CS3102 Theory of Computation

Context Free Languages

- For a PDA M, the language of M (denoted $L(M)$) refers to the set of strings accepted by the machine
  - $L(M) = \{s \in \Sigma^* | M \text{ accepts } s\}$
- The set of all languages decided by some PDA is call the **Context Free Languages**
  - Equivalent to the languages describable by Context Free Grammars
- A particular language decided by some FSA is called a **Context Free Language**
- All regular languages are context free (because if we choose not to use the stack, a PDA is a NFA)
- All context free languages can be decided by a Java program using only linear memory (relative to length of word)

Context Free Grammar

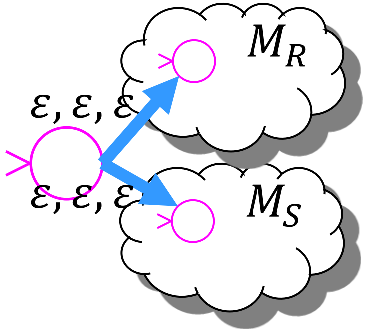
- Basic idea: Apply substitutions to construct strings
- Finite set of variables/non-terminals: $V = \{V_1, V_2, \dots, V_k\}$
- Finite set of terminals: $\Sigma \cup \{\varepsilon\}$
- Finite set of productions/subtitutions: $R: V \to (\Sigma \cup V \cup \{\varepsilon\})^*$
- Start symbol: S
- To produce a string:
  - Start with the start symbol,
  - As long as your string still contains non-terminals,
  - Subtitute a non-terminal using a production rule

Closure Properties of CFLs

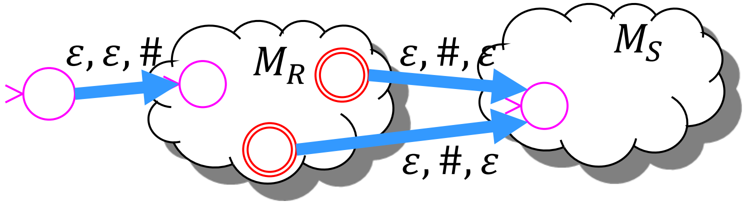
- Context Free Languages are closed under:
  - Union
  - Concatenation
  - Kleene Star
  - Intersection with Regular languages
- Context Free Languages are not closed under:
  - Complementation
  - Intersection with CFLs

CFLs closed under:

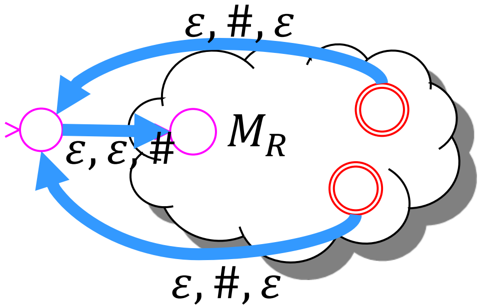
- Union



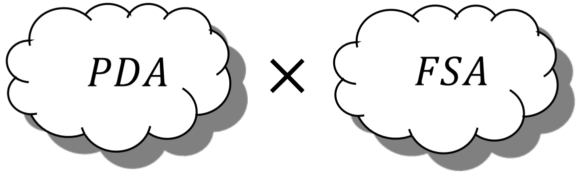
- Concatenation



- Kleene Star



- Intersection with regular languages



CFL Closure Redux

- We can also show closure of CFLs using CFGs
  - If we have CFGs for languages L_1, L_2 with start symbols S_1, S_2 respectively
- Union
  - Take production rules for both grammars, add $S \to S_1 | S_2$
- Concatenation
  - Take production rules for both grammars, add $S \to S_1 S_2$
- Kleene
  - Add to production rules of S_1, $S \to \varepsilon \mid SS \mid S_1$

