Regular expressions

Computational Linguistics (LING 455)

Rutgers University

Sep 28 & Oct 1, 2021

Patterns in text

Suppose you are were (writing) a chatbot

```
> Do you want to quit the chat?
```

```
import Regexp
```

```
*W5> r1 = ("Q" <|> "q") <.> "uit" <.> star anyc
*W5> match r1 "Quit"
True
*W5> match r1 "quit now!!!!!!!!!"
True
*W5> match r1 "Hi :)"
False
```

…Or you wanted to scrub bold text from a URL

```
*W5> r2 = "<b>" <.> star anyc <.> "</b>"
*W5> match r2 "<b>some bolded text</b>"
True
*W5> match r2 "<i>some italic text</i>"
False
```

…Or find the lines in an annotated XML Shakespeare corpus:

```
*W5> r3 = "<LINE>" <.> star anyc <.> "</LINE>"
*W5> match r3 "<LINE>To be, or not to be:</LINE>"
True
```

String patterns

Regular expressions (REs, regexps) help characterize string **patterns**.

- REs give a compact syntax for describing a **set of strings**
- This has obvious practical utility, but also some deeper upshots.

What is a **language**, after all, but a pattern?

- We will see later that REs can be used to describe certain parts of our capacity language, and certain ways they fall short

Alphabet and star-closures

An **alphabet** is a fixed set of symbols

- 0, 1 — binary digits
- ASCII (␣, !, … A, B, … Z, … a, b, …) — text
- A, C, G, T — DNA
- NP, VP, S, CP, … — syntactic categories
- C, V — syllable structure

Alphabet and star-closures

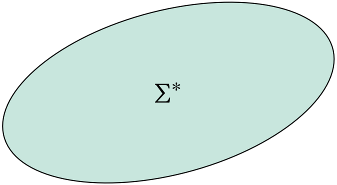
An **alphabet** is a fixed set of symbols

- 0, 1 — binary digits
- ASCII (␣, !, … A, B, … Z, … a, b, …) — text
- A, C, G, T — DNA
- NP, VP, S, CP, … — syntactic categories
- C, V — syllable structure

Given an alphabet Σ, Σ^* is all of the strings built from ‘letters’ in Σ:

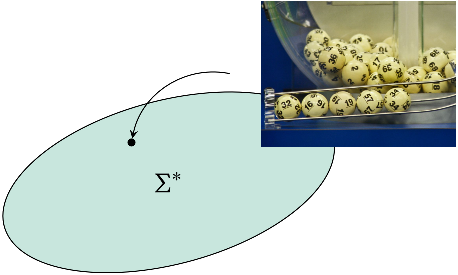
- 101, 10000001, 100100100, …
- `I am a String!`, `aslkehlqw;lsadj`, `To be or not to be`, …
- AGTAGCTATAG, TGAGAGACAATA, GATTACA, …
- NP CP VP PP, …
- CV, CVC, CCVC, …

If Σ is ASCII, Σ^* includes…



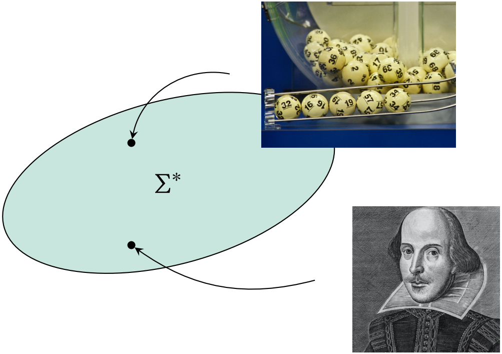
If Σ is ASCII, Σ^* includes...

- the next winning powerball numbers



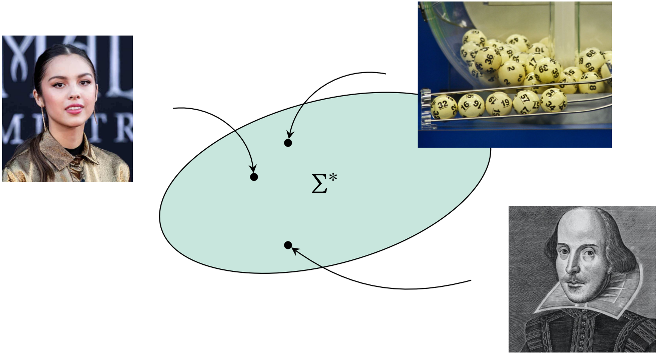
If Σ is ASCII, Σ^* includes…

- the next winning powerball numbers
- the complete works of Shakespeare



If Σ is ASCII, Σ^* includes…

- the next winning powerball numbers
- the complete works of Shakespeare
- the lyrics to every song Olivia Rodrigo will ever write



A short Haskell program that computes ASCII*

```
*W5> mset (star anyc)
-- off we go!
```

If I let this run forever, eventually it'd stumble upon everything that could be written, said, or thought…

A short Haskell program that computes ASCII*

```
*W5> mset (star anyc)
-- off we go!
```

If I let this run forever, eventually it'd stumble upon everything that could be written, said, or thought…

The signal and the noise

This is no exaggeration: ASCII* really does contain all of these things. **BUT** that doesn't mean that much, in any practical sense, because these needles are buried in an enormous haystack.

- ASCII* is essentially Borges' *Library of Babel* (explore here)

What we want is a way to filter out all of the noise, and concentrate on the meaningful or relevant stuff (given some task or goal).

- This is what Regular Expressions help us do.
- They allow us to describe **patterns** in strings, and what is a pattern but a way of focusing on something that *matters*?

```
*W5> r2 = "<b>" <.> star anyc <.> "</b>"
--                 ^^^^^^^^^ library of babel
```

Regular expressions as languages

A language is, from one point of view, just a set of strings:

- Certain strings count as English: `I walked the dog`
- Certain strings do not: `Dog I the walked`

The pattern that a RE describes is a (sometimes very small) language.
And operations on multiple REs can be used to form new languages!

Regexps

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

Examples

Suppose our starting alphabet is the lowercase letters, $\Sigma = \{\mathsf{a}, \ldots, \mathsf{z}\}$:

Regexp	**Matching strings**
b	

[1] Note: here I suppress some concatenation dots for readability. As we will see, we can take advantage of a similar notational shortcut working with REs in Haskell.

Examples

Suppose our starting alphabet is the lowercase letters, $\Sigma = \{\mathsf{a}, \ldots, \mathsf{z}\}$:

Regexp	**Matching strings**
b	b
$\mathsf{b} \cdot \mathsf{c}$	

[1] Note: here I suppress some concatenation dots for readability. As we will see, we can take advantage of a similar notational shortcut working with REs in Haskell.

Examples

Suppose our starting alphabet is the lowercase letters, $\Sigma = \{\mathsf{a}, \ldots, \mathsf{z}\}$:

Regexp	**Matching strings**
b	b
$\mathsf{b} \cdot \mathsf{c}$	bc
$(\mathsf{a} \mid \mathsf{b}) \cdot \mathsf{c}$	

[1] Note: here I suppress some concatenation dots for readability. As we will see, we can take advantage of a similar notational shortcut working with REs in Haskell.

Examples

Suppose our starting alphabet is the lowercase letters, $\Sigma = \{\mathsf{a}, \ldots, \mathsf{z}\}$:

Regexp	**Matching strings**
b	b
$\mathsf{b} \cdot \mathsf{c}$	bc
$(\mathsf{a} \mid \mathsf{b}) \cdot \mathsf{c}$	ac, bc
$((\mathsf{a} \mid \mathsf{b}) \cdot \mathsf{c})^*$	

[1] Note: here I suppress some concatenation dots for readability. As we will see, we can take advantage of a similar notational shortcut working with REs in Haskell.

Examples

Suppose our starting alphabet is the lowercase letters, $\Sigma = \{\mathsf{a}, \ldots, \mathsf{z}\}$:

Regexp	**Matching strings**
b	b
$\mathsf{b} \cdot \mathsf{c}$	bc
$(\mathsf{a} \mid \mathsf{b}) \cdot \mathsf{c}$	ac, bc
$((\mathsf{a} \mid \mathsf{b}) \cdot \mathsf{c})^*$	$\varepsilon, \mathsf{ac}, \mathsf{bc}, \mathsf{acac}, \mathsf{acbc}, \mathsf{bcac}, \ldots$
$\emptyset \cdot ((\mathsf{a} \mid \mathsf{b}) \cdot \mathsf{c})^*$	

[1] Note: here I suppress some concatenation dots for readability. As we will see, we can take advantage of a similar notational shortcut working with REs in Haskell.

