# Tyre : Typed Regular Expressions

Are you even regular?

Gabriel Radanne

6 December 2016

```
[a-zA-Z]+[0-9]*
```

`[a-zA-Z]+[0-9]*`

[0] One or several letters followed by several numbers

```
[0-9]+(.[0-9]*)?([eE][+-]?[0-9]+)?
```

```
[0-9]+(.[0-9]*)?([eE][+-]?[0-9]+)?
```

[0] Floating point numbers in the OCaml language.

Source: The OCaml lexer.

```
^([Hh][Tt][Tt][Pp][Ss]?)://
([0-9a-zA-Z.-]+|\[[0-9A-Fa-f:.]+\])
(:([0-9]+))?/([^\?]*)(\?(.*))?$
```

```
^([Hh][Tt][Tt][Pp][Ss]?)://
([0-9a-zA-Z.-]+|\[[0-9A-Fa-f:.]+\])
(:([0-9]+))?/([^\?]*)(\?(.*))?$
```

[0]Url addresses (it's wrong, don't use it).

Source: The Ocsigen Web Server.

```
^([0369]
 |[258][0369]*[147]
 |[147]([0369]|[147][0369]*[258])*[258]
 |[258][0369]*[258]([0369]|[147][0369]*[258])
     *[258]
 |[147]([0369]
 |[147][0369]*[258])*[147][0369]*[147]
 |[258][0369]*[258]
  ([0369]|[147][0369]*[258])*[147][0369]*[147]
)*$
```

```
^([0369]
 |[258][0369]*[147]
 |[147]([0369]|[147][0369]*[258])*[258]
 |[258][0369]*[258]([0369]|[147][0369]*[258])
      *[258]
 |[147]([0369]
 |[147][0369]*[258])*[147][0369]*[147]
 |[258][0369]*[258]
   ([0369]|[147][0369]*[258])*[147][0369]*[147]
)*$
```

[0]Multiple of 3 in decimal representation.

```
(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?
\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.
\t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\
](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\
(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]
?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*
 \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?
)*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(
)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r
*:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[
\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])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*)|(?:[^()<>@,;:\\".\[\] \
:(?:\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)?[
@,;:\\".\[\] \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])*))*|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:
?:(?:\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])*)?(?:[^()<>
00-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \
;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(
(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r
^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[
?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\
".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?
\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(
])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?
\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)?[ \
()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)
?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*(?:
 \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\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\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*))*)?;\s*)
```

```
(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?
\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.
\t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\
](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\
(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]
?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*
 \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?
)*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(
)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r
*:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[
\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])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*)|(?:[^()<>@,;:\\".\[\] \
:(?:\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)?[
@,;:\\".\[\] \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])*))*|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:
?:(?:\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])*)?(?:[^()<>
00-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \
;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(
(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r
^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[
?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\
".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?
\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(
])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?
\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)?[ \
()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)
?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*(?:
 \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\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\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*))*)?;\s*)
```

[0] Email addresses according to the RFC.

There are several more pages.

```
\[^\t]+\t\\\[\[0-9]+
\\(0x(\[0-9a-fA-F]+)\\)]
```

```
\[^\t]+\t\\\[\[0-9]+
\\(0x(\[0-9a-fA-F]+)\\)]
```

[0] Find the number of escapes.

Source: Tcl’s tutorial on regular expressions.

# Why do we even use this?

# Regular expressions

Pros:

- A formal definition
- Not very expressive (this is a feature)
- Fast

  Matching takes a time proportional to the length of the string.
- Easy to analyze

  Pretty much everything is decidable.

# Irregular expressions

Cons:

- Insane syntaxes (glob, grep, php, pcre, sed, ...)
  - Compose badly
  - String quoting hell
- Irregular expressions

  Do not follow the formal definition (pcre, ...)
- Slow implementations. Mostly two reasons:
  - Exponential complexity (see point above)
  - Crappy implementations
- Extracting group information is difficult

  Stringly typing!

## Irregular expressions

Cons:

- Insane syntaxes (glob, grep, php, pcre, sed, . . . )
    - Compose badly
    - String quoting hell
- Irregular expressions

  Do not follow the formal definition (pcre, . . . )
- Slow implementations. Mostly two reasons:
    - Exponential complexity (see point above)
    - Crappy implementations
- Extracting group information is difficult

  Stringly typing!

