Non-regular languages

Pumping Lemma: Let L be a regular language. **There exists** an integer p ("pumping length") for which **every** $w \in L$ with $|w| \geq p$ can be written as

$$w = xyz$$

such that

1. for every $i \geq 0$, $xy^i z \in L$, and
2. $|y| > 0$, and
3. $|xy| \leq p$.

Idea: if L has a DFA, then for every large enough word in L, its path in the state machine goes through a small cycle

Non-regular languages

Using the Pumping Lemma to prove L is not regular:

- assume L is regular
- then there exists a pumping length p
- select a string $w \in L$ of length at least p
- argue that **for every way** of writing $w = xyz$ that satisfies (2) and (3) of the Lemma, pumping on y yields a string not in L.
- contradiction.

Pumping Lemma examples

Theorem

$L = \{0^n 1^n \ : \ n \geq 0\}$ *is not regular.*

proof:

- let p be the pumping length for L
- choose $w = 0^p 1^p$

$$w = \underbrace{000000000\ldots0}_{p}\underbrace{111111111\ldots1}_{p}$$

- $w = xyz$, with $|y| > 0$ and $|xy| \leq p$.

Pumping Lemma examples

- 3 possibilities for where pumpable block occurs:

$$w = \underbrace{00000}_{x}\underbrace{000}_{y}\underbrace{0\ldots0111111111\ldots1}_{z}$$

$$w = \underbrace{000000000\ldots011}_{x}\underbrace{11111}_{y}\underbrace{11\ldots1}_{z}$$

$$w = \underbrace{0000000}_{x}\underbrace{00\ldots011111}_{y}\underbrace{1111\ldots1}_{z}$$

- Last two cases are ruled out by fact that pumping lemma requires $|xy| \leq p$
- in first case[1], pumping on y gives a string not in language L.

[1] indeed, also in other 2 cases

Pumping Lemma examples

Theorem

$L = \{w \ : \ w \text{ has an equal number of 0s and 1s}\}$ *is not regular.*

Proof:

- let p be the pumping length for L
- choose $w = 0^p 1^p$

$$w = \underbrace{000000000\ldots0}_{p}\underbrace{111111111\ldots1}_{p}$$

- $w = xyz$, with $|y| > 0$ and $|xy| \leq p$. Note that y is a non-empty string of 0's, so xy^2z is not in L.

There are other ways to prove languages are non-regular, which we will go over in exercises.

Summary in terms of "Course Program"

- Defined a simple programming model: **Deterministic Finite Automata**
- Positive results about this model:
  - The languages computed by this model are **closed** under union, concatenation, and star.
  - A more powerful model, NFAs, recognize **exactly the same** languages that DFAs do.
  - A convenient syntax, Regular expressions, describe **exactly the same** languages that DFAs (and NFAs) recognize.
- Negative results:
  - Some languages are **not regular**. This can be proved using the **Pumping Lemma**.

A more powerful machine

- limitation of FA related to fact that they can only "remember" a bounded amount of information
- What is the **simplest** alteration that adds unbounded "memory" to our machine?
- Should be able to recognize, e.g., $\{0^n 1^n \ : \ n \geq 0\}$.

Pushdown Automata

Pushdown automata (PDAs) have the same expressive power as *context-free grammars*; the languages they accept are *context-free languages*.

Pushdown Automata

Pushdown automata (PDAs) have the same expressive power as *context-free grammars*; the languages they accept are *context-free languages*.
A pushdown automaton is like a NFA but with an additional "memory stack" which can hold sequences of symbols from a memory alphabet.

Pushdown Automata

Pushdown automata (PDAs) have the same expressive power as *context-free grammars*; the languages they accept are *context-free languages*.
A pushdown automaton is like a NFA but with an additional "memory stack" which can hold sequences of symbols from a memory alphabet.
Automaton scans an input from left to right - at each step it may push a symbol onto the stack, or pop the stack. It cannot read other elements of the stack.

Pushdown Automata

Pushdown automata (PDAs) have the same expressive power as *context-free grammars*; the languages they accept are *context-free languages*.

