HKN x acm

CS/ECE374~~A~~B Midterm 1

Regular languages and context free languages

Disclaimer and logistics

- This is being recorded. Recording will be on HKN website.
- Slides will be available on EdStem and HKN website later.
- Some of us are CAs, but we have not seen the exam. We have no idea what the questions are. However, we've reviewed the practice exams, and taken the course, so we have *suspicions* as to what the questions will be like.
- Slides that contain information that's not on your cheatsheet will be indicated as such!

Regular expressions

- A regular expression r describes a regular language $L(r)$
- Built inductively on three operators:
  - $+$ is the union operator. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$

Regular expressions

- A regular expression r describes a regular language $L(r)$
- Built inductively on three operators:
  - $+$ is the union operator. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$
  - $*$ is the Kleene star. $L(r_1^*) = L(r_1)^*$ $0 \ni \varepsilon, 0, 00, \ldots$

Regular expressions

- A regular expression r describes a regular language $L(r)$
- Built inductively on three operators:
  - $+$ is the union operator. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$
  - $*$ is the Kleene star. $L(r_1^*) = L(r_1)^*$
  - $()$ are used to group expressions $(0+1)^*$

Regular expressions

- A regular expression r describes a regular language $L(r)$
- Built inductively on three operators:
  - $+$ is the union operator. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$
  - $*$ is the Kleene star. $L(r_1^*) = L(r_1)^*$
  - $()$ are used to group expressions
  - (implicit) concatenation operator: $L(r_1 r_2) = \{xy : x \in L_1, y \in L_2\}$

Regular expressions

- A regular expression r describes a regular language $L(r)$
- Built inductively on three operators:
  - $+$ is the union operator. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$
  - $*$ is the Kleene star. $L(r_1^*) = L(r_1)^*$
  - $()$ are used to group expressions
  - (implicit) concatenation operator:
    $L(r_1 r_2) = \{xy : x \in L_1, y \in L_2\}$
- Precedence: first $()$, then *, then concatenation, then $+$

$a^* b$

Regular expressions

- A regular expression r describes a regular language $L(r)$
- Built inductively on three operators:
  - $+$ is the union operator. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$
  - $*$ is the Kleene star. $L(r_1^*) = L(r_1)^*$
  - $()$ are used to group expressions
  - (implicit) concatenation operator: $L(r_1 r_2) = \{xy : x \in L_1, y \in L_2\}$
- Precedence: first $()$, then *, then concatenation, then $+$
- Regular expressions must use a *finite* number of these operators

Aside: what does “regular” mean?

- A language is regular if there exists a DFA/NFA/regular language that *decides* it (that is, accepts every string in the language and rejects every string not in the language)

Aside: what does “regular” mean?

- A language is regular if there exists a DFA/NFA/regular language that *decides* it (that is, accepts every string in the language and rejects every string not in the language)
- But what does that *mean*? What’s special about a DFA or an NFA or a regular expression?

Aside: what does “regular” mean?

- A language is regular if there exists a DFA/NFA/regular language that *decides* it (that is, accepts every string in the language and rejects every string not in the language)
- But what does that *mean*? What’s special about a DFA or an NFA or a regular expression?
- Memory! Given enough paper, you can *always* draw out the DFA/NFA/regex for *any* regular language!

Aside: what does "regular" mean?

- A language is regular if there exists a DFA/NFA/regular language that *decides* it (that is, accepts every string in the language and rejects every string not in the language)
- But what does that *mean*? What's special about a DFA or an NFA or a regular expression?
- Memory! Given enough paper, you can *always* draw out the DFA/NFA/regex for *any* regular language!
- You can implement a DFA/NFA for your regular language in code using only $O(1)$ memory per state

Aside: what does “regular” mean?

- A language is regular if there exists a DFA/NFA/regular language that *decides* it (that is, accepts every string in the language and rejects every string not in the language)
- But what does that *mean*? What’s special about a DFA or an NFA or a regular expression?
- Memory! Given enough paper, you can *always* draw out the DFA/NFA/regex for *any* regular language!
- You can implement a DFA/NFA for your regular language in code using only $O(1)$ memory

Approaching regular expressions

- Most common type of question: given a language, write a regular expression for it

Approaching regular expressions

- Most common type of question: given a language, write a regular expression for it
- First write down some strings in the language. Are there any common patterns you notice?

Approaching regular expressions

- Most common type of question: given a language, write a regular expression for it
- First write down some strings in the language. Are there any common patterns you notice?
  - For example, number of consecutive ones, number of zeroes after a one, etc

Approaching regular expressions

- Most common type of question: given a language, write a regular expression for it
- First write down some strings in the language. Are there any common patterns you notice?
  - For example, number of consecutive ones, number of zeroes after a one, etc
- Next write down some strings *not* in the language. What makes these strings different?

Approaching regular expressions

- Most common type of question: given a language, write a regular expression for it
- First write down some strings in the language. Are there any common patterns you notice?
  - For example, number of consecutive ones, number of zeroes after a one, etc
- Next write down some strings *not* in the language. What makes these strings different?
- Can you decompose the given language into a combination of smaller languages?

Approaching regular expressions

- Most common type of question: given a language, write a regular expression for it
- First write down some strings in the language. Are there any common patterns you notice?
  - For example, number of consecutive ones, number of zeroes after a one, etc
- Next write down some strings *not* in the language. What makes these strings different?
- Can you decompose the given language into a combination of smaller languages?
- Build a regular expression based on your observations and test it. Does it accept / reject the strings you thought of above?

Approaching regular expressions

- Most common type of question: given a language, write a regular expression for it
- First write down some strings in the language. Are there any common patterns you notice?
  - For example, number of consecutive ones, number of zeroes after a one, etc
- Next write down some strings *not* in the language. What makes these strings different?
- Can you decompose the given language into a combination of smaller languages?
- Build a regular expression based on your observations and test it. Does it accept / reject the strings you thought of above?
- Explain, in English, each part of your regular expression with a short sentence. Does the explanation match the language?

Practice with regular expressions

- Example: let L be the set of all strings w such that in every prefix of w, the number of 0s and 1s differs by at most 1

0110

Practice with regular expressions

- Example: let L be the set of all strings w such that in every prefix of w, the number of 0s and 1s differs by at most 1
  - Some strings in this language: $0, 1, 01, 10101, 0110$

Practice with regular expressions

- Example: let L be the set of all strings w such that in every prefix of w, the number of 0s and 1s differs by at most 1
  - Some strings in this language: $0, 1, 01, 10101, 0110$
  - Some strings not in this language: $11, 1011, 0011$

Practice with regular expressions

- Example: let L be the set of all strings w such that in every prefix of w, the number of 0s and 1s differs by at most 1
  - Some strings in this language: $0, 1, 01, 10101, 0110$
  - Some strings not in this language: $11, 1011, 0011$
- What is the pattern here? Strings can't start with two 0s or two 1s, and can't have more than 2 0s or 1s in a row.

Practice with regular expressions

- Example: let L be the set of all strings w such that in every prefix of w, the number of 0s and 1s differs by at most 1
  - Some strings in this language: $0, 1, 01, 10101, 0110$
  - Some strings not in this language: $11, 1011, 0011$
- What is the pattern here? Strings can't start with two 0s or two 1s, and can't have more than 2 0s or 1s in a row.
- Can you simplify this pattern? Strings are formed by repeated $(01 + 10)$, and may end with a 1, 0, or nothing.

