brackets
```

Quantifier greediness

In real life, we don't just match regexps, we look for substrings matching them. Which substring?

Real-life regexps are non-deterministic, but with deterministic non-determinism.

Consider `<.*>` to find an XML tag in
`<irritating>spurious</irritating>angle brackets`
fails because `*` is implemented as greedy (by everybody)

Quantifier greediness

In real life, we don't just match regexps, we look for substrings matching them. Which substring?

Real-life regexps are non-deterministic, but with deterministic non-determinism.

Consider `<.*>` to find an XML tag in
`<irritating>spurious</irritating>angle brackets`
fails because `*` is implemented as greedy (by everybody)

Could use `<[^>]*>`, but it's easier to say
`<.*?>` with the non-greedy quantifier.

Is is obvious in general that constructs controlling the back-tracking behaviour don't increase power?

Back-references

One of the oldest power-increasing extensions.

Match (ignoring escapes) Javascript string: `(["'']).*?\1`

In that case, clearly just succinct.

Back-references

One of the oldest power-increasing extensions.

Match (ignoring escapes) Javascript string: `(["'])\.*?\1`

In that case, clearly just succinct. But:

`(a*)b\1` is not regular

Back-references

One of the oldest power-increasing extensions.

Match (ignoring escapes) Javascript string: `(["'])\.*?\1`

In that case, clearly just succinct. But:

`(a*)b\1` is not regular; and

`(.*)\1` is not context-free.

Back-references

One of the oldest power-increasing extensions.

Match (ignoring escapes) Javascript string: `(["'])\.*?\1`
In that case, clearly just succinct. But:

`(a*)b\1` is not regular; and

`(.*)\1` is not context-free.

On the other hand, I can't see how to capture $a^n b^n$ just with back-references (and the other stuff mentioned so far).

So what exactly is the power of (say) standard regular expressions plus back-references?

Look-around

Perl generalizes the zero-width assertions to arbitrary look-ahead and fixed-width look-behind:

Look-around

Perl generalizes the zero-width assertions to arbitrary look-ahead and fixed-width look-behind:

`\d+(?=\w+)` matches a string of digits followed by a word (without including the word)

Look-around

Perl generalizes the zero-width assertions to arbitrary look-ahead and fixed-width look-behind:

`\d+(?=\w+)` matches a string of digits followed by a word (without including the word)

`\d+(?![Ee]-?\d+)` matches a number *not* followed by an exponent. (Exercise: no, it doesn't – what's the bug?)

Look-around

Perl generalizes the zero-width assertions to arbitrary look-ahead and fixed-width look-behind:

`\d+(?=\w+)` matches a string of digits followed by a word (without including the word)

`\d+(?![Ee]-?\d+)` matches a number *not* followed by an exponent. (Exercise: no, it doesn't – what's the bug?)

Look-behind: see title.

Note this allows intersection of (whole) regexps:
`^(?=`$patt_1$`$)`$patt_2$`$`

I think this adds succinctness, but no power.

Backtracking control

Default (greedy) quantifiers:

`foo.*ba` matches `foo bar baz`

Backtracking control

Default (greedy) quantifiers:

`foo.*ba` matches `foo bar baz`

Non-greedy:

`foo.*?ba` matches `foo bar baz`

Backtracking control

Default (greedy) quantifiers:

`foo.*ba` matches `foo bar baz`

Non-greedy:

`foo.*?ba` matches `foo bar baz`

This relies on backtracking, which is expensive. *Possessive* quantifiers `a*+` forbid backtracking inside the `a*`.

Backtracking control

Default (greedy) quantifiers:

`foo.*ba` matches `foo bar baz`

Non-greedy:

`foo.*?ba` matches `foo bar baz`

This relies on backtracking, which is expensive. *Possessive* quantifiers `a*+` forbid backtracking inside the `a*`.

More generally, `(?>`*patt*`)` cannot be backtracked into (but can be backtracked over as a whole).

I *think* this only adds succinctness, but by now I have little idea. . .

Backtracking control

Default (greedy) quantifiers:

`foo.*ba` matches `foo bar baz`

Non-greedy:

`foo.*?ba` matches `foo bar baz`

This relies on backtracking, which is expensive. *Possessive* quantifiers `a*+` forbid backtracking inside the `a*`.

More generally, `(?>`*patt*`)` cannot be backtracked into (but can be backtracked over as a whole).

I *think* this only adds succinctness, but by now I have little idea. . .

Worse: you can insert *backtracking control verbs* at arbitrary points in the expression.

Recursive regexps

To match balanced parentheses, of course:

Recursive regexps

To match balanced parentheses, of course:

`\([^()]*+(?R)?[^()]*+\)`

Or `a(?R)?b` to match $a^n b^n$.

What the heck . . .

`(??{code})`

inserts a dynamically computed sub-re (which knows what's been matched against so far).

Extracts from the manual

Regular expressions provide a terse and powerful programming language. As with most other power tools, power comes together with the ability to wreak havoc.

This document varies from difficult to understand to completely and utterly opaque. The wandering prose riddled with jargon is hard to fathom in several places.

And finally...

For those who haven't seen it before, this is the *right* regexp to validate an email address (and it doesn't even cope with comments):

```
(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t]
)+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:
\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(
?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^"\r\\]|\\.|(?:(?:\r\n)?[
\t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\0
31]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\
](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+
(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:
(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z
|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)
?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\
r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[
 \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)
?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t]
)*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
 \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*
)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t]
)+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*)
*:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+
|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r
\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:
\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^"\r\\]|\\.|(?:(?:\r\n)?[ \t
]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031
]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](
?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?
:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?
:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*)|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?
:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^"\r\\]|\\.|(?:(?:\r\n)?
```