Formal Languages and Groups

Newcastle Junior Algebra Seminar

Graham Campbell

School of Mathematics, Statistics and Physics, Newcastle University, UK

December 2019

Strings

Definition 1 (Alphabet)

An alphabet Σ is a finite set. We call $\sigma \in \Sigma$ a symbol, letter, or character.

Strings

Definition 1 (Alphabet)

An alphabet Σ is a finite set. We call $\sigma \in \Sigma$ a symbol, letter, or character.

Definition 2 (String)

A string over Σ is a finite sequence of symbols from Σ. We usually write these symbols side-by-side (e.g. $w = abcabc$ is a string over $\{a, b, c\}$). We denote the set of all strings over Σ by Σ^*.

Strings

Definition 1 (Alphabet)

An alphabet Σ is a finite set. We call $\sigma \in \Sigma$ a symbol, letter, or character.

Definition 2 (String)

A string over Σ is a finite sequence of symbols from Σ. We usually write these symbols side-by-side (e.g. $w = abcabc$ is a string over $\{a, b, c\}$). We denote the set of all strings over Σ by Σ^*.

Definition 3

A language over Σ is simply a subset $L \subseteq \Sigma^*$.

Strings

Definition 1 (Alphabet)

An alphabet Σ is a finite set. We call $\sigma \in \Sigma$ a symbol, letter, or character.

Definition 2 (String)

A string over Σ is a finite sequence of symbols from Σ. We usually write these symbols side-by-side (e.g. $w = abcabc$ is a string over $\{a, b, c\}$). We denote the set of all strings over Σ by Σ^*.

Definition 3

A language over Σ is simply a subset $L \subseteq \Sigma^*$.

All languages are countable sets, but there are uncountably many languages over a fixed alphabet. Concatenation of strings is concatenation of sequences, and we denote the empty string by ϵ.

Rules

Definition 4 (Substring)

Given strings u, v over Σ, we say that u is a substring of v ($u \sqsubseteq v$) exactly when u is a subsequence of v.

Rules

Definition 4 (Substring)

Given strings u, v over Σ, we say that u is a substring of v ($u \sqsubseteq v$) exactly when u is a subsequence of v.

Definition 5 (Rewriting Rule)

A string rewriting rule over Σ is a pair of strings $p = (l, r)$.

Rules

Definition 4 (Substring)

Given strings u, v over Σ, we say that u is a substring of v ($u \sqsubseteq v$) exactly when u is a subsequence of v.

Definition 5 (Rewriting Rule)

A string rewriting rule over Σ is a pair of strings $p = (l, r)$.

Definition 6 (Rule Application)

We say that $p = (l, r)$ can be applied to w iff $l \sqsubseteq w$. We write $w \Rightarrow_p z$ to indicate that p has been applied to w, replacing l with r to yield z.

Rules

Definition 4 (Substring)

Given strings u, v over Σ, we say that u is a substring of v ($u \sqsubseteq v$) exactly when u is a subsequence of v.

Definition 5 (Rewriting Rule)

A string rewriting rule over Σ is a pair of strings $p = (l, r)$.

Definition 6 (Rule Application)

We say that $p = (l, r)$ can be applied to w iff $l \sqsubseteq w$. We write $w \Rightarrow_p z$ to indicate that p has been applied to w, replacing l with r to yield z.

There may be no ways to apply a rule to a string, or there may be multiple ways. We can view $\Rightarrow_p$ as a finitely branching binary relation, and given a set of rules $\mathcal{R}$, we can extend the relation to $\Rightarrow_{\mathcal{R}}$ on Σ^*.

SRSs

Definition 7 (SRS)

Given an alphabet Σ and a finite set of rules $\mathcal{R}$ over Σ, we call the pair $(\Sigma, \mathcal{R})$ a string rewriting system (SRS).

SRSs

Definition 7 (SRS)

Given an alphabet Σ and a finite set of rules $\mathcal{R}$ over Σ, we call the pair $(\Sigma, \mathcal{R})$ a string rewriting system (SRS).

Definition 8

Given a SRS $(\Sigma, \mathcal{R})$, we define:

1. $\mathcal{R}^{-1} := \{(r, l) \mid (l, r) \in \mathcal{R}\}$;
2. $\Rightarrow_{\mathcal{R}}^{*} := \bigcup_{n \in \mathbb{N}} \Rightarrow_{\mathcal{R}}^{n}$ and $\Leftrightarrow_{\mathcal{R}}^{*} := \Rightarrow_{\mathcal{R}}^{*} \cup \Rightarrow_{\mathcal{R}^{-1}}^{*}$.

SRSs

Definition 7 (SRS)

Given an alphabet Σ and a finite set of rules $\mathcal{R}$ over Σ, we call the pair $(\Sigma, \mathcal{R})$ a string rewriting system (SRS).

Definition 8

Given a SRS $(\Sigma, \mathcal{R})$, we define:

1. $\mathcal{R}^{-1} := \{(r, l) \mid (l, r) \in \mathcal{R}\}$;
2. $\Rightarrow_{\mathcal{R}}^{*} := \bigcup_{n \in \mathbb{N}} \Rightarrow_{\mathcal{R}}^{n}$ and $\Leftrightarrow_{\mathcal{R}}^{*} := \Rightarrow_{\mathcal{R}}^{*} \cup \Rightarrow_{\mathcal{R}^{-1}}^{*}$.