$(01+10)^*(1+0+\varepsilon)$

Practice with regular expressions

- Example: let L be the set of all strings w such that in every prefix of w, the number of 0s and 1s differs by at most 1
  - Some strings in this language: $0, 1, 01, 10101, 0110$
  - Some strings not in this language: $11, 1011, 0011$
- What is the pattern here? Strings can't start with two 0s or two 1s, and can't have more than 2 0s or 1s in a row.
- Can you simplify this pattern? Strings are formed by repeated $(01 + 10)$, and may end with a 1, 0, or nothing.
- Solution: $(01 + 10)^*(\epsilon + 0 + 1)$

Practice with regular expressions

- Example: let L be the set of all strings w such that in every prefix of w, the number of 0s and 1s differs by at most 1
  - Some strings in this language: 0, 1, 01, 10101, 0110
  - Some strings not in this language: 11, 1011, 0011
- What is the pattern here? Strings can't start with two 0s or two 1s, and can't have more than 2 0s or 1s in a row.
- Can you simplify this pattern? Strings are formed by repeated $(01 + 10)$, and may end with a 1, 0, or nothing.
- Solution: $(01 + 10)^*(\epsilon + 0 + 1)$
- Exercise: which, if any, of the following are also correct:
  $(\epsilon + 0 + 1)^*(01 + 10)^*$, 1110 X
  $(01 + 10 + \cancel{1010 + 0101 + 0110 + 1001})^*(\epsilon + 0 + 1)$ ✓

Practice with regular expressions

- Example: let L be the set of all strings w such that in every prefix of w, the number of 0s and 1s differs by at most 1
  - Some strings in this language: 0, 1, 01, 10101, 0110
  - Some strings not in this language: 11, 1011, 0011
- What is the pattern here? Strings can't start with two 0s or two 1s, and can't have more than 2 0s or 1s in a row.
- Can you simplify this pattern? Strings are formed by repeated $(01 + 10)$, and may end with a 1, 0, or nothing.
- Solution: $(01 + 10)^*(\epsilon + 0 + 1)$
- Exercise: which, if any, of the following are also correct: $(\epsilon + 0 + 1)^*(01 + 10)^*$, $(01 + 10 + 1010 + 0101 + 0110 + 1001)^*(\epsilon + 0 + 1)$
  - The first is wrong with counterexmaple 001. The second is correct. $1010, 0101, 0110, 1001$ are all the strings of length 4 in $(01 + 10)^*$

Practice with regular expressions 2

- Let L be the set of all strings where every run of 1s has odd length.

Practice with regular expressions 2

- Let L be the set of all strings where every run of 1s has odd length.
  - Some strings in this language: $0, 00, 101, 111$

Practice with regular expressions 2

- Let L be the set of all strings where every run of 1s has odd length.
  - Some strings in this language: 0, 00, 101, 111
  - Some strings not in this language: 11, 1101, 0011

$$0^*\left(1(11)^*\,0\,0^*\right)^*\left(\varepsilon + 1(11)^*\right)$$

Practice with regular expressions 2

- Let L be the set of all strings where every run of 1s has odd length.
  - Some strings in this language: 0, 00, 101, 111
  - Some strings not in this language: 11, 1101, 0011
- A string containing an odd number of 1s is given by $(11)^*1$.

Practice with regular expressions 2

- Let L be the set of all strings where every run of 1s has odd length.
  - Some strings in this language: 0, 00, 101, 111
  - Some strings not in this language: 11, 1101, 0011
- A string containing an odd number of 1s is given by $(11)^*1$.
- Above, we notice each such string is separated by 1 or more 0s: $((11)^*10^+)^*$

Practice with regular expressions 2

- Let L be the set of all strings where every run of 1s has odd length.
    - Some strings in this language: 0, 00, 101, 111
    - Some strings not in this language: 11, 1101, 0011
- A string containing an odd number of 1s is given by $(11)^*1$.
- Above, we notice each such string is separated by 1 or more 0s: $((11)^*10^+)^*$
- What is this missing?

Practice with regular expressions 2

- Let L be the set of all strings where every run of 1s has odd length.
  - Some strings in this language: $0, 00, 101, 111$
  - Some strings not in this language: $11, 1101, 0011$
- A string containing an odd number of 1s is given by $(11)^*1$.
- Above, we notice each such string is separated by 1 or more 0s: $((11)^*10^+)^*$
- What is this missing?
  - There may be *no* zeroes!

Practice with regular expressions 2

- Let L be the set of all strings where every run of 1s has odd length.
  - Some strings in this language: $0, 00, 101, 111$
  - Some strings not in this language: $11, 1101, 0011$
- A string containing an odd number of 1s is given by $(11)^*1$.
- Above, we notice each such string is separated by 1 or more 0s: $((11)^*10^+)^*$
- What is this missing?
  - There may be *no* zeroes!
- Final answer: $0^*((11)^*10^+)^* + (11)^*1$

Closure properties

- What does it mean to be “closed under” an operation?

Closure properties

- What does it mean to be “closed under” an operation?
- A set X is closed under $\circ$ if for $a, b \in X$, $a \circ b \in X$

Closure properties

- What does it mean to be "closed under" an operation?
- A set X is closed under $\circ$ if for $a, b \in X$, $a \circ b \in X$
  - Example: a vector space is closed under addition

Closure properties

- What does it mean to be “closed under” an operation?
- A set X is closed under $\circ$ if for $a, b \in X$, $a \circ b \in X$
  - Example: a vector space is closed under addition
- The set of all regular languages is closed under the following:

Closure properties

- What does it mean to be “closed under” an operation?
- A set X is closed under $\circ$ if for $a, b \in X$, $a \circ b \in X$
  - Example: a vector space is closed under addition
- The set of all regular languages is closed under the following:
  - Union ($\cup$), intersection ($\cap$), concatenation ($\cdot$), kleene star (*), complement (^{C}), set difference ($\setminus$), and reverse (^{R})

Closure properties

- What does it mean to be "closed under" an operation?
- A set X is closed under $\circ$ if for $a, b \in X$, $a \circ b \in X$
  - Example: a vector space is closed under addition
- The set of all regular languages is closed under the following:
  - Union ($\cup$), intersection ($\cap$), concatenation ($\cdot$), kleene star (*), complement (^{C}), set difference ($\setminus$), and reverse (^{R})
- How do we know this? Think of NFA/DFA transformations from Thompson's algorithm and product construction (we did a special case of set difference earlier!)
- Regular languages are closed only under *finite* applications of these operations - this stems from the fact that a DFA must have a finite number of states, so we can't do an infinite product construction.
  - Note: sometimes an infinite sequence of these operations *will* result in a language that's still regular, but *not always*!

Short answer regular languages

- Is $(0+1+L)^*$ regular for *any* language L? $\rightarrow (0+1)^*$

Short answer regular languages

- Is $(0+1+L)^*$ regular for *any* language L?
  - Yes! $(0+1+L)^* = (0+1)^*$ since $L \subseteq (0+1)^*$

Short answer regular languages

- Is $(0+1+L)^*$ regular for *any* language L?
  - Yes! $(0+1+L)^* = (0+1)^*$ since $L \subseteq (0+1)^*$
- Tip: *Don't forget about* ϵ*!*

Short answer regular languages

- Is $(0+1+L)^*$ regular for *any* language L?
  - Yes! $(0+1+L)^* = (0+1)^*$ since $L \subseteq (0+1)^*$
- Tip: *Don't forget about ϵ!*
- Is $L = \{xyy^R z : x, y, z \in \Sigma^*\}$ regular?

Short answer regular languages

- Is $(0+1+L)^*$ regular for *any* language L?
  - Yes! $(0+1+L)^* = (0+1)^*$ since $L \subseteq (0+1)^*$
- Tip: *Don't forget about ϵ!*
- Is $L = \{xyy^Rz : x, y, z \in \Sigma^*\}$ regular?
  - Intuition about palindromes: When a palindrome has a *minimum* length, but no maximum, language is irregular. Why? If minimum but no maximum, we need $O(n)$ memory to keep track of what might be in the palindrome.

