Regular Expressions

1

Rethinking Regular Languages

- We currently have several tools for showing a language L is regular:
  - Construct a DFA for L.
  - Construct an NFA for L.
  - Combine several simpler regular languages together via closure properties to form L

Today's Topic

- Idea: Build up all regular languages as follows:
  - Start with a small set of simple languages we already know to be regular
  - Using closure properties, combine these simple languages together to form more elaborate languages

- Bottom-up approach

2

Regular Expressions

- Consider a language $L = \{ w \in \Sigma^* \mid w \text{ contains } \mathbf{aa} \text{ as a substring} \}$
  - Difficult to manipulate with verbal descriptions
- Regular expressions are a way of describing a language via a string representation
  - **Easy to apply language operators**
  - Shortened to ***regex***
- Wide applicability
  - JavaScript: for data validation
  - UNIX ***grep*** and flex tools: search files and build compilers.
  - Used to clean and scrape data for largescale analysis projects.
- Conceptually, regular expressions are strings describing how to assemble a larger language out of smaller pieces

Atomic Regular Expressions

- The regular expressions begin with three simple building blocks
- The symbol **Ø** is a regular expression that represents the empty language Ø
- Any **single character** a ∈ Σ, the symbol a is a regular expression for the language {a}
- The symbol **ε** is a regular expression that represents the language {ε}
  - Remember: {ε} ≠ Ø
  - Remember: {ε} ≠ ε

Compound Regular Expressions

- If R1 and R2 are regular expressions, R1 R2 is a regular expression for the **concatenation** of the languages of R1 and R2
- If R1 and R2 are regular expressions, R1 ∪ R2 is a regular expression for the **union** of the languages of R1 and R2
- If R is a regular expression, ***R**** is a regular expression for the Kleene closure of the language of R
- If R is a regular expression, ***(R)*** is a regular expression with the same meaning as R

- How about **complement** and **intersection**?

Operator Precedence

- Here's the operator precedence for regular expressions:

  (R)

  R*

  R1 R2

  R1 ∪ R2

- ab*c∪d is parsed as ((a(b*))c)∪d

Regex Examples

Regular Expression Examples

- The regular expression trick∪treat represents the language
  { trick, treat }
- The regular expression booo* represents the regular language
  { boo, booo, boooo, … }
- The regular expression candy!(candy!)* represents the regular language
  { candy!, candy!candy!, candy!candy!candy!, … }.

Regular Expressions, Formally

- The language of a regular expression is the language described by that regular expression

- Formally:
  - $\mathcal{L}(\varepsilon) = \{\varepsilon\}$
  - $\mathcal{L}(\emptyset) = \emptyset$
  - $\mathcal{L}(a) = \{a\}$
  - $\mathcal{L}(R1\ R2) = \mathcal{L}(R1)\,\mathcal{L}(R2)$
  - $\mathcal{L}(R1 \cup R2) = \mathcal{L}(R1) \cup \mathcal{L}(R2)$
  - $\mathcal{L}(R^*) = \mathcal{L}(R)^*$
  - $\mathcal{L}((R)) = \mathcal{L}(R)$

Designing Regular Expressions

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

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

  $\Sigma^*aa\Sigma^*$

- Let $\Sigma = \{a, b\}$.
- Let $L = \{ w \in \Sigma^* \mid |w| = 4 \}$

  Length of a string w

  $\Sigma\Sigma\Sigma\Sigma$

  Σ^4

Designing Regular Expressions

- Let Σ = {a, b}
- Let L = { w ∈ Σ* | w contains at most one a }

- Which of the following expressions are correct (Multiple choices)?
  ① Σ*aΣ*
  ② b*ab*
  ③ b*
  ④ b*(a ∪ ε)b*
  ⑤ b*a*b* ∪ b*
  ⑥ b*(a* ∪ ε)b*

Use ? For one or none occurrence of character a? ➔ (a ∪ ε)

A More Elaborate Design

- Let Σ = { a, ., @ }, where a represents "some letter"
- Let's make a regex for email addresses
- Legitimate email addresses
  - anon2022@gmail.com
  - Gil.dong.hong@kentech.ac.kr