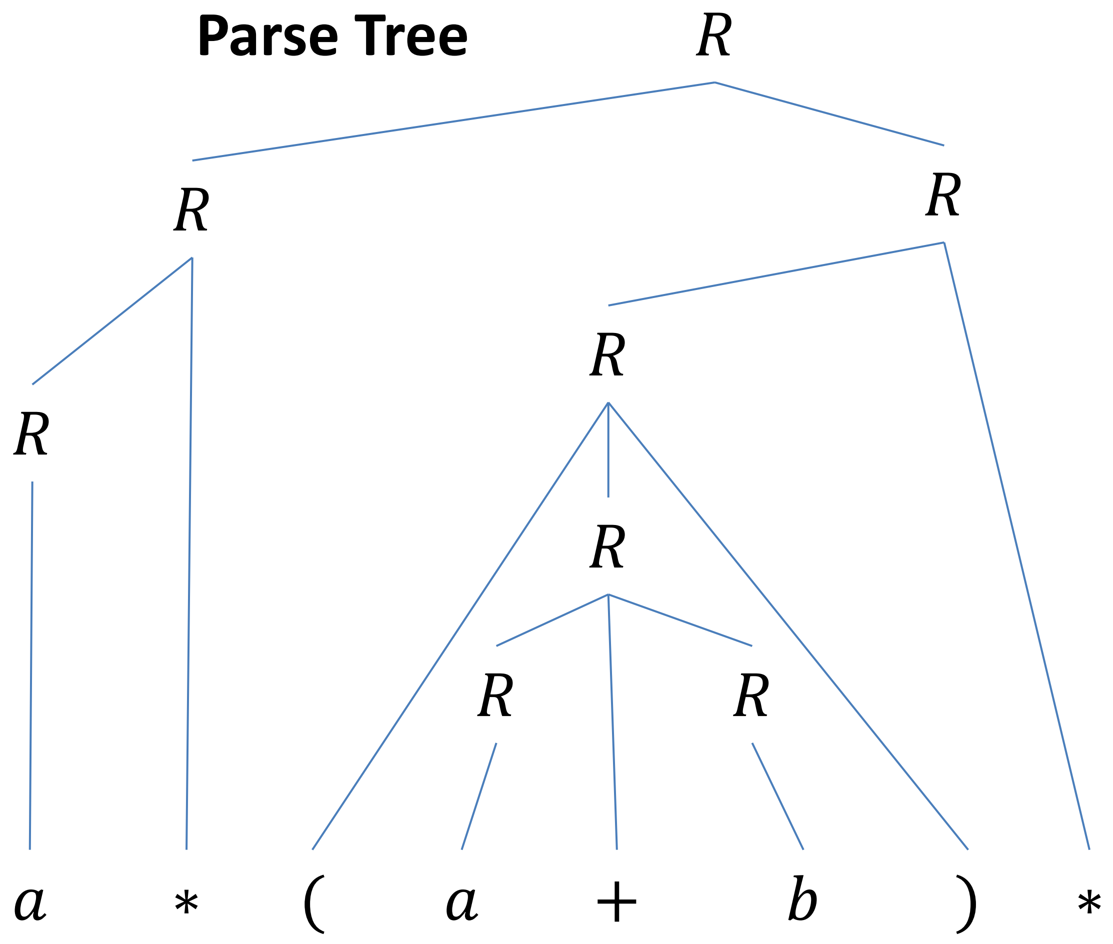
CFG for Regular Expressions

$$R \to a \mid b \mid \varepsilon \mid R + R \mid RR \mid R^* \mid (R)$$

To produce: $a^*(a+b)^*$

Current String	Production rule applied
R	$R \to RR$
RR	$R \to R^*$
R^*R	$R \to R^*$
R^*R^*	$R \to a$
a^*R^*	$R \to (R)$
$a^*(R)^*$	$R \to R + R$
$a^*(R+R)^*$	$R \to a$
$a^*(a+R)^*$	$R \to b$
$a^*(a+b)^*$	All terminals