A pushdown automaton is like a NFA but with an additional "memory stack" which can hold sequences of symbols from a memory alphabet.

Automaton scans an input from left to right - at each step it may push a symbol onto the stack, or pop the stack. It cannot read other elements of the stack.

Start with empty stack; accept if at end of string state is in subset $F \subseteq Q$ of accepting states and stack is empty. (**alternative definition:** the stack need not be empty in order to accept.)

Transitions

notation $\mathcal{P} = (Q, \Sigma, \Gamma, \delta, q_0, F)$ where Γ is stack alphabet and δ is transition function.

Action taken by machine is allowed to depend on top element of stack, input letter being read, and state. Action consists of new state, and possibly push/pop the stack.

Formally:

$\delta : Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \rightarrow \mathbf{P}(Q \times (\Gamma \cup \{\epsilon\}))$

i.e. for each combination of state, letter being read, and topmost stack symbol, we are given a set of allowable new states, and actions on stack.

For example:

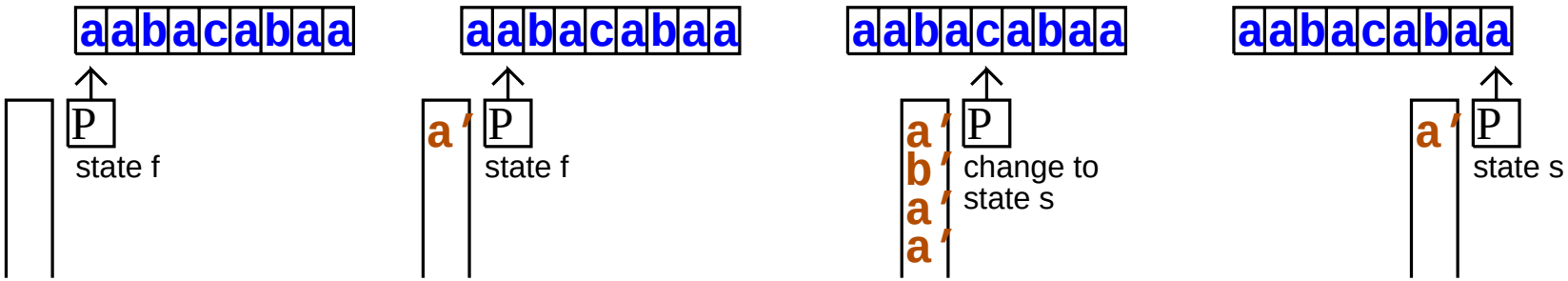
$\delta(q, \mathtt{a}, \mathtt{m}) = \{(q2, \mathtt{m2}), (q3, \mathtt{m3})\}$
means that in state q, if you read $\mathtt{a}$ with $\mathtt{m}$ at top of stack, you may move to state $q2$ and replace $\mathtt{m}$ with $\mathtt{m2}$. Alternatively you may move to state $q3$ and replace $\mathtt{m}$ with $\mathtt{m3}$.
$\delta(q, \mathtt{a}, \epsilon) = \{(q2, \mathtt{m2})\}$
means in state q with input $\mathtt{a}$, go to state $q2$ and push $\mathtt{m2}$ on top of stack.

Example: Palindromes

Input alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$
Use stack alphabet $\Gamma = \{\mathtt{a}', \mathtt{b}', \mathtt{c}'\}$

states $Q = \{f, s\}$ (f is "reading first half", s is "reading second half")
Initial state f.
Accepting states $F = \{s\}$
Transitions:
$\delta(f, \mathtt{a}, \epsilon) = \{(f, \mathtt{a}'), (s, \epsilon), (s, \mathtt{a}')\}$
$\delta(f, \mathtt{b}, \epsilon) = \{(f, \mathtt{b}'), (s, \epsilon), (s, \mathtt{b}')\}$
$\delta(f, \mathtt{c}, \epsilon) = \{(f, \mathtt{c}'), (s, \epsilon), (s, \mathtt{c}')\}$
$\delta(s, \mathtt{a}, \mathtt{a}') = \{(s, \epsilon)\}$; $\delta(s, \mathtt{b}, \mathtt{b}') = \{(s, \epsilon)\}$; $\delta(s, \mathtt{c}, \mathtt{c}') = \{(s, \epsilon)\}$;
$\delta(\text{anything else}) = \emptyset$

