# We have the following problem

- need to check a lot of files (the 70 or so files comprising the source for this book, actually) to confirm that each file contained `**SetSize** ‘ exactly as often (or as rarely) as it contained `**ResetSize** '. To complicate matters, I needed to disregard capitalization (such that, for example, `**setSIZE** ' would be counted just the same as `**SetSize** ').

From the book “Mastering Regular Expressions” by Jeffrey E. F. Friedl

# Regular Expressions

ES 2016/2017

Adam Belloum

a.s.z.belloum@uva.nl

Material Prepared by Eelco Schatborn

# Today: Regular Expressions and Grammars

- What is a regular expression
- Formal Languages
- Context-free grammars; BNF, ABNF
- Type of regex engines
- Unix Regular Expressions

# Useful to Know …

- A more formal approach  to teaching regular expressions
  - Regular Expressions, lecture given at UCDavis, starting from (10mn to the end of the Video)
  - http://www.youtube.com/watch?v=B72XAeFO9ZE

# Regular Expressions

- A regular expression (regexp, regex, regxp) is a string (a word), that, **according to certain syntax rules, describes a set of strings** (a language).

- Regular Expressions are used in many text (unix) editors, tools and programming languages to **search for patterns** in a **text**, and for **substitution** of string

"Regular expressions" […] are only marginally related to real regular expressions. Nevertheless, the term has grown with the capabilities of our pattern matching engines, so I'm not going to try to fight linguistic necessity here. I will, however, generally call them "regexes" (or "regexen", when I'm in an Anglo-Saxon mood). Larry Wall, Perl 6

# Sometimes Intelligible: Obscure Regexes

- **Suppose to** "Validate" emails following the RFC 5322 standard
  - Tested the regular expression
    - http://leaverou.github.io/regexplained/
    - http://www.rubular.com/

```
(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*| "(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f] |\\[\x01-\x09\x0b\x0c\x0e-\x7f])*“)@(?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?|\[(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?|[a-z0-9-]*[a-z0-9]:(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])+)\])
```

http://www.regular-expressions.info/email.html (October 1,2013)

# Reasons to Learn and Use Regular Expressions

1. *Regular Expressions are **integrated** into **many** programmer **tools** and programming **languages**.*
2. *Each time you **use Regular Expressions**, you are invoking a powerful **search engine***
3. *Regular Expression syntax is **very simple**, **(**If your **search string** is **simple**)*

# Reasons to Learn and Use Regular Expressions

1. *Regular Expressions can be used **to match just** about anything!*
   1. *You can match **upper** or **lower** case*
   2. *You can match **either one** string or another*
   3. you can match whole classes of characters
   4. *you to match **any character** by using a **period***
   5. *…*

# History

- origins of regular expressions lie in
  - Theory of formal languages
  - and automata theory

- In 1950s Kleene's algebra of regular sets is introduced
  - **Stephen Kleene** best known as a founder of the branch of mathematical logic also known as recursion theory
- Ken Thompson introduced the RE notation to QED
- Regex is used in grep, awk, emacs, vi, perl.

# Today: Regular Expressions and Grammars

- What is a regular expression
- Formal Languages
- Context-free grammars; BNF, ABNF
- Type of regex engines
- Unix Regular Expressions

# Regular Expressions in formal languages theory

- A regular expression **represents** **a set of strings/words** (a language).

- Regexp are made up of **constants** and **operators** and must satisfy the Basis, inductive, and external clauses

- Given an alphabet $\Sigma$ , the following constants are defined as regular expressions:

| | | |
|---|---|---|
| **(empty set)** | $\varnothing$ | represents the empty set |
| **(empty string)** | $\varepsilon$ | represents the set $\{\varepsilon\}$ |
| **(literal character)** a in $\Sigma$ | | represents the set $\{a\}$ |