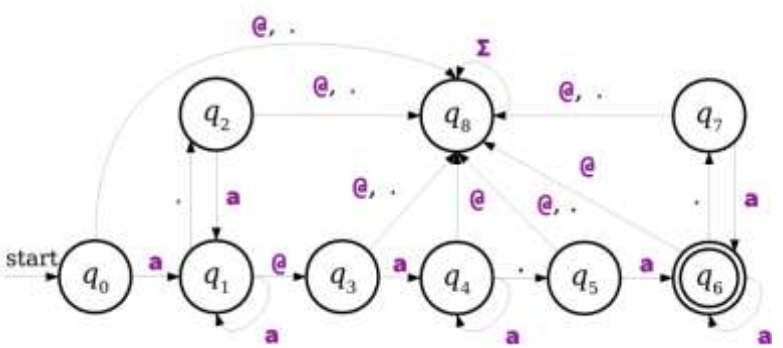
aa* (. aa*)* @ aa* (. aa*) (. aa*)*

aa* = $\mathbf{a}^+$

$\mathbf{a}^+$ (. $\mathbf{a}^+$)* @ $\mathbf{a}^+$ (. $\mathbf{a}^+$)$^+$

Regex vs DFA

- $\mathbf{a}^+ \ (. \mathbf{a}^+)^* \ @ \ \mathbf{a}^+ \ (. \mathbf{a}^+)^+$



Important Symbols

- $\mathbf{R}^n$ is shorthand for RR … R (n times)
  - Edge case: define $R^0 = \varepsilon$.
- **Σ** is shorthand for "any character in Σ"
- **R?** is shorthand for $(R \cup \varepsilon)$, meaning "zero or one copies of R"
- $\mathbf{R}^+$ is shorthand for RR*, meaning "one or more copies of R"

15

Question

AI

Language you can build a DFA/NFA for → Regular Languages

Language you can write a Regex for → Languages (RGL)

Regular language (RL) and Regex language (RGL)
RL ⊆≡RGL
RL ⊇ RGL
RL ∩ RGL ≠ ∅
RL ∩ RGL ≡ ∅
RL ≡ RGL

16

The Power of Regular Expressions

- Theorem: If R is a regular expression, then $\mathcal{L}(R)$ is regular.

- Proof idea: Use induction!
  - The atomic regular expressions all represent regular languages
  - The combination steps represent closure properties
  - So anything you can make from them must be regular!

Thompson's Algorithm

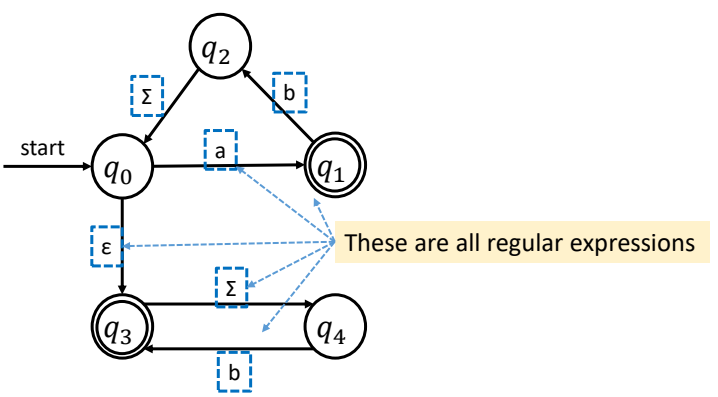
- In practice, many regex matchers use an algorithm called Thompson's algorithm to convert **regular expressions into NFAs** (and, from there, to DFAs).
  - Read Sipser for details

Ken Thompson (1943) is an American pioneer of computer science. At Bell Labs., he designed and implemented the original Unix OS and invented the B programming language, the direct Predecessor to the C. At Google, he co-developed the Go programming language. He won the Turing Award in 1983 with his long-term colleague Dennis Ritchie

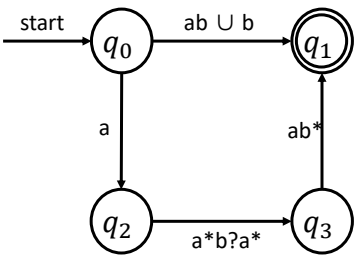
The Power of Regular Expressions

- Theorem: If L is a regular language, then there is a regular expression for L
- This is not obvious!
- Proof idea: Show how to convert an arbitrary NFA into a regular expression.

Generalizing NFAs

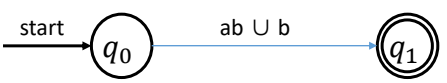


Generalizing NFAs

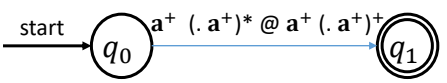


Key Idea 1: Imagine that we can label transitions in an NFA with arbitrary regular expressions

NFA & Regex



Regex: ab ∪ b

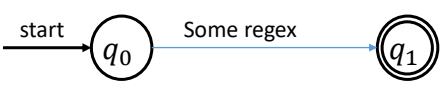


Regex: $\mathbf{a}^+$ (. $\mathbf{a}^+$)* @ $\mathbf{a}^+$ (. $\mathbf{a}^+$)$^+$

Generalizing NFAs

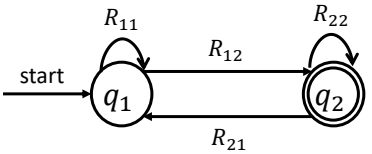
AI

Key Idea 2: If we can convert an NFA into a generalized NFA that looks like this.