010000

$L_Y = (00+01)^*$

$L_Y L_Y^R$

0000010000

Short answer regular languages

- Is $(0 + 1 + L)^*$ regular for *any* language L?
  - Yes! $(0 + 1 + L)^* = (0 + 1)^*$ since $L \subseteq (0 + 1)^*$
- Tip: *Don't forget about ϵ!*
- Is $L = \{xyy^R z : x, y, z \in \Sigma^*\}$ regular?
  - Intuition about palindromes: When a palindrome has a *minimum* length, but no maximum, language is irregular. Why? If minimum but no maximum, we need $O(n)$ memory to keep track of what might be in the palindrome.
  - Note that we can set $y = \epsilon$ so then $L = \{xz : x, z \in \Sigma^*\} = \Sigma^*$ so L is regular!

Short answer regular languages 2

- If L is not regular, then is L^* also not regular?

$L = 0^p$ p prime $\quad L^* = 00^*$, reg.

$L = 1 + 0 + \underbrace{L}_{\text{bad}} \rightarrow L^* = (1+0)^*$

Short answer regular languages 2

- If L is not regular, then is L^* also not regular?
  - No! Let $L = \{0^n 1^n : n \geq 0\} \cup \{0, 1\}$. Since $0, 1 \in L$, $L^* \supseteq \{0, 1\}^*$ so $L^* = \{0, 1\}^*$.

Short answer regular languages 2

- If L is not regular, then is L^* also not regular?
  - No! Let $L = \{0^n 1^n : n \geq 0\} \cup \{0, 1\}$. Since $0, 1 \in L$, $L^* \supseteq \{0, 1\}^*$ so $L^* = \{0, 1\}^*$.
- If L^* is not regular, then is L also not regular?

Short answer regular languages 2

- If L is not regular, then is L^* also not regular?
  - No! Let $L = \{0^n 1^n : n \geq 0\} \cup \{0, 1\}$. Since $0, 1 \in L$, $L^* \supseteq \{0, 1\}^*$ so $L^* = \{0, 1\}^*$.
- If L^* is not regular, then is L also not regular?
  - Yes! If L was regular, then L^* would be regular.

Short answer regular languages 2

- If L is not regular, then is L^* also not regular?
  - No! Let $L = \{0^n 1^n : n \geq 0\} \cup \{0, 1\}$. Since $0, 1 \in L$, $L^* \supseteq \{0, 1\}^*$ so $L^* = \{0, 1\}^*$.
- If L^* is not regular, then is L also not regular?
  - Yes! If L was regular, then L^* would be regular.
- If L_1 is not regular and L_2 is regular, then must $L_1 L_2$ be irregular?

Short answer regular languages 2

- If L is not regular, then is L^* also not regular?
  - No! Let $L = \{0^n1^n : n \geq 0\} \cup \{0, 1\}$. Since $0, 1 \in L$, $L^* \supseteq \{0, 1\}^*$ so $L^* = \{0, 1\}^*$.
- If L^* is not regular, then is L also not regular?
  - Yes! If L was regular, then L^* would be regular.
- If L_1 is not regular and L_2 is regular, then must L_1L_2 be irregular?
  - No. Consider $L_1 = \{0^{2^n} : n \geq 1\} \cup \{\epsilon\}$ and $L_2 = 0^*$. Then, $L_1L_2 = 0^*$, which is regular.

Building DFAs and NFAs

- Begin by decomposing your language into basic parts that can be combined using logical operators such as $\wedge$ and $\vee$

Building DFAs and NFAs

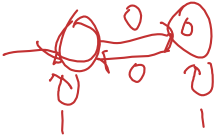
- Begin by decomposing your language into basic parts that can be combined using logical operators such as $\wedge$ and $\vee$
- For each “simple” sub-language L_i, write down some strings that are and are not in the language

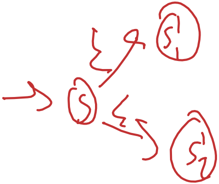
Building DFAs and NFAs

- Begin by decomposing your language into basic parts that can be combined using logical operators such as $\wedge$ and $\vee$
- For each "simple" sub-language L_i, write down some strings that are and are not in the language
- See what patterns you observe, and sketch a DFA/NFA

Building DFAs and NFAs

- Begin by decomposing your language into basic parts that can be combined using logical operators such as $\wedge$ and $\vee$
- For each "simple" sub-language L_i, write down some strings that are and are not in the language
- See what patterns you observe, and sketch a DFA/NFA
- Validate your DFA/NFA. If asked, make sure to define **all five** elements of the tuple $(Q, \Sigma, \delta, s, A)$! This is an easy way to lose points!





Building DFAs and NFAs

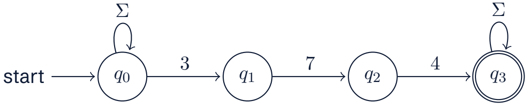
- Begin by decomposing your language into basic parts that can be combined using logical operators such as $\wedge$ and $\vee$
- For each "simple" sub-language L_i, write down some strings that are and are not in the language
- See what patterns you observe, and sketch a DFA/NFA
- Validate your DFA/NFA. If asked, make sure to define **all five** elements of the tuple $(Q, \Sigma, \delta, s, A)$! This is an easy way to lose points!
  - And, make sure your set of states is finite!

Practice building an NFA/DFA

- Let $L = \{w \in \Sigma^* : 374 \text{ is a substring of } w\}$

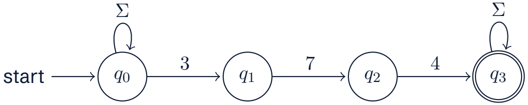
Practice building an NFA/DFA

- Let $L = \{w \in \Sigma^* : 374 \text{ is a substring of } w\}$



Practice building an NFA/DFA

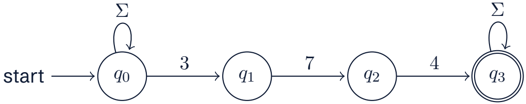
- Let $L = \{w \in \Sigma^* : 374 \text{ is a substring of } w\}$



- What about the missing transitions?

Practice building an NFA/DFA

- Let $L = \{w \in \Sigma^* : 374$ is a substring of $w\}$



- What about the missing transitions? Simply write "all missing transitions lead to reject"!

Practice building an NFA/DFA 2

- Let $L_z = \{w \in \Sigma^* : z \text{ is a prefix of } w\}$

Practice building an NFA/DFA 2

- Let $L_z = \{w \in \Sigma^* : z$ is a prefix of $w\}$

$$
\begin{aligned}
Q &= \{0, \ldots, |z|\} \\
s &= 0 \\
A &= \{|z|\} \\
\delta(i, z_{i+1}) &= i + 1 \text{ for } i < |z| \\
\delta(|z|, x) &= |z| \ \forall x
\end{aligned}
$$

Practice building an NFA/DFA 2

- Let $L_z = \{w \in \Sigma^* : z$ is a prefix of $w\}$

$$Q = \{0, \ldots, |z|\}$$
$$s = 0$$
$$A = \{|z|\}$$
$$\delta(i, z_{i+1}) = i + 1 \text{ for } i < |z|$$
$$\delta(|z|, x) = |z| \; \forall x$$

- What about the missing transitions?