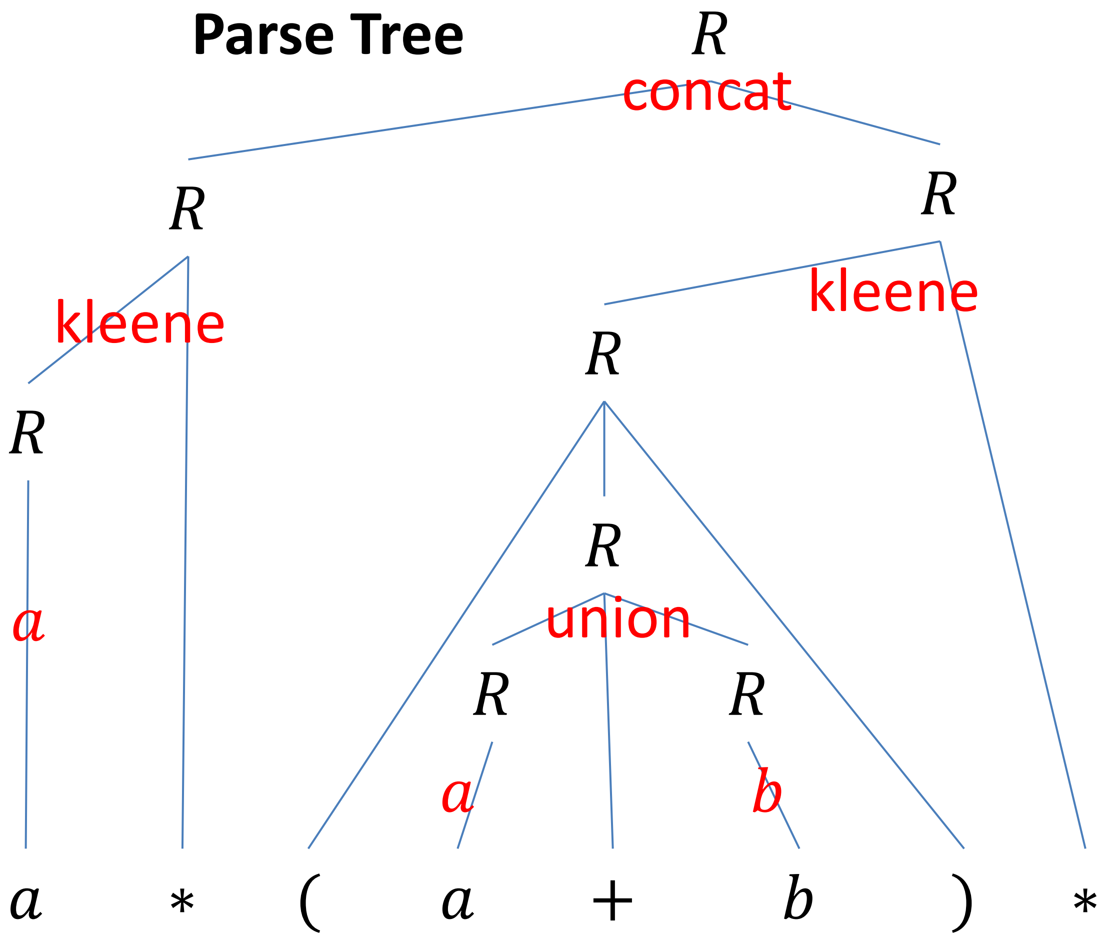
Parse Tree

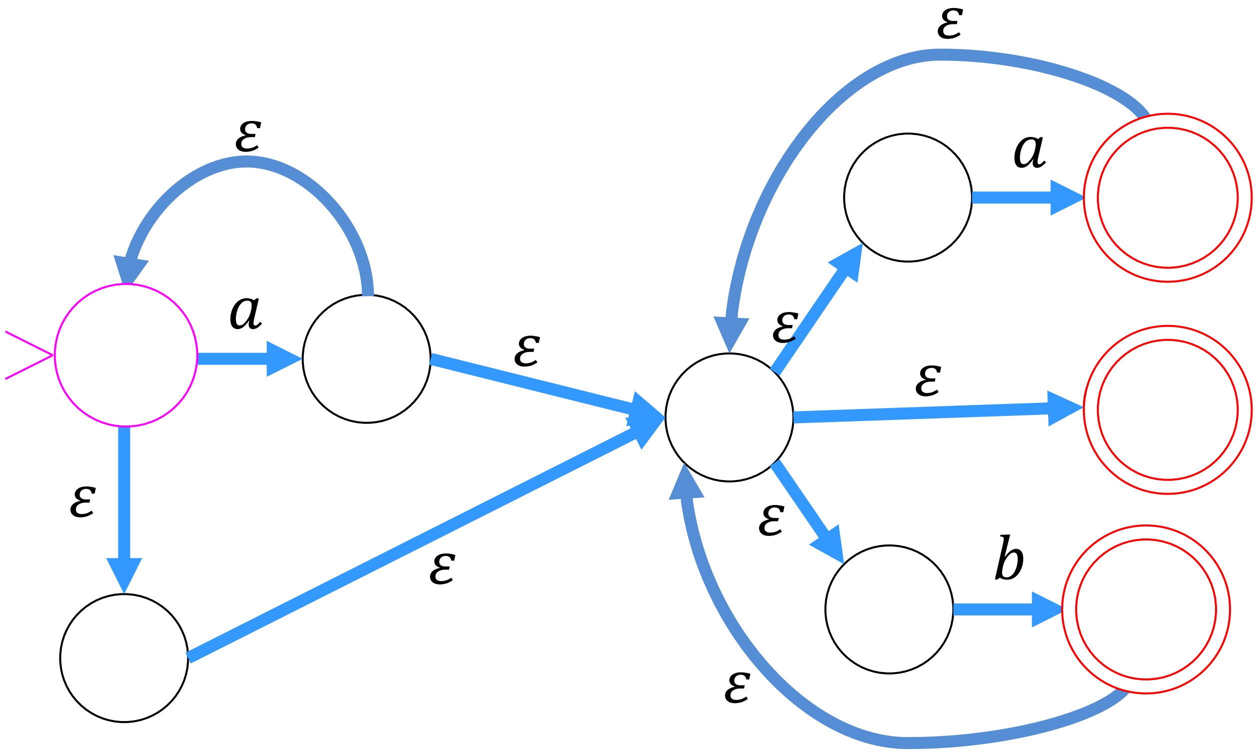


Usefulness of Parse Tree

- Typically with CFGs, each production rule has some “meaning” associated
- E.g. in Regex, each rule was an operation
- From the parse tree, we know which operations to apply to which languages in what order to get the describe language