# Regular Expressions in formal languages theory

## Operators

- Assume Regular Expressions R and S

  **Concatenation** RS represents the set $\{\alpha\beta, \alpha \in R, \beta \in S\}$

  **Choice** R|S represents a choice between R and S $\{\alpha, \alpha \in R, \alpha \in S\}$

  **Iteration** R* represents the closure of R under concatenation;

$$R^* = \{\varepsilon\} \cup R \cup RR \cup RRR \cup \ldots$$

- **Binding strength**: Kleene star > concatenation > choice

  $((ab)c)$ can be written as $abc$

  $(a|(b(c)^*))$ as $a|bc^*$

# Regular Expressions in formal languages theory

Examples

- Let $\sum = \{0,1\}$, then
- `00` represents {00}
- `0100100` represents {0100100}
- `0|1` represents {0,1}
- `10|01` represents {10,01}
- `0*` represents { $\varepsilon$,0,00,000, …}
- `(01)*` represents { $\varepsilon$ ,01,0101,010101, …}
- `(0|00)1*` represents {0,00,01,001,011,0011, …}
- `(0|00)1*` = `01*|001*`

# Regular Expressions in formal languages theory

## More Examples

`a|b*` represents

`ab*(c |ε)` represents the set of strings that

- begin with a single **a**
- followed by **zero or more** b
- ending in an **optional** c

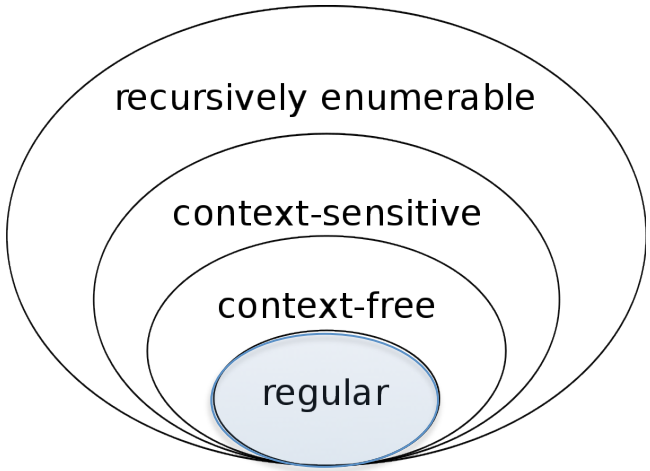
Unix regex : `ab*c?`

# Regular Languages

- Languages represented by Regular Expressions are called **regular languages**
- they correspond to "**type 3**" grammars in the Chomsky hierarchy

Example of a **Regular grammar**

$$\begin{aligned} \mathbf{S} &\rightarrow a\mathbf{S} \\ \mathbf{S} &\rightarrow b\mathbf{A} \\ \mathbf{A} &\rightarrow \varepsilon \\ \mathbf{A} &\rightarrow c\mathbf{A} \end{aligned}$$

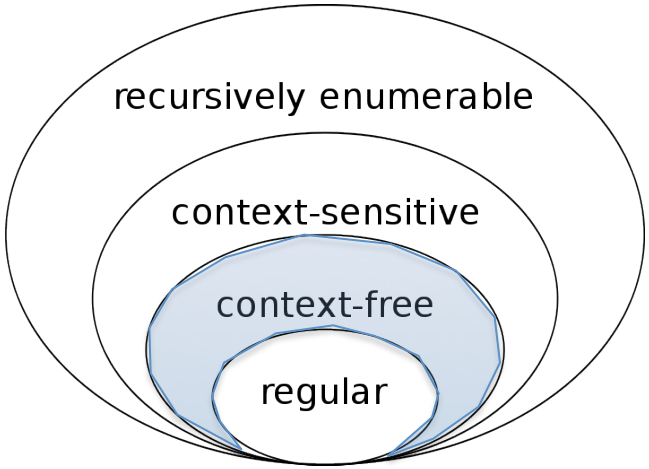



with start **symbol S** corresponds to the **regular expression** **a*bc***

# Context-Free languages

- Regular Expressions are **less expressive than** context-free grammars (CFGs)
- Example Context-Free Language $a^k b^k, k \geq 1$

$$S \rightarrow aSb$$
$$S \rightarrow ab$$




This language **cannot** be **expressed** by a **regular expression (Pumping Lemma).**

Note:

- Regular expressions in Perl are **not strictly** regular.
- Extra expressiveness can be detrimental to effectiveness Worst case complexity or matching a string against a Perl RE is time exponential to the length of the input

# Context-Free languages

Here is a context-free grammar for syntactically correct infix algebraic expressions in the variables x, y and z:

1. S → x
2. S → y
3. S → z
4. S → S + S
5. S → S - S
6. S → S * S
7. S → S / S
8. S → ( S )

This grammar can, for example,

( x + y ) * x - z * y / ( x + x )

as follows:

S (the start symbol)
→ S - S (by rule 5)
→ S * S - S (by rule 6, applied to the leftmost S)
→ S * S - S / S (by rule 7, applied to the rightmost S)
→ ( S ) * S - S / S (by rule 8, applied to the leftmost S)
→ ( S ) * S - S / ( S ) (by rule 8, applied to the rightmost S)
→ ( S + S ) * S - S / ( S ) (etc.)
→ ( S + S ) * S - S * S / ( S )
→ ( S + S ) * S - S * S / ( S + S )
→ ( x + S ) * S - S * S / ( S + S )
→ ( x + y ) * S - S * S / ( S + S )
→ ( x + y ) * x - S * y / ( S + S )
→ ( x + y ) * x - S * y / ( x + S )
→ ( x + y ) * x - z * y / ( x + S )
→ ( x + y ) * x - z * y / ( x + x )

https://en.wikipedia.org/wiki/Context-free_grammar

# notation: BNF

- **BNF (Backus Normal Form** or **Backus–Naur Form)** I
  - a notation technique for context-free grammars,
  - a format for defining CFG's

```
<bit>  ::= 0 | 1
<expr> ::= <bit> | (<expr> + <expr>) |
  (<expr> * <expr>)
```

  - Non-terminals  <bit>, <expr>
  - Terminals  0, 1, (, ), *, +, (space)
  - This BNF grammar generates the strings

    0, 1, (0 + 1), (1 * (1 + 1)), . . .

# notation: **A**ugmented**BNF**

- **Augmented Backus–Naur Form (ABNF)** is a **metalanguage** based on Backus–Naur Form (BNF), but it has its own **syntax** and **derivation rules**.

- The motive principle for ABNF is to describe a formal system of a language to be used as a bidirectional communications protocol.

- Defined by *Internet Standard 68*, which as of Feb 2010 is RFC 5234, and it often serves as the definition language for IETF communication protocols.

- RFC 2234, Augmented BNF for Syntax Specifications: ABNF. Obsolete.
- RFC 4234, Augmented BNF for Syntax Specifications: ABNF. Obsolete.
- RFC 5234 - Augmented BNF for Syntax Specifications: ABNF (Jan 2008)

# ABNF: **Rules**

- Rules have names like

  `"elements", "rule0" and "char-a-z"`

  - **Rule names may** be put inside brackets `<elements>`

  - **Rule names** are case-insensitive:

    `<rulename>, rULeNamE, <RULENAME>` refer to the **same rule**

- A rule is defined by a sequence of:

  `name = elements ; comment crlf`

# ABNF: **Terminal Values**

- **Terminal** values: specified by one/more numeric characters.
- A numeric character encoding like US-ASCII may be used

```
%b1000001  (binary 65, US-ASCII "A")
%x42       (hexadecimal 66, US-ASCII "B")
%d67       (decimal 67, US-ASCII "C")
%d13.10    (the sequence CR, LF)
```

# ABNF: Literal Text

- Literal text strings in ABNF are case insensitive (RFC 5234), to define case **insensitive rule** (rule1 and rule2).
- in the proposed standard (RFC 7405) Literal can be both case sensitive or insensitive. The form of matching used with a literal text string is denoted by a prefix to the quoted string
  - %s = case-sensitive
  - %i = case-insensitive (rule4) or simple string quotation (rule3)
- In RFC 5234 to define a case sensitive rule was necessary to use numerical specification of individual characters (rule6, rule7).

| RFC 5234 | RFC 7405 |
|---|---|
| `rule1= "abc"`<br>`rule2= "Abc"` | `rule3= "abc"`<br>`rule4= %i"Abc"`<br>`rule5= %s"abc"` |
| `rule1, rule2,rule3, rule4 will` match the set<br>`{abc, Abc, aBc, abC, ABc, aBC, AbC, ABC}` | |
| `rule6=%d97 %d98 %d99 ; a b c`<br>`rule7=%d97.98.99 ;` | |
| `rule5, rule6, rule7, will` match the set `{ abc }` | |

# ABNF: **Basic operators**

- Concatenation " "
- Alternatives "|"
- Repetition "*"
- Grouping "()"
- Comments ";"