Proposition 9

1. $\Rightarrow_{\mathcal{R}^{-1}}$ *equals* $\Rightarrow_{\mathcal{R}}^{-1}$;
2. $\Rightarrow_{\mathcal{R}}^{*}$ *is the reflexive transitive closure of* $\Rightarrow_{\mathcal{R}}$;
3. $\Leftrightarrow_{\mathcal{R}}^{*}$ *is the sym. closure of* $\Rightarrow_{\mathcal{R}}^{*}$, *is the smallest ER containing* $\Rightarrow_{\mathcal{R}}$.

Grammars

Definition 10 (Grammar)

A grammar is a 4-tuple $\mathcal{G} = (T, N, \mathcal{R}, S)$ where T and N are finite disjoint sets called terminals and non-terminals respectively, $\mathcal{R}$ is a finite set of rules over $T \cup N$ such that each rule's LHS contains at least one non-terminal, and $S \in N$ is the start symbol.

Grammars

Definition 10 (Grammar)

A grammar is a 4-tuple $\mathcal{G} = (T, N, \mathcal{R}, S)$ where T and N are finite disjoint sets called terminals and non-terminals respectively, $\mathcal{R}$ is a finite set of rules over $T \cup N$ such that each rule's LHS contains at least one non-terminal, and $S \in N$ is the start symbol.

Define the accepted language to be the set of all terminal strings derivable from s in 0 or more steps: $\mathrm{L}(\mathcal{G}) := \{w \in T^* \mid s \Rightarrow_{\mathcal{R}}^* w\}$.

Grammars

Definition 10 (Grammar)

A grammar is a 4-tuple $\mathcal{G} = (T, N, \mathcal{R}, S)$ where T and N are finite disjoint sets called terminals and non-terminals respectively, $\mathcal{R}$ is a finite set of rules over $T \cup N$ such that each rule's LHS contains at least one non-terminal, and $S \in N$ is the start symbol.

Define the accepted language to be the set of all terminal strings derivable from s in 0 or more steps: $\mathrm{L}(\mathcal{G}) := \{w \in T^* \mid s \Rightarrow_{\mathcal{R}}^* w\}$.

Example 11

Non-terminals really do add generational power. Consider the language of all palindromes over $\Sigma = \{a, b\}$.

Grammars

Definition 10 (Grammar)

A grammar is a 4-tuple $\mathcal{G} = (T, N, \mathcal{R}, S)$ where T and N are finite disjoint sets called terminals and non-terminals respectively, $\mathcal{R}$ is a finite set of rules over $T \cup N$ such that each rule's LHS contains at least one non-terminal, and $S \in N$ is the start symbol.

Define the accepted language to be the set of all terminal strings derivable from s in 0 or more steps: $\mathrm{L}(\mathcal{G}) := \{w \in T^* \mid s \Rightarrow_{\mathcal{R}}^* w\}$.

Example 11

Non-terminals really do add generational power. Consider the language of all palindromes over $\Sigma = \{a, b\}$.

$(\{a, b\}), \{S\}, \{S \to aSa, S \to bSb, S \to \epsilon\}, S)$ generates the language.

Grammars

Definition 10 (Grammar)

A grammar is a 4-tuple $\mathcal{G} = (T, N, \mathcal{R}, S)$ where T and N are finite disjoint sets called terminals and non-terminals respectively, $\mathcal{R}$ is a finite set of rules over $T \cup N$ such that each rule's LHS contains at least one non-terminal, and $S \in N$ is the start symbol.

Define the accepted language to be the set of all terminal strings derivable from s in 0 or more steps: $\mathrm{L}(\mathcal{G}) := \{w \in T^* \mid s \Rightarrow_{\mathcal{R}}^* w\}$.

Example 11

Non-terminals really do add generational power. Consider the language of all palindromes over $\Sigma = \{a, b\}$.

$(\{a, b\}), \{S\}, \{S \to aSa, S \to bSb, S \to \epsilon\}, S)$ generates the language.

There is no grammar with $N = \emptyset$ that generates the language.

Free Monoids

Definition 12 (Universal Mapping Property (UMP))

$M(\Sigma)$ satisfies the UMP iff there is a fixed function $i : \Sigma \to |M(\Sigma)|$ (where $|\cdot|$ is underlying set functor on **Mon**), such that given any monoid N and function $f : \Sigma \to |N|$, there is a unique monoid homomorphism $\bar{f} : M(\Sigma) \to N$ such that $|f| \circ i = f$.

Free Monoids

Definition 12 (Universal Mapping Property (UMP))

$M(\Sigma)$ satisfies the UMP iff there is a fixed function $i : \Sigma \to |M(\Sigma)|$ (where $|\cdot|$ is underlying set functor on **Mon**), such that given any monoid N and function $f : \Sigma \to |N|$, there is a unique monoid homomorphism $\bar{f} : M(\Sigma) \to N$ such that $|f| \circ i = f$.

Theorem 13

For each Σ, Σ^* is a monoid under concatenation with identity ϵ. Moreover, Σ^* satisfies the UMP, and is unique up to unique isomorphism. We call it the free monoid on Σ.

Free Monoids

Definition 12 (Universal Mapping Property (UMP))

$M(\Sigma)$ satisfies the UMP iff there is a fixed function $i : \Sigma \to |M(\Sigma)|$ (where $|\cdot|$ is underlying set functor on **Mon**), such that given any monoid N and function $f : \Sigma \to |N|$, there is a unique monoid homomorphism $\bar{f} : M(\Sigma) \to N$ such that $|f| \circ i = f$.

Theorem 13

For each Σ, Σ^* is a monoid under concatenation with identity ϵ. Moreover, Σ^* satisfies the UMP, and is unique up to unique isomorphism. We call it the free monoid on Σ.