Regex parse tree to NFA



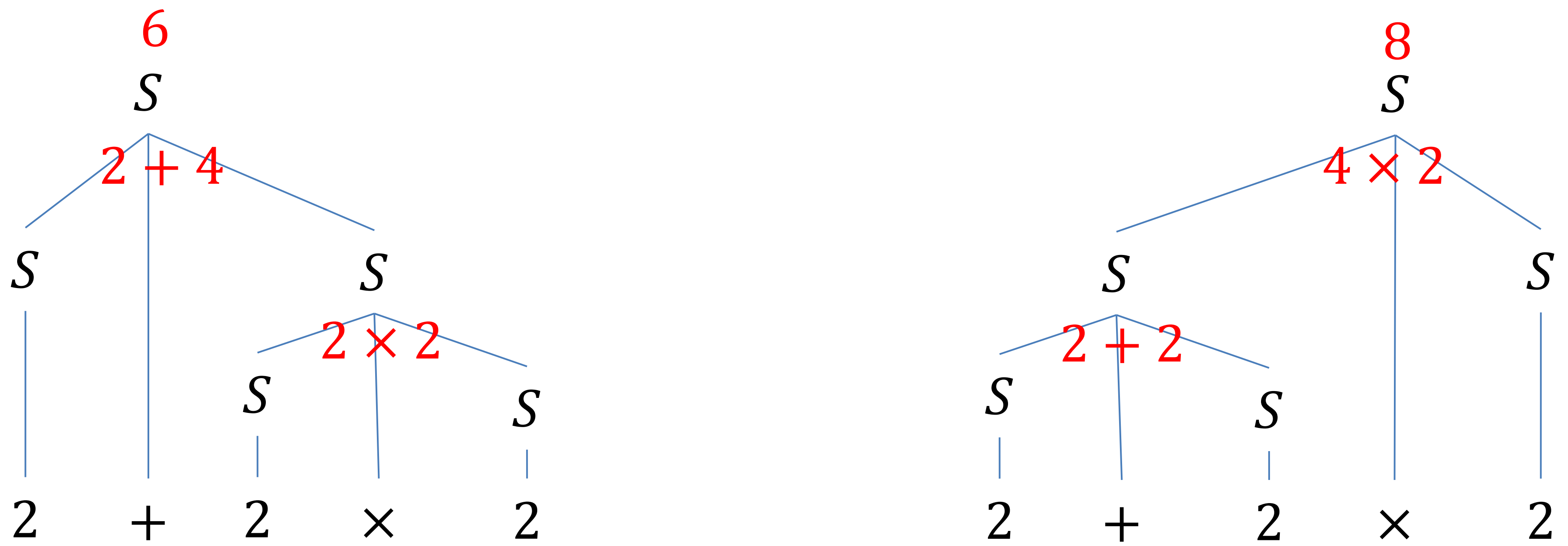


CFGs and Smug People on Social Media

- Consider this CFG for arithemetic expressions:

$$S \rightarrow S + S \mid S \times S \mid (S) \mid 2$$

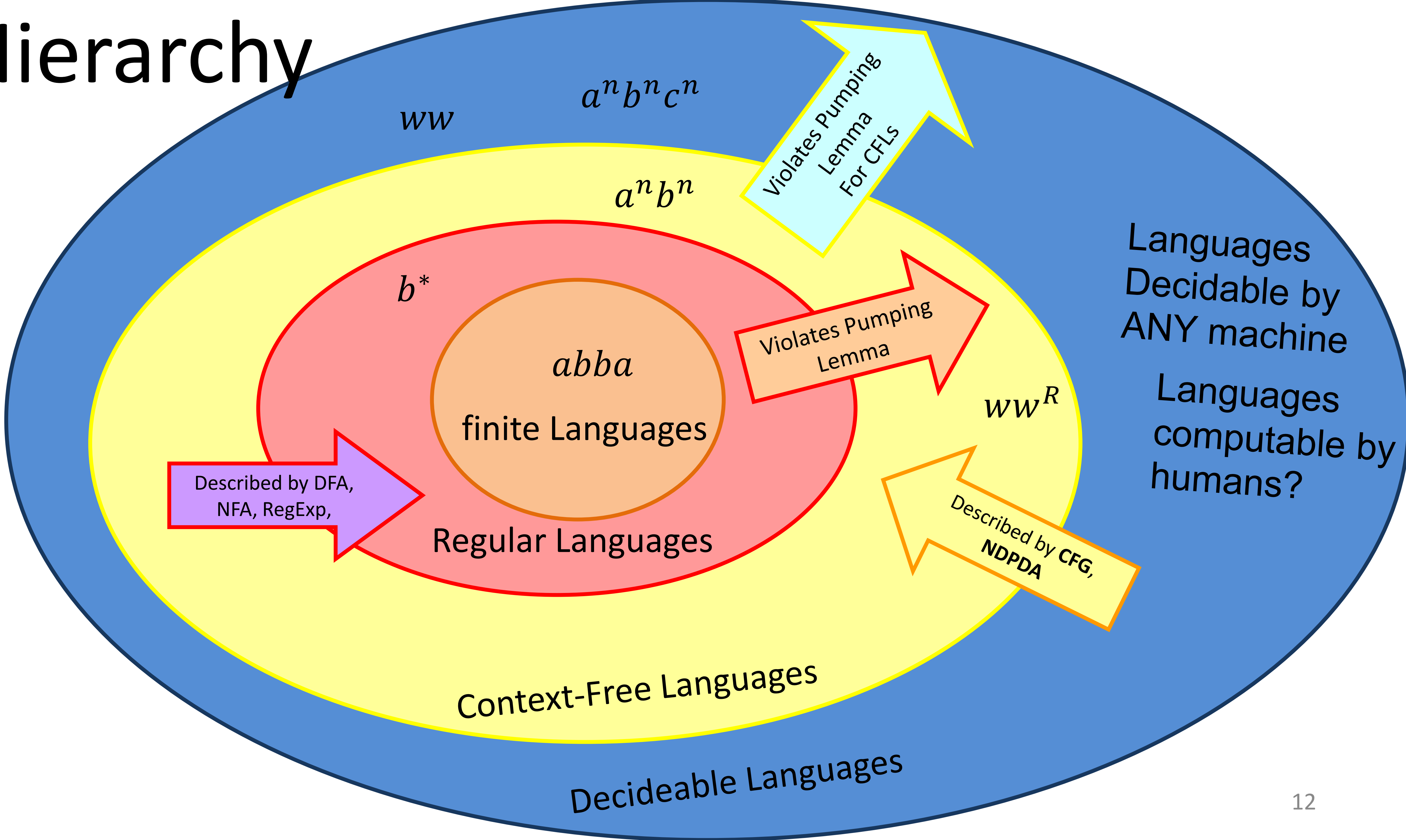
How could we get $2 + 2 \times 2$?