An Accepting Computation



Example of a *deterministic* PDA

Consider palindromes over $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ which contain exactly one $\mathtt{c}$.
Use stack alphabet $M = \{\mathtt{a}', \mathtt{b}'\}$

states $Q = \{f, s\}$ (f is "reading first half", s is "reading second half")
Initial state f, accepting states $T = \{s\}$
Transitions:
$\delta(f, \mathtt{a}, \epsilon) = (f, \mathtt{a}')$
$\delta(f, \mathtt{b}, \epsilon) = (f, \mathtt{b}')$
$\delta(f, \mathtt{c}, \epsilon) = (s, \epsilon)$
$\delta(s, \mathtt{a}, \mathtt{a}') = (s, \epsilon)$; $\delta(s, \mathtt{b}, \mathtt{b}') = (s, \epsilon)$;
$\delta(\text{anything else})$ is undefined (reject input).

So, deterministic PDAs can recognise certain non-regular languages (but can't recognise all PDA languages)

Another deterministic example

PDA to recognise "well-formed" strings of parentheses

- A single state s (accepting)
- Input alphabet $\{(,)\}$
- Memory alphabet $\{\mathtt{x}\}$

$\delta(s, (, \epsilon) = \{(s, \mathtt{x})\}$
$\delta(s,), \mathtt{x}) = \{(s, \epsilon)\}$

Comments

- The number of $\mathtt{x}$'s on the stack is the number of $($'s read so far minus number of $)$'s read.

Exercise

Design a NPDA for the language

$$\{a^i b^j c^k \ : \ i, j, k \geq 0 \text{ and } i = j \text{ or } i = k\}$$

Results for NPDA

Closure properties

Languages are closed under: Union, Concatenation, Kleene Star
But not, e.g. intersection

Alternative language-representation mechanism

Recall Kleene's Theorem, N/DFA $\leftrightarrow$ regular expression...

Similar result for NPDA:

- languages recognized by a **NPDA** are exactly the languages described by **context-free grammars**, and they are called the **context-free languages**.

Context-Free Grammars

- **Alphabet** Σ, a finite collection of symbols
- Set of **variables** (also called **non-terminals**), one of which is the **starting symbol** (usually letter **S**).
- set of **rules** (also called **productions**): a rule says that some **variable** may be replaced[2] by some string of letters/variables.

Start with S, apply rules until a string in Σ^* results...

[2] wherever it occurs in a string, i.e. regardless of context

Derivations in CFGs

A simple CFG

Alphabet **0**,**1**,**2**.

$S \to \mathbf{0}S\mathbf{1}$

$S \to B$

$B \to \mathbf{2}$

A derivation

$S \Rightarrow \mathbf{0}S\mathbf{1} \Rightarrow \mathbf{00}S\mathbf{11} \Rightarrow$
$\mathbf{000}S\mathbf{111} \Rightarrow$
$\mathbf{000}B\mathbf{111} \Rightarrow \mathbf{0002111}$

Another derivation

$S \Rightarrow B \Rightarrow \mathbf{2}$

- set of all strings generated using grammar G is the **language of the grammar** $L(G)$.
- called a **Context-free Language** (CFL)

Example CFG

programming language syntax often described using CFGs (*Backus-Naur form* notation often used), e.g.

