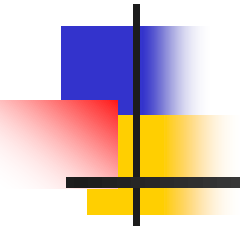
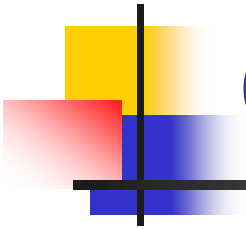


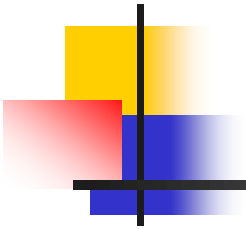
Final Course Review

Reading: Chapters 1-9



Objectives

- Introduce concepts in automata theory and theory of computation
- Identify different formal language classes and their relationships
- Design grammars and recognizers for different formal languages
- Prove or disprove theorems in automata theory using its properties
- Determine the decidability and intractability of computational problems



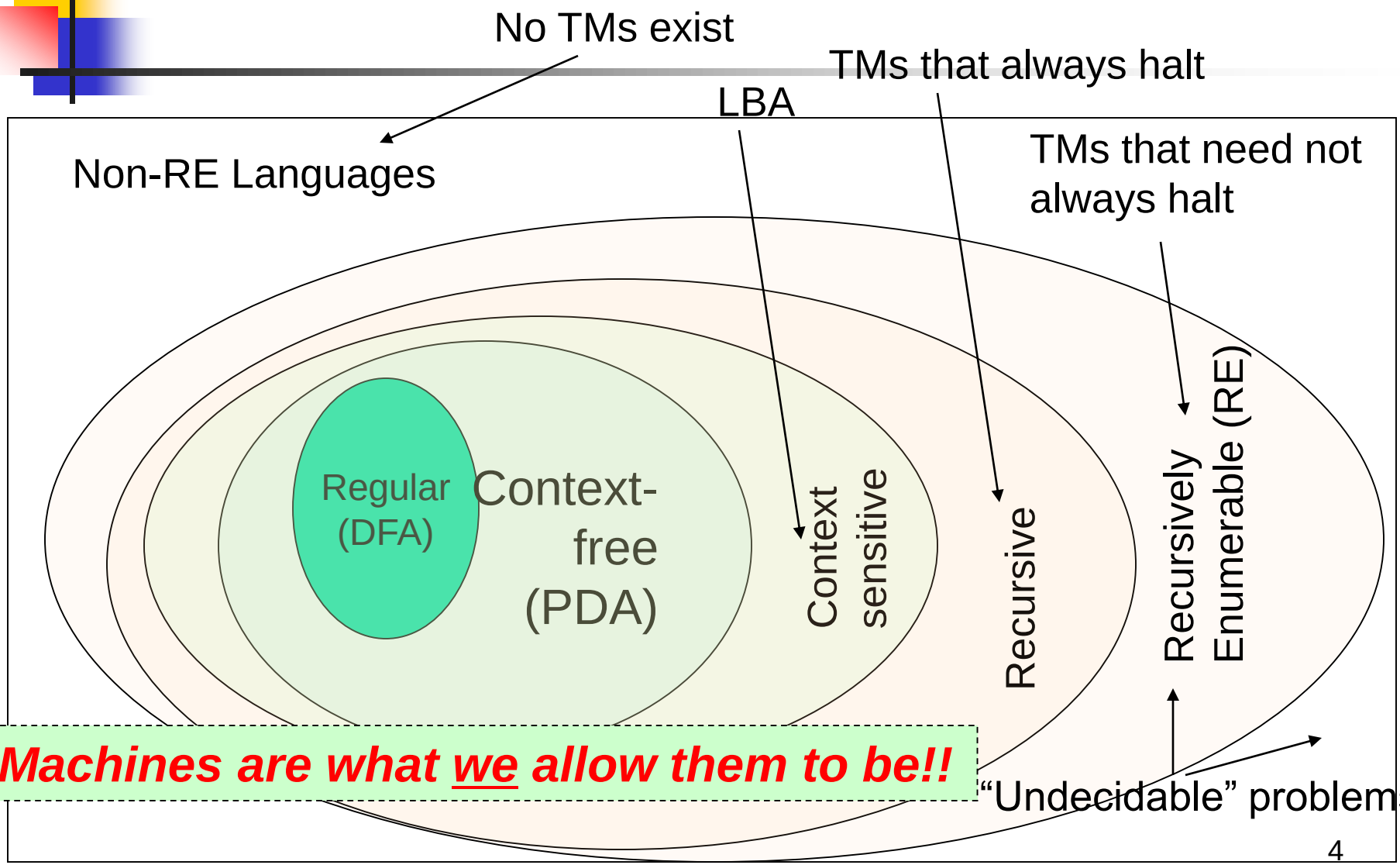
Main Topics