Proof: It's easy to check Σ^* is a monoid. To see the UMP, choose i to be the inclusion which takes each symbol to its length one string, and define $\bar{f}$ to operate on each character by f. The rest is easy to check... □

Monoid Presentations

Definition 14 (Monoid Presentation)

Given a SRS $(\Sigma, \mathcal{R})$, define ${}^{\Sigma^*}/_{\mathcal{R}} := \{[w]_{\Leftrightarrow^*_{\mathcal{R}}} \mid w \in \Sigma^*\}$.

Monoid Presentations

Definition 14 (Monoid Presentation)

Given a SRS $(\Sigma, \mathcal{R})$, define ${}^{\Sigma^*}/_{\mathcal{R}} := \{[w]_{\Leftrightarrow^*_{\mathcal{R}}} \mid w \in \Sigma^*\}$.

Proposition 15

Then ${}^{\Sigma^*}/_{\mathcal{R}}$ is a monoid with operation $[u][v] = [uv]$. Moreover, all monoids are isomorphic to such a quotient of the free monoid (not necessarily with Σ or $\mathcal{R}$ finite).

Monoid Presentations

Definition 14 (Monoid Presentation)

Given a SRS $(\Sigma, \mathcal{R})$, define ${}^{\Sigma^*}/_{\mathcal{R}} := \{[w]_{\Leftrightarrow^*_{\mathcal{R}}} \mid w \in \Sigma^*\}$.

Proposition 15

Then ${}^{\Sigma^*}/_{\mathcal{R}}$ is a monoid with operation $[u][v] = [uv]$. Moreover, all monoids are isomorphic to such a quotient of the free monoid (not necessarily with Σ or $\mathcal{R}$ finite).

Proof: The first part is due to the fact that $\Leftrightarrow^*_{\mathcal{R}}$ is a congruence. The second part can be seen setting Σ equal to the monoid, and using the multiplication table to generate $\mathcal{R}$. □

Monoid Presentations

Definition 14 (Monoid Presentation)

Given a SRS $(\Sigma, \mathcal{R})$, define ${}^{\Sigma^*}/_{\mathcal{R}} := \{[w]_{\Leftrightarrow^*_{\mathcal{R}}} \mid w \in \Sigma^*\}$.

Proposition 15

Then ${}^{\Sigma^*}/_{\mathcal{R}}$ is a monoid with operation $[u][v] = [uv]$. Moreover, all monoids are isomorphic to such a quotient of the free monoid (not necessarily with Σ or $\mathcal{R}$ finite).

Proof: The first part is due to the fact that $\Leftrightarrow^*_{\mathcal{R}}$ is a congruence. The second part can be seen setting Σ equal to the monoid, and using the multiplication table to generate $\mathcal{R}$. □

Those monoids for which there exists a presentation with Σ finite are called finitely generated, and if $\mathcal{R}$ is finite too, we call them finitely presented.

Overview

Theorem 16 (Regular Language Characterisations)

Given $L \subseteq \Sigma^*$, the following are equivalent:

1. *L is a recognisable subset of Σ^*;*
2. *L is a rational subset of Σ^*;*
3. *L has a finite syntactic monoid;*
4. *L is generated by a regular grammar;*
5. *L is accepted by a finite state automaton (FSA);*
6. *L is accepted by a deterministic FSA (DFSA);*
7. *L is accepted by a read-only Turing machine;*
8. *$L \setminus \{\epsilon\}$ is monadic second order (MSO) definable.*

Overview

Theorem 16 (Regular Language Characterisations)

Given $L \subseteq \Sigma^*$, the following are equivalent:

1. *L is a recognisable subset of Σ^*;*
2. *L is a rational subset of Σ^*;*
3. *L has a finite syntactic monoid;*
4. *L is generated by a regular grammar;*
5. *L is accepted by a finite state automaton (FSA);*
6. *L is accepted by a deterministic FSA (DFSA);*
7. *L is accepted by a read-only Turing machine;*
8. *$L \setminus \{\epsilon\}$ is monadic second order (MSO) definable.*

We don't have time to look at all of these. We'll only look at 1, 2, 4, 5, 6, only very briefly at 1 and 2.

Recognisable and Rational Sets

Definition 17 (Recognisable Set)

Given a monoid M, a subset L is called recognisable iff there is a finite monoid N and a monoid homomorphism $h : M \to N$ such that $h^{-1}(h(L)) = L$.

Recognisable and Rational Sets

Definition 17 (Recognisable Set)

Given a monoid M, a subset L is called recognisable iff there is a finite monoid N and a monoid homomorphism $h : M \to N$ such that $h^{-1}(h(L)) = L$.

Definition 18 (Rational Set)

Given a monoid M, the rational subsets $\mathrm{RAT}(M)$ are defined inductively:

1. the finite subsets of M are rational;
2. if $L_1, L_2 \in \mathrm{RAT}(M)$ then $L_1 \cup L_2 \in \mathrm{RAT}(M)$;
3. if $L_1, L_2 \in \mathrm{RAT}(M)$ then $L_1 L_2 \in \mathrm{RAT}(M)$;
4. if $L \in \mathrm{RAT}(M)$, then the generated submonoid $L^* \in \mathrm{RAT}(M)$.

Recognisable and Rational Sets

Definition 17 (Recognisable Set)

Given a monoid M, a subset L is called recognisable iff there is a finite monoid N and a monoid homomorphism $h : M \to N$ such that $h^{-1}(h(L)) = L$.