$\langle$stmt$\rangle \rightarrow \langle$if-stmt$\rangle \mid \langle$while-stmt$\rangle \mid \langle$begin-stmt$\rangle \mid \langle$asgn-stmt$\rangle$
$\langle$if-stmt$\rangle \rightarrow$ **IF** $\langle$bool-expr$\rangle$ **THEN** $\langle$stmt$\rangle$ **ELSE** $\langle$stmt$\rangle$
$\langle$while-stmt$\rangle \rightarrow$ **WHILE** $\langle$bool-expr$\rangle$ **DO** $\langle$stmt$\rangle$
$\langle$begin-stmt$\rangle \rightarrow$ **BEGIN** $\langle$stmt-list$\rangle$ **END**
$\langle$stmt-list$\rangle \rightarrow \langle$stmt$\rangle \mid \langle$stmt$\rangle$; $\langle$stmt-list$\rangle$
$\langle$asgn-stmt$\rangle \rightarrow \langle$var$\rangle$:= $\langle$arith-expr$\rangle$
$\langle$bool-expr$\rangle \rightarrow \langle$arith-expr$\rangle \langle$compare-op$\rangle \langle$arith-expr$\rangle$
$\langle$compare-op$\rangle \rightarrow < \mid > \mid \leq \mid \geq \mid =$
$\langle$arith-expr$\rangle \rightarrow \langle$var$\rangle \mid \langle$const$\rangle \mid (\langle$arith-expr$\rangle \langle$arith-op$\rangle \langle$arith-expr$\rangle)$
$\langle$arith-op$\rangle \rightarrow + \mid - \mid * \mid /$
$\langle$const$\rangle \rightarrow$ **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
$\langle$var$\rangle \rightarrow$ **a** | **b** | **c** | ... | **x** | **y** | **z**

CFG formal definition

A **context-free grammar** is a 4-tuple

$$(V, \Sigma, R, S)$$

where

- V is a finite set called the **non-terminals**
- Σ is a finite set (disjoint from V) called the **terminals**
- R is a finite set of **productions** (or **rules**) where each production is a non-terminal and a string of terminals and non-terminals.
- $S \in V$ is the **start variable** (or start non-terminal)

CFG formal definition

Suppose u, v, w are strings of non-terminals and terminals, and $A \to w$ is a production.

- "uAv **yields** uwv" notation: $uAv \Rightarrow uwv$
  also: "**yields in 1 step**" notation: $uAv \Rightarrow^1 uwv$
- in general:
  "**yields in k steps**" notation: $u \Rightarrow^k v$
  meaning: there exists strings $u_1, u_2, \ldots u_{k-1}$ for which
  $u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_{k-1} \Rightarrow v$
- notation: $u \Rightarrow^* v$
  - meaning: $\exists k \geq 0$ and strings $u_1, ..., u_{k-1}$ for which
    $u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow ... \Rightarrow u_{k-1} \Rightarrow v$
- if $u = S$ (the start symbol), the above is a **derivation of** v
- The **language of** G, denoted $L(G)$ is:

$$\{w \in \Sigma^* \ : \ S \Rightarrow^* w\}$$

One more example

arithmetic expressions using infix operators $+$, $*$, parentheses and "atoms" $\mathtt{x}$ and $\mathtt{y}$.

A set of arithmetic expressions

start symbol: E (no other variable symbols)
alphabet: $+$, $*$, $($, $)$, $\mathtt{x}$, $\mathtt{y}$
Rules:
$E \to E{*}E$
$E \to E{+}E$
$E \to (E)$
$E \to \mathtt{x}$
$E \to \mathtt{y}$

To see that $\mathtt{x} * (\mathtt{x} + \mathtt{y})$ is in the language, find a derivation.
$E \Rightarrow E{*}E \Rightarrow E{*}(E) \Rightarrow E{*}(E{+}E)$
$\Rightarrow \mathtt{x} * (E{+}E) \Rightarrow \mathtt{x} * (\mathtt{x}{+}E) \Rightarrow \mathtt{x} * (\mathtt{x} + \mathtt{y})$
notation: $E \stackrel{*}{\Rightarrow} \mathtt{x} * (\mathtt{x} + \mathtt{y})$

NPDA, CFG equivalence

Theorem

a language L is recognized by a NPDA **iff** *L is described by a CFG.*

Must prove **two** directions:
($\Rightarrow$) L is recognized by a NPDA **implies** L is described by a CFG.
($\Leftarrow$) L is described by a CFG **implies** L is recognized by a NPDA.