# ABNF: **Concatenation**

- `Rule = Rule1 Rule2`

Example

```
magic = xyzzy foo bar
xyzzy = "xyzzy"
foo = "foo"
bar = "bar”

magic → "xyzzyfoobar”
```

# ABNF: **Alternatives**

- `Rule = Rule1 / Rule2`
- Sometimes uses pipe (|) instead of /

Example

```
magic = xyzzy / foo / bar
magic → "xyzzy", but also
magic → "foo", and also
magic → "bar"
```

# ABNF: **Variable Repetition**

- `Rule = <n>*<m>Rule1`
- n and m are optional decimal values
- Default for n is 0 and for m is infinity

- Example

  `magic = <2> * <3> xyzzy`

  `magic → "xyzzyxyzzy"`

  `magic → "xyzzyxyzzyxyzzy"`

# ABNF: **Grouping**

- `Rule = (Rule1 Rule2)`
- Only used for parsing (syntax)
- Has **no semantic** counterpart

## Example

`magictoo = (magic)`

`magictoo` has the same production as `magic`

## Example

```
Rule1 = elem (foo / bar) blat   VS   Rule2=elem foo / bar blat
```

Use grouping to avoid misunderstanding

`Rule2` is the same as `(elem foo) / (bar blat)`

# ABNF: **Comment**

- `Rule = . . . ; Followed by an explanation`

- Example

```
magic = xyzzy",”foo",”bar ; comma
  separated magic

magic → "xyzzy,foo,bar"
```

# ABNF: More operators

- Incremental alternative
- Value ranges
- Optional presence
- Specific repetition

# ABNF: **Incremental alternative**

- Alternatives may be added later in extra rules
- `Rule =/ Rule1`

Example

The rule

```
magic = "xyzzy" / "foo" / "bar"
```

is equivalent to

```
magic = "xyzzy"
magic =/ "foo"
magic =/ "bar"
```

# ABNF: **Value ranges**

- Uses "-" as range indicator in **terminal** specifications
  - Useful to create **case sensitive rules**, without having to list all the possible characters

Example

```
DIGIT = %x30-39 ; "0" / "1" / . . . / "9"
UPPER = %x41-5A ; "A" / "B" / . . . / "Z"
ALPHA = %x41-5A / %x61-7A ; A-Z / a-z
```

# ABNF: **Optional presence**

- `Rule = [Rule1]`

Equivalent to: `Rule = *<1>Rule1`

Example

magic = [ "xyzzy" ]

magic → "xyzzy", but also

magic → ""

# ABNF: **Specific repetition**

- `Rule = <n>Rule1`

Equivalent to : `Rule = <n>*<n>Rule1`

- Example

```
magic = <3>"xyzzy"
magic → "xyzzyxyzzyxyzzy"
```

# BNF, ABNF for U.S. postal address

## BNF for a U.S. postal address

```
<postal-address> ::= <name-part> <street-address> <zip-part>
<name-part>      ::= <personal-part> <last-name> <opt-suffix-part>
                     <EOL>  | <personal-part> <name-part>

 <personal-part>   ::= <first-name> | <initial> "."
 <street-address> ::= <house-num> <street-name> <opt-apt-num>
<EOL>

 <zip-part>          ::= <town-name> "," <state-code> <ZIP-code>
                     <EOL>

<opt-suffix-part>  ::= "Sr." | "Jr." | <roman-numeral> | ""
<opt-apt-num>     ::= <apt-num> | ""
...
```

## ABNF for a U.S. postal address

```
postal-address   = name-part street zip-part
name-part        = *(personal-part SP) last-name [SP suffix]
CRLF
name-part        =/ personal-part CRLF
personal-part    = first-name / (initial ".")
first-name       = *ALPHA
initial          = ALPHA
last-name        = *ALPHA
suffix           = ("Jr." / "Sr." / 1*("I" / "V" / "X"))
street           = [apt SP] house-num SP street-name CRLF
apt              = 1*4DIGIT
house-num        = 1*8(DIGIT / ALPHA)
street-name      = 1*VCHAR
zip-part         = town-name "," SP state 1*2SP zip-code
CRLF
town-name        = 1*(ALPHA / SP)
state            = 2ALPHA
zip-code         = 5DIGIT ["-" 4DIGIT]
```

# Today: Regular Expressions and Grammars

- what is a regular expression
- Formal Languages
- Context-free grammars; BNF, ABNF
- Type of Regex engines
- Unix Regular Expressions

# Regex engines

- Two kinds of regular expression engines:
  - **text-directed** engines (DFA engines)
  - **regex-directed** engines (NFA engines).