Part 1) Regular Languages

Part 2) Context-Free Languages

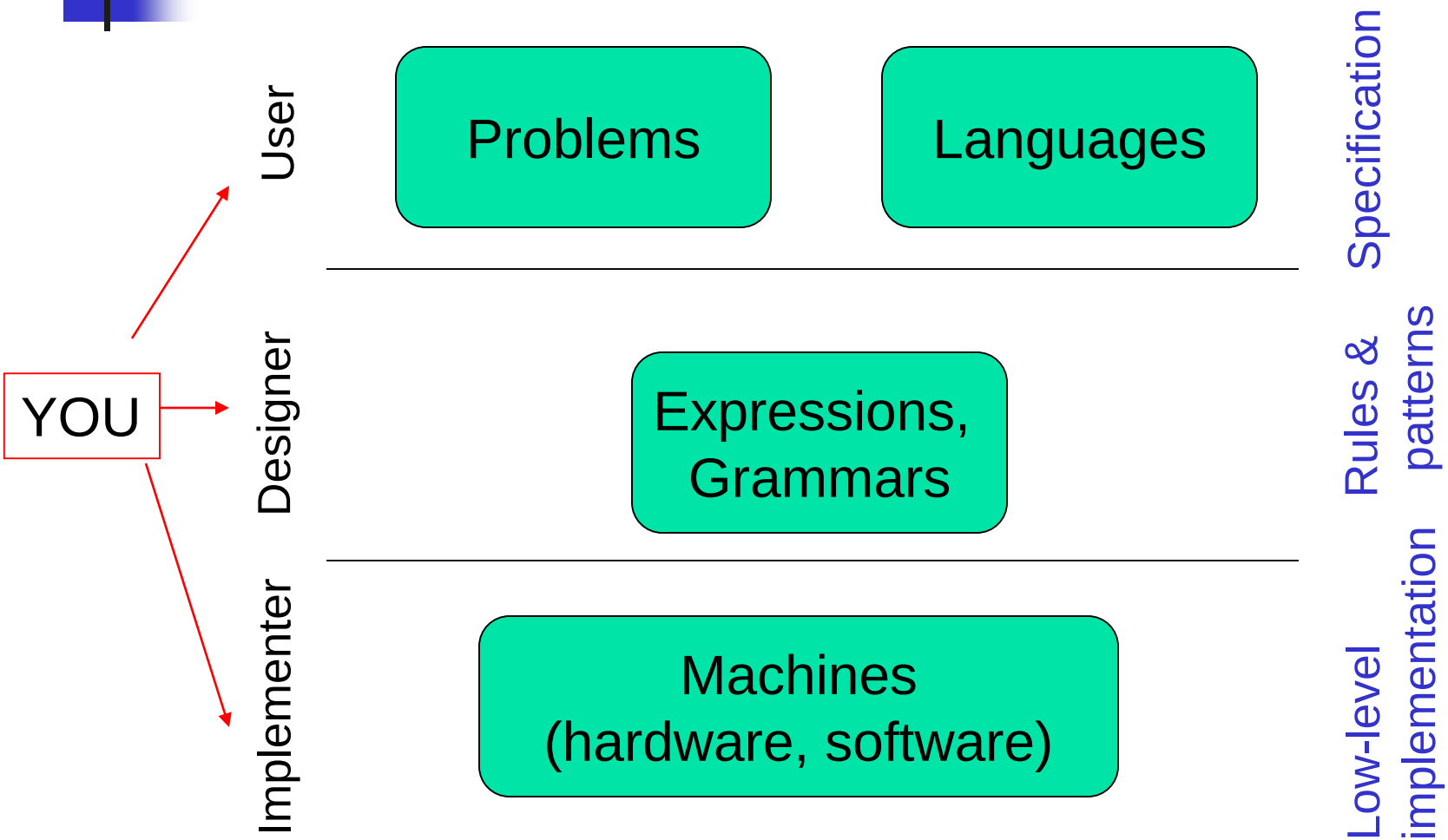
Part 3) Turing Machines & Computability

The Chomsky hierarchy for formal languages

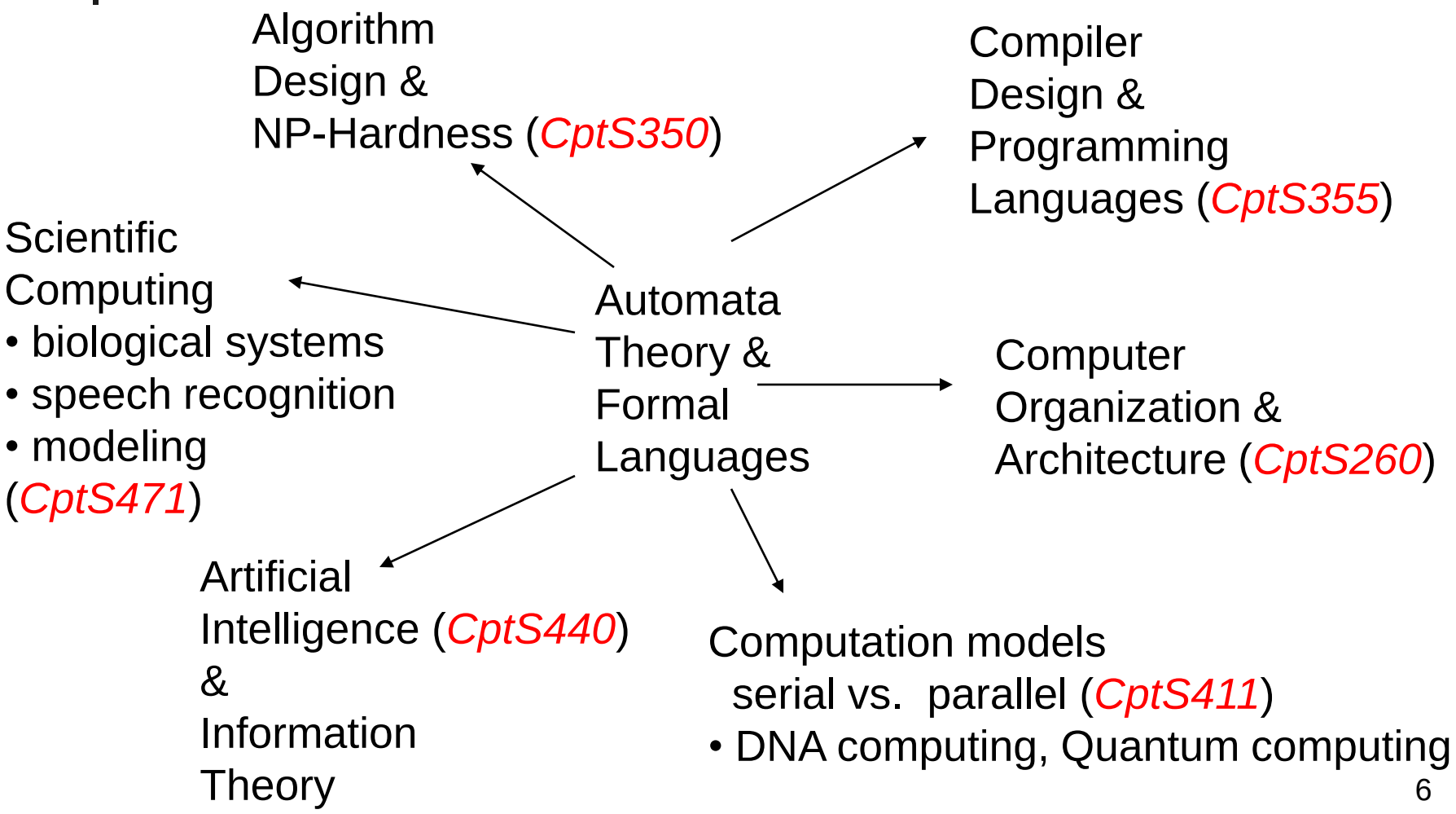


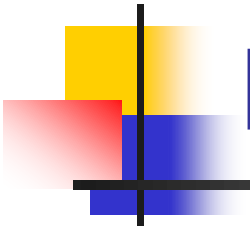
Machines are what we allow them to be!!