bacon

Practice building an NFA/DFA 2

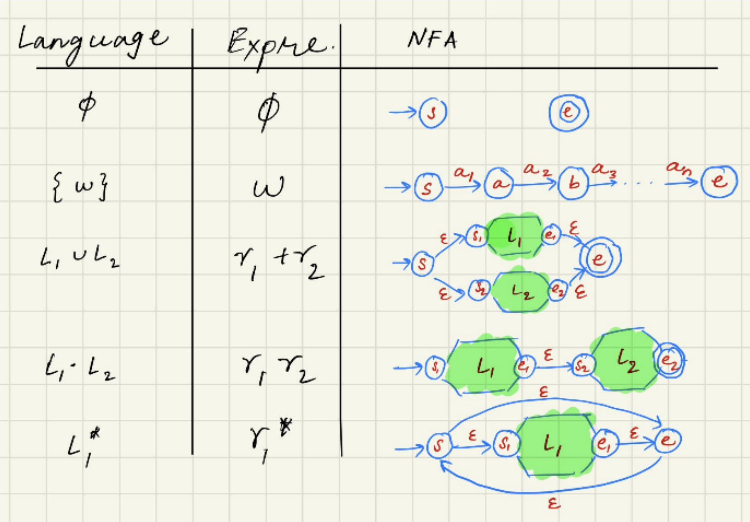
- Let $L_z = \{w \in \Sigma^* : z \text{ is a prefix of } w\}$

$$
\begin{aligned}
Q &= \{0, \ldots, |z|\} \\
s &= 0 \\
A &= \{|z|\} \\
\delta(i, z_{i+1}) &= i + 1 \text{ for } i < |z| \\
\delta(|z|, x) &= |z| \; \forall x
\end{aligned}
$$

- What about the missing transitions? Simply write “all missing transitions lead to reject”!

Combine with Thompson's Algorithm

Note: not all of this is not on your cheatsheet! You should make sure you remember it, just in case.



Combine with product construction

- Given some languages $L_1, \ldots, L_n$ we want a DFA that accepts strings w satisfying $f(w \in L_1, \ldots, w \in L_n)$ where f is some logical function.

Combine with product construction

- Given some languages $L_1, \ldots, L_n$ we want a DFA that accepts strings w satisfying $f(w \in L_1, \ldots, w \in L_n)$ where f is some logical function.
  - For example, L is the set of strings in L_1 and not in L_2

Combine with product construction

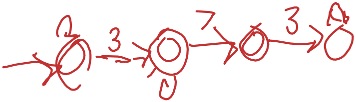
- Given some languages $L_1, \ldots, L_n$ we want a DFA that accepts strings w satisfying $f(w \in L_1, \ldots, w \in L_n)$ where f is some logical function.
  - For example, L is the set of strings in L_1 and not in L_2
- Create a DFA/NFA for L using the following *rough* format:
  - $Q = Q_1 \times \cdots \times Q_n$
  - $\delta'(q_1, \ldots, q_n) = (\delta_1(q_1), \ldots, \delta_2(q_2))$
  - $s = (s_1, \ldots, s_n)$
  - $A' = \{\text{convert } f \text{ into a set expression}\}$

Practice product construction

- Let L be the set of strings that contain the substring 374 and not the substring 373. Create a DFA for L.

Practice product construction

- Let L be the set of strings that contain the substring 374 and not the substring 373. Create a DFA for L.
- We already know these two languages are regular, so we'll skip the process of creating machines M_{374} and M_{373} (but you should show this on the exam!)



Practice product construction

- Let L be the set of strings that contain the substring 374 and not the substring 373. Create a DFA for L.
- We already know these two languages are regular, so we'll skip the process of creating machines M_{374} and M_{373} (but you should show this on the exam!)
- Let's set up the machinery to run M_{373} and M_{374} in parallel:

3 5 7 374

Practice product construction

- Let L be the set of strings that contain the substring 374 and not the substring 373. Create a DFA for L.
- We already know these two languages are regular, so we'll skip the process of creating machines M_{374} and M_{373} (but you should show this on the exam!)
- Let's set up the machinery to run M_{373} and M_{374} in parallel:
  - $Q = Q_{373} \times Q_{374}$
  - $\delta((q_1, q_2), x) = (\delta_{373}(x), \delta_{374}(x))$
  - $s = (s_{373}, s_{374})$

Practice product construction

- Let L be the set of strings that contain the substring 374 and not the substring 373. Create a DFA for L.
- We already know these two languages are regular, so we'll skip the process of creating machines M_{374} and M_{373} (but you should show this on the exam!)
- Let's set up the machinery to run M_{373} and M_{374} in parallel:
  - $Q = Q_{373} \times Q_{374}$
  - $\delta((q_1, q_2), x) = (\delta_{373}(x), \delta_{374}(x))$
  - $s = (s_{373}, s_{374})$
- Now, what is A? We accept when M_{373} is *not* in an accept state, **and** when M_{374} is in an accept state:

Practice product construction

- Let L be the set of strings that contain the substring 374 and not the substring 373. Create a DFA for L.
- We already know these two languages are regular, so we'll skip the process of creating machines M_{374} and M_{373} (but you should show this on the exam!)
- Let's set up the machinery to run M_{373} and M_{374} in parallel:
  - $Q = Q_{373} \times Q_{374}$
  - $\delta((q_1, q_2), x) = (\delta_{373}(x), \delta_{374}(x))$
  - $s = (s_{373}, s_{374})$
- Now, what is A? We accept when M_{373} is *not* in an accept state, **and** when M_{374} is in an accept state:
  - $A' = (Q_{373} \setminus A_{373}) \times A_{374}$

$A' = (q_1 \in A_{373}$ and $q_2 \in A_{374})$
or
!

Language transformations

- Skip something: ex Half$(L) = \{$cut$(w) : w \in L\}$ where cut removes characters at odd indices (strings are 1-indexed, cut(hi) = i)

Language transformations

- Skip something: ex Half$(L) = \{\text{cut}(w) : w \in L\}$ where cut removes characters at odd indices (strings are 1-indexed, cut(hi) = i)
- Insert something: ex Double$(L) = \{w : \text{cut}(w) \in L\}$ where cut is defined as above

Language transformations

- Skip something: ex Half$(L) = \{\text{cut}(w) : w \in L\}$ where cut removes characters at odd indices (strings are 1-indexed, cut(hi) = i)
- Insert something: ex Double$(L) = \{w : \text{cut}(w) \in L\}$ where cut is defined as above
- Change something: ex: Quiet$(L) = \{\text{quiet}(w) : w \in L\}$ where quiet replaces any ! with a .

Language transformations

- Skip something: ex Half$(L) = \{$cut$(w) : w \in L\}$ where cut removes characters at odd indices (strings are 1-indexed, cut(hi) = i)
- Insert something: ex Double$(L) = \{w :$ cut$(w) \in L\}$ where cut is defined as above
- Change something: ex: Quiet$(L) = \{$quiet$(w) : w \in L\}$ where quiet replaces any ! with a .
- Your NFA should "undo" the operation. For example, in the "Skip something" case, un-cut the string. Always go left-to-right!

Language transformations

- Skip something: ex Half$(L) = \{\text{cut}(w) : w \in L\}$ where cut removes characters at odd indices (strings are 1-indexed, cut(hi) = i)
- Insert something: ex Double$(L) = \{w : \text{cut}(w) \in L\}$ where cut is defined as above
- Change something: ex: Quiet$(L) = \{\text{quiet}(w) : w \in L\}$ where quiet replaces any ! with a .
- Your NFA should "undo" the operation. For example, in the "Skip something" case, un-cut the string. Always go left-to-right!
- Approach: first create a language L that will be impacted by the transformation

Language transformations

- Skip something: ex Half$(L) = \{\text{cut}(w) : w \in L\}$ where cut removes characters at odd indices (strings are 1-indexed, cut(hi) = i)
- Insert something: ex Double$(L) = \{w : \text{cut}(w) \in L\}$ where cut is defined as above
- Change something: ex: Quiet$(L) = \{\text{quiet}(w) : w \in L\}$ where quiet replaces any ! with a .
- Your NFA should "undo" the operation. For example, in the "Skip something" case, un-cut the string. Always go left-to-right!
- Approach: first create a language L that will be impacted by the transformation
    - Example: for Quiet we might pick $L = \{\text{Hi!}, \text{Bye.}, \text{Hi.}\}$