Ambiguous Grammars

- A CFG is ambiguous if there is at least 1 string that can be generated by at least 2 different parse trees
- Generally less desirable (harder to assign meaning to each production rule)
- Some languages have no non-ambiguous CFGs
- Determining whether a CFG is ambiguous is not computable

Hierarchy



ww
$a^n b^n c^n$
Violates Pumping Lemma For CFLs
$a^n b^n$
Languages Decidable by ANY machine
Languages computable by humans?
b^*
$abba$
finite Languages
Violates Pumping Lemma
ww^R
Described by DFA, NFA, RegExp,
Regular Languages
Described by **CFG, NDPDA**
Context-Free Languages
Decideable Languages

Historical Aside: David Hilbert

- Can we automate mathematics?
- Kurt Godel:
  - Every expressive axiomatic system is either inconsistent or incomplete
  - First Order logic is both consisten and complete (because it's not very expressive)
    - E.g. $\forall x \; \exists y \; \forall z \; p(x, y, z)$
- Hilbert, 1928:
  - Since every statement has a proof, let's find an "algorithm" for proving things!
  - Enscheidungsproblem (deciding problem)

Alonzo Church, Alan Turing

- What is an algorithm, anyway?
- Church:
  - Lambda Calculus
  - Algorithms consist of variables and functions
- Turing:
  - (Turing) Machine
  - Algorithms consist of memory that’s manipulated by a finite “controller”

On Computable Numbers, With an Application to the Entscheidungsproblem

- Alan Turing’s seminal work
- Contributions
  - Turing Machines
  - Proof that some “numbers” (equivalently functions) are expressible, but not computable by them
  - Proof that the Entscheidungsproblem cannot be solved by them
  - **Proof that they are of equivalent power to the lambda calculus**
  - **Demonstration that they are “universal”**
    - **There is one Turing Machine that can simulate all others**
  - **A philosophical argument that they are equivalent to humans**

Universal Machine

- We can have one Turing machine which can simulate any other Turing Machine
  – There is some machine M_u which given the description of another machine M' and an input w, it behaves the same way that M' would on w
- Why is this useful?
  – Turing machines are meant to be a model for all computers
  – Allows us to have one machine that we've built to simulate any other machine we describe

How do we describe a computer?