Interplay between different computing components



Automata Theory & Modern-day Applications



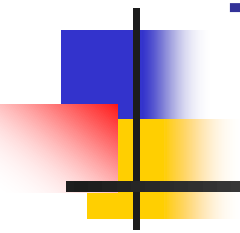


Final Exam

- **May 3, 2017, Wednesday, 8:00am – 10:00am**
- In class
- Comprehensive:
  - Everything covered in class until (& including) the closure properties for Recursive and Recursively Enumerable language classes.

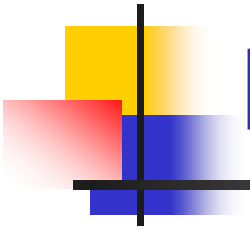
Thank You &

Course evaluations:
(fill out from my.wsu)



Topic Reviews

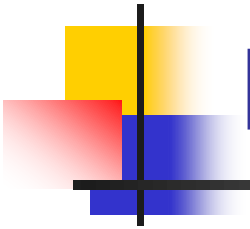
The following set of review slides are *not* meant to be comprehensive. So make sure you refer to them in conjunction with the midterm review slides, homeworks and most importantly, the original lecture slides!



Regular Languages Topics

- Simplest of all language classes

- Finite Automata
  - NFA, DFA, ε-NFA
- Regular expressions
- Regular languages & properties
  - Closure
  - Minimization



Finite Automata

- Deterministic Finite Automata (DFA)
  - The machine can exist in only one state at any given time
- Non-deterministic Finite Automata (NFA)
  - The machine can exist in multiple states at the same time

- ε-NFA is an NFA that allows ε-transitions

- What are their differences?
- Conversion methods

Deterministic Finite Automata

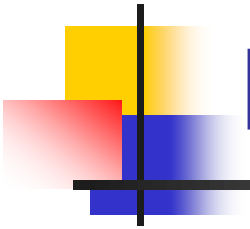
- A DFA is defined by the 5-tuple:
  - $\{Q, \sum, q_0, F, \delta\}$
- Two ways to represent:
  - State-diagram
  - State-transition table

- DFA construction checklist:
  - States & their meanings
  - Capture all possible combinations/input scenarios
    - break into cases & subcases wherever possible)
  - Are outgoing transitions defined for every symbol from every state?
  - Are final/accepting states marked?
  - Possibly, dead-states will have to be included

Non-deterministic Finite Automata

- A NFA is defined by the 5-tuple:
  - $\{Q, \sum , q_0, F, \delta \}$
- Two ways to represent:
  - State-diagram
  - State-transition table

- NFA construction checklist:
  - Introduce states only as needed
  - Capture only valid combinations
    - Ignore invalid input symbol transitions (allow that path to die)
  - Outgoing transitions defined only for valid symbols from every state
  - Are final/accepting states marked?

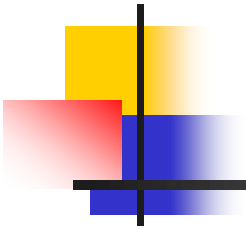


NFA to DFA conversion

- Checklist for NFA to DFA conversion
  - Two approaches:
    - Enumerate all possible subsets, or
    - Use *lazy construction* strategy (to save time)
      - Introduce subset states only as needed
  - Any subset containing an accepting state is also accepting in the DFA
  - Have you made a special entry for Φ, the empty subset?
    - This will correspond to dead state

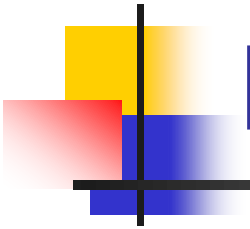
ε-NFA to DFA conversion

- Checklist for €-NFA to DFA conversion
  - First take ECLOSE(start state)
  - New start state = ECLOSE(start state)
  - Remember: ECLOSE(q) include q
  - Same two approaches as NFA to DFA:
    - Enumerate all possible subsets, or
    - Use *lazy construction* strategy (to save time)
      - Introduce subset states only as needed
  - Only difference: take ECLOSE *both before & after* transitions
  - The subset Φ corresponds to a “dead state”



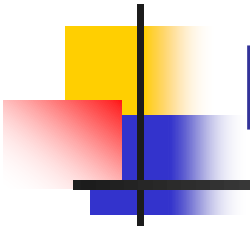
Regular Expressions

- A way to express accepting patterns
- Operators for Reg. Exp.
  - (E), L(E+F), L(EF), L(E^*)..
- Reg. Language ➔ Reg. Exp. (checklist):
  - Capture all cases of valid input strings
  - Express each case by a reg. exp.
  - Combine all of them using the + operator
  - Pay attention to operator precedence