Language transformations

- Skip something: ex Half$(L) = \{\text{cut}(w) : w \in L\}$ where cut removes characters at odd indices (strings are 1-indexed, cut(hi) = i)
- Insert something: ex Double$(L) = \{w : \text{cut}(w) \in L\}$ where cut is defined as above
- Change something: ex: Quiet$(L) = \{\text{quiet}(w) : w \in L\}$ where quiet replaces any ! with a .
- Your NFA should "undo" the operation. For example, in the "Skip something" case, un-cut the string. Always go left-to-right!
- Approach: first create a language L that will be impacted by the transformation
  - Example: for Quiet we might pick $L = \{\text{Hi!}, \text{Bye.}, \text{Hi.}\}$
- Next, compute the transformed language $f(L)$

Language transformations

- Skip something: ex $\text{Half}(L) = \{\text{cut}(w) : w \in L\}$ where cut removes characters at odd indices (strings are 1-indexed, cut(hi) = i)
- Insert something: ex $\text{Double}(L) = \{w : \text{cut}(w) \in L\}$ where cut is defined as above
- Change something: ex: $\text{Quiet}(L) = \{\text{quiet}(w) : w \in L\}$ where quiet replaces any ! with a .
- Your NFA should “undo” the operation. For example, in the “Skip something” case, un-cut the string. Always go left-to-right!
- Approach: first create a language L that will be impacted by the transformation
    - Example: for Quiet we might pick $L = \{\text{Hi!}, \text{Bye.}, \text{Hi.}\}$
- Next, compute the transformed language $f(L)$
    - Example: $\text{Quiet}(L) = \{\text{Hi.}, \text{Bye.}\}$

Language transformations

- Skip something: ex Half$(L) = \{$cut$(w) : w \in L\}$ where cut removes characters at odd indices (strings are 1-indexed, cut(hi) = i)
- Insert something: ex Double$(L) = \{w :$ cut$(w) \in L\}$ where cut is defined as above
- Change something: ex: Quiet$(L) = \{$quiet$(w) : w \in L\}$ where quiet replaces any ! with a .
- Your NFA should "undo" the operation. For example, in the "Skip something" case, un-cut the string. Always go left-to-right!
- Approach: first create a language L that will be impacted by the transformation
  - Example: for Quiet we might pick $L = \{\text{Hi!}, \text{Bye.}, \text{Hi.}\}$
- Next, compute the transformed language $f(L)$
  - Example: Quiet$(L) = \{\text{Hi.}, \text{Bye.}\}$
- Figure out what you need to go from a string in $f(L)$ to the original string in L. *This is what your NFA for $f(L)$ needs to do!*

Language transformations

- Skip something: ex Half$(L) = \{$cut$(w) : w \in L\}$ where cut removes characters at odd indices (strings are 1-indexed, cut(hi) = i)
- Insert something: ex Double$(L) = \{w :$ cut$(w) \in L\}$ where cut is defined as above
- Change something: ex: Quiet$(L) = \{$quiet$(w) : w \in L\}$ where quiet replaces any ! with a .
- Your NFA should "undo" the operation. For example, in the "Skip something" case, un-cut the string. Always go left-to-right!
- Approach: first create a language L that will be impacted by the transformation
  - Example: for Quiet we might pick $L = \{\text{Hi!}, \text{Bye.}, \text{Hi.}\}$
- Next, compute the transformed language $f(L)$
  - Example: Quiet$(L) = \{\text{Hi.}, \text{Bye.}\}$
- Figure out what you need to go from a string in $f(L)$ to the original string in L. *This is what your NFA for $f(L)$ needs to do!*
  - Example: we need to change Hi. to Hi!, and Bye. stays the

Practice language transformations

- $\text{Quiet}(L) = \{\text{quiet}(w) : w \in L\}$

Practice language transformations

- Quiet$(L) = \{$quiet$(w) : w \in L\}$
- When our NFA sees a . in the input string, it *might* need to replace it with a !, but it *might* not

Hi.

Hi!

Practice language transformations

- Quiet$(L) = \{$quiet$(w) : w \in L\}$
- When our NFA sees a . in the input string, it *might* need to replace it with a !, but it *might* not
- Intuition: our NFA should simulate the DFA M_Q for Quiet(L), but override the transition function for .

Practice language transformations

- Quiet$(L) = \{$quiet$(w) : w \in L\}$
- When our NFA sees a . in the input string, it *might* need to replace it with a !, but it *might* not
- Intuition: our NFA should simulate the DFA M_Q for Quiet(L), but override the transition function for .

Hi.

- Proposed solution:
  - $Q = Q_Q$
  - $s = s_Q$
  - $\delta(q, x) = \{\delta_Q(q, x)\}$ if $x \neq .$, otherwise $\delta(q, .) = \{\delta_Q(q, .), \delta_Q(q, !)\}$
  - $A = A_Q$

Practice language transformations

- Quiet$(L) = \{$quiet$(w) : w \in L\}$
- When our NFA sees a . in the input string, it *might* need to replace it with a !, but it *might* not
- Intuition: our NFA should simulate the DFA M_Q for Quiet(L), but override the transition function for .
- Proposed solution:
  - $Q = Q_Q$
  - $s = s_Q$
  - $\delta(q, x) = \{\delta_Q(q, x)\}$ if $x \neq$., otherwise $\delta(q, .) = \{\delta_Q(q, .), \delta_Q(q, !)\}$
  - $A = A_Q$
- What's wrong here? Does M accept Hi!? Should it?

Practice language transformations

- Quiet$(L) = \{$quiet$(w) : w \in L\}$
- When our NFA sees a . in the input string, it *might* need to replace it with a !, but it *might* not
- Intuition: our NFA should simulate the DFA M_Q for Quiet(L), but override the transition function for .
- Proposed solution:
  - $Q = Q_Q$
  - $s = s_Q$
  - $\delta(q, x) = \{\delta_Q(q, x)\}$ if $x \neq .$, otherwise $\delta(q, .) = \{\delta_Q(q, .), \delta_Q(q, !)\}$
  - $A = A_Q$
- What's wrong here? Does M accept Hi!? Should it?
- Fix: $\delta(q, !) = \{$reject$\}$

Practice language transformations

- Quiet$(L) = \{$quiet$(w) : w \in L\}$
- When our NFA sees a . in the input string, it *might* need to replace it with a !, but it *might* not
- Intuition: our NFA should simulate the DFA M_Q for Quiet(L), but override the transition function for .
- Proposed solution:
  - $Q = Q_Q$
  - $s = s_Q$
  - $\delta(q, x) = \{\delta_Q(q, x)\}$ if $x \neq .$, otherwise $\delta(q, .) = \{\delta_Q(q, .), \delta_Q(q, !)\}$
  - $A = A_Q$
- What's wrong here? Does M accept Hi!? Should it?
- Fix: $\delta(q, !) = \{$reject$\}$
- Why did we need an NFA here?

Practice language transformations

- Quiet$(L) = \{$quiet$(w) : w \in L\}$
- When our NFA sees a . in the input string, it *might* need to replace it with a !, but it *might* not
- Intuition: our NFA should simulate the DFA M_Q for Quiet(L), but override the transition function for .
- Proposed solution:
  - $Q = Q_Q$
  - $s = s_Q$
  - $\delta(q, x) = \{\delta_Q(q, x)\}$ if $x \neq$., otherwise $\delta(q, .) = \{\delta_Q(q, .), \delta_Q(q, !)\}$
  - $A = A_Q$
- What's wrong here? Does M accept Hi!? Should it?
- Fix: $\delta(q, !) = \{$reject$\}$
- Why did we need an NFA here?Usually we have to consider multiple options, and an NFA is the easiest way to do this.