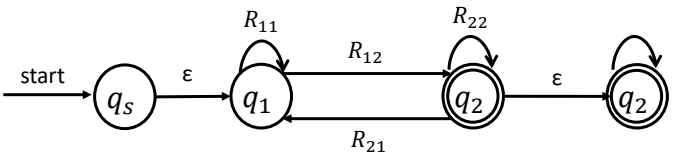


then we can easily read off a regular expression for the original NFA.

NFA ➔ Regex

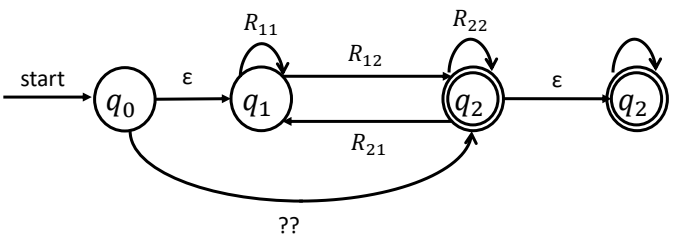


First, add two redundant states; start and accept states



Goal: Remove states q_1 and q_2 AND connects q_s and q_f directly

NFA ➔ Regex

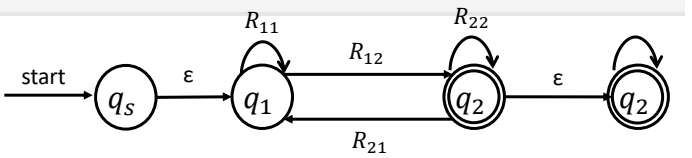


The State-Elimination Algorithm

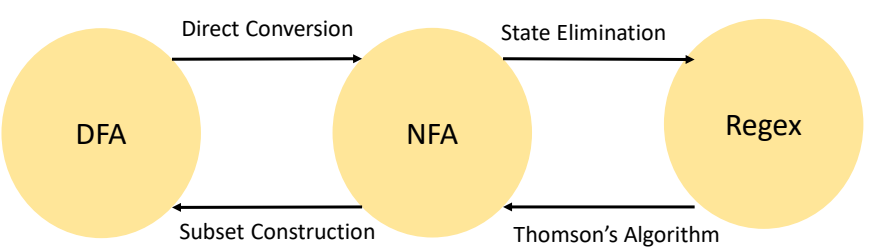
- Start with an NFA N for the language L.
- Add a new start state q_s and accept state q_f to the NFA.
  - Add an ε-transition from q_s to the old start state of N.
  - Add ε-transitions from each accepting state of N to q_f , then mark them as not accepting
- Repeatedly remove states other than q_s and q_f from the NFA by "shortcutting" them until only two states remain: q_s and q_f
- The transition from q_s to q_f is then a regular expression for the NFA

The State-Elimination Algorithm

- To eliminate a state **q** from the automaton, do the following for each pair of states q_0 and q_1, where there's a transition from q_0 into **q** and a transition from **q** into q_1:
  - Let Rin be the regex on the transition from q_0 to q
  - Let Rout be the regex on the transition from q to q_1
  - If there is a regular expression Rstay on a transition from q to itself, add a new transition from q_0 to q_1 labeled ((Rin)(Rstay)*(Rout)).
  - If there isn't, add a new transition from q_0 to q_1 labeled ((Rin)(Rout))
- If a pair of states has multiple transitions between them labeled R_1, R_2, …, R_k, replace them with a single transition labeled $R_1 \cup R_2 \cup \ldots \cup R_k$



Transformations



Equivalences

- Theorem: The following are all equivalent: ·
  - L is a regular language
  - There is a DFA D such that $\mathcal{L}(D) = L$
  - There is an NFA N such that $\mathcal{L}(N) = L$
  - There is a regular expression R such that $\mathcal{L}(R) = L$

- The equivalence of regular expressions and finite automata has practical relevance
  - Regular expression matchers have all the power available to them of DFAs and NFAs
- This also is hugely theoretically significant: the regular languages can be assembled "from scratch" using a small number of operations!

Nonregular Languages

Computers as Finite Automata

- My computer has 12GB of RAM and about 150GB of SSD space
- That's a total of 162GB of memory, which is 1,391,569,403,904 bits
- There are "only" $2^{1,391,569,403,904}$ possible configurations of the memory in my computer
- You could in principle build a DFA representing my computer, where there's one symbol per type of input the computer can receive.