Definition 18 (Rational Set)

Given a monoid M, the rational subsets $\mathrm{RAT}(M)$ are defined inductively:

1. the finite subsets of M are rational;
2. if $L_1, L_2 \in \mathrm{RAT}(M)$ then $L_1 \cup L_2 \in \mathrm{RAT}(M)$;
3. if $L_1, L_2 \in \mathrm{RAT}(M)$ then $L_1 L_2 \in \mathrm{RAT}(M)$;
4. if $L \in \mathrm{RAT}(M)$, then the generated submonoid $L^* \in \mathrm{RAT}(M)$.

Proposition 19

Given a monoid M, the rational and recognisable sets coincide.

FSAs

Definition 20 (Finite State Automaton (FSA))

An FSA is a 5-tuple $\mathcal{A} = (Q, \Sigma, \delta, i, F)$ where Q is a finite set of states, Σ is the finite input alphabet, $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ is the transition relation, $i \in Q$ is the initial state, and $F \subseteq Q$ are the final states.

FSAs

Definition 20 (Finite State Automaton (FSA))

An FSA is a 5-tuple $\mathcal{A} = (Q, \Sigma, \delta, i, F)$ where Q is a finite set of states, Σ is the finite input alphabet, $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ is the transition relation, $i \in Q$ is the initial state, and $F \subseteq Q$ are the final states.

Definition 21 (Extended Transition Relation)

Define the extended transition relation $\bar{\delta} \subseteq Q \times \Sigma^* \times Q$ inductively:

1. if $(s, a, t) \in \delta$, then $(s, a, t) \in \bar{\delta}$;
2. if $(s, a, q) \in \delta$ and $(q, w, t) \in \bar{\delta}$, then $(s, aw, t) \in \bar{\delta}$.

FSAs

Definition 20 (Finite State Automaton (FSA))

An FSA is a 5-tuple $\mathcal{A} = (Q, \Sigma, \delta, i, F)$ where Q is a finite set of states, Σ is the finite input alphabet, $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ is the transition relation, $i \in Q$ is the initial state, and $F \subseteq Q$ are the final states.

Definition 21 (Extended Transition Relation)

Define the extended transition relation $\bar{\delta} \subseteq Q \times \Sigma^* \times Q$ inductively:

1. if $(s, a, t) \in \delta$, then $(s, a, t) \in \bar{\delta}$;
2. if $(s, a, q) \in \delta$ and $(q, w, t) \in \bar{\delta}$, then $(s, aw, t) \in \bar{\delta}$.

Definition 22 (Language Accepted by an FSA)

Define $\mathrm{L}(\mathcal{A}) := \{w \in \Sigma^* \mid \exists q \in F \text{ with } (i, w, q) \in \bar{\delta}\}$.

FSAs

Definition 20 (Finite State Automaton (FSA))

An FSA is a 5-tuple $\mathcal{A} = (Q, \Sigma, \delta, i, F)$ where Q is a finite set of states, Σ is the finite input alphabet, $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ is the transition relation, $i \in Q$ is the initial state, and $F \subseteq Q$ are the final states.

Definition 21 (Extended Transition Relation)

Define the extended transition relation $\bar{\delta} \subseteq Q \times \Sigma^* \times Q$ inductively:

1. if $(s, a, t) \in \delta$, then $(s, a, t) \in \bar{\delta}$;
2. if $(s, a, q) \in \delta$ and $(q, w, t) \in \bar{\delta}$, then $(s, aw, t) \in \bar{\delta}$.

Definition 22 (Language Accepted by an FSA)

Define $\mathrm{L}(\mathcal{A}) := \{w \in \Sigma^* \mid \exists q \in F \text{ with } (i, w, q) \in \bar{\delta}\}$.

Call $\mathcal{A}$ deterministic whenever δ is functional on $Q \times \Sigma \cup \{\epsilon\}$, and without ϵ-transitions (no $(s, \epsilon, t) \in \delta$ for any $s, t \in Q$).

FSA Equivalence

Proposition 23

Let Σ be some fixed alphabet. Then the FSAs over Σ accept the same class of languages as the deterministic FSAs over Σ.

FSA Equivalence

Proposition 23

Let Σ be some fixed alphabet. Then the FSAs over Σ accept the same class of languages as the deterministic FSAs over Σ.

Proof: The reverse inclusion is obvious. To see the forward inclusion, we can explicitly construct a deterministic FSA by taking the states to be the powerset of the original states, and recursively adding transitions between sets of states. Correctness is by induction on transition length, looking at (non-)reachability of final states. □

FSA Equivalence

Proposition 23

Let Σ be some fixed alphabet. Then the FSAs over Σ accept the same class of languages as the deterministic FSAs over Σ.

Proof: The reverse inclusion is obvious. To see the forward inclusion, we can explicitly construct a deterministic FSA by taking the states to be the powerset of the original states, and recursively adding transitions between sets of states. Correctness is by induction on transition length, looking at (non-)reachability of final states. □

Proposition 24

The rational sets of Σ^* are exactly the languages accepted by an FSA.

FSA Equivalence

Proposition 23

Let Σ be some fixed alphabet. Then the FSAs over Σ accept the same class of languages as the deterministic FSAs over Σ.

Proof: The reverse inclusion is obvious. To see the forward inclusion, we can explicitly construct a deterministic FSA by taking the states to be the powerset of the original states, and recursively adding transitions between sets of states. Correctness is by induction on transition length, looking at (non-)reachability of final states. □

Proposition 24

The rational sets of Σ^* are exactly the languages accepted by an FSA.