Practice language transformations 2

- $\text{Double}(L) = \{w : \text{cut}(w) \in L\}$
- When our DFA M' sees a character at an odd index, it shouldn't send the character into M, since we want $\text{cut}(w) \in L$.

Practice language transformations 2

- Double$(L) = \{w : \text{cut}(w) \in L\}$
- When our DFA M' sees a character at an odd index, it shouldn't send the character into M, since we want cut$(w) \in L$.
- Intuition: track whether the next character should be skipped, and flip this every time we read a character

Practice language transformations 2

- Double$(L) = \{w : \text{cut}(w) \in L\}$
- When our DFA M' sees a character at an odd index, it shouldn't send the character into M, since we want cut$(w) \in L$.
- Intuition: track whether the next character should be skipped, and flip this every time we read a character
- Partial solution:
  - $Q' = Q \times \{T, F\}$

Practice language transformations 2

- $\text{Double}(L) = \{w : \text{cut}(w) \in L\}$
- When our DFA M' sees a character at an odd index, it shouldn't send the character into M, since we want $\text{cut}(w) \in L$.
- Intuition: track whether the next character should be skipped, and flip this every time we read a character
- Partial solution:
  - $Q' = Q \times \{T, F\}$
  - $\delta'((q, T), a) = (q, F)$

Practice language transformations 2

- Double$(L) = \{w : \text{cut}(w) \in L\}$
- When our DFA M' sees a character at an odd index, it shouldn't send the character into M, since we want cut$(w) \in L$.
- Intuition: track whether the next character should be skipped, and flip this every time we read a character
- Partial solution:
  - $Q' = Q \times \{T, F\}$
  - $\delta'((q, T), a) = (q, F)$
  - $\delta'((q, F), a) = (\delta(q, a), T)$

Practice language transformations 2

- Double$(L) = \{w : \text{cut}(w) \in L\}$
- When our DFA M' sees a character at an odd index, it shouldn't send the character into M, since we want cut$(w) \in L$.
- Intuition: track whether the next character should be skipped, and flip this every time we read a character
- Partial solution:
  - $Q' = Q \times \{T, F\}$
  - $\delta'((q, T), a) = (q, F)$
  - $\delta'((q, F), a) = (\delta(q, a), T)$
- What should s' be?

Practice language transformations 2

- Double$(L) = \{w : \text{cut}(w) \in L\}$
- When our DFA M' sees a character at an odd index, it shouldn't send the character into M, since we want cut$(w) \in L$.
- Intuition: track whether the next character should be skipped, and flip this every time we read a character
- Partial solution:
  - $Q' = Q \times \{T, F\}$
  - $\delta'((q, T), a) = (q, F)$
  - $\delta'((q, F), a) = (\delta(q, a), T)$
- What should s' be? We need to skip the first character, so $s' = (s, T)$

Practice language transformations 2

- Double$(L) = \{w : \text{cut}(w) \in L\}$
- When our DFA M' sees a character at an odd index, it shouldn't send the character into M, since we want cut$(w) \in L$.
- Intuition: track whether the next character should be skipped, and flip this every time we read a character
- Partial solution:
  - $Q' = Q \times \{T, F\}$
  - $\delta'((q, T), a) = (q, F)$
  - $\delta'((q, F), a) = (\delta(q, a), T)$
- What should s' be? We need to skip the first character, so $s' = (s, T)$
- What should A' be?

Practice language transformations 2

- Double$(L) = \{w : \text{cut}(w) \in L\}$
- When our DFA M' sees a character at an odd index, it shouldn't send the character into M, since we want cut$(w) \in L$.
- Intuition: track whether the next character should be skipped, and flip this every time we read a character
- Partial solution:
  - $Q' = Q \times \{T, F\}$
  - $\delta'((q, T), a) = (q, F)$
  - $\delta'((q, F), a) = (\delta(q, a), T)$
- What should s' be? We need to skip the first character, so $s' = (s, T)$
- What should A' be? We don't care about our skipping state, so $A' = A \times \{T, F\}$

Irregularity

- What should you think when you see regular?

Irregularity

- What should you think when you see regular? Finite!

Irregularity

- What should you think when you see regular? Finite!
- An *irregular* language would need an infinite number of states

Irregularity

- What should you think when you see regular? Finite!
- An *irregular* language would need an infinite number of states
- In CS374, we use fooling sets to show that a language is irregular

Irregularity

- What should you think when you see regular? Finite!
- An *irregular* language would need an infinite number of states
- In CS374, we use fooling sets to show that a language is irregular
- A fooling set F for a language L is a set such that for any $x, y \in F$, there exists a suffix z so that $xz \in L$ and $yz \notin L$

Irregularity

- What should you think when you see regular? Finite!
- An *irregular* language would need an infinite number of states
- In CS374, we use fooling sets to show that a language is irregular
- A fooling set F for a language L is a set such that for any $x, y \in F$, there exists a suffix z so that $xz \in L$ and $yz \notin L$
  - By Myhill-Nerode, if a language L has a fooling set of size $|F|$, any DFA for that language must have at least $|F|$ states

Irregularity

- What should you think when you see regular? Finite!
- An *irregular* language would need an infinite number of states
- In CS374, we use fooling sets to show that a language is irregular
- A fooling set F for a language L is a set such that for any $x, y \in F$, there exists a suffix z so that $xz \in L$ and $yz \notin L$
  - By Myhill-Nerode, if a language L has a fooling set of size $|F|$, any DFA for that language must have at least $|F|$ states
  - This implies that the existence of an *infinite* fooling set for a language L means L is irregular!

Irregularity

- What should you think when you see regular? Finite!
- An *irregular* language would need an infinite number of states
- In CS374, we use fooling sets to show that a language is irregular
- A fooling set F for a language L is a set such that for any $x, y \in F$, there exists a suffix z so that $xz \in L$ and $yz \notin L$
  - By Myhill-Nerode, if a language L has a fooling set of size $|F|$, any DFA for that language must have at least $|F|$ states
  - This implies that the existence of an *infinite* fooling set for a language L means L is irregular!
- Intuition: when you need to keep track of unbounded information (i.e. occurrences of a character), the language is probably irregular

Practice fooling sets

- Find the size (or, a rough bound on it) of the largest fooling set for each language. Which is regular?
  - $L_1 = \{w \in \{0,1\}^* : \#_1(w) - \#_0(w) \geq 374\}$
  - $L_2 = \{w \in \{0,1\}^* : \#_1(w) - \#_0(w) \text{ mod } 3 \geq 1\}$

Practice fooling sets

- Find the size (or, a rough bound on it) of the largest fooling set for each language. Which is regular?
  - $L_1 = \{w \in \{0,1\}^* : \#_1(w) - \#_0(w) \geq 374\}$
  - $L_2 = \{w \in \{0,1\}^* : \#_1(w) - \#_0(w) \text{ mod } 3 \geq 1\}$
- What did this show us? mod usually has finitely many states!

Practice fooling sets

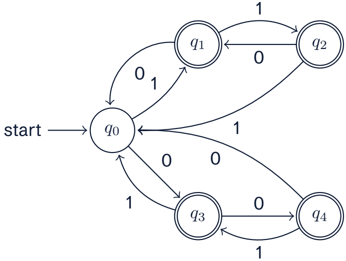
- Find the size (or, a rough bound on it) of the largest fooling set for each language. Which is regular?
  - $L_1 = \{w \in \{0,1\}^* : \#_1(w) - \#_0(w) \geq 374\}$
  - $L_2 = \{w \in \{0,1\}^* : \#_1(w) - \#_0(w) \text{ mod } 3 \geq 1\}$
- What did this show us? mod usually has finitely many states!
  - L_1 has the following infinite fooling set: $F = \{1^i : i > 500\}$

Practice fooling sets

- Find the size (or, a rough bound on it) of the largest fooling set for each language. Which is regular?
  - $L_1 = \{w \in \{0,1\}^* : \#_1(w) - \#_0(w) \geq 374\}$
  - $L_2 = \{w \in \{0,1\}^* : \#_1(w) - \#_0(w) \text{ mod } 3 \geq 1\}$
- What did this show us? mod usually has finitely many states!
  - L_1 has the following infinite fooling set: $F = \{1^i : i > 500\}$
    - For $x = 1^i$ and $y = 1^j$ where $i < j$, $z = 0^{i-373}$ is a distinguising suffix since $i - (i - 373) = 373 < 374$ but since $j \geq i + 1$, $j - (i - 373) \geq 374$.