A Powerful Intuition

- Regular languages correspond to problems that can be solved with finite memory
  - At each point in time, we only need to store one of finitely many pieces of information
- Nonregular languages, in a sense, correspond to problems that cannot be solved with finite memory
- Since every computer ever built has finite memory, in a sense, nonregular languages correspond to problems that cannot be solved by physical computers!

Finding Nonregular Languages

- To prove that a language is regular, we can just find a DFA, NFA, or regex for it
- How to prove that a certain language is nonregular?

➔ We need to prove that there are no possible DFAs, NFAs, or regexes for it

Challenging!

- Claim: We can actually just prove that there's no DFA for it

A Simple Famous Language

- Let Σ = {a, b} and consider the following language:

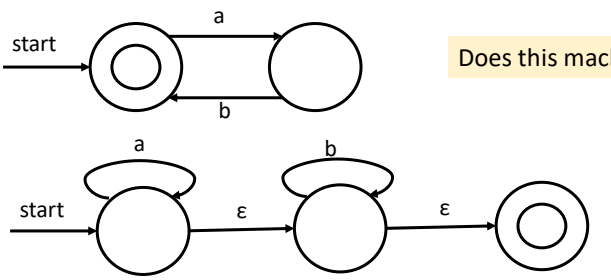
$$E = \{a^n b^n \mid n \in \mathbb{N}\}$$

- E is the language of all strings of n a's followed by n b's:
  - { ε, ab, aabb, aaabbb, aaaabbbb, … }

- Regex for *E*
  - How many of the following are regex for *E*?
    - ① $a^n b^n$
    - ② $a^* b^*$
    - ③ $(ab)^n$
    - ④ ε ∪ ab ∪ $a^2 b^2$ ∪ $a^3 b^3$

Another Attempt - Build NFA

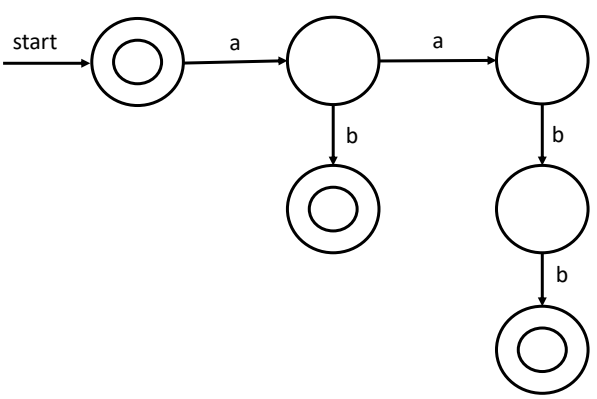
- Try to design an NFA for $E = \{a^n b^n \mid n \in \mathbb{N}\}$



Does this machine work?

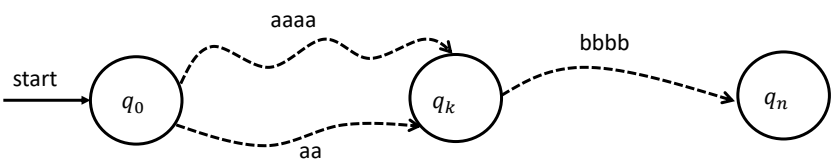
How about this one?

Build NFA



Distinguishability

- Let D be a DFA for $E = \{a^n b^n \mid n \in \mathbb{N}\}$
- Lemma: Any two strings of the form a^m and a^n, where m ≠ n, cannot end in the same state when run through D
- Proof: Suppose that a^m and a^n, where m ≠ n, end in the same state



  - Assume that from q_k input bbbb leads to a state q_n
  - If q_n is an accepting state, then aabbbb is accepted
  - If q_n is not an accepting state, then aaaabbbb is not accepted
  - Contradiction!

Observation

- Strings of the form a^{m} and a^{n}, where m ≠ n, transition to different states
  - ➔ Infinitely many states
  - ➔ Nonregular

Distinguishability - Formal

- Let *L* be an arbitrary language over Σ
- Two strings x ∈ Σ* and y ∈ Σ* are called *distinguishable relative to L* if there is a string w ∈ Σ* such that exactly one of xw and yw is in *L*
- We denote this by writing x $\not\equiv_L$ y

Lemma

- Consider a language: E = $\{a^n b^n \mid n \in \mathbb{N}\}$
- Lemma: If m, n $\in \mathbb{N}$ and m ≠ n, then $a^m \not\equiv_L a^n$
- Proof:
  - Let a^m and a^n be strings where m ≠ n
  - Then $a^m b^m \in$ E and $a^n b^m \notin$ E.
  - Therefore, we see that $a^m \not\equiv_L a^n$, as required