Proof: From a rational expression for L, it's easy to inductively construct an FSA accepting L. For the reverse inclusion, we can build a rational expression for the language accepted by an FSA by considering paths between states in the FSA. □

Regular Grammars

Definition 25

A grammar $\mathcal{G} = (T, N, \mathcal{R}, S)$ is regular iff each rule is of the form $A \to xB$ where $A, B \in N$ and $x \in T \cup \{\epsilon\}$.

Regular Grammars

Definition 25

A grammar $\mathcal{G} = (T, N, \mathcal{R}, S)$ is regular iff each rule is of the form $A \to xB$ where $A, B \in N$ and $x \in T \cup \{\epsilon\}$.

Example 26

$L = \{a^n b^m \mid n, m \in \mathbb{N}\}$ is generated by the regular grammar $(\{a, b\}, \{S, B\}, \{S \to aS, S \to bB, S \to \epsilon, B \to bB, B \to \epsilon\}, S)$.

Regular Grammars

Definition 25

A grammar $\mathcal{G} = (T, N, \mathcal{R}, S)$ is regular iff each rule is of the form $A \to xB$ where $A, B \in N$ and $x \in T \cup \{\epsilon\}$.

Example 26

$L = \{a^n b^m \mid n, m \in \mathbb{N}\}$ is generated by the regular grammar $(\{a, b\}, \{S, B\}, \{S \to aS, S \to bB, S \to \epsilon, B \to bB, B \to \epsilon\}, S)$.

Proposition 27

Let Σ be some fixed alphabet. Then the FSAs with input alphabets Σ accept the same class of languages as the regular grammars with terminals Σ.

Regular Grammars

Definition 25

A grammar $\mathcal{G} = (T, N, \mathcal{R}, S)$ is regular iff each rule is of the form $A \to xB$ where $A, B \in N$ and $x \in T \cup \{\epsilon\}$.

Example 26

$L = \{a^n b^m \mid n, m \in \mathbb{N}\}$ is generated by the regular grammar $(\{a, b\}, \{S, B\}, \{S \to aS, S \to bB, S \to \epsilon, B \to bB, B \to \epsilon\}, S)$.

Proposition 27

Let Σ be some fixed alphabet. Then the FSAs with input alphabets Σ accept the same class of languages as the regular grammars with terminals Σ.

Proof: The bijective correspondence is clear, treating non-terminals as states, and terminals (or ϵ) as transitions! Non-terminals that can be rewritten to ϵ are exactly the final states. □

The Pumping Lemma

Theorem 28 (Pumping Lemma)

Let $L \subseteq \Sigma^*$ be a regular language. Then there exists some positive integer n such that for every $w \in L$ with length at least n, we can decompose w into three strings ($w = xyz$) such that:

1. *y has length at least 1;*
2. *xy has length at most n;*
3. *for every $k \geq 0$, $xy^k z \in L$.*

The Pumping Lemma

Theorem 28 (Pumping Lemma)

Let $L \subseteq \Sigma^*$ be a regular language. Then there exists some positive integer n such that for every $w \in L$ with length at least n, we can decompose w into three strings ($w = xyz$) such that:

1. *y has length at least 1;*
2. *xy has length at most n;*
3. *for every $k \geq 0$, $xy^k z \in L$.*

Proof: Take a string of length at least n. Its path in a DFSA from an initial state to a final state must visit a state more than once by the pigeonhole principle, and so we have a cycle that we can pump around. □

The Pumping Lemma

Theorem 28 (Pumping Lemma)

Let $L \subseteq \Sigma^*$ be a regular language. Then there exists some positive integer n such that for every $w \in L$ with length at least n, we can decompose w into three strings ($w = xyz$) such that:

1. *y has length at least 1;*
2. *xy has length at most n;*
3. *for every $k \geq 0$, $xy^k z \in L$.*

Proof: Take a string of length at least n. Its path in a DFSA from an initial state to a final state must visit a state more than once by the pigeonhole principle, and so we have a cycle that we can pump around. □

Example 29

The language $L = \{a^n b^n \mid n \in \mathbb{N}\}$ is not regular.

Group Presentations I

The results we showed for monoids also hold for groups (as quotients of the free group). Moreover:

Group Presentations I

The results we showed for monoids also hold for groups (as quotients of the free group). Moreover:

Theorem 30

Let F_n be the free group of rank n and $R \subseteq F_n$. Define $\mathcal{R} = \{(r, \epsilon) \mid r \in R\}$ and N to be the normal closure of R. Then $F_n/N \cong F_n/\mathcal{R}$.

Group Presentations I

The results we showed for monoids also hold for groups (as quotients of the free group). Moreover:

Theorem 30

Let F_n be the free group of rank n and $R \subseteq F_n$. Define $\mathcal{R} = \{(r, \epsilon) \mid r \in R\}$ and N to be the normal closure of R. Then $F_n/N \cong F_n/\mathcal{R}$.

The usual approach is to view groups as quotients of the free group, but sometimes we want to make the handling of inverses explicit. In which case, we can view groups as a quotient of the free monoid, where we must throw in the rewrite rules to handle $aa^{-1} = 1 = a^{-1}a$ for all a. This is more also appropriate, since formally the elements of the free group are equivalence classes of strings, whereas the free monoid has elements that are single strings.

Group Presentations II

Proposition 31

Let Σ be a an alphabet, set $\bar{\Sigma}$ to be a disjoint copy of Σ, and let $X = \Sigma \cup \bar{\Sigma}$. If $\mathcal{R} = \{(a\bar{a}, \epsilon), (\bar{a}a, \epsilon) \mid a \in \Sigma\}$, then $F_{|\Sigma|} \cong X^*/\mathcal{R}$).

Group Presentations II

Proposition 31

Let Σ be a an alphabet, set $\bar{\Sigma}$ to be a disjoint copy of Σ, and let $X = \Sigma \cup \bar{\Sigma}$. If $\mathcal{R} = \{(a\bar{a}, \epsilon), (\bar{a}a, \epsilon) \mid a \in \Sigma\}$, then $F_{|\Sigma|} \cong X^*/\mathcal{R}$).

But, the free group need not be defined over an inverse closed alphabet...

Group Presentations II

Proposition 31

Let Σ be a an alphabet, set $\bar{\Sigma}$ to be a disjoint copy of Σ, and let $X = \Sigma \cup \bar{\Sigma}$. If $\mathcal{R} = \{(a\bar{a}, \epsilon), (\bar{a}a, \epsilon) \mid a \in \Sigma\}$, then $F_{|\Sigma|} \cong X^*/\mathcal{R}$).

But, the free group need not be defined over an inverse closed alphabet...

Example 32

Let $\Sigma = \{a, b, c\}$ and define $\mathcal{R} = \{(abc, \epsilon), (bca, \epsilon), (cab, \epsilon)\}$.

Group Presentations II

Proposition 31

Let Σ be a an alphabet, set $\bar{\Sigma}$ to be a disjoint copy of Σ, and let $X = \Sigma \cup \bar{\Sigma}$. If $\mathcal{R} = \{(a\bar{a}, \epsilon), (\bar{a}a, \epsilon) \mid a \in \Sigma\}$, then $F_{|\Sigma|} \cong {}^{X^*}/_{\mathcal{R}})$.

But, the free group need not be defined over an inverse closed alphabet...

Example 32

Let $\Sigma = \{a, b, c\}$ and define $\mathcal{R} = \{(abc, \epsilon), (bca, \epsilon), (cab, \epsilon)\}$.

Then $F_2 \cong {}^{\Sigma^*}/_{\mathcal{R}}$.

Group Presentations II

Proposition 31

Let Σ be a an alphabet, set $\bar{\Sigma}$ to be a disjoint copy of Σ, and let $X = \Sigma \cup \bar{\Sigma}$. If $\mathcal{R} = \{(a\bar{a}, \epsilon), (\bar{a}a, \epsilon) \mid a \in \Sigma\}$, then $F_{|\Sigma|} \cong {}^{X^*}/_{\mathcal{R}}$).

But, the free group need not be defined over an inverse closed alphabet...

Example 32

Let $\Sigma = \{a, b, c\}$ and define $\mathcal{R} = \{(abc, \epsilon), (bca, \epsilon), (cab, \epsilon)\}$.

Then $F_2 \cong {}^{\Sigma^*}/_{\mathcal{R}}$.

There are no explicit inverse letters! These three rewriting rules are known as the Dyck-system, and appeared on Dyck's foundational papers on the abstract theory of free groups.

The Word Problem

The word problem for monoids is classically defined to be:

The Word Problem

The word problem for monoids is classically defined to be:

Definition 33 ((Universal) Word Problem: Monoids)

Input: A (monoid presentation and) strings u, v.
Question: Is u equal to v in the monoid?

The Word Problem

The word problem for monoids is classically defined to be:

Definition 33 ((Universal) Word Problem: Monoids)

Input: A (monoid presentation and) strings u, v.
Question: Is u equal to v in the monoid?

But, in the case of groups, testing if two strings are equal in the group is the same as testing if $uv^{-1} = 1$ in the group. We thus define the word problem as follows:

The Word Problem

The word problem for monoids is classically defined to be:

Definition 33 ((Universal) Word Problem: Monoids)

Input: A (monoid presentation and) strings u, v.
Question: Is u equal to v in the monoid?

But, in the case of groups, testing if two strings are equal in the group is the same as testing if $uv^{-1} = 1$ in the group. We thus define the word problem as follows:

Definition 34 ((Universal) Word Problem: Groups)

Input: A (group presentation and a) string w.
Question: Is w equal to the identity in the group?

Group Languages

Definition 35 (Group Language)

Given a group presentation over X, the word problem is the language of words over X equal to the identity in the group. We call such a language a group language, whenever X is finite.

Group Languages

Definition 35 (Group Language)

Given a group presentation over X, the word problem is the language of words over X equal to the identity in the group. We call such a language a group language, whenever X is finite.

Theorem 36 (Parkes and Thomas (2002))

$L \subseteq X^*$ is a group language iff both:

1. *If $u \in X^*$, then $\exists v \in X^*$ such that $uv \in L$;*
2. *If $u, v \in L$ and $u \sqsubseteq v$, then v with u deleted is in L too.*

Group Languages

Definition 35 (Group Language)

Given a group presentation over X, the word problem is the language of words over X equal to the identity in the group. We call such a language a group language, whenever X is finite.

Theorem 36 (Parkes and Thomas (2002))

$L \subseteq X^*$ is a group language iff both:

1. *If $u \in X^*$, then $\exists v \in X^*$ such that $uv \in L$;*
2. *If $u, v \in L$ and $u \sqsubseteq v$, then v with u deleted is in L too.*

Theorem 37 (Anisimov (1971))

Call a group regular iff there exists a finite presentation for it with a regular word problem. A group is regular iff it is finite.