Practice fooling sets

- Find the size (or, a rough bound on it) of the largest fooling set for each language. Which is regular?
  - $L_1 = \{w \in \{0,1\}^* : \#_1(w) - \#_0(w) \geq 374\}$
  - $L_2 = \{w \in \{0,1\}^* : \#_1(w) - \#_0(w) \text{ mod } 3 \geq 1\}$
- What did this show us? mod usually has finitely many states!
  - L_1 has the following infinite fooling set: $F = \{1^i : i > 500\}$
    - For $x = 1^i$ and $y = 1^j$ where $i < j$, $z = 0^{i-373}$ is a distinguising suffix since $i - (i - 373) = 373 < 374$ but since $j \geq i + 1$, $j - (i - 373) \geq 374$.
  - We can write a DFA for L_2 with 5 states, so $|F| \leq 5$



Context-free languages

- A context-free language L is generated by a context-free grammar $G = (V, T, P, S)$

Context-free languages

- A context-free language L is generated by a context-free grammar $G = (V, T, P, S)$
- What does context-free mean? Concatenation, union, *and recursion*

$S \to AB$

$S \to A \mid B$

$S \to \ldots S \ldots$

Context-free languages

- A context-free language L is generated by a context-free grammar $G = (V, T, P, S)$
- What does context-free mean? Concatenation, union, *and recursion*
  - This means every regular language is context free, but not every context free language is regular

Context-free languages

- A context-free language L is generated by a context-free grammar $G = (V, T, P, S)$
- What does context-free mean? Concatenation, union, *and recursion*
  - This means every regular language is context free, but not every context free language is regular
- CFLs only closed under union, kleene star, and concatenation

Context-free languages

- A context-free language L is generated by a context-free grammar $G = (V, T, P, S)$
- What does context-free mean? Concatenation, union, *and recursion*
  - This means every regular language is context free, but not every context free language is regular
- CFLs only closed under union, kleene star, and concatenation
- Intuition 1: a CFL "builds" strings, going from the outside in, you can choose any rule to add characters at the left / right

Context-free languages

- A context-free language L is generated by a context-free grammar $G = (V, T, P, S)$
- What does context-free mean? Concatenation, union, *and recursion*
  - This means every regular language is context free, but not every context free language is regular
- CFLs only closed under union, kleene star, and concatenation
- Intuition 1: a CFL "builds" strings, going from the outside in, you can choose any rule to add characters at the left / right
  - Example $L = \{0^i 1^{2i} : i > 0\}$. To build 001111, we only need the production rule $S \to 0S11|\epsilon$. We start with $0S11$, then replace the S with $0S11$ to get $00S1111$ and then take the other case, ϵ

Context-free languages

- A context-free language L is generated by a context-free grammar $G = (V, T, P, S)$
- What does context-free mean? Concatenation, union, *and recursion*
  - This means every regular language is context free, but not every context free language is regular
- CFLs only closed under union, kleene star, and concatenation
- Intuition 1: a CFL "builds" strings, going from the outside in, you can choose any rule to add characters at the left / right
  - Example $L = \{0^i 1^{2i} : i > 0\}$. To build 001111, we only need the production rule $S \to 0S11|\epsilon$. We start with $0S11$, then replace the S with $0S11$ to get $00S1111$ and then take the other case, ϵ
- Intuition 2: a CFL "peels back" strings, starting from a chosen pivot point.

Context-free languages

- A context-free language L is generated by a context-free grammar $G = (V, T, P, S)$
- What does context-free mean? Concatenation, union, *and recursion*
  - This means every regular language is context free, but not every context free language is regular
- CFLs only closed under union, kleene star, and concatenation
- Intuition 1: a CFL "builds" strings, going from the outside in, you can choose any rule to add characters at the left / right
  - Example $L = \{0^i 1^{2i} : i > 0\}$. To build 001111, we only need the production rule $S \to 0S11|\epsilon$. We start with $0S11$, then replace the S with $0S11$ to get $00S1111$ and then take the other case, ϵ
- Intuition 2: a CFL "peels back" strings, starting from a chosen pivot point.
  - Example: follow the above example, but in reverse

Context-free languages

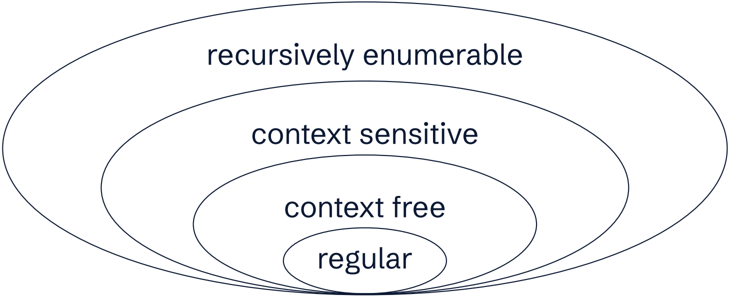
- A context-free language L is generated by a context-free grammar $G = (V, T, P, S)$
- What does context-free mean? Concatenation, union, *and recursion*
  - This means every regular language is context free, but not every context free language is regular
- CFLs only closed under union, kleene star, and concatenation
- Intuition 1: a CFL "builds" strings, going from the outside in, you can choose any rule to add characters at the left / right
  - Example $L = \{0^i 1^{2i} : i > 0\}$. To build 001111, we only need the production rule $S \to 0S11|\epsilon$. We start with $0S11$, then replace the S with $0S11$ to get $00S1111$ and then take the other case, ϵ
- Intuition 2: a CFL "peels back" strings, starting from a chosen pivot point.
  - Example: follow the above example, but in reverse
- Both ways to think about CFLs are completely fine!

The Chomsky Hierarchy

- Remember, all regular languages are context-free!

The Chomsky Hierarchy

- Remember, all regular languages are context-free!
- You'll learn about other classes of languages later in the course.



Practice with CFLs

- Let $L = \{0^j 1^i 0^k : i + j \neq k\}$

$i+j < k$

$i+j > k$

Practice with CFLs

$i \neq k - j$

CFLs are NOT closed under negation or intersection

- Let $L = \{0^j 1^i 0^k : i + j \neq k\}$
- First try:

$$S \to 0S0 | N$$
$$N \to 1N0 | 1 | 0$$

$A^m B^m C^n \leftarrow$ CFL

$A^m B^n C^n \leftarrow$

$A^n B^n C^n \leftarrow$ NOT CFL

Practice with CFLs

- Let $L = \{0^j 1^i 0^k : i + j \neq k\}$
- First try:

$$S \to 0S0|N$$
$$N \to 1N0|1|0$$

- What's wrong with this?

Practice with CFLs

- Let $L = \{0^j 1^i 0^k : i + j \neq k\}$
- First try:

11111

$$S \to 0S0 | N$$
$$N \to 1N0 | 1 | 0$$

1 . . . 1 1 0 . . . 0
0

- What's wrong with this? Forces $j = k$. How can we solve this?

Practice with CFLs

- Let $L = \{0^j 1^i 0^k : i + j \neq k\}$
- First try:

$$S \to 0S0|N$$
$$N \to 1N0|1|0$$

- What's wrong with this? Forces $j = k$. How can we solve this?
- Casework!