## Reminder: Regular Expressions

- Letters: `a`, `b`,...
- Sequence: $\texttt{e}_1\texttt{e}_2$
- Empty word: $\varepsilon$
- Alternative: $\texttt{e}_1 | \texttt{e}_2$
- Repetition: `e*`
- Group: `(e)`

Things you can add:

- Sets: `[abc]` = `a|b|c`
- Plus: `e+` = `e|e*`
- Option: `e?` = $\varepsilon$`|e`
- Various constants: `^`, `$`, ...

# Regular Expressions – combinator edition

```
type t
(** Type of a regular expression *)

val char : char -> t
val seq : t list -> t (* e₁e₂... *)
val empty : t (* ε *)
val alt : t list -> t (* e₁|e₂|... *)
val rep : t -> t (* e* *)
val group : t -> t (* (e) *)
...
```

⇒ The `re` library (it's great, use it)

# Regular Expressions – combinator edition

```
type t
(** Type of a regular expression *)

val char : char -> t
val seq : t list -> t (* e1e2... *)
val empty : t (* ε *)
val alt : t list -> t (* e1|e2|... *)
val rep : t -> t (* e* *)
val group : t -> t (* (e) *)
...
```

⇒ The `re` library (it's great, use it)

# Url in Regular Expressions

```
open Re

let scheme = seq [rep (compl [set "/:?#"]); str "://"]
let host = rep (compl [set "/:?#"])
let port = seq [char ':'; rep1 digit]
let path = rep (seq [char '/'; rep (compl [set "/?#"])])
let query = seq [char '?'; rep (compl [set "#"])]
let fragment = seq [char '#'; rep any]
let url = seq [
  scheme ; host ; opt port ;
  path ; opt query ; opt fragment ;
]
```

# Url in Regular Expressions

```
open Re
let cset chars = compl [set chars]

let scheme = seq [rep (cset "/:?#"); str "://"]
let host = rep (cset "/:?#")
let port = seq [char ':'; rep1 digit]
let path = rep (seq [char '/'; rep (cset "/?#")])
let query = seq [char '?'; rep (cset "#")]
let fragment = seq [char '#'; rep any]
let url = seq [
  scheme ; host ; opt port ;
  path ; opt query ; opt fragment ;
]
```

# Url in Regular Expressions

```
open Re
let cset chars = compl [set chars]
let prefixed head tail = seq [char head; rep tail]

let scheme = seq [rep (cset "/:?#"); str "://"]
let host = rep (cset "/:?#")
let port = prefixed ':' digit
let path = rep (prefixed '/' (cset "/?#"))
let query = prefixed '?' (cset "#")
let fragment = prefixed '#' any
let url = seq [
  scheme ; host ; opt port ;
  path ; opt query ; opt fragment ;
]
```

⇒ The `uri` library (it's great, use it)

# Url in Regular Expressions

```
open Re
let cset chars = compl [set chars]
let prefixed head tail = seq [char head; rep tail]

let scheme = seq [rep (cset "/:?#"); str "://"]
let host = rep (cset "/:?#")
let port = prefixed ':' digit
let path = rep (prefixed '/' (cset "/?#"))
let query = prefixed '?' (cset "#")
let fragment = prefixed '#' any
let url = seq [
  scheme ; host ; opt port ;
  path ; opt query ; opt fragment ;
]
```

⇒ The `uri` library (it's great, use it)

# How do I extract information?

# Re groups

We use groups !

```
val group : t -> t
(** Delimit a group *)
```

# Url with groups

```
let scheme = seq [group (rep (cset "/:?#")); str "://"]
let host = group (rep (cset "/:?#"))
let port = seq [char ':'; group (rep1 digit)]
let path =
  group (rep (seq [char '/'; rep (cset "/?#")]))
let query = seq [char '?'; group (rep (cset "#"))]
let fragment = seq [char '#'; group (rep any)]
let url = seq [
  scheme ; host ; opt port ;
  path ; opt query ; opt fragment ;
]
```