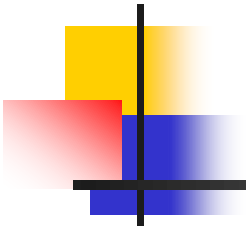
Regular Expressions…

- DFA to Regular expression
  - Enumerate all paths from start to every final state
  - Generate regular expression for each segment, and concatenate
  - Combine the reg. exp. for all each path using the + operator
- Reg. Expression to ε-NFA conversion
  - Inside-to-outside construction
  - Start making states for every atomic unit of RE
  - Combine using: concatenation, + and * operators as appropriate
  - For connecting adjacent parts, use ε-jumps
  - Remember to note down final states



Regular Expressions…

- Algebraic laws
  - Commutative
  - Associative
  - Distributive
  - Identity
  - Annihiliator
  - Idempotent
  - Involving Kleene closures (* operator)



English description of lang.

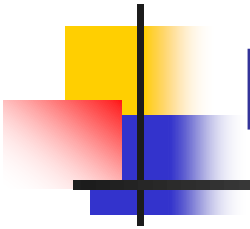
- For finite automata
- For Regular expressions
- When asked for “English language descriptions”:
  - Always give the description of *the underlying language that is accepted by that machine or expression*

    (and not of the machine or expression)

Pumping Lemma

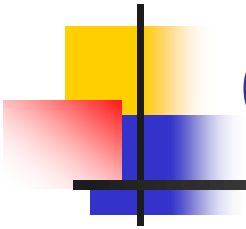
- Purpose: Regular or not? Verification technique

- Steps/Checklist for Pumping Lemma:
  - Let n ← pumping lemma constant
  - Then construct input w which has n or more characters
  - Now w=xyz should satisfy P/L
  - Check all three conditions
  - Then use one of these 2 strategies to arrive at contradiction for some other string constructed from w:
    - Pump up (k >= 2)
    - Pump down (k=0)



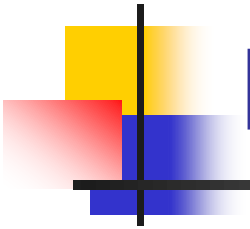
Reg. Lang. Properties

- Closed under:
  - Union
  - Intersection
  - Complementation
  - Set difference
  - Reversal
  - Homomorphism & inverse homomorphism
- Look at all DFA/NFA constructions for the above



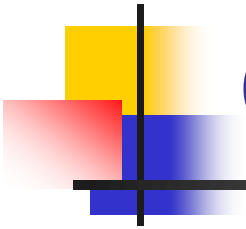
Other Reg. Lang. Properties

- Membership question
- Emptiness test
  - Reachability test
- Finiteness test
  - Remove states that are:
    - Unreachable, or cannot lead to accepting
  - Check for cycle in left-over graph
  - Or the reg. expression approach



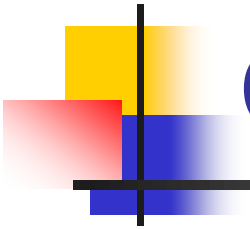
DFA minimization

- Steps:
  - Remove unreachable states first
  - Detect equivalent states
- Table-filing algorithm (checklist):
  - First, mark X for accept vs. non-accepting
  - Pass 1:
    - Then mark X where you can distinguish by just using one symbol transition
    - Also mark = whenever states are equivalent.
  - Pass 2:
    - Distinguish using already distinguished states (one symbol)
  - Pass 3:
    - Repeat for 2 symbols (on the state pairs left undistinguished)
    - …
  - Terminate when all entries have been filled
  - Finally modify the state diagram by keeping one representative state for every equivalent class



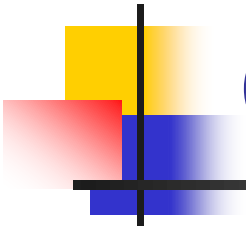
Other properties

- Are 2 DFAs equivalent?
  - Application of table filling algo



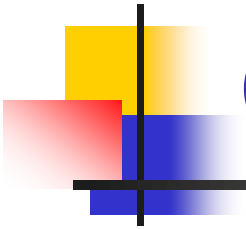
CFL Topics

- CFGs
- PDAs
- CFLs & pumping lemma
- CFG simplification & normal forms
- CFL properties