Proof

Proof: For the reverse direction, we can simply set the generating set to be all the elements in the group, and then have an FSA simply evaluate the finite multiplication table for the group, by having a state for each group element and having a transition between elements whenever the state multiplied by the edge label equals the label of the target state. Make the identity-labelled state the initial and final state.

Proof

Proof: For the reverse direction, we can simply set the generating set to be all the elements in the group, and then have an FSA simply evaluate the finite multiplication table for the group, by having a state for each group element and having a transition between elements whenever the state multiplied by the edge label equals the label of the target state. Make the identity-labelled state the initial and final state.
For the forward direction, suppose by contradiction that the group is infinite. Then there must be arbitrarily long string such that no non-empty substring equals the identity in the group, since if there were a bound on the length of such words, then every element of the group would be represented by only finitely many strings... a contradiction.

Proof

Proof: For the reverse direction, we can simply set the generating set to be all the elements in the group, and then have an FSA simply evaluate the finite multiplication table for the group, by having a state for each group element and having a transition between elements whenever the state multiplied by the edge label equals the label of the target state. Make the identity-labelled state the initial and final state.
For the forward direction, suppose by contradiction that the group is infinite. Then there must be arbitrarily long string such that no non-empty substring equals the identity in the group, since if there were a bound on the length of such words, then every element of the group would be represented by only finitely many strings... a contradiction. Now, let $\mathcal{A}$ be an FSA for the word problem, with n states, and choose a string w of length at least $n+1$ such that no non-empty substring is equal to the identity in the group. The, machine must visit a q state twice. Say, if visits q after reading u and again after then reading v, then $w = uvy$. Now, if $uu^{-1} = 1$ but $uvu^{-1} \neq 1$, $\mathcal{A}$ must accept or reject both strings... a contradiction! □

Context-Free Languages I

Definition 38

A grammar $\mathcal{G} = (T, N, \mathcal{R}, S)$ is context-free iff each rule is of the form $A \to w$ where $A \in N$ and $x \in (T \cup N)^*$.

Context-Free Languages I

Definition 38

A grammar $\mathcal{G} = (T, N, \mathcal{R}, S)$ is context-free iff each rule is of the form $A \to w$ where $A \in N$ and $x \in (T \cup N)^*$.

A pushdown automaton is like an FSA, but we also have a stack where we can push (zero or more stack symbols at once) and pop (exactly one stack symbol). The definition(s) (of acceptance) are more fiddly, so we omit them here, due to time constraints.

Context-Free Languages I

Definition 38

A grammar $\mathcal{G} = (T, N, \mathcal{R}, S)$ is context-free iff each rule is of the form $A \to w$ where $A \in N$ and $x \in (T \cup N)^*$.

A pushdown automaton is like an FSA, but we also have a stack where we can push (zero or more stack symbols at once) and pop (exactly one stack symbol). The definition(s) (of acceptance) are more fiddly, so we omit them here, due to time constraints.

Theorem 39 (Context-Free Language Characterisations)

Given $L \subseteq \Sigma^*$, the following are equivalent:

1. *L is generated by a context-free grammar;*
2. *L is accepted by a pushdown automaton.*

The deterministic pushdown automata recognise a strict subset of the context-free languages, called the deterministic context-free languages.

Context-Free Languages II

Theorem 40 (Muller, Schupp (1983))

A finitely generated group is virtually free iff it has a context-free word problem iff it has a deterministic context-free word problem.

Context-Free Languages II

Theorem 40 (Muller, Schupp (1983))

A finitely generated group is virtually free iff it has a context-free word problem iff it has a deterministic context-free word problem.

Example 41

The word problem of $\mathbb{Z}$ is context-free, but $\mathbb{Z}^2$ is not.

Context-Free Languages II

Theorem 40 (Muller, Schupp (1983))

A finitely generated group is virtually free iff it has a context-free word problem iff it has a deterministic context-free word problem.

Example 41

The word problem of $\mathbb{Z}$ is context-free, but $\mathbb{Z}^2$ is not.

Let $X = \{a, \bar{a}\}$ and $\mathcal{R} = \{a\bar{a} \to \epsilon, \bar{a}a \to \epsilon\}$, then ${}^{X^*}/_{\mathcal{R}} \cong \mathbb{Z}$. To see that $\mathbb{Z}$ is context-free, all we need to do is produce a context-free grammar for the language $\{w \in X^* \mid |w|_a = |w|_{\bar{a}}\}$, where $|w|_\sigma$ denotes the number of occurrences of σ in w.

Context-Free Languages II

Theorem 40 (Muller, Schupp (1983))

A finitely generated group is virtually free iff it has a context-free word problem iff it has a deterministic context-free word problem.

Example 41

The word problem of $\mathbb{Z}$ is context-free, but $\mathbb{Z}^2$ is not.

Let $X = \{a, \bar{a}\}$ and $\mathcal{R} = \{a\bar{a} \to \epsilon, \bar{a}a \to \epsilon\}$, then ${}^{X^*}/_{\mathcal{R}} \cong \mathbb{Z}$. To see that $\mathbb{Z}$ is context-free, all we need to do is produce a context-free grammar for the language $\{w \in X^* \mid |w|_a = |w|_{\bar{a}}\}$, where $|w|_\sigma$ denotes the number of occurrences of σ in w.

Such a grammar is $\mathcal{G} = (X, \{S\}, \{S \to aS\bar{a}S, S \to \bar{a}SaS, S \to \epsilon\}, S)$.