## Re – extraction

```
type groups
(** Information about groups in a match *)

val exec : re -> string -> groups
(** [exec re str] matches [str] against the
    compiled expression [re], and returns the
    matched groups if any. *)

val gets : groups -> int -> string
(** Raise [Not_found] if the group did not
    match. *)
```

# Extraction in `ocaml-uri`

```
let get_opt s n =
  try Some (Re.get s n)
  with Not_found -> None
in
let subs = Re.exec Uri_re.uri_reference s in
let scheme = get_opt subs 2 in
let userinfo, host, port =
  match get_opt_encoded subs 4 with
  |None -> None, None, None
  |Some a ->
    let subs' = Re.exec Uri_re.authority (Pct.uncast_encoded a) in
    let userinfo = match get_opt subs' 1 with
      | Some x -> Some (Userinfo.userinfo_of_encoded (Pct.
          uncast_encoded x))
      | None -> None
    in
    let host = get_opt subs' 2 in
    let port =
      match get_opt subs' 3 with
      |None -> None
      |Some x ->
        (try
           Some (int_of_string (Pct.uncast_decoded x))
         with _ -> None)
    in
    userinfo, host, port
in
...
```

# Extraction

The bad news:

- Very sensitive to off-by-one errors.
- Everything is string.
- Manual type cast.
- Separation between the regex definition and the extraction code.
- The regex doesn't inform us about un-parsing.

The good news: We can solve it all in one go.

# Extraction

The bad news:

- Very sensitive to off-by-one errors.
- Everything is string.
- Manual type cast.
- Separation between the regex definition and the extraction code.
- The regex doesn't inform us about un-parsing.

The good news: We can solve it all in one go.

# Extraction

The bad news:

- Very sensitive to off-by-one errors.
- Everything is string.
- Manual type cast.
- Separation between the regex definition and the extraction code.
- The regex doesn't inform us about un-parsing.

The good news: We can solve it all in one go.

# Tyre

# Regex with a type parameter

```
type 'a t
(** A regular expression which result is of
   type 'a *)

val list : 'a t -> 'a list t
val opt : 'a t -> 'a option t

val seq : ?? t list -> ?? t
val alt : ?? t list -> ?? t
...
```

# Regex with a type parameter

```
type 'a t
(** A regular expression which result is of
   type 'a *)

val list : 'a t -> 'a list t
val opt : 'a t -> 'a option t

val seq : ?? t list -> ?? t
val alt : ?? t list -> ?? t
...
```

## Regex with a type parameter: seq

```
let e1 : int t = ...
let e2 : string t = ...
let e3 : float t = ...

let e : ?? t = rep [e1; e2; e3]
```

How to type `e`?

- `int * string * float` ?
  Not expressible in OCaml.
- `int * (string * float)` ?
  Would need special lists as argument of `rep`.

We can't even type `[e1; e2; e3]` with vanilla lists!

## Regex with a type parameter: seq

```
let e1 : int t = ...
let e2 : string t = ...
let e3 : float t = ...

let e : ?? t = rep [e1; e2; e3]
```

How to type `e`?

- `int * string * float` ?
  Not expressible in OCaml.
- `int * (string * float)` ?
  Would need special lists as argument of `rep`.

We can't even type `[e1; e2; e3]` with vanilla lists!

# Regex with a type parameter: seq

The best solution is to give up on lists here:

```
val seq : 'a t -> 'b t -> ('a * 'b) t
val (<&>) : 'a t -> 'b t -> ('a * 'b) t
```

```
let e1 : int t = ...
let e2 : string t = ...
let e3 : float t = ...

let e : ((int * string) * float) t
      = e1 <&> e2 <&> e3
```

# Regex with a type parameter: seq

The best solution is to give up on lists here:

```
val seq : 'a t -> 'b t -> ('a * 'b) t
val (<&>) : 'a t -> 'b t -> ('a * 'b) t

let e1 : int t = ...
let e2 : string t = ...
let e3 : float t = ...

let e : ((int * string) * float) t
      = e1 <&> e2 <&> e3
```

# Regex with a type parameter: alt

Similar solution for `alt`:

```
type ('a, 'b) lr = [
  | `Left of 'a
  | `Right of 'b
]

val alt : 'a t -> 'b t -> (('a, 'b) lr) t
val (<|>) : 'a t -> 'b t -> (('a, 'b) lr) t
```

# Regex with custom types

We also want to create regex matching base/new types.

```
val regex : Re.t -> string t

val map : ('a -> 'b) -> 'a t -> 'b t

let pos_int : int t =
  let re = rep1 digit in
  map int_of_string (regex re)
```

## Regex with custom types

We also want to create regex matching base/new types.

```
val regex : Re.t -> string t

val map : ('a -> 'b) -> 'a t -> 'b t

let pos_int : int t =
  let re = rep1 digit in
  map int_of_string (regex re)
```

# Regex with custom types

We also want to create regex matching base/new types.

```
val regex : Re.t -> string t

val map : ('a -> 'b) -> 'a t -> 'b t

let pos_int : int t =
  let re = rep1 digit in
  map int_of_string (regex re)
```

# Problem: uninteresting parts

```
let port = char ':' <&> pos_int
val port : (char * int) t
```

We don't care about the char!

Could add `map snd` to get rid of it, but inefficient.

Solution: Introduce a version of `seq` that throw away one side:

```
val prefix : _ t -> 'a t -> 'a t
val ( *> ) : _ t -> 'a t -> 'a t
val ( <* ) : 'a t -> _ t -> 'a t
```

```
let port = char ':' *> pos_int
val port : int t
```

The arrows point to the part that we keep.

# Problem: uninteresting parts

```
let port = char ':' <&> pos_int
val port : (char * int) t
```

We don't care about the char!

Could add `map snd` to get rid of it, but inefficient.

Solution: Introduce a version of `seq` that throw away one side:

```
val prefix : _ t -> 'a t -> 'a t
val ( *> ) : _ t -> 'a t -> 'a t
val ( <* ) : 'a t -> _ t -> 'a t
```

```
let port = char ':' *> pos_int
val port : int t
```

The arrows point to the part that we keep.

# The main functions

```
val compile : 'a t -> 'a re

val exec :
  'a re -> string -> ('a, 'a error) result
```

We need and error type because several errors are possibles:

- No match
- A converter failed

# Url with Regular Expressions – revisited

```
open Tyre
let cset chars = Re.(compl [set chars])
let pref c chars : string t =
  char c *> regex Re.(rep (cset chars))

let scheme: string t =
  regex Re.(rep (cset "/:?#")) <* str "://"
let host: string t =
  regex Re.(rep (cset "/:?#"))
let port: int t = char ':' *> pos_int
let path: string list t = rep (pref '/' "/?#")
let query: string t = pref '?' "#"
let fragment: string t =
  char '#' *> regex Re.(rep any)
```

# Url with Regular Expressions – revisited

```
let url_raw
  = scheme <&> host <&> opt port <&>
    path <&> opt query <&> opt fragment

let to_url (((((sch, h), p), pa), q), f) =
  { .... }

let url : Url.t t = map to_url url_raw
```

## Url with Regular Expressions – revisited

```
let myurl = Tyre.exec url "http://ocaml.org/a#b"
val myurl : (Url.t, Url.t error) result =
  Result.Ok
  {Url.sch = "http"; h = "ocaml.org"; p = None;
   pa = ["a"]; q = None; f = Some "b"}
```

# uri and tyre

While writing this presentation, I ended up using tyre to replace ocaml-uri's handwritten extraction code: `https://github.com/mirage/ocaml-uri/pull/93`

The end result is as efficient as the previously handwritten extraction.

# Unparsing

Another main function:

```
val eval : 'a t -> 'a -> string
(** [eval tyre v] returns a string [s] such
    that [exec (compile tyre) s = v].
*)
```

Automatically obtain a printing function as soon as you have a typed regex for a format.

# Unparsing – difficulties – map

`map` doesn't work anymore: Not enough information to go back.

We replace it by `conv`:

```
val conv
  : ('a -> 'b) -> ('b -> 'a) -> 'a t -> 'b t
```

`map` doesn't work anymore: Not enough information to go back.

We replace it by `conv`:

```
val conv
  : ('a -> 'b) -> ('b -> 'a) -> 'a t -> 'b t
```

# Unparsing – difficulties – prefix

Let us consider this code:

```
let re : int t =
  string *> pos_int

let s = eval re 3
```

We need to print part of the regex for which we have no values!

**We are regular!**

Given a regex *e*, finding *s* such that *e* matches *s* is decidable. We can make up a string that will fit the ignored part.

Regular languages are nice because all the properties are easy to find. The price to pay is limited expressivity (no html parsing using regexes ...).

## Unparsing – difficulties – prefix

Let us consider this code:

```
let re : int t =
  string *> pos_int

let s = eval re 3
```

We need to print part of the regex for which we have no values!

**We are regular!**
Given a regex *e*, finding *s* such that *e* matches *s* is decidable. We can make up a string that will fit the ignored part.

Regular languages are nice because all the properties are easy to find. The price to pay is limited expressivity (no html parsing using regexes ...).

# How does it all work?

## Implementation

We rely on `ocaml-re`.

- `Tyre.t` is a GADT describing everything in a typed regex.
- From a `Tyre.t`, we build two things:
  - An untyped regular expression `Re.t`
  - A witness that describes how to apply all the conversion functions on each group.
- A compiled typed regex is composed of
  - A compiled regex `Re.re`
  - A witness
- When executing the regex, we use the witness to do the extraction.

# Implementation

We rely on `ocaml-re`.

- `Tyre.t` is a GADT describing everything in a typed regex.
- From a `Tyre.t`, we build two things:
  - An untyped regular expression `Re.t`
  - A witness that describes how to apply all the conversion functions on each group.
- A compiled typed regex is composed of
  - A compiled regex `Re.re`
  - A witness
- When executing the regex, we use the witness to do the extraction.

## Implementation

We rely on `ocaml-re`.

- `Tyre.t` is a GADT describing everything in a typed regex.
- From a `Tyre.t`, we build two things:
  - An untyped regular expression `Re.t`
  - A witness that describes how to apply all the conversion functions on each group.
- A compiled typed regex is composed of
  - A compiled regex `Re.re`
  - A witness
- When executing the regex, we use the witness to do the extraction.

# Implementation

We rely on `ocaml-re`.

- `Tyre.t` is a GADT describing everything in a typed regex.
- From a `Tyre.t`, we build two things:
  - An untyped regular expression `Re.t`
  - A witness that describes how to apply all the conversion functions on each group.
- A compiled typed regex is composed of
  - A compiled regex `Re.re`
  - A witness
- When executing the regex, we use the witness to do the extraction.

# Implementation – difficulties

- For `alt` to work, we need a new concept in `re`: marks.

  Allow to mark sub-regexes and to know after execution if that specific part was matched.
- It's not possible to extract groups under a repetition [1]

  Instead, we proceed in two steps:
  - First match without extraction, to find how long the repetition is.
  - Then use a different regex with only the content of the repetition, and extract all the elements.

  Nesting repetitions is not great.

  Made it a bit better by enumerating the result of a repetition lazily.

[1] This would amount to counting in an automaton, which is not possible

# Implementation – bonus

The implementation is quite readable and is a good example of GADT at work.

```
type 'a t =
  | Regexp : Re.t * Re.re Lazy.t -> string t
  | Conv   : 'a t * ('a, 'b) conv -> 'b t
  | Opt    : 'a t -> ('a option) t
  | Alt    : 'a t * 'b t ->
             [`Left of 'a | `Right of 'b] t
  | Seq    : 'a t * 'b t -> ('a * 'b) t
  | Prefix : _ t * 'a t -> 'a t
  | Suffix : 'a t * _ t -> 'a t
  | Rep    : 'a t -> 'a gen t
  | Mod    : (Re.t -> Re.t) * 'a t -> 'a t
```

# Implementation – routing

Thanks to Marks, we get efficient routing for free.

```
let my_regex_with_routing =
  route [
    (str "foo-" *> pos_int)
       --> (fun i -> Foo i) ;
    (str "bar-" *> pos_int <&> str "-" *> string)
       --> (fun (i, s) -> Bar (i, s)) ;
  ]
```

# Go play:

`opam install tyre`

**`https://github.com/Drup/tyre`**