- All modern regex flavors are based on **regex-directed** engines for two very useful features.
  - lazy quantifiers
  - and back references.

Chapter 4 of the book “Mastering Regular Expressions” by Jeffrey E. F. Friedl

# A regex-directed engine

- walks through the regex, attempting to match the next token in the regex to the next character.
  - IF a match is found, THEN
    - the engine advances through the **regex** **and** the **subject string**.
  - IF no match, THEN
    - the engine backtracks to a previous position in the regex and the subject string where it can try a different path through the regex.

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

# NFA Engine: Regex-Directed

| Progress in the regex | Progress in the String to match | |
|---|---|---|
| …<br>t o (nite \| knight \| night) | …<br>Some text ….tonight…. | |
| to (nite \| knight \| night) | Some text ….tonight…. | |
| …<br>to (nite \| knight \| night) | …<br>Some text ….tonight | |
| to (nite \| knight \| night) | …<br>Some text ….tonight | |
| to (nite \| knight \| night) | Some text ….tonight | |
| to (nite \| knight \| night) | Some text ….tonight | A fail → Backtrack to n |
| …<br>to (nite \| knight \| night) | …<br>Some text ….tonight | |

# A text-directed engine

- walks through the subject string, **attempting all** permutations of the regex before advancing to the **next character in the string**.
- A text-directed engine **never backtracks**.

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

# DFA engines: text-directed engines

| regex | String to match |
|---|---|
| t o (nite \| knight \| night) | … Some text ….tonight…. |
| to (nite \| knight \| night) | Some text ….tonight…. … |
| to (nite \| knight \| night) | Some text ….tonight … |
| to (nite \| knight \| night) | Some text ….tonight |
| to (nite \| knight \| night) | Some text ….tonight |
| to (nite \| knight \| night) | Some text ….tonight … |

# Regex Engines in various linux tools

*Table 4-1: Some Tools and Their Regex Engines*

| Engine Type | Programs |
|---|---|
| DFA | Awk(1), egrep(1),Flex, lex, MySQL, Procmail |
| Traditional NFA | GNU Emacs, Java, grep(1), less, mode, .NET languages, PCRE library, Perl, PHP , Python, Ruby, sed(1), vi |
| POSIX NFA | Mawk, Mortice Kern Systems utilities, GNU Emacs |
| Hybrid NFA/DFA | GNU awk, GNU grep/egrep, Tcl |

(1) Most versions

Chapter 4 of the book “Mastering Regular Expressions” by Jeffrey E. F. Friedl

# Today: Regular Expressions and Grammars

- what is a regular expression
- Formal Languages
- Context-free grammars; BNF, ABNF
- Type of Regex engines
- Unix Regular Expressions

# Unix regexps

The **following syntax** is more or less standard in many unix **tools** and **programming languages**

| Basic rules | Description |
|---|---|
| Every **printable character** | that is not a **meta-character** is a regular expression that represents itself, for example a for the letter `a' |
| . | represents **any single character** except newline |
| ^ | represents the **beginning of a line** |
| $ | represents the **ending of a line** |
| \ | followed by a **meta-character** represents the character itself. Meaning \. represents a dot. t `.' |
| [E] | represents a **single character**. The characterization of E is put in brackets. |

# Unix regexps: Examples

| Basic rules | Description |
|---|---|
| **[a]** | represents a |
| **[abc]** | represents **a character** a, b, or c |
| **[a-z]** | represents **a character** in the **range** a-z (ordered according to the ASCII notation) |
| **[A-Za-z0-9]** | represents **a digit** or **a character**<br>represents every character that is not |
| **[^E]** | represents every character that is not represented by `[E]` |
| **[acq-z]** | represents `a, c,` or a character in `q-z`. |

# Unix regexps: Remarks

The following characters : “{" , “}", “<", “>", “(", “)” **preceded** by a **backslash** have a special meaning

- **Match a specific number of sets** with \{ and \}

[a-z]\{4,8\} → match 4, 5, 6, 7 or 8 lower case letters

- **Match words** with \< and \>

\<[tT]he\> → match "the” or “The”

- **Back references** - Remembering patterns with \(, \) and \1

[a-z][a-z] will match

\([a-z]\)\1 will also match

# POSIX **Extended** Regular Expressions

- The meaning of meta-characters escaped with a backslash is **reversed** in the POSIX Extended Regular Expression syntax
  - a **backslash** causes the metacharacter to be treated as a literal character
  - "\{" , "\}", "\<", "\>", "\(", "\)" . → are treated as literals
  - "{" , "}", "<", ">", "(", ")" → are treated as meta-characters

- Two programs use the **extended** regular expression: *egrep* and *awk*.

# Example

- We wan to match 5 letter palindrome like “radar”, “civic”, “level” …

```
\([a-z]\)\([a-z]\)[a-z]\2\1
```

# Unix regexps (1)

- Inductive rules if `A` and `B` are Regular Expressions then
  1. `AB` is a regular expression (concatenation)
  2. `A|B` is a regular expression (choice/unification)
  3. `A*` is a regular expression (**zero** or **more**)
  4. `A+` is a regular expression (**one** or **more**)
  5. `A?` is a regular expression (**zero** or **one**)
  6. `(A)` is a regular expression (**awk**, **egrep**, perl)

# Unix regexps (2)

- Inductive rules if `A` and `B` are Regular Expressions then
  7. `A\{m, n\}` for integers `m` and `n` **represents** m to n concatenated instances of A
  8. `A\{m\}` **represents** `m` concatenated instances of `A`
  9. Iteration **binds stronger** than **concatenation** which **binds stronger** than **choice**, so
     - `A|(B(C)*)=`

`A|BC*`

# grep support both unix and POSIX regex ( -E)

```
$ cat file
   (aap)
    aap

$ grep (aap) file
   (aap)
```

Why: in unix regex ()are not metacharacters

```
$ grep -E (aap) file
   (aap)
    aap
```

Why: in POSIX regex () metacharacters

```
$ cat file
    not
   noot
   nooot

$ grep o\{2\} file
    noot
    nooot

$ grep -E o{3} file
    nooot
```

# More Examples

| regex | | |
|---|---|---|
| `A.` | A9, Aa, AA | aA, AAA |
| `a.c` | abc, aac, a4c, a+c | ABC, abcd, abbc |
| `a\.c` | a.c | abc |
| `.ap` | aap, lap, hap | |
| `[al]ap` | aap, lap | |
| `[^al]ap` | hap, kap | aap, lap |
| `[al]+ap` | aap, lap, aaap, alap, laap, llap | |
| `[^A-Z]` | 5, b | A, Q, W |
| `[abc]*` | aaab, cba | |
| `Iets.*` | lets, lets is beter dan niets | |
| `Iets.+` | lets is beter dan niets | lets |
| `[ab]{4}` | abba, baba | aba |
| `a(bc)*d` | ad, abcd, abcbcd | abcxd, bcbcd |

# Dutch Examples

- Telephone number (land line) → `020-5253705`

  `^[-0-9+() ]*$`

  `^[0-9]{3,4}-[0-9]{6,7}$`

  (it has a false positive it allows phone number with 11 numbers not valid in the Netherlands, try to fix this)

- Postal code → 1098XH or 1098 XH

  `^[1-9]{1}[0-9]{3}[:space:]?[A-Z]{2}`

# More Examples

Email → E.P.Schatborn@uva.nl

```
[A-Za-z0-9_-]+([.]{1}[A-Za-z0-9_-]+)*@[A-
  Za-z0-9-]+([.]{1}[A-Za-z0-9-]+)+
```

Mobile → 06-20369972

```
^06[-]?[0-9]{8}$
```

# One More Example

```
$ cat sedscr
s/\([A-Z][A-Z]*\){\([^}]*\)}/\1<\2>/

$ cat b
a{ab} A{d}
aBc{aba} ABC(a) ABC{a}ab

$ sed -f sedscr b
a{ab} A<d>
aBc{aba} ABC(a) ABC<a>ab
```

# Online tool for writing regex

- Practice regular expression http://www.ccl.net/cgi-bin/ccl/regexp/test_re.pl
- http://www.rubular.com
- http://regexpal.com
- http://gskinner.com/RegExr/