Beyond Context-Free

A Turing Machine is much like a pushdown automaton, only instead of a stack with operations push and pop, we have an infinite tape with a head which can read and write at its current position, and can move left/right.

Beyond Context-Free

A Turing Machine is much like a pushdown automaton, only instead of a stack with operations push and pop, we have an infinite tape with a head which can read and write at its current position, and can move left/right.

Definition 42

1. The context-sensitive languages are exactly those accepted by Turing Machines for which the number of steps is bounded by a fixed constant multiple of the length of the input string.

Beyond Context-Free

A Turing Machine is much like a pushdown automaton, only instead of a stack with operations push and pop, we have an infinite tape with a head which can read and write at its current position, and can move left/right.

Definition 42

1. The context-sensitive languages are exactly those accepted by Turing Machines for which the number of steps is bounded by a fixed constant multiple of the length of the input string.
2. The recursive languages are exactly those accepted by Turing Machines which halt on all inputs.

Beyond Context-Free

A Turing Machine is much like a pushdown automaton, only instead of a stack with operations push and pop, we have an infinite tape with a head which can read and write at its current position, and can move left/right.

Definition 42

1. The context-sensitive languages are exactly those accepted by Turing Machines for which the number of steps is bounded by a fixed constant multiple of the length of the input string.
2. The recursive languages are exactly those accepted by Turing Machines which halt on all inputs.
3. The recursively enumerable languages (r.e.) are those which are accepted by Turing Machines which don't necessarily halt.

Beyond Context-Free

A Turing Machine is much like a pushdown automaton, only instead of a stack with operations push and pop, we have an infinite tape with a head which can read and write at its current position, and can move left/right.

Definition 42

1. The context-sensitive languages are exactly those accepted by Turing Machines for which the number of steps is bounded by a fixed constant multiple of the length of the input string.
2. The recursive languages are exactly those accepted by Turing Machines which halt on all inputs.
3. The recursively enumerable languages (r.e.) are those which are accepted by Turing Machines which don't necessarily halt.

Theorem 43

The unrestricted grammars generate the r.e. languages.

Decidability I

Theorem 44

Fix some encoding of Turing Machines as strings. Then the language of all Turing Machines that accept themselves is not recursive, but is recursively enumerable.

Decidability I

Theorem 44

Fix some encoding of Turing Machines as strings. Then the language of all Turing Machines that accept themselves is not recursive, but is recursively enumerable.

Theorem 45

There are only countably many recursively enumerable languages.

Decidability I

Theorem 44

Fix some encoding of Turing Machines as strings. Then the language of all Turing Machines that accept themselves is not recursive, but is recursively enumerable.

Theorem 45

There are only countably many recursively enumerable languages.

Corollary 46

There are uncountably many languages that are not even recursively enumerable!

Decidability I

Theorem 44

Fix some encoding of Turing Machines as strings. Then the language of all Turing Machines that accept themselves is not recursive, but is recursively enumerable.

Theorem 45

There are only countably many recursively enumerable languages.

Corollary 46

There are uncountably many languages that are not even recursively enumerable!

Theorem 47 (Novikov (1955))

There is a finitely presented group whose word problem is not recursive.

Decidability II

The Church-Turing thesis says that every effective procedure can be carried out by a Turing machine...

Decidability II

The Church-Turing thesis says that every effective procedure can be carried out by a Turing machine...

Definition 48 (Decidability)

A problem is decidable if it can be solved in general by a Turing Machine. That is, the language of positive inputs under some encoding is recursive. Otherwise, it's called undecidable.

Decidability II

The Church-Turing thesis says that every effective procedure can be carried out by a Turing machine...

Definition 48 (Decidability)

A problem is decidable if it can be solved in general by a Turing Machine. That is, the language of positive inputs under some encoding is recursive. Otherwise, it's called undecidable.

Example 49 (Halting Problem)

The following problem is undecidable in general:
Input: A Turing Machine T.
Question: Does T halt on all inputs?

Decidability II

The Church-Turing thesis says that every effective procedure can be carried out by a Turing machine...

Definition 48 (Decidability)

A problem is decidable if it can be solved in general by a Turing Machine. That is, the language of positive inputs under some encoding is recursive. Otherwise, it's called undecidable.

Example 49 (Halting Problem)

The following problem is undecidable in general:
Input: A Turing Machine T.
Question: Does T halt on all inputs?

There are plenty more undecidable problems... in fact, there are only countably many non-equivalent decidable problems! For example, given a group presentation, we can't even decide if it admits a finite group!

Beyond Strings

The notation of rewriting generalised beyond strings, and is fundamental in computer science: lambda calculus, term rewriting, etc.

Beyond Strings

The notation of rewriting generalised beyond strings, and is fundamental in computer science: lambda calculus, term rewriting, etc.

An area in which I research is called algebraic graph transformation, where we look at rewriting on graphs using basic concepts from category theory.

Beyond Strings

The notation of rewriting generalised beyond strings, and is fundamental in computer science: lambda calculus, term rewriting, etc.

An area in which I research is called algebraic graph transformation, where we look at rewriting on graphs using basic concepts from category theory.

Graph rewrite rules are spans of the form $L \leftarrow K \rightarrow R$, and rule application to a graph G involves finding an injective morphism $g : L \rightarrow G$, constructing a pushout complement, then a pushout:

$$\begin{array}{ccccc} L & \longleftarrow & K & \longrightarrow & R \\ \downarrow{\scriptstyle g} & & \downarrow & & \downarrow{\scriptstyle h} \\ G & \longleftarrow & D & \longrightarrow & H \end{array}$$