Lambda Calculus = Turing Machines

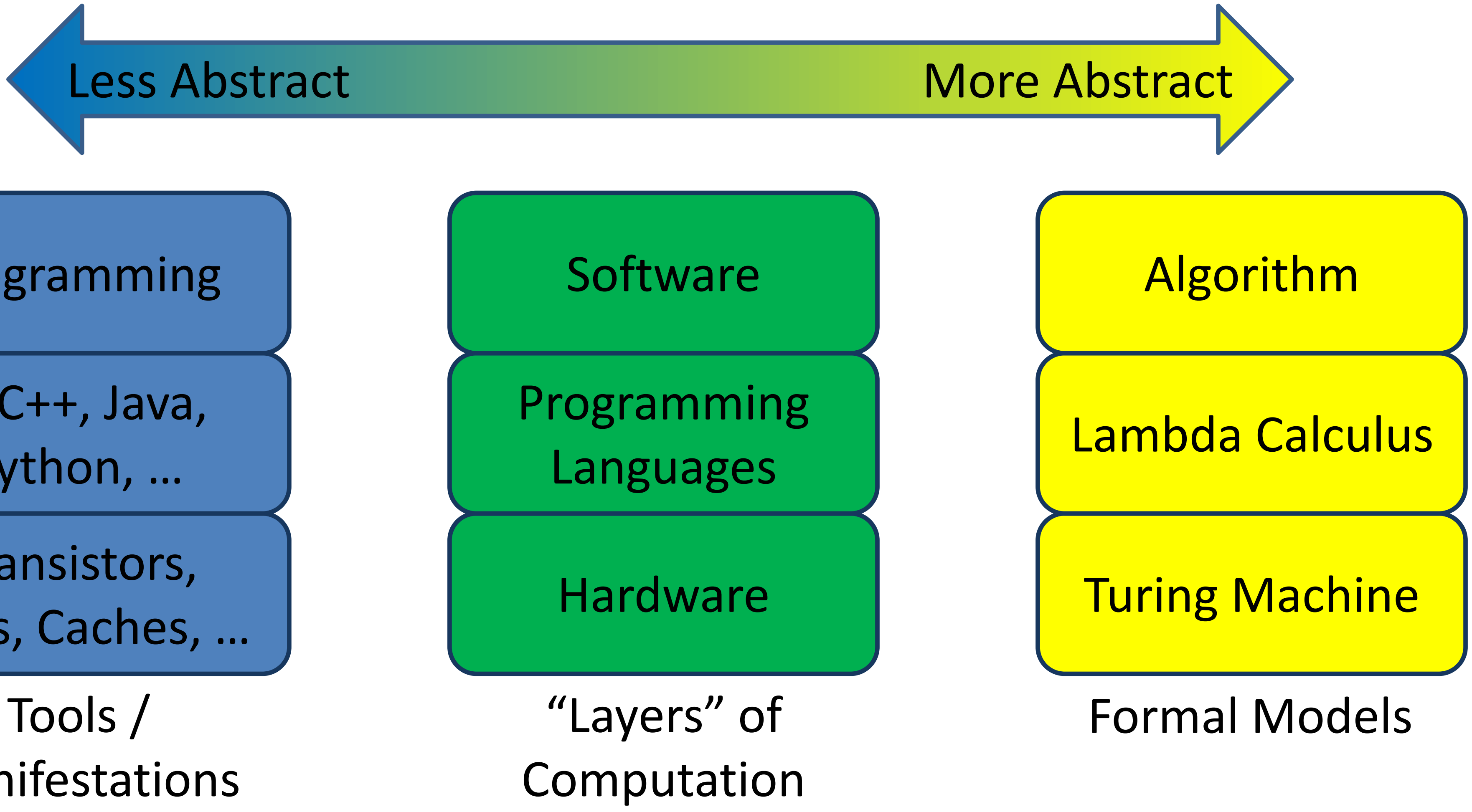
- Lambda Calculus:
  - Computes using variables and functions to manipulate them
  - Precursor to modern programming languages
- Turing Machines:
  - Precursor to modern CPUs
- Their equivalence means:
  - Anything we can do in hardware, we can also do in software (and the reverse)

How do we compute?