Nonregular Language

- Theorem: The language *E* = $\{a^n b^n \mid n \in \mathbb{N}\}$ is not regular
- Proof:
  - Suppose for the sake of contradiction that *E* is regular
  - Let D be a DFA for *E*, and let k be the number of states in D
  - Consider the strings $a^0, a^1, a^2 \ldots, a^k$
  - This is a collection of k+1 strings and there are only k states in D
  - Therefore, by the pigeonhole principle, there must be two distinct strings a^m and a^n that end in the same state when run through D
  - The previous lemma shows that $a^m \not\equiv_L a^n$, where m ≠ n
  - That is a^m and a^n cannot end in the same state when run through D
  - But this is impossible, since we know that a^m and a^n do end in the same state when run through D.
  - We have reached a contradiction, so our assumption must have been wrong
  - Therefore, *E* is not regular

Limit of Finite Memory

- We've just hit the limit of finite memory computation
- To build a DFA for $E = \{a^n b^n \mid n \in \mathbb{N}\}$, we need to have different memory configurations (states) for all possible strings of the form a^n
- There's no way to do this with finitely many possible states!

Another Nonregular Language

- Consider the following language over the alphabet Σ = {a, b, $\stackrel{?}{=}$}:
- *EQ* = { w$\stackrel{?}{=}$w | w ∈ {a, b}*}
- *EQ* is the language of all strings that have the form, "*sub-string1* $\stackrel{?}{=}$ *sub-string2*", where *sub-string1* and *sub-string2* are the same
- Examples:

  ab$\stackrel{?}{=}$ ab ∈ *EQ*

  bbb $\stackrel{?}{=}$ bbb ∈ *EQ*

  $\stackrel{?}{=}$ ∈ *EQ*

  ab $\stackrel{?}{=}$ ba ∉ *EQ*

  bbb $\stackrel{?}{=}$ aaa ∉ *EQ*

  b$\stackrel{?}{=}$ ∉ *EQ*

Myhill-Nerode Theorem

Distinguishing Sets

- Definition:
  - Let L be a language over Σ
  - A ***distinguishing set*** for L is a set S ⊆ Σ* where the following is true: ∀x ∈ S. ∀y ∈ S. (x ≠ y → x $\not\equiv_L$ y)

- Example
  - A distinguishing set for *E* = {$a^n b^n$ | n ∈ ℕ }:
    ➔ S = {a^n | n ∈ ℕ }
  - A distinguishing set for EQ = { w$\stackrel{?}{=}$w | w ∈ {a, b}*}:
    ➔ S = {a, b}*

Myhill-Nerode Theorem

- Theorem: If *L* is a language and S is a distinguishing set for *L* that contains infinitely many strings, then *L* is not regular

- Proof:
  - Let *L* be an arbitrary language over Σ and let S be a distinguishing set for *L* that contains infinitely many strings
  - Suppose for the sake of contradiction that *L* is regular
  - ➔ There must be some DFA D for *L*
  - Let k be the number of states in D
  - Since there are infinitely many strings in S, we can choose k+1 distinct strings from S and consider what happens when we run D on all of those strings
  - Because there are only k states in D and we've chosen k+1 strings from S, by the pigeonhole principle we know that at least two strings from S must end in the same state in D
  - Choose any two such strings and call them x and y

Myhill-Nerode Theorem

- Proof continue
  - Because x ≠ y and S is a distinguishing set for *L*, we know that x $\not\equiv_L$ y
  - Our earlier theorem therefore tells us that when we run D on inputs x and y, they must end up in different states
  - But this is impossible – we chose x and y precisely because they end in the same state when run through D
  - We have reached a contradiction, so our assumption must have been wrong
  - Thus *L* is not a regular language

Use of Myhill-Nerode Theorem

- Theorem: The language *E* = {$a^n b^n$ | n ∈ ℕ }: is not regular.

- Proof:
  - Let S = {a^n | n ∈ ℕ }
  - To see that S is infinite, note that S contains one string for each natural number
  - To see that S is a distinguishing set for *E*, consider any strings a^m , a^n ∈ S where m ≠ n
  - Note that a^m b^m ∈ *E* and that a^n b^m ∉ E. Therefore, we see that $a^m \not\equiv_L a^n$, as required
  - Since S is infinite and is a distinguishing set for *E*, by the Myhill-Nerode theorem we see that *E* is not regular.