Practice with CFLs

- Let $L = \{0^j 1^i 0^k : i + j \neq k\}$
- First try:

$$S \to 0S0|N$$
$$N \to 1N0|1|0$$

- What's wrong with this? Forces $j = k$. How can we solve this?
- Casework! Create production rules L and G for when $j < k$ and $j > k$, respectively:

Practice with CFLs

- Let $L = \{0^j 1^i 0^k : i + j \neq k\}$
- First try:

$$S \to 0S0|N$$
$$N \to 1N0|1|0$$

- What's wrong with this? Forces $j = k$. How can we solve this?
- Casework! Create production rules L and G for when $j < k$ and $j > k$, respectively:

$$S \to 0S0|N|L0|0G$$
$$L \to L0|N0$$
$$G \to 0G|0N$$
$$N \to 1N0|1|0$$

0 1111110 i+j+

$\{0^j 1^i 0^k : i+j \leq k\}$

i+j ≥ k

$0^j 1^i 0^m 0^k$

$0^j 1^i 1^m 0^k$

$0^j 0^m 1^i 1^k$

$S \to (N)S \mid \varepsilon$

$0^m 1^n$ where $m<n$

$0^m 1^{k+m}$

$N \to 0N1 \mid N1 \mid 1$

$\{(0+1)^* \mid$ every run of 0's is followed by a longer run of 1's.$\}$

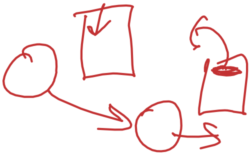
$S \to \varepsilon \mid NS$

$N \to 1 \mid 0N1$

$\underbrace{0011111}_{N}\underbrace{011}_{N}\underbrace{000111}_{N}\underbrace{1}_{N}$

Push Down Automata

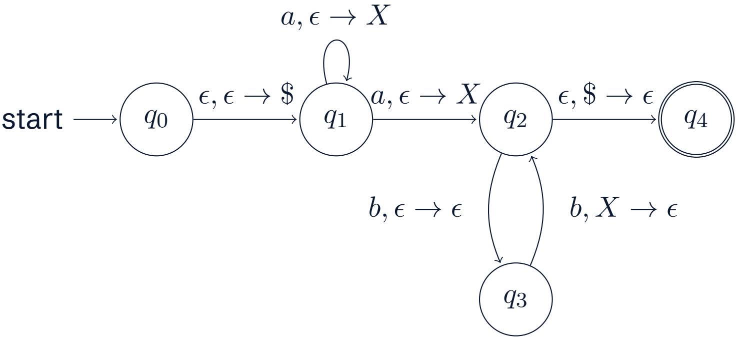
DFA + stack



- Unlikey to be on exam, alternative to using a CFG to prove a language is context free.
- Adds a stack to a FSM, making it able to decide context free grammars.
- Each transition can interact with the stack by popping or pushing a character from the stack alphabet.
- In order to take a transition, the top of the stack and the input must match the transition rule. Unless they are ϵ.
- Formally defined as a 6-Tuple: $(Q, \Sigma, \Gamma, s, A, \delta)$

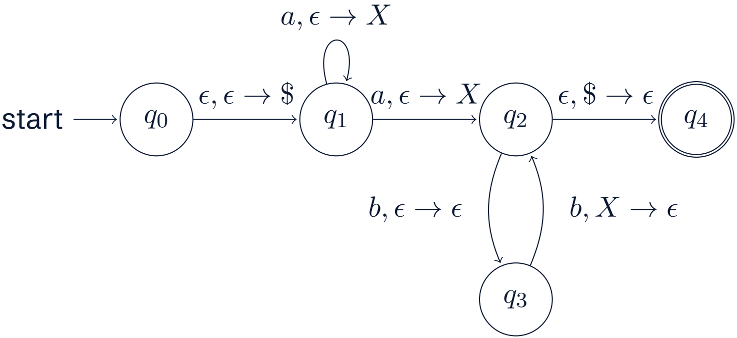
Practice with PDAs

- What language does this PDA accept?



Practice with PDAs

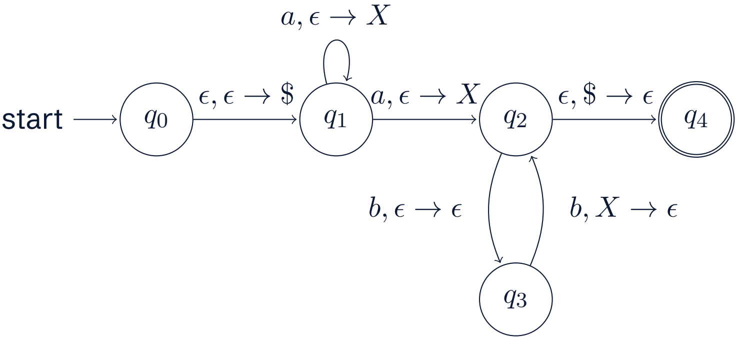
- What language does this PDA accept?



- $L = \{a^n b^{2n} \mid n \geq 1\}$

Practice with PDAs

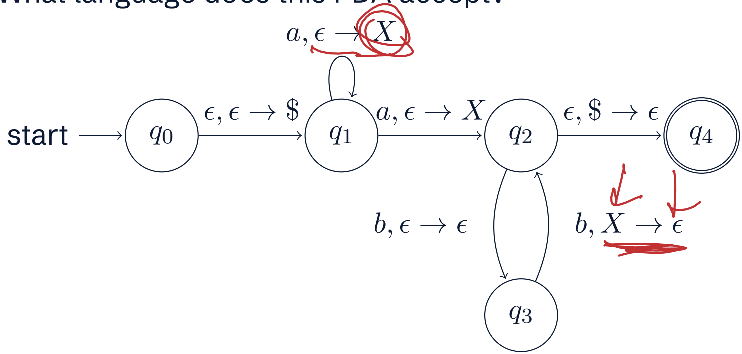
- What language does this PDA accept?



- $L = \{a^n b^{2n} \mid n \geq 1\}$
- What happens if we change $q_1 \to q_2$ to $\epsilon, \epsilon \to \epsilon$?

Practice with PDAs

- What language does this PDA accept?



- $L = \{a^n b^{2n} \mid n \geq 1\}$
- What happens if we change $q_1 \to q_2$ to $\epsilon, \epsilon \to \epsilon$?
- $L = \{a^n b^{2n} \mid n \geq 0\}$

Advice and tips

- If you know a language is regular, remember that there is always an NFA, DFA, and regular expression for the language

Advice and tips

- If you know a language is regular, remember that there is always an NFA, DFA, and regular expression for the language
- When you see recursive definitions, you should almost always use induction!

Advice and tips

- If you know a language is regular, remember that there is always an NFA, DFA, and regular expression for the language
- When you see recursive definitions, you should almost always use induction!
- Writing something is better than nothing

Advice and tips

- If you know a language is regular, remember that there is always an NFA, DFA, and regular expression for the language
- When you see recursive definitions, you should almost always use induction!
- Writing something is better than nothing
- Go easiest to hardest

Advice and tips

- If you know a language is regular, remember that there is always an NFA, DFA, and regular expression for the language
- When you see recursive definitions, you should almost always use induction!
- Writing something is better than nothing
- Go easiest to hardest
- Sanity check your answers (consider edge cases: strings of length 1 or 0)

Advice and tips

- If you know a language is regular, remember that there is always an NFA, DFA, and regular expression for the language
- When you see recursive definitions, you should almost always use induction!
- Writing something is better than nothing
- Go easiest to hardest
- Sanity check your answers (consider edge cases: strings of length 1 or 0)
- Practice until you could explain the concept to a friend

Closing Remarks

- Good luck on Midterm 1!
- Please fill out feedback form for this review session.
- Worksheet links and feedback form: