Regular Expressions

13 February 2020
A language \( L \) is a \textit{regular language} if there is a DFA \( D \) such that \( L(D) = L \).

\textbf{THEOREM.} The following are equivalent:

- \( L \) is a regular language.
- There is a DFA for \( L \).
- There is an NFA for \( L \).
Closure properties
Last class, we saw that the class of regular languages was closed under the following operations:

- Complement
- Union
- Intersection
- Concatenation

There’s one more closure property we want to establish.
**Concatenation of strings:**

If $w \in \Sigma^*$ and $x \in \Sigma^*$, then $wx$ is the concatenation of $w$ and $x$.

**Concatenation of languages:**

If $L_1$ and $L_2$ are languages over $\Sigma$, the concatenation of $L_1$ and $L_2$ is the language $L_1L_2$ defined as

$$L_1L_2 = \{wx \mid w \in L_1 \text{ and } x \in L_2\}$$

Example: If $L_1 = \{a, \text{ba}, \text{bb}\}$ and $L_2 = \{\text{aa}, \text{bb}\}$, then

$$L_1L_2 = \{\text{aaa}, \text{abb}, \text{baaa}, \text{babb}, \text{bbaa}, \text{bbbb}\}$$
Lots of concatenation

Consider the language \( L = \{aa, b\} \)

\( LL \) is the set of strings formed by concatenating pairs of strings in \( L \):

\[
\{aaaa, aab, baa, bb\}
\]

\( LLL \) is the set of strings formed by concatenating triples of strings in \( L \):

\[
\{aaaaaa, aaaaab, aabaa, aabb, baaaa, baab, bbaa, bbb\}
\]

\( LLLLL \) is the set of strings formed by concatenating quadruples of strings in \( L \)...
Language exponentiation

We can define what it means to “exponentiate” a language as follows:

\[ L^0 = \{ \varepsilon \} \]

**Base case:** Any string formed by concatenating zero strings together is just the empty string.

\[ L^{n+1} = LL^n \]

**Recursive case:** Concatenating \( n+1 \) strings together works by concatenating \( n \) strings, then concatenating one more.
Kleene star
Kleene (star) closure

An important operation on languages is the **Kleene closure**, which is defined as

\[ L^* = \{ w \in \Sigma^* \mid \exists n \in \mathbb{N}_0 . w \in L^n \} \]

A word is in \( L^* \) iff it’s in one of the languages \( L^0, L^1, L^2, \ldots \)

That is, \( L^* \) consists of all the possible ways of concatenating zero or more strings in \( L \).
If $L = \{a, \, bb\}$, then $L^* = \{
\varepsilon, 
\ a, \ bb, 
\ aa, \ abb, \ bba, \ bbbb, 
\ aaa, \ aabb, \ abba, \ abbbbb, \ bbaa, \ bbabb, \ bbbbaa, \ bbbbbbb, 
\ bbbbbbb, 
\ ...
\}
If $L$ is a regular language, is $L^*$ necessarily regular?
A **bad** line of reasoning

If $L$ is regular,

- $L^0 = \{ \varepsilon \}$ is regular.
- $L^1 = L$ is regular.
- $L^2 = LL$ is regular.
- $L^3 = L(LL)$ is regular.

... 

Regular languages are closed under union.

So, the union of all these languages is regular.
Reasoning about infinity
Reasoning about infinity
Reasoning about infinity
Reasoning about infinity
Reasoning about infinity
Reasoning about infinity

\[ x \neq 2x \]
Reasoning about infinity

0.9 < 1
Reasoning about infinity

0.99 < 1
Reasoning about infinity

0.999 < 1
Reasoning about infinity

0.9999 < 1
Reasoning about infinity

0.999999 < 1
Reasoning about infinity

\[ 0.9 \not\leq 1 \]
Reasoning about infinity

Strange but true!

\[ 0.\bar{9} = 1 \]

\[
\begin{align*}
x &= 0.\bar{9} \\
10x &= 9.\bar{9} & \text{multiply both sides by 10} \\
9x &= 9 & \text{subtract } x \text{ from both sides} \\
x &= 1 & \text{divide both sides by 9}
\end{align*}
\]
Reasoning about infinity

1 is finite
Reasoning about infinity

2 is finite
Reasoning about infinity

3 is finite
Reasoning about infinity

4 is finite
Reasoning about infinity

∞ is not finite
Even if a series of finite objects all have some property, the “limit” of that process doesn’t necessarily have that property.

In general, it’s not safe to conclude that some property that holds in the finite case must hold in the infinite case.

So, our earlier argument based on $L^* = L^0 \cup L^1 \cup \cdots$ isn’t going to work.

We need a different line of reasoning.
Can we convert an NFA for a language \( L \) into an NFA for \( L^* \)?
The new machine has the option of jumping back to the start state to read another piece that $N_1$ accepts.
The new machine has the option of jumping back to the start state to read another piece that $N_1$ accepts.
\[ L(N) = L(N_1)^* \]
Why add a new start state instead of making $N_1$’s a final state?
This construction proves the class of regular languages is closed under Kleene star.
Closure properties

THEOREM. If $L_1$ and $L_2$ are regular languages over an alphabet $\Sigma$, then so are the following languages:

- $\overline{L_1}$
- $L_1 \cup L_2$
- $L_1 \cap L_2$
- $L_1 L_2$
- $L_1^*$

These properties are *closure properties of the regular languages.*
Regular expressions
We’ve seen we can show a language is regular by
constructing a DFA for it
constructing an NFA for it (with or without $\epsilon$-transitions)

We can also show a language is regular by
constructing it out of simpler regular languages
using closure properties.
Regular expressions are a concise notation for describing how to assemble a larger language out of smaller pieces.
A bottom-up approach to the regular languages:

Start with a small set of simple languages we know to be regular
Use closure properties to combine these to form more elaborate languages

Regular expressions provide a string representation for describing a language in this way.
Atomic regular expressions

Regular expressions start with three simple building blocks:

For any symbol $\alpha \in \Sigma$, the regular expression $\alpha$ represents the language $\{ \alpha \}$

The symbol $\epsilon$ is a regular expression representing the language $\{ \epsilon \}$

The symbol $\emptyset$ is a regular expression for the empty language $\emptyset$
Compound regular expressions

New regular expressions are built out of existing ones using symbols for the regular operations:

- union,
- concatenation, and
- Kleene star.
Operation: Union

If $R_1$ and $R_2$ are regular expressions, $(R_1 \cup R_2)$ is a regular expression for the union of the languages of $R_1$ and $R_2$:

$$L((R_1 \cup R_2)) = L(R_1) \cup L(R_2).$$
Operation: Concatenation

If $R_1$ and $R_2$ are regular expressions,

$$(R_1 \circ R_2) \text{ or } (R_1 R_2)$$

is a regular expression for the concatenation of the languages of $R_1$ and $R_2$. 
Operation: Kleene star

If $R$ is a regular expression, $(R^*)$ is a regular expression for the Kleene closure of the language of $R$. 
Formal definition of regular expressions

DEFINITION. $R$ is a *regular expression* if $R$ is

1. $\alpha$ for some $\alpha \in \Sigma$
2. $\varepsilon$
3. $\emptyset$
4. $(R_1 \cup R_2)$, where $R_1$ and $R_2$ are regular expressions
5. $(R_1 \circ R_1)$, where $R_1$ and $R_1$ are regular expressions
6. $(R_1^*)$, where $R_1$ is a regular expression

Every regular expression arises by a finite number of applications of these six rules.
Order of operations

We can omit parentheses to make regular expressions more compact, but this makes them ambiguous unless we define precedence:

0. Parentheses – \((R)\)
1. Kleene star – \(R^*\)
2. Concatenation – \(R_1 R_2\) or \(R_1 \circ R_2\)
3. Union – \(R_1 u R_2\)
Empty strings, empty sets

Do not confuse the regular expressions:

ε – the language containing only the empty string

∅ – the language containing no strings

Identities:

\[ R \cup \emptyset = R \]

\[ R \circ \varepsilon = R \]
R is a regular expression if R is

1. \( \alpha \) for some \( \alpha \in \Sigma \)
2. \( \varepsilon \)
3. \( \emptyset \)
4. \( (R_1 \cup R_2) \), where \( R_1 \) and \( R_2 \) are regular expressions
5. \( (R_1 \circ R_1) \), where \( R_1 \) and \( R_1 \) are regular expressions
6. \( (R_1^*) \), where \( R_1 \) is a regular expression

**EXAMPLE:** To prove \( (((a(b^*)) \cup a) \) is a regular expression over \( \Sigma = \{a, b\} \), show it can be constructed according to the rules:

1. \( b \) is regular by **Rule 1**
2. \( (b^*) \) is regular by **Rule 6**
3. \( a \) is regular by **Rule 1**
4. \( (a(b^*)) \) is regular by **Rule 5**
5. \( ((a(b^*)) \cup a) \) is regular by **Rule 4** applied to expressions (4) and (3)
Examples

$L(\text{hi}) = \{\text{hi}\}$

$L(\text{hi} \cup \text{heyy}^*) = \{\text{hi}, \text{hey}, \text{heyy}, \text{heyyy}, \ldots\}$

$L((\emptyset(\emptyset \cup 1))^*) = \text{the set of strings of 0s and 1s, of even length, such that every odd position has a 0}$
A few more examples…

\(ab^*a\)

\(a^*b^*\)

\((ab)^*\)

Is this the same as \(a^*b^*\)?

\(a^*b^*a^*\)

Is \(baa\) in this?

\(L = \{x^{\text{odd}}\} = x(xx)^* \text{ or } (xx)^*x \text{ but not } x^*xx^*\)

All strings of \(a\)s and \(b\)s of exactly length 3

\(L = \{aaa, aab, aba, abb, baa, bab, bba, bbb\}\)

or \((a \cup b) (a \cup b) (a \cup b)\)

or \((a \cup b)^3\)
The *language of a regular expression* is the language described by that regular expression.

Formally:

\[ L(\varepsilon) = \{\varepsilon\} \]

\[ L(\emptyset) = \emptyset \]

\[ L(a) = \{a\} \]

\[ L(R_1 R_2) = L(R_1) L(R_2) \]

\[ L(R_1 \cup R_2) = L(R_1) \cup L(R_2) \]

\[ L(R^*) = L(R)^* \]

\[ L((R)) = L(R) \]
Designing regular expressions
Designing regular expressions

Let $\Sigma = \{a, b\}$

Let $L = \{w \in \Sigma^* \mid w \text{ contains } aa \text{ as a substring}\}$
Designing regular expressions

Let $\Sigma = \{a, b\}$

Let $L = \{w \in \Sigma^* \mid w$ contains $aa$ as a substring$\}$

$$(a \cup b)^*aa(a \cup b)^*$$
Designing regular expressions

Let $\Sigma = \{a, b\}$

Let $L = \{w \in \Sigma^* \mid w \text{ contains } aa \text{ as a substring}\}$

$(aub)^*aa(aub)^*$
Designing regular expressions

Let $\Sigma = \{a, b\}$

Let $L = \{w \in \Sigma^* | w$ contains $aa$ as a substring$\}$

$$(a \cup b)^*aa(a \cup b)^*$$

$bbabbbbaaabab$

$aaaa$

$bbbbbbabbbbaabbbbb$
Designing regular expressions

Let \( \Sigma = \{a, b\} \)

Let \( L = \{w \in \Sigma^* \mid w \text{ contains } aa \text{ as a substring}\} \)

\[(a \cup b)^*aa(a \cup b)^*\]

\[bbabbbbaabab\]
\[aaaa\]
\[bbbbbbabbbbaabbbbb\]
Designing regular expressions

Let $\Sigma = \{a, b\}$

Let $L = \{w \in \Sigma^* \mid w \text{ contains } aa \text{ as a substring}\}$

$\Sigma^*aa\Sigma^*$

A convenient shorthand

bbabbaaabab

aaaa

bbbbbbabbbbaabbbbbb
Designing regular expressions

Let $\Sigma = \{a, b\}$

Let $L = \{w \in \Sigma^* \mid |w| = 4\}$
Designing regular expressions

Let $\Sigma = \{a, b\}$

Let $L = \{w \in \Sigma^* \mid |w| = 4\}$

*Recall: $|w|$ denotes the length of string $w$.**
Designing regular expressions

Let $\Sigma = \{a, b\}$

Let $L = \{w \in \Sigma^* \mid |w| = 4\}$
Designing regular expressions

Let \( \Sigma = \{a, b\} \)

Let \( L = \{ w \in \Sigma^* \mid |w| = 4 \} \)

\[ \Sigma \Sigma \Sigma \Sigma \]

aaaa
baba
bbbb
baaa
Designing regular expressions

Let $\Sigma = \{a, b\}$

Let $L = \{w \in \Sigma^* \mid |w| = 4\}$

$\Sigma \Sigma \Sigma \Sigma$

aaaaa
baba
bbbbb
baaa
Designing regular expressions

Let $\Sigma = \{a, b\}$

Let $L = \{w \in \Sigma^* \mid |w| = 4\}$

$\Sigma^4$

aaaa
baba
bbbb
baaa
Designing regular expressions

Let $\Sigma = \{a, b\}$

Let $L = \{w \in \Sigma^* \mid |w| = 4\}$

Another shorthand

$\Sigma^4$

aaaa
baba
bbbb
baaa
Designing regular expressions

Let $\Sigma = \{a, b\}$

Let $L = \{w \in \Sigma^* \mid w \text{ contains at most one } a\}$

Here are some candidates regular expressions for $L$. Which are correct?

- $\Sigma^*a\Sigma^*$
- $b^*ab^*ub^*$
- $b^*(a \cup \varepsilon)b^*$
- $b^*a^*bub^*$
- $b^*(a^*u\varepsilon)b^*$
Designing regular expressions

Let $\Sigma = \{a, b\}$

Let $L = \{ w \in \Sigma^* \mid w$ contains at most one $a \}$

$$b^*(a \cup \varepsilon)b^*$$
Designing regular expressions

Let $\Sigma = \{a, b\}$

Let $L = \{w \in \Sigma^* \mid w \text{ contains at most one } a\}$

$b^* (a \cup \varepsilon) b^*$
Designing regular expressions

Let $\Sigma = \{a, b\}$

Let $L = \{w \in \Sigma^* | w$ contains at most one $a\}$

$$b^*(a \cup \varepsilon)b^*$$

bbbabbb

bbbbbb

abbb

a
Designing regular expressions

Let $\Sigma = \{a, b\}$

Let $L = \{w \in \Sigma^* \mid w \text{ contains at most one } a\}$

$$b^*(a \cup \varepsilon)b^*$$
Designing regular expressions

Let $\Sigma = \{a, b\}$

Let $L = \{w \in \Sigma^* \mid w \text{ contains at most one } a\}$

$\quad b^*a?b^*$

$\quad bbbbabbb$

$\quad bbbbbbb$

$\quad aabbb$

$\quad a$
Designing regular expressions

Let $\Sigma = \{a, b\}$

Let $L = \{w \in \Sigma^* \mid w \text{ contains at most one } a\}$

Another shorthand

$$b^*a?b^*$$
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

\[ aa^* \]

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

```
  aa*
```

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

```
aa*(.aa*)*
```

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

$$aa*(.aa*)*$$

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

```
 aa*(.aa*)*@  
```

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

$aa* (.aa*)*@$

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

$$aa*(.aa*)*@aa*\.aa*$$

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

$$aa^*(.aa^*)*@aa^*.aa^*$$

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
A more elaborate design

Let \( \Sigma = \{a, \ ., \ @\} \), where \( a \) represents “any letter”.

Let’s make a regular expression for email addresses.

\[
\text{aa}\ast\left(\text{.aa}\ast\right)\ast\text{@aa}\ast\text{.aa}\ast\left(\text{.aa}\ast\right)\ast
\]

\[
\text{mvassar@vassar.edu}
\]
\[
\text{matthew.vassar@vassarbrewery.com}
\]
\[
\text{matt@cs.vassar.edu}
\]
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

aa*(.aa*)*@aa*.aa*(.aa*)*

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

$$ a^+ (.aa*)*@aa* .aa* (.aa*)* $$

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

You guessed it – another shorthand

\[
a^+ (a^*).@a^*.a^*(a^*)^*
\]

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

```
a^+(.a^+)*@a^+.a^+(.a^+)*
```

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

$$a^+(.a^+)*@a^+(.a^+)^+$$

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
For comparison

\[ a^+ (.a^+) @ a^+ (.a^+) ^+ \]
Convenient shorthands

\( \Sigma \) is a shorthand for “any character in \( \Sigma \)”

\( R^n \) is a shorthand for \( RR \ldots R \) (\( n \) times)

\( R? \) is shorthand for \( (R \cup \epsilon) \) – that is, zero or one copies of \( R \).

\( R^+ \) is a shorthand for \( RR^* \) – that is, one or more copies of \( R \).
Regular expressions in the real world
UNIX regular expressions

From the beginning (of time), UNIX has used regular expressions in many places, including the grep command.

\texttt{grep} = \texttt{global} (search for a) \texttt{regular expression} and \texttt{print}

Many UNIX commands use an extended RE notation, but it still expresses only the regular languages.
UNIX RE notation

\[[a_1 a_2 \ldots a_n]\] is shorthand for \( a_1 \cup a_2 \cup \cdots \cup a_n \).

Ranges are indicated by first-dash-last and brackets, using ASCII character order, e.g.,

\[[a\text{–}z]\] = any lowercase letter

\[[a\text{–}zA\text{–}Z]\] = any letter

Dot (\( . \)) = any character (like our shorthand \( \Sigma \))
UNIX RE notation, *continued*

Since characters like brackets, dashes, and dots have special meaning, if you want to match them, you need to quote with backslash (\).

Union operator is represented with a bar (|)

Includes our + shorthand for “one or more”, e.g.,

\[a\text{-}z\]+ = one or more lowercase letter
Perl, Python, Emacs, …

Include additional extensions, notably character classes like `\b` for word boundary characters, `\w` for word characters, etc.

With each implementation of regular expressions, they become less standard, so what you write for one language or application won’t work in another.
grep lets you search files for text

$ grep bananas foo.txt

Here are some of my favourite grep command line arguments!

- **E** use if you want regexps like ".+" to work. Otherwise you need to use ".\+

- **-v** invert match: find all lines that don't match

- **-i** case insensitive

- **-l** only show the filenames of the files that matched

- **-a** only print the matching part of the line (not the whole line)

- **-o** search binaries: treat binary data like it's text instead of ignoring it!

- **-A** show context for your search.
  
  $ grep -A 3 foo

  will show 3 lines of context after a match

- **-B** don't treat the match string as a regex
  
  eg $ grep -F ...

- **-C** grep alternatives

  - **ack**
  - **ag**
  - **ripgrep**

  (better for searching code!)

Recursive! Search all the files in a directory.

https://twitter.com/b0rk/status/991880504805871616
Lexical analysis

The first thing a compiler does is break a program into *tokens*, which are substrings that together represent a unit, e.g.,

- identifiers
- reserved words like "if"
- meaningful single characters like ";" or "+
- multi-character operators like "<="
Lexical analysis, continued

There are tools like lex or flex that let you write a regular expression for each kind of token.

E.g., in UNIX notation, identifiers are something like \([A-Za-z][A-Za-z0-9_]\)*

Each RE has an associated action like returning a code for the token found or adding it to a symbol table.
Acknowledgments

This lecture incorporates material from:

- Nancy Ide
- Keith Schwarz
- Michael Sipser
- Jeffrey Ullman