- We start with a universal Turing machine that we’ve built (a CPU)
- We use the lambda calculus (programming languages) to write algorithms
- Since the lambda calculus is equivalent to Turing machines, and our machine was universal, we can perform the operation described

What is Computer Science?

Computer Science studies all layers of abstractions of computing machines

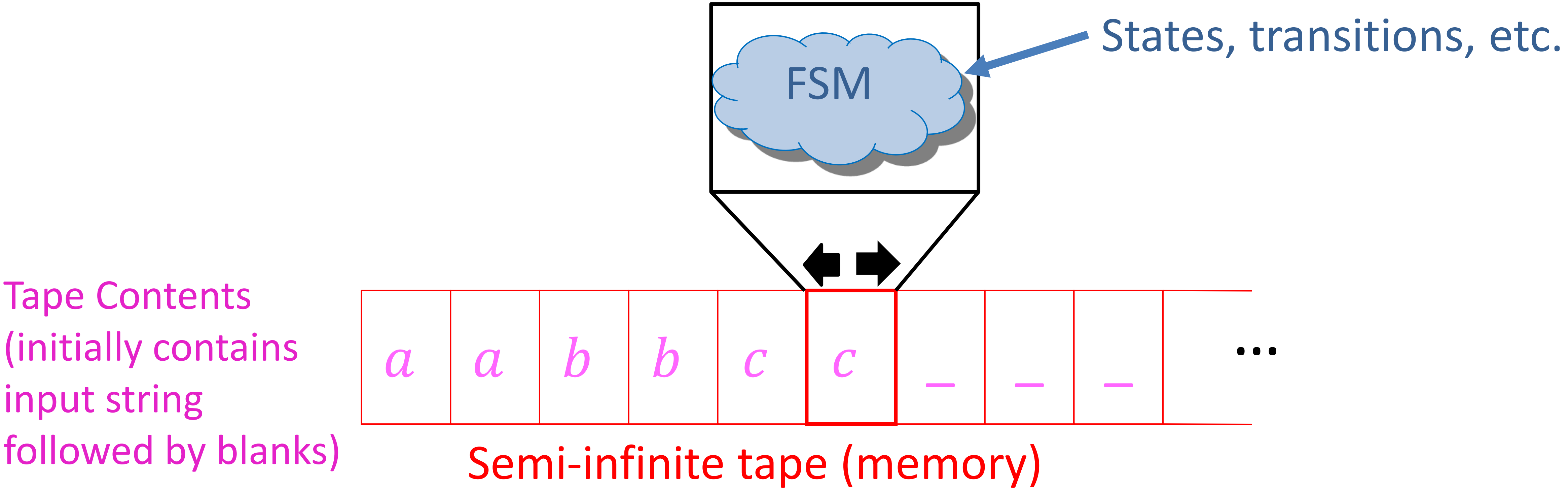


Turing Machines

- NFA/DFA:
  - Finite number of states,
  - read-once input,
  - transition using input character and state
- PDA:
  - Finite number of states,
  - read-once input,
  - stack (memory)
  - Transition using input character, state, and stack
  - Can push to the stack on transition

- Turing Machine:
  - Finite number of states,
  - Read-once input,
  - Semi-infinite tape (memory)
  - Transition using input character, state, and “current symbol” on tape
  - Can overwrite current symbol, move left/right on tape

Turing Machine



Operation: transitions outgoing from each state match on current character on the tape, when transitioning you can overwrite that character and move which cell you're reading

Can We do better?

- Church-Turing Thesis:
  - No!
  - Turing Machines can compute anything a human can compute
- Why is this reasonable?