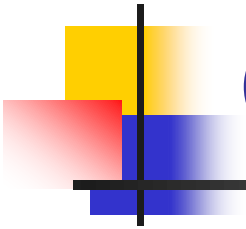
CFGs

- G=(V,T,P,S)
- Derivation, recursive inference, parse trees
  - Their equivalence
- Leftmost & rightmost derivation
  - Their equivalence
  - Generate from parse tree
- Regular languages vs. CFLs
  - Right-linear grammars



CFGs

- Designing CFGs
  - Techniques that can help:
    - Making your own start symbol for combining grammars
      - Eg., S => S_1 | S_2 (or) S => S_1 S_2
    - Matching symbols: (e.g., S => a S a | …)
    - Replicating structures side by side: (e.g., S => a S b S)
    - Use variables for specific purposes (e.g., specific sub-cases)
    - To go to an acceptance from a variable
      - ==> end the recursive substitution by making it generate terminals directly
      - A => w
    - Conversely, to *not* go to acceptance from a variable, have productions that lead to other variables
- Proof of correctness
  - Use induction on the string length



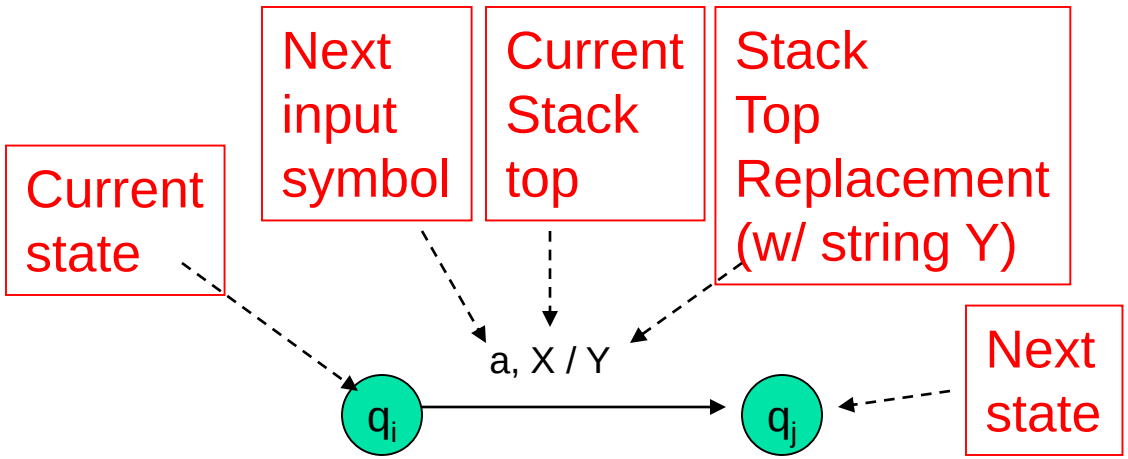
CFGs…

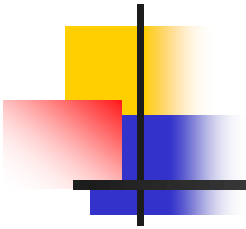
- Ambiguity of CFGs
  - One string <==> more than one parse tree
  - Finding one example is sufficient
- Converting ambiguous CFGs to non-ambiguous CFGs
  - Not always possible
  - If possible, uses ambiguity resolving techniques (e.g., precedence)
- Ambiguity of CFL
  - It is not possible to build even a single unambiguous CFG

There can be only 1 stack top symbol
There can be many symbols for the replacement

PDAs

- PDA ==> ε-NFA + “a stack”
- P = (Q,$\sum$,Γ, δ,q_0,Z_0,F)
- $\delta(q,a,X) = \{(p,Y), \ldots\}$
- ID : (q, aw, XB) |--- (p,w,AB)
- State diagram way to show the design of PDAs





Designing PDAs

- Techniques that can help:
  - Two types of PDAs
    - Acceptance by empty stack
      - If no more input and stack becomes empty
    - Acceptance by final state
      - If no more input and end in final state
  - Convert one form to another
  - Assign state for specific purposes
  - Pushing & popping stack symbols for matching
  - Convert CFG to PDA
  - Introducing new stack symbols may help
  - Take advantage of non-determinism

CFG Simplification

1. Eliminate ε-productions: A => ε
   - ==> substitute for A (with & without)
   - Find nullable symbols first and substitute next
2. Eliminate unit productions: A=> B
   - ==> substitute for B directly in A
   - Find unit pairs and then go production by production
3. Eliminate useless symbols
   - Retain only reachable and generating symbols

