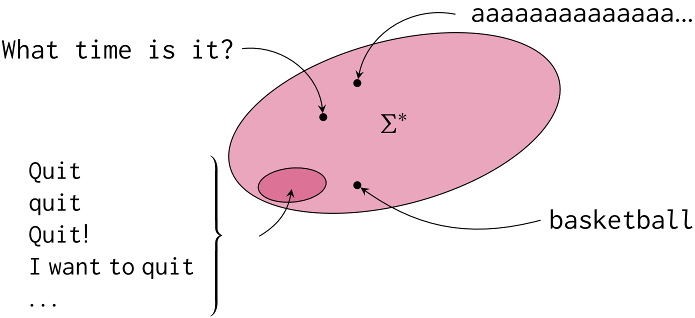
More regular expressions

Computational Linguistics (LING 455)

Rutgers University October 8, 2021

REcap

REs pick out sets of strings



What is a regular expression?

A regular expression given some starting alphabet Σ is:

- $\emptyset$ matches **nothing**
- ε matches **the empty string**
- c matches a **character** in Σ
- $r \cdot s$ the **concatenation** of two REs
- $r \mid s$ a **choice** between two REs
- r^* zero or more **repetitions** of an RE

The language of `Regexp`'s in Haskell

```
data Regexp = Zero                  -- nothing (0)
            | One                   -- empty string (eps)
            | Lit  Char             -- single character
            | Cat  Regexp Regexp    -- concatenation (.)
            | Plus Regexp Regexp    -- union (|)
            | Star Regexp           -- repetition (*)
            deriving Show
```

These would be very annoying to write and work with directly, so I have set things up so that you can use *string syntax* to specify a RE:

```
*W6> "a" :: Regexp
Lit 'a'
*W6> "ab" :: Regexp
Cat (Lit 'a') (Lit 'b')
```

Constructing `Regexp`'s in Haskell

Regexp	**Haskell**
$\emptyset$	`zero`
ε	`one`
$r \cdot s$	`r <.> s`
$r \mid s$	`r <|> s`
r^*	`star r`

```
*W6> star "re"
Star (Cat (Lit 'r') (Lit 'e'))

*W6> ("a" <|> "b") <.> "c"
Cat (Plus (Lit 'a') (Lit 'b')) (Lit 'c')
```

Practice

What patterns do these RE's express?

- $r \mid \varepsilon$

Practice

What patterns do these RE's express?

- $r \mid \varepsilon$ optionally r
- $r \cdot r^*$

Practice

What patterns do these RE's express?

- $r \mid \varepsilon$ optionally r
- $r \cdot r^*$ at least one r

Practice

What patterns do these RE's express?

- $r \mid \varepsilon$ optionally r
- $r \cdot r^*$ at least one r

We may define such operators for ourselves as conveniences!

```
optionally :: Regexp -> Regexp
optionally r = r <|> one

starPlus :: Regexp -> Regexp
starPlus r = r <.> star r
```

```
*W6> take 5 (mset ("hell" <.> starPlus "o" <.> "?"))
["hello?","helloo?","hellooo?","helloooo?","hellooooo?"]
```

More practice

What patterns do the following RE's express?

- $\mathsf{a} \cdot (\mathsf{a} \cdot \mathsf{a})^*$

More practice

What patterns do the following RE's express?

- $\mathsf{a} \cdot (\mathsf{a} \cdot \mathsf{a})^*$ odd numbers of a's
- $(\mathsf{a} \mid \mathsf{b})^* \cdot \mathsf{b} \cdot (\mathsf{a} \mid \mathsf{b})^*$

More practice

What patterns do the following RE's express?

- $\mathsf{a} \cdot (\mathsf{a} \cdot \mathsf{a})^*$ odd numbers of a's
- $(\mathsf{a} \mid \mathsf{b})^* \cdot \mathsf{b} \cdot (\mathsf{a} \mid \mathsf{b})^*$ strings of a's and b's with $\geqslant 1$ b
- $(C^* \cdot V \cdot C^* \cdot V \cdot C^*)^*$

More practice

What patterns do the following RE's express?

- $\mathsf{a} \cdot (\mathsf{a} \cdot \mathsf{a})^*$ — odd numbers of a's
- $(\mathsf{a} \mid \mathsf{b})^* \cdot \mathsf{b} \cdot (\mathsf{a} \mid \mathsf{b})^*$ — strings of a's and b's with $\geqslant$1 b
- $(C^* \cdot V \cdot C^* \cdot V \cdot C^*)^*$ — strings of C's and V's with even # V's

```
*W6> take 5 (mset ("a" <.> star "aa"))
["a","aaa","aaaaa","aaaaaaa","aaaaaaaaa"]

*W6> r = star ("a" <|> "b") <.> "b" <.> star ("a" <|> "b")
*W6> match r "aaaaaaaba"
True

*W6> s = star (star "C" <.> "V" <.> star "C" <.> "V" <.> star "C")
*W6> match s "CVCVCVCCV"
True
```

Matching as parsing

Matching sets, given by $[\![\cdot]\!]$

$[\![\text{RE}]\!]$	**Set of strings**
$[\![\emptyset]\!]$	$= \{\ \}$
$[\![\varepsilon]\!]$	$= \{\varepsilon\}$
$[\![c]\!]$	$= \{c\}$
$[\![r_1 \cdot r_2]\!]$	$= \{u \mathbin{+\!\!+} v \mid u \in [\![r_1]\!], v \in [\![r_2]\!]\}$
$[\![r_1 \mid r_2]\!]$	$= [\![r_1]\!] \cup [\![r_2]\!]$
$[\![r^*]\!]$	$= [\![\varepsilon]\!] \cup [\![r]\!] \cup [\![r \cdot r]\!] \cup [\![r \cdot r \cdot r]\!] \cup \ldots$

```
*W6> let r = star (("a" <|> "b") <.> "c")
*W6> take 5 (mset r)
["","ac","bc","acac","acbc"]
*W6> elem "bcac" (mset r)
True
```

Extending RE syntax

It is often useful to talk about anti-matches for an RE:

- $\overline{r}$ the strings that **don't** match r

Given this specification, the meaning of $\overline{r}$ is easy to characterize:

- $[\![\overline{r}]\!] = \{u \in \Sigma^* \mid u \notin [\![r]\!]\}$ (remember: Σ^* is the strings over Σ)

Or we could define some shorthand for n repetitions of r, with the obvious meaning in terms of matching sets:

- $[\![r\{n\}]\!] = [\![r_1 \cdot r_2 \cdot \ldots \cdot r_n]\!]$

Challenge exercise: how would you extend `Regexp.hs` to handle such cases, along the lines of `starPlus` and `optionally`?

Matching sets

Formally useful, though computationally pretty infeasible:

- Enumerating matches is highly inefficient:

```
*W6> elem "abcd" (mset (star anyc))
True
(48.61 secs, 46,031,685,560 bytes)
```

- If no match in an infinite matching set, `elem` searches forever:

```
*W6> elem "x" (mset (star "a"))
-- loops endlessly
^CInterrupted.
```

Instead of mindlessly enumerating all matches for the `Regexp`, we should begin with the `String` to be matched against the `Regexp`.

- It's trivial to see that `"x"` is neither `""` nor a sequence of `"a"`'s

More efficient matching with `match`

```
*W6> :t match
match :: Regexp -> String -> Bool

*W6> match (star "a") "x"
False
*W6> match (star anyc) "aslkdjal k$/dj$kd!jaskldjaskl."
True

*W6> let r10 = star (("a" <|> "b") <.> "c")
*W6> match r10 "acacbcbcacacbcbc"
True

*W6> let r2 = "<b>" <.> star anyc <.> "</b>"
*W6> match r2 "<b>some bolded text</b>"
True
```

Defining `match`

Much of `match` is straightforward to define:

```
match :: Regexp -> String -> Bool
match Zero         _ = False
match One          u = u==""
match (Lit c)      u = u==[c]
match (Plus r s) u = match r u || match s u
```

Defining `match`

Much of `match` is straightforward to define:

```
match :: Regexp -> String -> Bool
match Zero         _ = False
match One          u = u==""
match (Lit c)      u = u==[c]
match (Plus r s) u = match r u || match s u
```

The trickier cases are `Cat` and `Star`:

```
match (Cat  r s) u = undefined -- ??
match (Star r) ""  = True
match (Star r) u   = undefined -- ??
```

Matching a `Cat`

The tricky thing about `Cat` is that `u` matches `Cat r s` if `u` can be cut into two parts matching `r` and `s`. *But where should we make the cut?*

`"abc"` should `match` **all** of the following `Regexp`'s (and more!):

```
 ""   <.> "abc"
 "a"  <.>  "bc"
 "ab" <.>   "c"
"abc" <.>    ""
```

Making all the cuts with `splitAt` (in Prelude)

```
*W6> splitAt 0 "abc"
("","abc")
*W6> splitAt 1 "abc"
("a","bc")
*W6> splitAt 2 "abc"
("ab","c")
*W6> splitAt 3 "abc"
("abc","")
```

```
splits :: String -> [(String, String)]
splits u = [splitAt i u | i <- [0..length u]]
```

```
*W6> splits "abc"
[("","abc"),("a","bc"),("ab","c"),("abc","")]
--r? s?       r?  s?      r?   s?     r?     s?
```

Matching a `Cat` (cont)

```
*W6> splits "abc"
[("","abc"),("a","bc"),("ab","c"),("abc","")]
--r? s?       r?  s?       r?   s?     r?     s?
```

Once we make all the cuts, `u` matches `Cat r s` if one of those cuts gives a left-half matching `r` and a right-half matching `s`:

```
match (Cat r s) u =
  or [ match r v && match s w | (v,w) <- splits u ]
```

`or ts` is `True` if there's any `True`'s in `ts` — if one of the cuts made by `splits` yields a left-match and a right-match.

An example

Let's walk through how `match ("ab" <.> "c") "abc"` is checked:

- Enumerate all the `splits` of `"abc"`
- Is there a `(v,w)` s.t. `match "ab" v && match "c" w` is `True`?
  - There is! Among `splits "abc"` is `("ab","c")`

In fact, our string-y notation for `Regexp`'s is so handy that it's easy to lose sight of the fact that `match "ab" "ab"` involves some work:

- Used as a `Regexp`, `"ab"` stands for `Cat (Lit 'a') (Lit 'b')`!
- So we must also check that `"ab"` can be cut into a match for `Lit 'a'` and a match for `Lit 'b'`. Of course, it can.

Matching a `Star`

`""` always matches `Star r`. But how about non-empty `String`'s?

```
match (Star r) "" = True
match (Star r) u  = undefined -- ??
```

The logic is similar to `Cat`: if we can split up `u` into a first part that matches `r`, and a second part that matches `Star r`, we match!

```
match (Star r) u = or [ match r v && match (Star r) w
                      | (v,w) <- tail (splits u) ]
```

Challenge exercise: figure out why `tail` is required here.

Another example

Let's walk through how `match (star "a") "aa"` is checked:

- Enumerate all the `splits` of `"aa"`
- Is there a `(v,w)` s.t. `match "a" v && match (star "a") w`?
  - There is! The first thing in `("a","a")` matches `"a"`.
  - The second is a match for `Star "a"`, since it can be split into `("a","")`, whose first thing matches `"a"` and whose second matches `Star "a"`.

Why does `match` do better than `elem ... mset ...`?

The `mset` strategy **constructs** all the matches from the RE.

- In an infinite set of matches, we can never be sure we've looked long enough for a non-matching string.
- In a very big set of matches, we might have to plow through a ton of irrelevant matching strings before we get where we need.

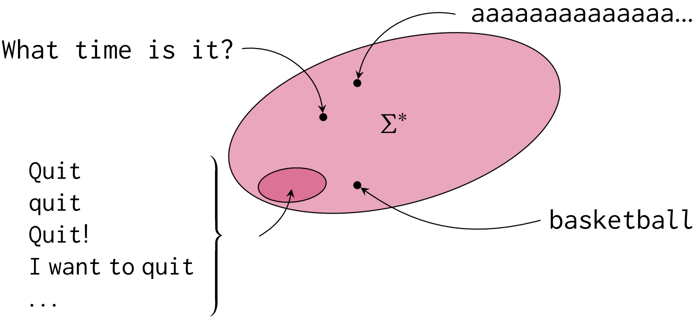
`match` works differently — it **deconstructs** the `String` using `splits`:

- `match (Star "a") "x"` $\Longrightarrow^*$ `... match "a" "x" ...` ☠

Deconstructing a potential match (instead of enumerating a haystack and looking for a needle) is known as **parsing**.

REs and natural language

REs pick out sets of strings



String sets and natural language

What else picks out sets of strings? The human language faculty!

For example, some strings of sounds are recognized as licit by English speakers, and others are not. This is known as **phonotactics**:

a.	thole	e.	mgla
b.	fslux	f.	plast
c.	msuklha	g.	flitch
d.	rtut	h.	dnom

String sets and natural language

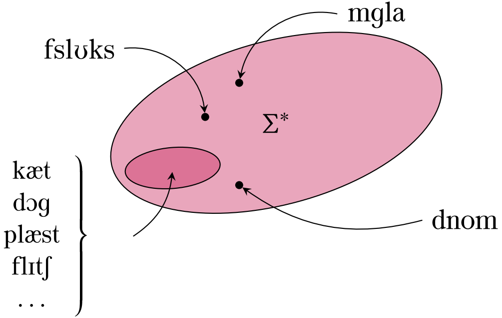
What else picks out sets of strings? The human language faculty!

For example, some strings of sounds are recognized as licit by English speakers, and others are not. This is known as **phonotactics**:

a.	thole	e.	mgla
b.	fslux	f.	plast
c.	msuklha	g.	flitch
d.	rtut	h.	dnom

Phonological strings

Speakers distinguish **the set of possible strings of segments** in their language from impossible ones:



$\Sigma = \{$m,p,b,ŋ,f,v,θ,…,i,ɪ,e,ɛ,…$\}$ (the IPA)

REs for phonology

Can we write a RE for English words?

- Yes! Phonology is **regular** (Heinz and Idsardi, *Science*, 2011)

Building up a (partial) RE for English syllables:

- $(\mathrm{p} \mid \mathrm{t} \mid \mathrm{k} \mid \varepsilon) \cdot V \cdot (\mathrm{p} \mid \mathrm{t} \mid \mathrm{k} \mid \mathrm{m} \mid \mathrm{n} \mid \text{ŋ} \mid \varepsilon)$
  {kæt,piːk, …}

Where V abbreviates the RE for some vowel or other:

- $V =$ iː | ɪ | eː | ɛ | æ | uː | ʊ | oː | ɔ | ɑ

REs for phonology

Can we write a RE for English words?

- Yes! Phonology is **regular** (Heinz and Idsardi, *Science*, 2011)

Building up a (partial) RE for English syllables:

- $(\text{s} \mid \varepsilon) \cdot (\text{p} \mid \text{t} \mid \text{k} \mid \varepsilon) \cdot V \cdot (\text{p} \mid \text{t} \mid \text{k} \mid \text{m} \mid \text{n} \mid \text{ŋ} \mid \varepsilon)$
  {kæt,piːk,skɪm,stɛp, …}

Where V abbreviates the RE for some vowel or other:

- $V = \text{iː} \mid \text{ɪ} \mid \text{eː} \mid \text{ɛ} \mid \text{æ} \mid \text{uː} \mid \text{ʊ} \mid \text{oː} \mid \text{ɔ} \mid \text{ɑ}$

REs for phonology

Can we write a RE for English words?

- Yes! Phonology is **regular** (Heinz and Idsardi, *Science*, 2011)

Building up a (partial) RE for English syllables:

- $(\mathrm{s} \mid \varepsilon) \cdot (\mathrm{p} \mid \mathrm{t} \mid \mathrm{k} \mid \varepsilon) \cdot (\mathrm{r} \mid \varepsilon) \cdot V \cdot (\mathrm{p} \mid \mathrm{t} \mid \mathrm{k} \mid \mathrm{m} \mid \mathrm{n} \mid \text{ŋ} \mid \varepsilon)$
  {kæt,piːk,skɪm,stɛp,strɪŋ,kriːm, …}

Where V abbreviates the RE for some vowel or other:

- $V =$ iː | ɪ | eː | ɛ | æ | uː | ʊ | oː | ɔ | ɑ

The regularity of phonology

All known phonological patterns are regular!

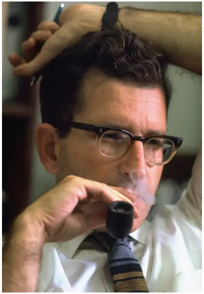
- This has influenced computational approaches to phonology

This is a subject of recent research in phonological theory: might phonology be *sub-regular* — less complex even than REs?

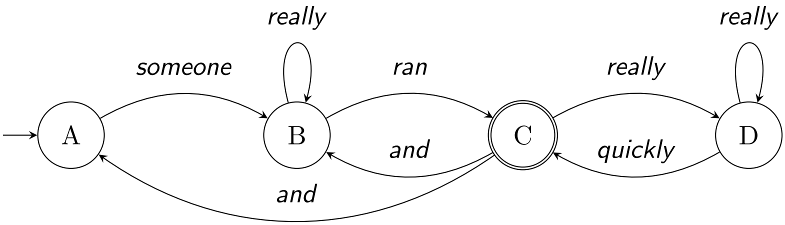
The **non**-regularity of syntax

Noam Chomsky (1929–)

- **Not all** patterns in language are regular
- Chomsky in 1956: There are syntactic patterns that are not regular

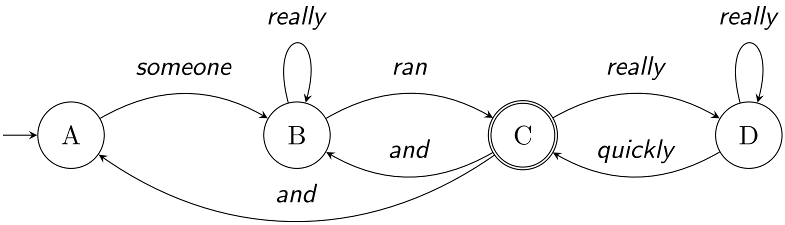


REs for syntax?



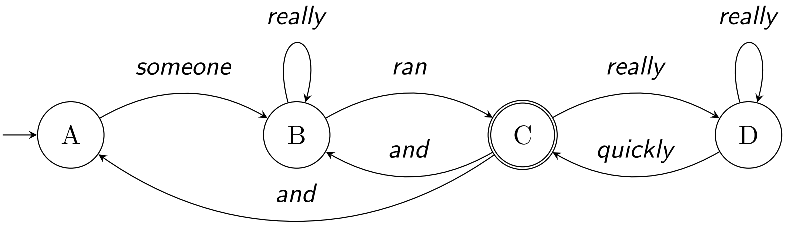
someone ·

REs for syntax?



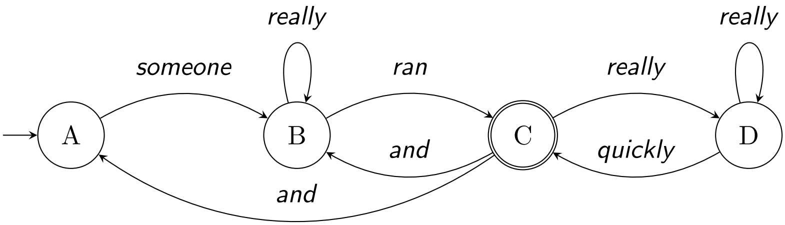
$$\text{someone} \cdot \text{really}^* \cdot$$

REs for syntax?



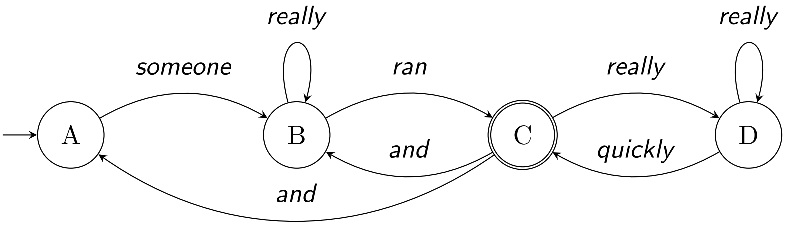
$$\text{someone} \cdot \text{really}^* \cdot \text{ran} \cdot$$

REs for syntax?



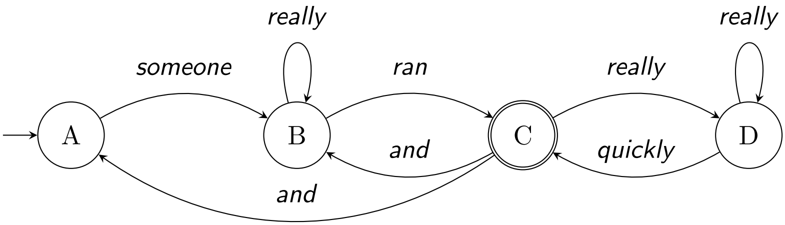
$$\text{someone} \cdot \text{ really}^* \cdot \text{ran} \cdot (\text{really} \cdot$$

REs for syntax?



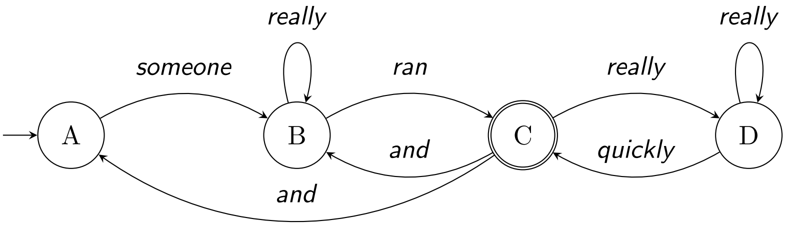
$$\text{someone} \cdot \text{ really}^* \cdot \text{ran} \cdot (\text{really} \cdot \text{really}^* \cdot$$

REs for syntax?



$$\text{someone} \cdot \underbrace{\text{really}^* \cdot \text{ran} \cdot (\text{really} \cdot \text{really}^* \cdot \text{quickly})^*}_{\mathbf{r}}$$

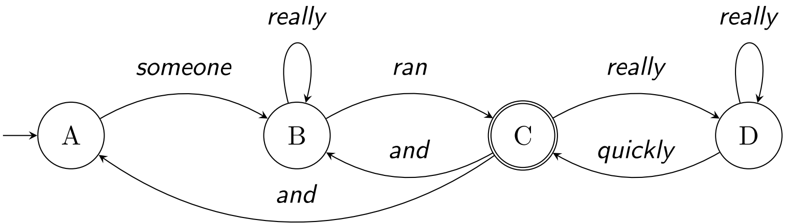
REs for syntax?



$$\text{someone} \cdot \underbrace{\text{really}^* \cdot \text{ran} \cdot (\text{really} \cdot \text{really}^* \cdot \text{quickly})^*}_{\mathbf{r}}$$

$$\underbrace{\text{someone} \cdot \mathbf{r} \cdot (\text{and} \cdot \mathbf{r})^*}_{\mathbf{s}}$$

REs for syntax?



$$\text{someone} \cdot \underbrace{\text{really}^* \cdot \text{ran} \cdot (\text{really} \cdot \text{really}^* \cdot \text{quickly})^*}_{\mathbf{r}}$$

$$\underbrace{\text{someone} \cdot \mathbf{r} \cdot (\text{and} \cdot \mathbf{r})^*}_{\mathbf{s}}$$

$$\mathbf{s} \cdot (\text{and} \cdot \mathbf{s})^*$$

Center embedding

The cat ran.

Center embedding

The cat	ran.
NP	VP

Center embedding

The cat	the dog	chased	ran.
NP	NP	VP	VP

Center embedding

The cat	the dog	Sam	owns	chased	ran.
NP	NP	NP	VP	VP	VP

Center embedding

	The cat	the dog	Sam	owns	chased	ran.
	NP	NP	NP	VP	VP	VP
*	The cat	the dog	Sam	owns		ran.
	NP	NP	NP	VP		VP

$a^n \cdot b^n$

Center embedding ($\Sigma = \{\text{NP}, \text{VP}\}$):

$$\underbrace{\text{NP} \quad \text{NP} \quad \text{NP} \quad \ldots}_{n} \; \underbrace{\ldots \quad \text{VP} \quad \text{VP} \quad \text{VP}}_{n}$$

$a^n \cdot b^n$

Center embedding ($\Sigma = \{\mathrm{NP}, \mathrm{VP}\}$):

$$\underbrace{\mathrm{NP} \quad \mathrm{NP} \quad \mathrm{NP} \quad \ldots}_{n} \; \underbrace{\ldots \quad \mathrm{VP} \quad \mathrm{VP} \quad \mathrm{VP}}_{n}$$

There is no regular expression for this set of strings!

- The matching set $\{a^n \cdot b^n \mid n \geqslant 1\}$ isn't generated by any RE!

There are many examples of such patterns in natural language syntax:

- *Either … or …*
- *Both … and …*
- *If … then …*
- *The woman who … saw …*

Why not?

Finite languages can always be enumerated. It could get tedious.

- Infinite lgs need $*$ somewhere. Maybe the short-ish strings don't, but eventually, you **will** get a string so long it could have only been generated using $*$. (And so for longer strings.)
- A RE can be as long as you want, but it will always end.
- The non-$*$ parts of any RE always generate some longest string.

Why not?

Finite languages can always be enumerated. It could get tedious.

- Infinite lgs need $*$ somewhere. Maybe the short-ish strings don't, but eventually, you **will** get a string so long it could have only been generated using $*$. (And so for longer strings.)
- A RE can be as long as you want, but it will always end.
- The non-$*$ parts of any RE always generate some longest string.

Call the minimal $*$-requiring length for some RE p. All strings at least as long as p used $*$, and hence have a part that can be repeated.

- Example: for $\mathsf{a} \cdot \mathsf{b}^* \cdot \mathsf{c}$, $p = 3$. Repeatable part: b.
- Every (non-finite) regular language has such a p, because every (non-finite) regular language relies on $*$ somewhere.

Why not? (cont)

Can there be such a p for $\mathcal{L} = \{a^n \cdot b^n \mid n \geqslant 1\}$?

- Maybe there could. Let's see where this leads us.
- But *now* consider $a^p \cdot b^p$. It must have *some* part that can be repeated since its length is $2p$, and $2p > p$.

What could the repeatable part be though? a? b? Some substring drawn straight down the middle with equal numbers of a's and b's?

- $a^p \cdot a \cdot b^p$ — repeating a once $\longrightarrow$ unbalanced
- $a^p \cdot b \cdot b^p$ — repeating b once $\longrightarrow$ unbalanced
- $a^p \cdot a \cdot b \cdot a \cdot b \cdot b^p$ — repeating $a \cdot b$ twice $\longrightarrow$ alternation

Any possible thing we could repeat (eventually) takes us outside $\mathcal{L}$. We must conclude there can be no p for $\mathcal{L}$. It's not regular.

The pumping lemma

This is the **pumping lemma**:

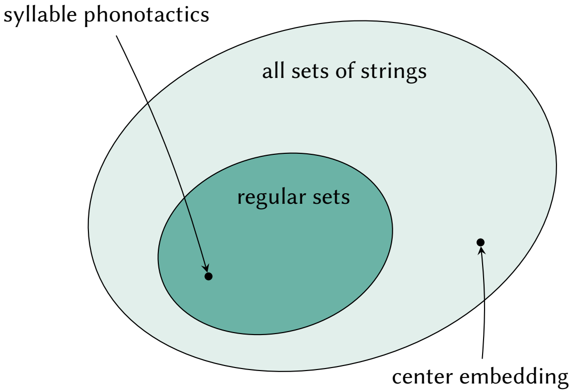
- Any (non-trivial, i.e., infinite) regular language eventually uses $*$
- At that point, the sub-string can be **pumped** (iterated/repeated)

Recall that a fundamental property of human languages is their **unboundedness** and their allowance for **creativity**.

- Human languages are by their nature **infinite**!

The way that REs allow unboundedness/creativity is via $*$, but this is not sufficient to capture the ways natural languages can work.

Computationally, syntax and phonology are different



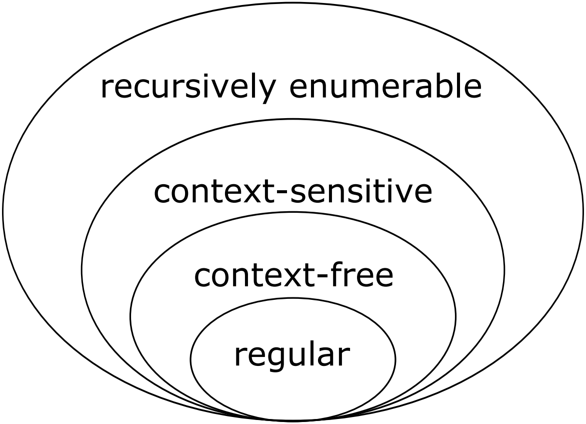
Relative complexity

Syntactic patterns are *more complex* than phonological patterns:

- Phonological patterns can be expressed with REs
- Syntactic patterns can't be. Syntax requires a more powerful tool for specifying patterns/languages.

This tool, as we'll see later, is **Context-Free Grammars** (CFGs).

The Chomsky Hierarchy (via wiki)



Wrapping up REs

Regular expressions are a powerful tool for **characterizing string patterns**, and **defining regular languages**…

- REs come from **formal language theory**, which is foundational to both computer science and linguistics
- REs find wide application in computational linguistics and programming/scripting in general

REs in computational linguistics

Regular expressions also illuminate the computational nature of the patterns that make up different parts of our linguistic competence:

- Phonological patterns can be captured with regular expressions
- But not all syntactic patterns can!

Next in the course

We will discuss **finite state (FS) machines**. These are closely related to both the *n*-gram models we considered previously, and to REs:

- *n*-gram models are a specific kind of FS machine
- REs and FS machines are **equivalent**! They are capable of generating exactly the same languages!