Examples

Suppose our starting alphabet is the lowercase letters, $\Sigma = \{a, \ldots, z\}$:

Regexp	Matching strings
b	b
$b \cdot c$	bc
$(a \mid b) \cdot c$	ac, bc
$((a \mid b) \cdot c)^*$	$\varepsilon, ac, bc, acac, acbc, bcac, \ldots$
$\emptyset \cdot ((a \mid b) \cdot c)^*$	
$\varepsilon \cdot ((a \mid b) \cdot c)^*$	

[1] Note: here I suppress some concatenation dots for readability. As we will see, we can take advantage of a similar notational shortcut working with REs in Haskell.

Examples

Suppose our starting alphabet is the lowercase letters, $\Sigma = \{\mathsf{a}, \dots, \mathsf{z}\}$:

Regexp	**Matching strings**
b	b
$\mathsf{b} \cdot \mathsf{c}$	bc
$(\mathsf{a} \mid \mathsf{b}) \cdot \mathsf{c}$	ac, bc
$((\mathsf{a} \mid \mathsf{b}) \cdot \mathsf{c})^*$	$\varepsilon, \mathsf{ac}, \mathsf{bc}, \mathsf{acac}, \mathsf{acbc}, \mathsf{bcac}, \dots$
$\emptyset \cdot ((\mathsf{a} \mid \mathsf{b}) \cdot \mathsf{c})^*$	
$\varepsilon \cdot ((\mathsf{a} \mid \mathsf{b}) \cdot \mathsf{c})^*$	$\varepsilon, \mathsf{ac}, \mathsf{bc}, \mathsf{acac}, \mathsf{acbc}, \mathsf{bcac}, \dots$
$\Sigma^* \cdot \mathsf{s}$	

[1] Note: here I suppress some concatenation dots for readability. As we will see, we can take advantage of a similar notational shortcut working with REs in Haskell.

Examples

Suppose our starting alphabet is the lowercase letters, $\Sigma = \{\mathsf{a}, \ldots, \mathsf{z}\}$:

Regexp	**Matching strings**
b	b
$\mathsf{b} \cdot \mathsf{c}$	bc
$(\mathsf{a} \mid \mathsf{b}) \cdot \mathsf{c}$	ac, bc
$((\mathsf{a} \mid \mathsf{b}) \cdot \mathsf{c})^*$	$\varepsilon, \mathsf{ac}, \mathsf{bc}, \mathsf{acac}, \mathsf{acbc}, \mathsf{bcac}, \ldots$
$\emptyset \cdot ((\mathsf{a} \mid \mathsf{b}) \cdot \mathsf{c})^*$	
$\varepsilon \cdot ((\mathsf{a} \mid \mathsf{b}) \cdot \mathsf{c})^*$	$\varepsilon, \mathsf{ac}, \mathsf{bc}, \mathsf{acac}, \mathsf{acbc}, \mathsf{bcac}, \ldots$
$\Sigma^* \cdot \mathsf{s}$	s, `matches`, `facts`, `zxcnbcxbcs`, $\ldots$
$\mathsf{a} \cdot \Sigma^* \cdot \mathsf{a}$	

[1] Note: here I suppress some concatenation dots for readability. As we will see, we can take advantage of a similar notational shortcut working with REs in Haskell.

Examples

Suppose our starting alphabet is the lowercase letters, $\Sigma = \{\mathsf{a}, \ldots, \mathsf{z}\}$:

Regexp	**Matching strings**
b	b
$\mathsf{b} \cdot \mathsf{c}$	bc
$(\mathsf{a} \mid \mathsf{b}) \cdot \mathsf{c}$	ac, bc
$((\mathsf{a} \mid \mathsf{b}) \cdot \mathsf{c})^*$	ε, ac, bc, acac, acbc, bcac, ...
$\emptyset \cdot ((\mathsf{a} \mid \mathsf{b}) \cdot \mathsf{c})^*$	
$\varepsilon \cdot ((\mathsf{a} \mid \mathsf{b}) \cdot \mathsf{c})^*$	ε, ac, bc, acac, acbc, bcac, ...
$\Sigma^* \cdot \mathsf{s}$	s, matches, facts, zxcnbcxbcs, ...
$\mathsf{a} \cdot \Sigma^* \cdot \mathsf{a}$	aa, alpaca, aqskwqieua, ...
$(\varepsilon \mid \mathsf{un}) \cdot \mathsf{box} \cdot (\mathsf{ed} \mid \mathsf{ing})$[1]	

[1] Note: here I suppress some concatenation dots for readability. As we will see, we can take advantage of a similar notational shortcut working with REs in Haskell.

Examples

Suppose our starting alphabet is the lowercase letters, $\Sigma = \{\mathtt{a}, \ldots, \mathtt{z}\}$:

Regexp	Matching strings
$\mathtt{b}$	b
$\mathtt{b} \cdot \mathtt{c}$	bc
$(\mathtt{a} \mid \mathtt{b}) \cdot \mathtt{c}$	ac, bc
$((\mathtt{a} \mid \mathtt{b}) \cdot \mathtt{c})^*$	ε, ac, bc, acac, acbc, bcac, ...
$\emptyset \cdot ((\mathtt{a} \mid \mathtt{b}) \cdot \mathtt{c})^*$	
$\varepsilon \cdot ((\mathtt{a} \mid \mathtt{b}) \cdot \mathtt{c})^*$	ε, ac, bc, acac, acbc, bcac, ...
$\Sigma^* \cdot \mathtt{s}$	s, matches, facts, zxcnbcxbcs, ...
$\mathtt{a} \cdot \Sigma^* \cdot \mathtt{a}$	aa, alpaca, aqskwqieua, ...
$(\varepsilon \mid \mathtt{un}) \cdot \mathtt{box} \cdot (\mathtt{ed} \mid \mathtt{ing})$[1]	boxed, boxing, unboxed, unboxing

[1] Note: here I suppress some concatenation dots for readability. As we will see, we can take advantage of a similar notational shortcut working with REs in Haskell.

The language of `Regexp`'s in Haskell

```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:

```
*W5> "a" :: Regexp
Lit 'a'
*W5> "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`

```
*W5> ("a" <|> "b") <.> "c"
Cat (Plus (Lit 'a') (Lit 'b')) (Lit 'c')

*W5> star ("re" <|> one)
Star (Plus (Cat (Lit 'r') (Lit 'e')) One)
```

More practice

Give Haskell code for these REs, and say what pattern they describe:[2]

Regexp	Haskell	Description
$(\mathtt{H} \cdot \mathtt{A})^*$		

[2] Note: string syntax allows us to use fewer dots. A `Regexp` equivalent to `star "HA"` is the more verbose expression: `star ("H" <.> "A")`. You can verify this yourself by entering both into `ghci` and viewing the desugared `Regexp` output.

More practice

Give Haskell code for these REs, and say what pattern they describe:[2]

Regexp	Haskell	Description
$(\mathtt{H} \cdot \mathtt{A})^*$	`star "HA"`	ε, HA, HAHA, ...
$\mathtt{b} \cdot (\mathtt{e} \mid \mathtt{i}) \cdot \mathtt{t}$		

[2] Note: string syntax allows us to use fewer dots. A `Regexp` equivalent to `star "HA"` is the more verbose expression: `star ("H" <.> "A")`. You can verify this yourself by entering both into `ghci` and viewing the desugared `Regexp` output.

More practice

Give Haskell code for these REs, and say what pattern they describe:[2]

Regexp	Haskell	Description
$(\mathtt{H} \cdot \mathtt{A})^*$	`star "HA"`	ε, HA, HAHA, ...
$\mathtt{b} \cdot (\mathtt{e} \mid \mathtt{i}) \cdot \mathtt{t}$	`"b" <.> ("e" <|> "i") <.> "t"`	bet, bit
$\varepsilon \mid (\mathtt{e} \cdot \mathtt{d})$		

[2] Note: string syntax allows us to use fewer dots. A `Regexp` equivalent to `star "HA"` is the more verbose expression: `star ("H" <.> "A")`. You can verify this yourself by entering both into `ghci` and viewing the desugared `Regexp` output.

More practice

Give Haskell code for these REs, and say what pattern they describe:[2]

Regexp	Haskell	Description
$(\mathtt{H} \cdot \mathtt{A})^*$	`star "HA"`	ε, HA, HAHA, ...
$\mathtt{b} \cdot (\mathtt{e} \mid \mathtt{i}) \cdot \mathtt{t}$	`"b" <.> ("e" <|> "i") <.> "t"`	bet, bit
$\varepsilon \mid (\mathtt{e} \cdot \mathtt{d})$	`one <|> "ed"`	ε, ed
$(\mathtt{b}^* \cdot (\mathtt{a} \cdot \mathtt{a})^* \cdot \mathtt{b}^*)^*$		

[2] Note: string syntax allows us to use fewer dots. A `Regexp` equivalent to `star "HA"` is the more verbose expression: `star ("H" <.> "A")`. You can verify this yourself by entering both into `ghci` and viewing the desugared `Regexp` output.

More practice

Give Haskell code for these REs, and say what pattern they describe:[2]

Regexp	Haskell	Description
$(\mathtt{H} \cdot \mathtt{A})^*$	`star "HA"`	ε, HA, HAHA, ...
$\mathtt{b} \cdot (\mathtt{e} \mid \mathtt{i}) \cdot \mathtt{t}$	`"b" <.> ("e" <|> "i") <.> "t"`	bet, bit
$\varepsilon \mid (\mathtt{e} \cdot \mathtt{d})$	`one <|> "ed"`	ε, ed
$(\mathtt{b}^* \cdot (\mathtt{a} \cdot \mathtt{a})^* \cdot \mathtt{b}^*)^*$	`star (star "b" <.> star ("aa") <.> star "b")`	even # a's only

`Regexp.hs` also defines some helpful abbreviations for you:

- `anyc` is any character $\quad$ `! | ... | A | ... | Z | ... | a | ... | z | ...`
- `alpha` matches any letter $\quad$ `A | ... | Z | ... | a | ... | z`
- `digit` matches any number $\quad$ `0 | 1 | ... | 9`

[2] Note: string syntax allows us to use fewer dots. A `Regexp` equivalent to `star "HA"` is the more verbose expression: `star ("H" <.> "A")`. You can verify this yourself by entering both into `ghci` and viewing the desugared `Regexp` output.

The "algebra" of REs

Concatenation has some things in common with multiplication

- $\varepsilon \cdot r = r = r \cdot \varepsilon$ $\qquad 1n = n = n1$
- $\emptyset \cdot r = \emptyset = r \cdot \emptyset$ $\qquad 0n = 0 = n0$

And choice has some things in common with addition

- $\emptyset \mid r = r = r \mid \emptyset$ $\qquad 0 + n = n = n + 0$

What's more, both of these operations are **associative** too. Because grouping in such cases doesn't matter, we may leave parentheses off.[3]

- $(r \cdot s) \cdot t = r \cdot (s \cdot t)$ We write: $r \cdot s \cdot t$
- $(r \mid s) \mid t = r \mid (s \mid t)$ We write: $r \mid s \mid t$

[3] These parallels go even further, but we won't dwell on them today. Technically, regular expressions form a type of structure called a **star semiring**.

Matching

Patterns as sets

So we know how to write REs, both inside and outside of Haskell. But how can we actually **use them** to enforce **patterns**?

- In other words, how can we **check** that a string matches a RE?