- Order is important : steps (1) => (2) => (3)

Chomsky Normal Form

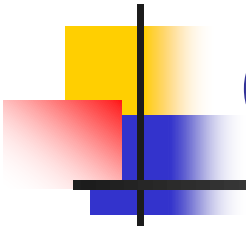
- All productions of the form:
  - A => BC or A=> a
- Grammar does not contain:
  - Useless symbols, unit and €-productions
- Converting CFG (without S=>* ε) into CNF
  - Introduce new variables that collectively represent a sequence of other variables & terminals
  - New variables for each terminal
- CNF ==> Parse tree size
  - If the length of the longest path in the parse tree is *n*, then $|w| \leq 2^{n-1}$.

Pumping Lemma for CFLs

- Then there exists a constant N, s.t.,
  - if z is any string in L s.t. $|z| \geq N$, then we can write $z=uvwxy$, subject to the following <u>conditions:</u>
    1. $|vwx| \leq N$
    2. $vx \neq \varepsilon$
    3. For all $k \geq 0$, uv^kwx^ky is in L

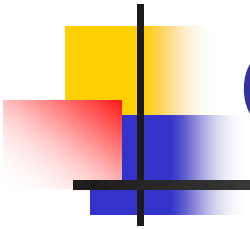
Using Pumping Lemmas for CFLs

- Steps:
  1. Let N be the P/L constant
  2. Pick a word z in the language s.t. $|z| \geq N$
     - (choice critical - an arbitrary choice may not work)
  3. z=uvwxy
  4. First, argue that because of conditions (1) & (2), the portions covered by vwx on the main string z will have to satisfy some properties
  5. Next, argue that by pumping up or down you will get a new string from z that is <u>not</u> in L



Closure Properties for CFL

- CFLs are closed under:
  - Union
  - Concatenation
  - Kleene closure operator
  - Substitution
  - Homomorphism, inverse homomorphism
- CFLs are *not* closed under:
  - Intersection
  - Difference
  - Complementation

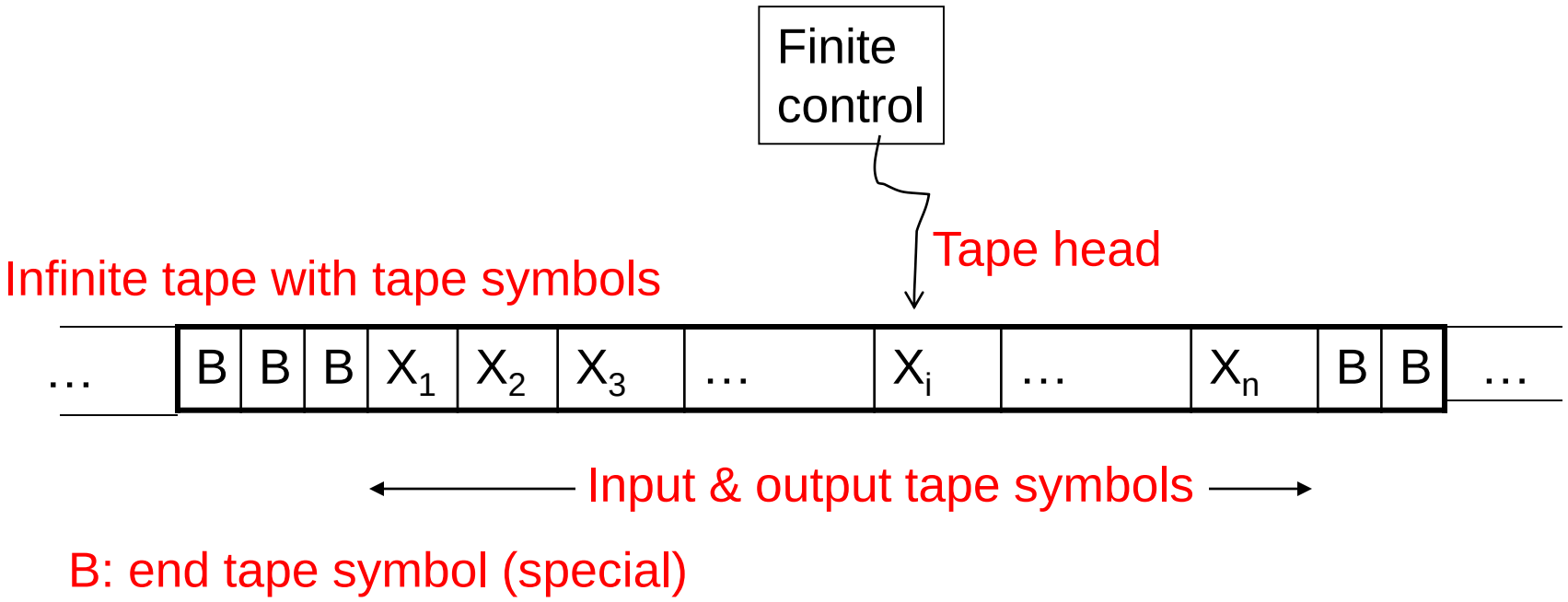


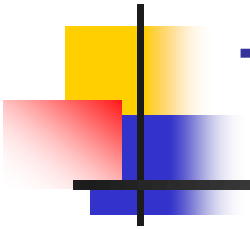
Closure Properties

- Watch out for
  - custom-defined operators
    - Eg.. Prefix(L), or “L x M”
  - Custom-defined symbols
    - Other than the standard 0,1,a,b,c..
    - E.g, #, c, ..

The Basic Turing Machine (TM)

- $M = (Q, \sum, \Gamma, \delta, q_0, B, F)$





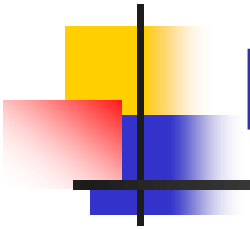
Turing Machines & Variations

- Basic TM
- TM w/ storage
- Multi-track TM
- Multi-tape TM
- Non-deterministic TM

Unless otherwise stated, it is OK to give TM design in the pseudocode format

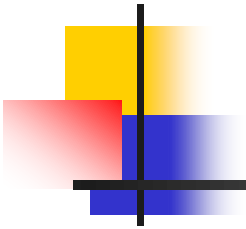
TM design

- Use any variant feature that may simplify your design
  - Storage - to remember last important symbol seen
  - A new track - to mark (without disturbing the input)
  - A new tape - to have flexibility in independent head motion in different directions
- Acceptance only by final state
- No need to show dead states
- Use ε-transitions if needed
- Invent your own tape symbols as needed



Recursive, RE, non-RE

- Recursive Language
  - TMs that always halt
- Recursively Enumerable
  - TMs that always halt only on acceptance
- Non-RE
  - No TMs exist that are guaranteed to halt even on accept
- Need to know the conceptual differences among the above language classes
  - Expect objective and/or true/false questions



Recursive Closure Properties

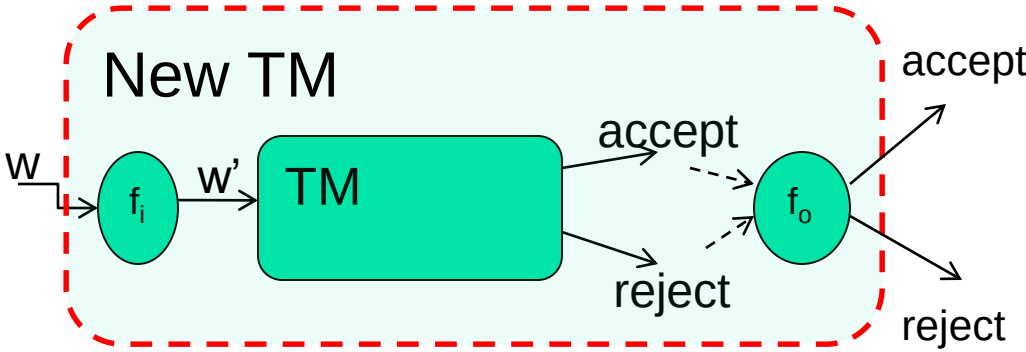
- Closed under:
  - Complementation, union, intersection, concatenation (discussed in class)
  - Kleene Closure, Homomorphism (not discussed in class but think of extending)

Tips to show closure properties on Recursive & RE languages

You need to define the input and output transformations (f_i and f_o)

Build a new machine that wraps around the TM for the input language(s)

- For Recursive languages:
  - The old TM is always going to halt (w/ accept or reject)
    => So will the new TM



- For Recursively Enumerable languages:
  - The old TM is guaranteed to halt only on acceptance
    => So will the new TM