Finding Distinguishing Set

- The challenge in using the Myhill-Nerode theorem is **finding the right set of strings**

- **General intuition**:
  - Start by thinking about what information a computer "must" remember in order to answer correctly
  - Choose a group of strings that all require different information
  - Prove that you have infinitely many strings an that the group of strings is a distinguishing set

DFA

- Note that each state in a DFA represent some key piece of information the automaton has to remember
- If you only need to remember one of finitely many pieces of information, that gives you a DFA
- If you need to remember one of infinitely many pieces of information, you can use the Myhill-Nerode theorem to prove that the language has no DFA

DFA and Computers

- We've ended up where we are now by trying to answer the question “what problems can you solve with a computer?”
- We defined a computer to be DFA, which means that the problems we can solve are precisely the regular languages
- We've discovered several equivalent ways to think about regular languages (DFAs, NFAs, and regular expressions) and used that to reason about the regular languages
- We now have a powerful intuition for where we ended up: DFAs are finite-memory computers, and regular languages correspond to problems solvable with finite memory
- Putting all of this together, we have a much deeper sense for what finite memory computation looks like – and what it doesn't look like!

Think About

- What does computation look like with unbounded memory?
- What problems can you solve with unbounded-memory computers?
- What does it even mean to “solve” such a problem?
- And how do we know the answers to any of these questions?

Context-Free Grammars

- A context-free grammar (or CFG) is a new formalism for defining a class of languages
- Goal: Give a description of a language by recursively describing the structure of the strings in the language
- CFGs are best explained by example...

Noam Chomsky (1928) is an American linguist, philosopher, cognitive scientist, social critic, and political activist. Inventor of CFGs (Context Free Grammars). Called "the father of modern linguistics", Chomsky is also a major figure in analytic philosophy and one of the founders of the field of cognitive science.

Arithmetic Expressions

- Suppose we want to describe all legal arithmetic expressions using addition, subtraction, multiplication, and division
- Here is one possible CFG:

E → int
E → E Op E
E → (E)
Op → +
Op → -
Op → *
Op → /

E
➔ E Op E
➔ E Op (E)
➔ E Op (E Op E)
➔ E * (E Op E)
➔ int * (E Op E)
➔ int * (int Op E)
➔ int * (int Op int)
➔ int * (int + int)

Context-Free Grammars

- Formally, a context-free grammar is a collection of four items:
  - A set of **nonterminal symbols** (also called **variables**)
  - A set of **terminal symbols** (the **alphabet** of the CFG)
  - A set of **production rules** saying how each nonterminal can be replaced by a string of terminals and nonterminals
  - A **start symbol** (which must be a nonterminal) that begins the derivation.

CFG Notation

- Nonterminals are represented by capital letters in **Bold Red Uppercase**
  - e.g. **A, B, C, D**
- Terminals are represented by lowercase letters in blue monospace
  - e.g. **t, u, v, w**
- Arbitrary strings of terminals and nonterminals are represented by lowercase Greek letters in *green italics*
  - e.g. ***α, γ, ω***

Shorthand Notation

E → int
E → E Op E
E → (E)
Op → +
Op → -
Op → *
Op → /

E → int | E Op E | (E)
Op → + | - | * | /

Derivations

- A sequence of steps where nonterminals are replaced by the right-hand side of a production is called a ***derivation***
- If string ***α*** derives string ***ω***, we write ***α*** ⇒* ***ω***
- In the example on the left, we see **E** ⇒* **int * (int + int)**

E
⇒* E Op E
⇒* E Op (E)
⇒* E Op (E Op E)
⇒* E * (E Op E)
⇒* int * (E Op E)
⇒* int * (int Op E)
⇒* int * (int Op int)
⇒* int * (int + int)

The Language of a Grammar

AI

- If G is a CFG with alphabet Σ and start symbol S, then *the language of G* is the set

$$\mathcal{L}(G) = \{ \omega \in \Sigma^* \mid S \Rightarrow^* \omega \}$$

- That is, $\mathcal{L}$ (G) is the set of strings of terminals derivable from the start symbol

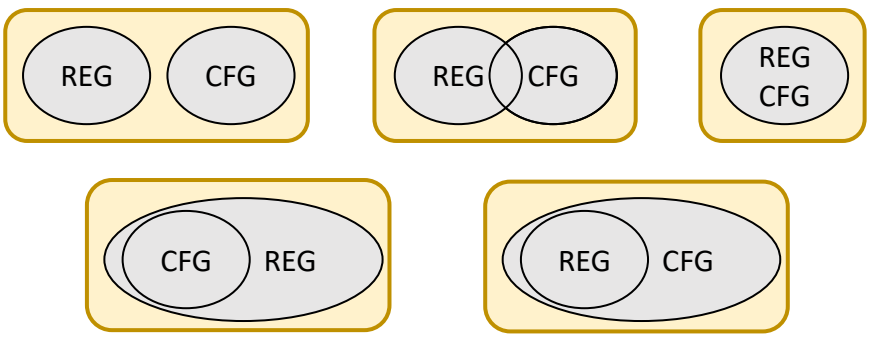
Consider the following CFG G over Σ = {a, b, c, d}:
S → Sa | dT
T → bTb | c

How many of the following strings are in $\mathcal{L}$ (G)?
dc
dca
cad
bcb
dTaa

CFG and RL

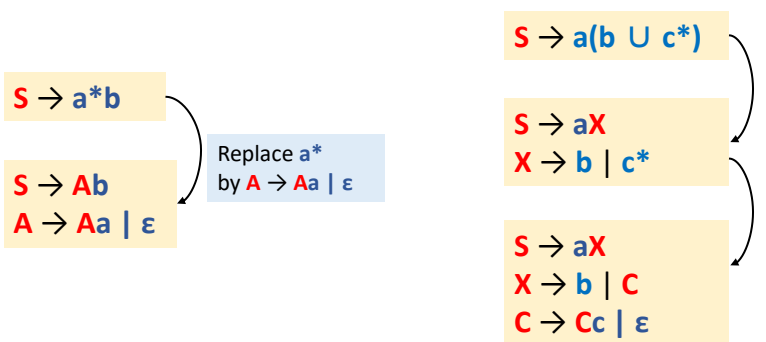
Context-Free Languages

- A language L is called a **Context-Free Language (CFL)** if there is a CFG G such that $L = \mathcal{L}(G)$
- Questions:
  - What languages are context-free?
  - How are context-free and regular languages related?



From Regexes to CFGs

- CFGs consist purely of production rules of the form $A \rightarrow \omega$
  - They do not have the regular expression operators * or ∪
- However, we can convert regular expressions to CFGs as follows:



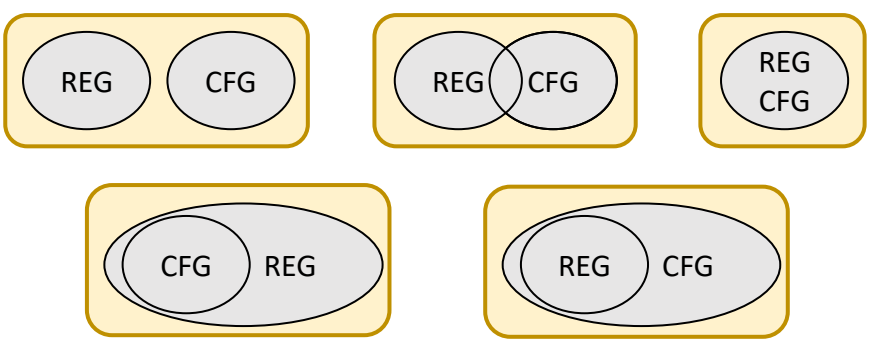
Regular Languages and CFLs

- Theorem: Every regular language (RL) is context-free language (CFL)
- Proof Idea: Use the construction techniques introduced before to convert a regular expression for *L* into a CFG for *L*
- You may convert a DFA/NFA (for a language *L*) into a CFG

  How?

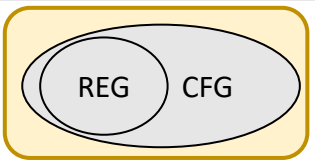
REG & CFG

- Which are possible now?



The Language of a Grammar

- Guess



- Show (at least) one case that CFG ∉ REG
- Consider the following CFG G:

  S → aSb | ε

- What strings does this generate?

CFG-Example

- Strings derived from a CFG

  S → aSb | ε

  a S b

  aa S bb

  aaa S bbb

  …

  $E = \{a^n b^n \mid n \in \mathbb{N}\}$

Why CFL ⊃ REG

- Why do CFGs have more power than regular expressions?
- Intuition: Derivations of strings have unbounded "memory."