We will discuss two ways:

- Matching sets, given by an `mset` function
- A `match` function that **parses** a string against a RE

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

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

A note on concatenation

The matching set of $r_1 \cdot r_2$ has **every possible way** of combining the things that match r_1 with the things that match r_2.

$$[\![r_1 \cdot r_2]\!] = \{ u +\!\!+\, v \mid u \in [\![r_1]\!], v \in [\![r_2]\!] \}$$

Thus, for example, if r_1 is a choice between a and b, and r_2 is a choice between y and z, the ways to match $r_1 \cdot r_2$ are given by:

	y	z
a	ay	az
b	by	bz

Walking through some cases

$$[\![(\mathsf{a} \mid \mathsf{b}) \cdot \mathsf{c}]\!] =$$

Walking through some cases

$$\begin{aligned} [\![(\mathsf{a} \mid \mathsf{b}) \cdot \mathsf{c}]\!] &= \{ u +\!\!+\, v \mid u \in [\![\mathsf{a} \mid \mathsf{b}]\!], v \in [\![\mathsf{c}]\!] \} \\ &= \end{aligned}$$

Walking through some cases

$$
\begin{aligned}
[\![(\mathsf{a} \mid \mathsf{b}) \cdot \mathsf{c}]\!] &= \{u +\!\!+ v \mid u \in [\![\mathsf{a} \mid \mathsf{b}]\!], v \in [\![\mathsf{c}]\!]\} \\
&= \{u +\!\!+ v \mid u \in [\![\mathsf{a}]\!] \cup [\![\mathsf{b}]\!], v \in [\![\mathsf{c}]\!]\} \\
&=
\end{aligned}
$$

Walking through some cases

$$\begin{aligned}
[\![(\mathsf{a} \mid \mathsf{b}) \cdot \mathsf{c}]\!] &= \{ u +\!\!+ v \mid u \in [\![\mathsf{a} \mid \mathsf{b}]\!], v \in [\![\mathsf{c}]\!] \} \\
&= \{ u +\!\!+ v \mid u \in [\![\mathsf{a}]\!] \cup [\![\mathsf{b}]\!], v \in [\![\mathsf{c}]\!] \} \\
&= \{ u +\!\!+ v \mid u \in \{\mathsf{a}\} \cup \{\mathsf{b}\}, v \in \{\mathsf{c}\} \} \\
&=
\end{aligned}$$

Walking through some cases

$$
\begin{aligned}
[\![(\mathsf{a} \mid \mathsf{b}) \cdot \mathsf{c}]\!] &= \{ u +\!\!+\, v \mid u \in [\![\mathsf{a} \mid \mathsf{b}]\!], v \in [\![\mathsf{c}]\!] \} \\
&= \{ u +\!\!+\, v \mid u \in [\![\mathsf{a}]\!] \cup [\![\mathsf{b}]\!], v \in [\![\mathsf{c}]\!] \} \\
&= \{ u +\!\!+\, v \mid u \in \{\mathsf{a}\} \cup \{\mathsf{b}\}, v \in \{\mathsf{c}\} \} \\
&= \{ u +\!\!+\, v \mid u \in \{\mathsf{a}, \mathsf{b}\}, v \in \{\mathsf{c}\} \} \\
&=
\end{aligned}
$$

Walking through some cases

$$\begin{aligned}
[\![(\mathsf{a} \mid \mathsf{b}) \cdot \mathsf{c}]\!] &= \{ u \mathbin{+\!\!+} v \mid u \in [\![\mathsf{a} \mid \mathsf{b}]\!], v \in [\![\mathsf{c}]\!] \} \\
&= \{ u \mathbin{+\!\!+} v \mid u \in [\![\mathsf{a}]\!] \cup [\![\mathsf{b}]\!], v \in [\![\mathsf{c}]\!] \} \\
&= \{ u \mathbin{+\!\!+} v \mid u \in \{\mathsf{a}\} \cup \{\mathsf{b}\}, v \in \{\mathsf{c}\} \} \\
&= \{ u \mathbin{+\!\!+} v \mid u \in \{\mathsf{a}, \mathsf{b}\}, v \in \{\mathsf{c}\} \} \\
&= \{ \mathsf{a} \mathbin{+\!\!+} \mathsf{c}, \mathsf{b} \mathbin{+\!\!+} \mathsf{c} \} \\
&=
\end{aligned}$$