  S → aSb | ε

Designing CFGs

- Like designing DFAs, NFAs, and regular expressions, designing CFGs is a craft

- When thinking about CFGs:
  - **Think recursively**: Build up bigger structures from smaller ones
  - **Have a construction plan**: Know in what order you will build up the string
  - **Store information in nonterminals**: Have each nonterminal correspond to some useful piece of information

Designing CFGs

- Let Σ = {a, b} and let L = {w ∈ Σ* | w is a palindrome }

- We can design a CFG for L by thinking inductively:
  - Base case: ε, a, and b are palindromes
  - If ω is a palindrome, then aωa and bωb are palindromes
  - No other strings are palindromes

S → ε | a | b | aSa | bSb

Designing CFGs

- Let Σ = {(,)} and let L = {w ∈ Σ* | w is a string of balanced parentheses }
  - Some sample strings in L:

    (( )) ( )

    (( ) ( )) (( ) ( ))

    ((((( ))) (( ))))

    ε

    ( ) ( )
- Let's think about this recursively
  - Base case: the empty string is a string of balanced parentheses
  - Recursive step: Look at the closing parenthesis that matches the first open parenthesis

    Removing the first parenthesis and the matching parenthesis forms two new strings of balanced parentheses

S → (S)S | ε

Designing CFGs

- Let Σ = {a, b} and let L = {w ∈ Σ* | w has the same number of a's and b's }
- How many of the following CFGs have language L?

① **S → aSb | bSa | ε**

② **S → abS | baS | ε**

③ **S → abSba | baSab | ε**

④ **S → SbaS | SabS | ε**

Designing CFGs: A Caveat

AI

- When designing a CFG for a language, make sure that it
  - Generates all the strings in the language ***(Most important)***
  - Never generates a string outside the language
  - Recursion actually terminates!

- Is the following grammar a CFG for the language { $a^n b^n$ | n ∈ ℕ }?

  S → aSb

- What strings in {a, b}* can you derive?
  - Answer: None!
- What is the language of the grammar?
  - Answer: Ø

  Infinite recursion !

Designing CFGs

AI

- When designing CFGs, remember that each nonterminal can be expanded out independently of the others
- Let Σ = {**a**, $\stackrel{?}{=}$} and let *L* = {$\boldsymbol{a^n \stackrel{?}{=} a^n}$ | n ∈ ℕ }
- Is the following a CFG for L?

S → X$\stackrel{?}{=}$X
X → aX | ε

S
⇒ X$\stackrel{?}{=}$X
⇒ aX$\stackrel{?}{=}$X
⇒ aaX$\stackrel{?}{=}$X
⇒ aa$\stackrel{?}{=}$X
⇒ aa$\stackrel{?}{=}$aX
⇒ aa$\stackrel{?}{=}$a

Finding a Build Order

- Let Σ = {**a**, $\stackrel{?}{=}$} and let $L = \{a^n \stackrel{?}{=} a^n \mid n \in \mathbb{N}\}$
- To build a CFG for L, we need to be more clever with how we construct the string
- If we build the strings of a's independently of one another, then we can't enforce that they have the same length
  - Idea: Build both strings of a's at the same time
  - One possible grammar based on that idea:

S → $\stackrel{?}{=}$ | aSa

S
⇒ aSa
⇒ aaSaa
⇒ aaaSaaa
⇒ aaa $\stackrel{?}{=}$ aaa

Applications

Function Prototypes

- Let Σ = {void, int, double, name, (,), ,, ;}
- Let's write a CFG for C-style function prototypes!
- Examples:
  - **void name(int name, double name);**
  - **int name();**
  - **int name(double name);**
  - **int name(int, int name, int);**
  - **void name(void);**

Function Prototypes

- Here's one possible grammar:

S → **Ret name (Args);**

Ret → **Type** | **void**

Type → **int** | **double**

Args → **ε** | **void** | **ArgList**

ArgList → **OneArg** | **ArgList**, **OneArg**

OneArg → **Type** | **Type name**

- What changes would you need to make to support pointer types?

Applications

- Programming Languages

```
BLOCK → STMT | { STMTS }
STMTS → ε | STMT STMTS
STMT → EXPR;
        | if (EXPR) BLOCK
        | while (EXPR) BLOCK
        | do BLOCK while (EXPR);
        | BLOCK
        | …
EXPR → identifier
        | constant
        | EXPR + EXPR
        | EXPR – EXPR
        | EXPR * EXPR
        | …
```


Grammars in Compilers

- One of the key steps in a compiler is figuring out what a program "means"
- This is usually done by defining a grammar showing the high-level structure of a programming language
- There are certain classes of grammars (LL(1) grammars, LR(1) grammars, LALR(1) grammars, etc.) for which it's easy to figure out how a particular string was derived
- Tools like yacc or bison automatically generate parsers from these grammars

Natural Language Processing (NLP)

AI

- By building context-free grammars for actual languages and applying statistical inference, it's possible for a computer to recover the likely meaning of a sentence
- CFGs were first called ***phrase-structure grammars*** and were introduced by Noam Chomsky in his seminal work Syntactic Structures
- CFGs were then adapted for use in the context of programming languages, where they were called ***Backus Naur forms***