Walking through some cases

$$
\begin{aligned}
[\![(\mathsf{a} \mid \mathsf{b}) \cdot \mathsf{c}]\!] &= \{u \mathbin{+\!\!+} v \mid u \in [\![\mathsf{a} \mid \mathsf{b}]\!], v \in [\![\mathsf{c}]\!]\} \\
&= \{u \mathbin{+\!\!+} v \mid u \in [\![\mathsf{a}]\!] \cup [\![\mathsf{b}]\!], v \in [\![\mathsf{c}]\!]\} \\
&= \{u \mathbin{+\!\!+} v \mid u \in \{\mathsf{a}\} \cup \{\mathsf{b}\}, v \in \{\mathsf{c}\}\} \\
&= \{u \mathbin{+\!\!+} v \mid u \in \{\mathsf{a}, \mathsf{b}\}, v \in \{\mathsf{c}\}\} \\
&= \{\mathsf{a} \mathbin{+\!\!+} \mathsf{c}, \mathsf{b} \mathbin{+\!\!+} \mathsf{c}\} \\
&= \{\mathsf{ac}, \mathsf{bc}\}
\end{aligned}
$$

Walking through some cases (cont)

$$[\![(\mathsf{a} \mid \mathsf{b})^*]\!] =$$

Walking through some cases (cont)

$$
\begin{aligned}
[\![(\mathsf{a} \mid \mathsf{b})^*]\!] &= [\![\varepsilon]\!] \cup \overbrace{[\![\mathsf{a} \mid \mathsf{b}]\!]}^{\{\mathsf{a},\mathsf{b}\}} \cup \overbrace{[\![(\mathsf{a} \mid \mathsf{b}) \cdot (\mathsf{a} \mid \mathsf{b})]\!]}^{\{\mathsf{aa},\mathsf{ab},\mathsf{ba},\mathsf{bb}\}} \cup \ldots \\
&\ldots \\
&=
\end{aligned}
$$

Walking through some cases (cont)

$$\llbracket (\mathsf{a} \mid \mathsf{b})^* \rrbracket = \llbracket \varepsilon \rrbracket \cup \overbrace{\llbracket \mathsf{a} \mid \mathsf{b} \rrbracket}^{\{\mathsf{a},\mathsf{b}\}} \cup \overbrace{\llbracket (\mathsf{a} \mid \mathsf{b}) \cdot (\mathsf{a} \mid \mathsf{b}) \rrbracket}^{\{\mathsf{aa},\mathsf{ab},\mathsf{ba},\mathsf{bb}\}} \cup \ldots$$

$$\ldots$$

$$= \{\varepsilon, \mathsf{a}, \mathsf{b}, \mathsf{aa}, \mathsf{ab}, \mathsf{ba}, \mathsf{bb}, \ldots\}$$

Any sequence of any length (including 0!) consisting only of a and b.

Matching in Haskell

Regexp.hs defines a `mset` function that returns the matching sets (technically, matching lists) for a RE:

```
mset :: Regexp -> [String]
mset Zero        = []    -- matched by nothing
mset One         = [""]  -- matched by ""
mset (Lit c)     = [[c]] -- matched by the String [c]
mset (Plus r s) = mset r ++ mset s -- unioning lists
mset (Cat  r s) = [u++v | u <- mset r, v <- mset s]
mset (Star r)    = concatMap (mset . dup r) [0..]
  where -- don't worry about the Star case :)
    dup r n = foldr (<.>) One (replicate n r)
```

All the clauses except for `Star` are direct Haskell translations of $[\![\cdot]\!]$.
Question: why do we use `++` in two places?

Matching in Haskell

Regexp.hs defines a `mset` function that returns the matching sets (technically, matching lists) for a RE:

```
mset :: Regexp -> [String]
mset Zero       = []    -- matched by nothing
mset One        = [""]  -- matched by ""
mset (Lit c)    = [[c]] -- matched by the String [c]
mset (Plus r s) = mset r ++ mset s -- unioning lists
mset (Cat  r s) = [u++v | u <- mset r, v <- mset s]
mset (Star r)   = concatMap (mset . dup r) [0..]
  where -- don't worry about the Star case :)
    dup r n = foldr (<.>) One (replicate n r)
```

All the clauses except for `Star` are direct Haskell translations of $[\![\cdot]\!]$.
Question: why do we use `++` in two places? With `Plus`, `++` combines two lists (matching sets). With `Cat`, `++` concatenates two `String`'s.

`mset` sample usage

Along with being able to define REs in Haskell, you should be able to **compute their matching sets**:

```
*W5> let u = ("a" <|> "b") <.> "c"
*W5> mset u
["ac","bc"]
```

Note that the matching sets for `star` are generally infinite, so you should use `take` to sample them:

```
*W5> take 8 (mset (star u))
["","ac","bc","acac","acbc","bcac","bcbc","acacac"]
```

Checking our algebraic properties

```
*W5> mset (zero <.> "ab")
[]
*W5> mset (one <.> "ab")
["ab"]
*W5> mset (zero <|> "ab")
["ab"]
```

```
*W5> mset (("ab" <.> "cd") <.> "e")
["abcde"]
*W5> mset ("ab" <.> ("cd" <.> "e"))
["abcde"]
```

```
*W5> mset (("ab" <|> "cd") <|> "e")
["ab","cd","e"]
*W5> mset ("ab" <|> ("cd" <|> "e"))
["ab","cd","e"]
```

Checking for membership in a matching set

We can use `elem` to check whether a `String` matches a RE:

```
*W5> let r8 = ("a" <|> "b") <.> "c"
*W5> elem "ac" (mset r8)
True
*W5> elem "xy" (mset r8)
False
```

And this even works for some infinite `mset`'s:

```
*W5> let r9 = star r8
*W5> elem "acbcacbcbcbcac" (mset r9)
True
```

Problems with `mset`?

Problems with `mset`?

How would you ever know something wasn't in an infinite set? For example, it is obvious to us that x does not match the pattern a^*. Yet:

```
*W5> elem "x" (mset (star "a")) -- dont try this at home
-- some time passes
^CInterrupted.
```

What happened?

Problems with `mset`?

How would you ever know something wasn't in an infinite set? For example, it is obvious to us that x does not match the pattern a*. Yet:

```
*W5> elem "x" (mset (star "a")) -- dont try this at home
-- some time passes
^CInterrupted.
```

What happened? `mset (star "a")` is an **infinite** list.

- The `elem` function walks through this list, checking to see whether any value is `"x"`.
- It won't ever encounter any, but it doesn't know that.

Slowness

Even when the `mset` strategy works, it is *slooooow:*

```
*W5> elem "bbbb" (mset (star anyc))
True
(48.97 secs, 46,733,644,048 bytes)
```

Oh my *god!* Why is it so slow?

Slowness

Even when the `mset` strategy works, it is *slooooow:*

```
*W5> elem "bbbb" (mset (star anyc))
True
(48.97 secs, 46,733,644,048 bytes)
```

Oh my *god!* Why is it so slow?

- Because it has to plow through **every** 1-, 2-, and 3-length `String` first, and a sizable chunk of the 4-length ones.

More efficient matching with `match`

```
*W5> :t match
match :: Regexp -> String -> Bool

*W5> match (star "a") "x"
False
*W5> match (star anyc) "aslkdjal k$/dj$kd!jaskldjaskl."
True

*W5> let r10 = star (("a" <|> "b") <.> "c")
*W5> match r10 "acacbcbcacacbcbc"
True

*W5> let r2 = "<b>" <.> star anyc <.> "</b>"
*W5> 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) [c']  = c==c'
match (Lit c) _      = False
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 `match (Cat r s) u` is that `u` must be made of parts matching `r` and `s`. But we don't know **which** parts exactly!

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

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

Introducing `splitAt`

```
*W5> splitAt 0 "abc"
("","abc")
*W5> splitAt 1 "abc"
("a","bc")
*W5> splitAt 2 "abc"
("ab","c")
*W5> splitAt 3 "abc"
("abc","")
```

Matching a `Cat` (cont)

We can try all of em and require that **one works**!

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

```
*W5> splits "abc"
[("","abc"),("a","bc"),("ab","c"),("abc","")]
--r? s?       r?  s?      r?   s?     r?    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` — thus, if there's any way of splitting up `u` into a part matching `r`, followed by one matching `s`.

Matching a `Star`

`""` always matches `Star r`. But how about other strings?

```
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.

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 starting string:

- `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**.