To prove this equivalence, it's useful to define some other mechanisms that turn out to have equivalent expressive power (first: "Chomsky normal form" CFGs).

Chomsky Normal Form

A restricted form of CFG that's easier to work with:

Definition

In Chomsky normal form every production has form

1. $A \to BC$
2. $S \to \epsilon$ (**note S is start symbol**)
3. $A \to a$

where A, B, C are any non-terminals, and a is any terminal.

Theorem

Every CFL is generated by a CFG in Chomsky Normal Form.

Proof: Transform any CFG into equivalent CFG in CNF. 4 steps:

- add a new start symbol, which can produce prior start
- remove "ϵ-productions" $A \to \epsilon$
- eliminate "unit productions" $A \to B$
- convert remaining rules into proper form

conversion to Chomsky Normal Form

(general idea; some detail missing)

- add a new start symbol S_0: add production $S_0 \to S$
- remove "ϵ-productions" $A \to \epsilon$
  - **for each production with A on RHS, add production with A's removed: e.g. for each rule $R \to uAv$, add $R \to uv$**

  ϵ rule may filter back to start symbol in doing this.
- eliminate "unit productions" $A \to B$
  - **for each production with B on LHS: $B \to u$, add rule $A \to u$ (do this for A=Start symbol also)**

Note. In removing "ϵ-productions" $A \to \epsilon$ if we had a production $R \to A$ we create a new production $R \to \epsilon$ *unless we already had removed such a production before*. Thus the process of removing these productions terminates!

conversion to Chomsky Normal Form

- convert remaining rules into proper form

replace production of form:

$A \to u_1 U_2 u_3 \ldots u_k$

with:

$A \to U_1 A_1 \qquad U_1 \to u_1$

$A_1 \to U_2 A_2$

$A_2 \to U_3 A_3 \qquad U_3 \to u_3$

$\vdots$

$A_{k-1} \to U_{k-1} U_k \qquad U_k \to u_k$

Introduce new non-terminals $A_1 \ldots A_{k-1}$ and also $U_1, U_3 \ldots U_k$ for every terminal in RHS

for any nonterminal on the RHS, like U_2, we don't need a new rule for it.

Implementing CFGs with an NPDA

New goal: Take a **Chomsky Normal Form** CFG, simulate any *leftmost* derivation with a **NPDA**

Example: CNF CFG for 0^n21^n

$A \to ZC$
$Z \to 0$
$C \to AD$
$D \to 1$
$A \to 2$

Typical leftmost derivation (with some steps skipped)
$A \Rightarrow^* 0C \Rightarrow 0AD \Rightarrow^* 00CD \Rightarrow 00ADD \Rightarrow 002DD \Rightarrow 0021D \Rightarrow 00211$

Always a set of terminals followed by a set of nonterminals

Think of an NPDA simulating this derivation where the terminals are the stuff we've already read, the nonterminals are the stack (leftmost NT is the top of stack)

Target of translation: Variation on NPDAs

Slight variations of NPDAs have the same expressiveness:

- allow to do "push-and-swap", rather than either a push or a swap
- demand acceptance by empty stack + final state

Call this a **"Demanding, Energetic NPDA"**
DENPDAs can be converted to normal NPDAs:

- convert push-and-swap to an ϵ-move swap followed by a push
- get rid of Demanding requirement by: recording the beginning of the stack by pushing symbol \$ on in the beginning, adding transitions popping \$ into a new accepting state

Now: suffices to translate CNF grammar to DENPDA, using idea on previous slide.

Implementing CFGs with a (DE)NPDA

1. $A \to ZC$
2. $Z \to 0$
3. $C \to AD$
4. $D \to 1$
5. $A \to 2$

Transitions of corresponding NPDA

1. Read nothing; replace A (on stack) with C; then add Z
2. Read a 0 and pop Z
3. Read nothing; replace C with D; add A
4. Read a 1 and pop D
5. Pop A and read a 2

Derivation
$A \Rightarrow ZC \Rightarrow 0C \Rightarrow 0AD \Rightarrow$
$0ZCD \Rightarrow 00CD \Rightarrow$
$00ADD \Rightarrow 002DD \Rightarrow 0021D$
$\Rightarrow 00211$

Implementing CFGs with DENPDAs: construction

Case 1: There is a rule $S \to \epsilon$

(So, empty string is accepted)

NPDA has two states q_0 and q_1, with first being start state and both being accept states.

Have a rule $\delta(q_0, \epsilon, \epsilon) \to (q_1, S)$ that pushes start symbol onto the stack, reading nothing

For each CFG rule $A \to BC$ add a push-and-swap that reads nothing

For each CFG rule $A \to a$ add a pop rule that reads an a

Case 2: there is no rule $S \to \epsilon$

NPDA has two states, q_0 and q_1, q_0 is the initial state and q_1 is an accept state

In q_0, have rule that pushes start symbol on to the stack and moves to q_1. In q_1 have rules as above, remaining in q_1.

Every NPDA can be programmed via a CFG

(The "⇒" part of what we are proving:) L is recognized by a NPDA **implies** L is described by a CFG.

Again use alternative form of NPDA

- single accept state
- demanding (stack must be empty when accepts)
- each transition *either* **pushes** *or* **pops** a symbol, but not both

Easy to convert to this form by adding epsilon transitions, new accept state.
Now suffices to convert every NPDA of this form to a CFG

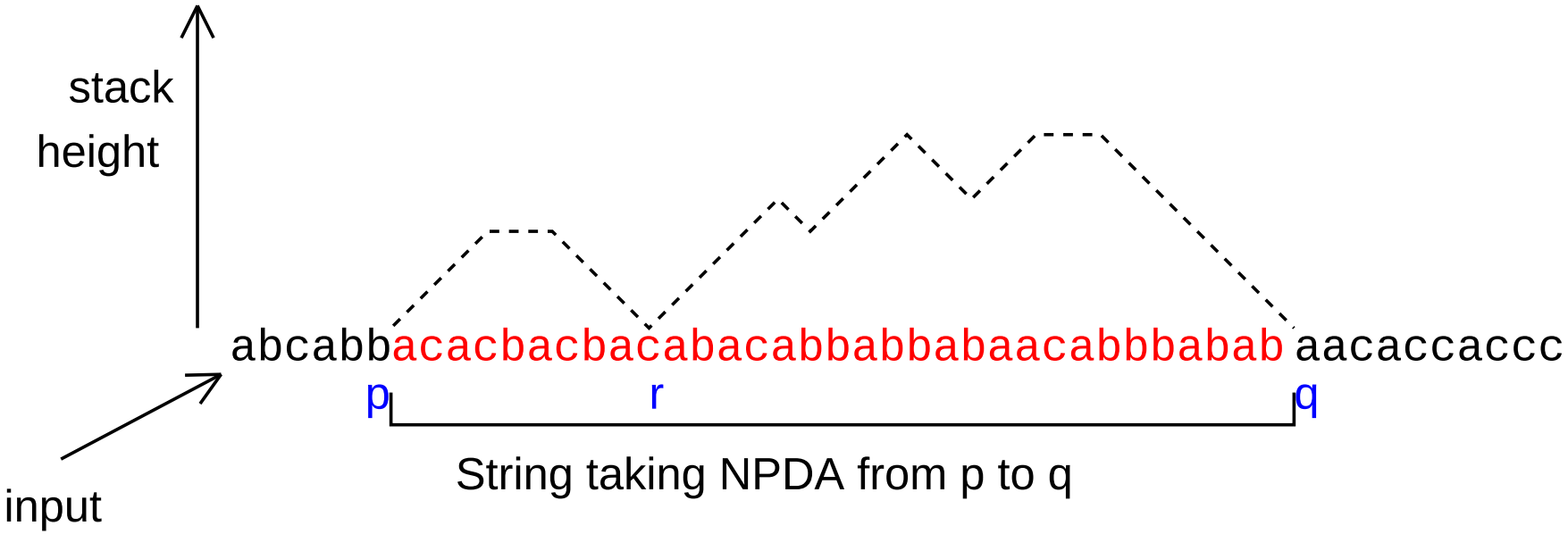
Converting PDA to CFG

- **main idea:** non-terminal $A_{p,q}$ generates exactly the strings that take the NPDA from state p (with empty stack) to state q (with empty stack)
- then $A_{start,accept}$ generates all of the strings in the language recognized by the NPDA.

Two possibilities to get from state p to q while reading a string w (starting with empty stack on state p, ending with empty stack in state q):

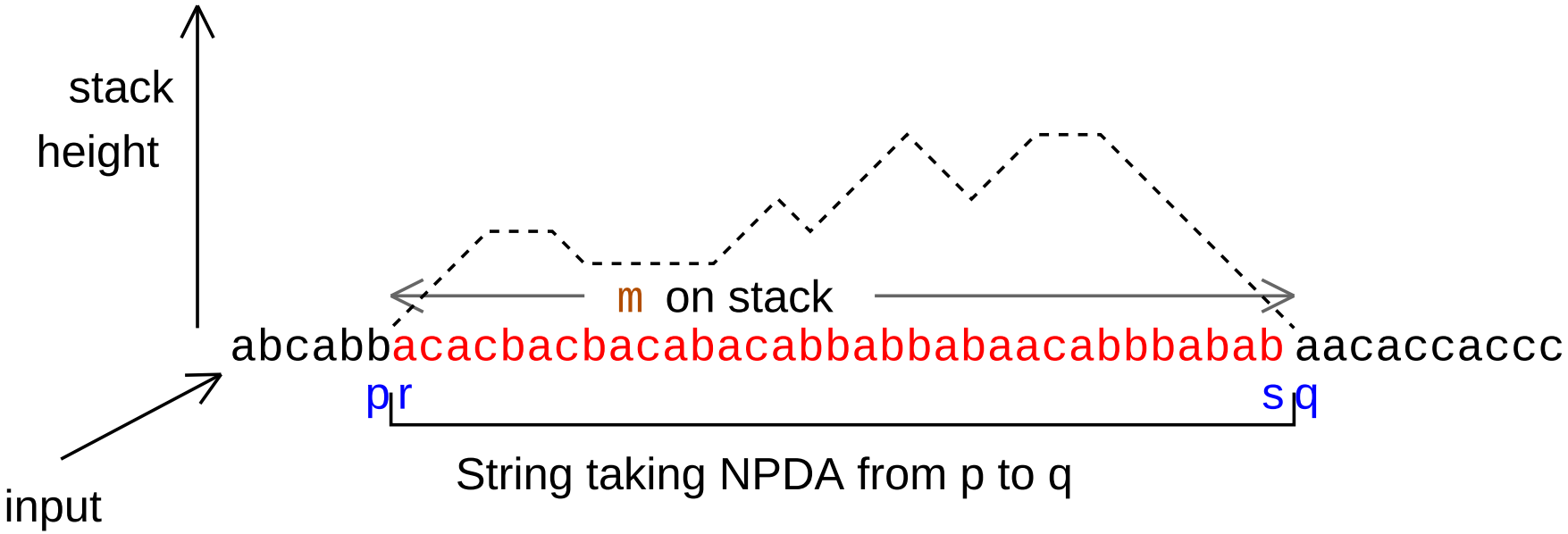
- stack becomes empty
- stack gets loaded with some symbol m which does not get unloaded until end of w

NPDA to CFG, case 1



- NPDA $(Q, \Sigma, \Gamma, \delta, i, \{t\})$
- CFG G contains
  - non-terminals $V = \{A_{p,q} \ : \ p, q \in Q\}$
  - start variable $A_{initial,accept}$
  - productions:
    for every $p, r, q \in Q$, add the rule $A_{p,q} \rightarrow A_{p,r}A_{r,q}$

NPDA to CFG, case 2



- NPDA $(Q, \Sigma, \Gamma, \delta, i, \{t\})$
- CFG G also contains
  - productions:
    for every $p, r, s, q \in Q$, $\mathtt{m} \in \Gamma$ and $a, b \in (\Sigma \cup \{\epsilon\})$
    if $(r, \mathtt{m}) \in \delta(p, a, \epsilon)$ and $(q, \epsilon) \in \delta(s, b, \mathtt{m})$, add the rule
    $A_{p,q} \rightarrow aA_{r,s}b$.

NPDA to CFG (last slide)

Finally,

- for every $p \in Q$, add the rule $A_{p,p} \rightarrow \epsilon$
- if $(r, \epsilon) \in \delta(p, a, \epsilon)$ add $A_{p,r} \rightarrow a$

Proving this translation works (general approach):

- show that any word w accepted by the NPDA can be generated by the CFG, and vice versa.
- An accepting computation of the NPDA corresponds with a derivation of the same word using the CFG.

Negative results for CFLs

CFL Pumping Lemma

Given a CFL, any sufficiently long string in that CFL has *two* substrings (at least one of which is non-empty) such that if both of these substrings are "pumped" you generate further words in that CFL.

Negative results for CFLs

CFL Pumping Lemma

Given a CFL, any sufficiently long string in that CFL has *two* substrings (at least one of which is non-empty) such that if both of these substrings are "pumped" you generate further words in that CFL.

More formally...

Given a CFL L, **there exists** a number p such that any string $\alpha \in L$ with $|\alpha| \geq p$ can be written as

$$\alpha = sxtyu$$

such that

$$sx^2ty^2u,\ sx^3ty^3u,\ sx^4ty^4u, \ldots$$

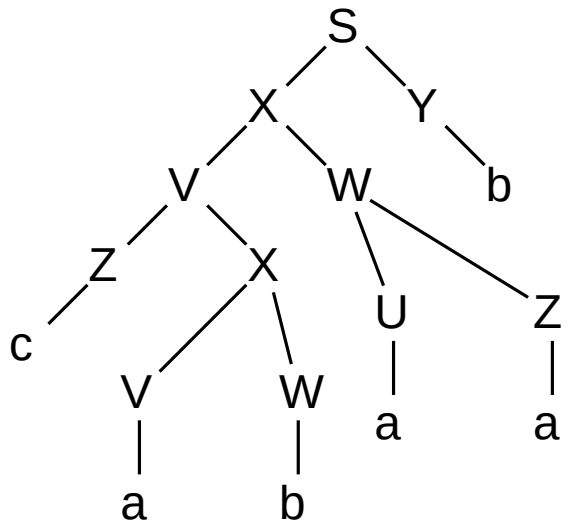
are all members of L, $|xty| \leq p$ and $|xy| > 0$ (which we need for all strings in this collection to be distinct).

How do we find suitable substrings x and y?

Consider the following (Chomsky normal form) grammar

$$
\begin{array}{rcl}
S & \longrightarrow & XY \\
U & \longrightarrow & \texttt{a} \\
V & \longrightarrow & ZX \mid \texttt{a} \mid \texttt{b} \\
X & \longrightarrow & VW \mid \texttt{a} \\
Y & \longrightarrow & \texttt{b} \mid \texttt{c} \\
Z & \longrightarrow & \texttt{a} \mid \texttt{c} \\
W & \longrightarrow & UZ \mid \texttt{b}
\end{array}
$$

The string `cabaab` belongs to the language. Also it contains "suitable substrings" x and y which we can find by looking at a *derivation tree* of `cabaab`.



We find two X's on the same path. We can say:

$$X \Longrightarrow^* \mathtt{c}X\mathtt{aa}$$

(via the sequence $X \Rightarrow VW \Rightarrow ZXW$
$\Rightarrow \mathtt{c}XW \Rightarrow \mathtt{c}XUZ \Rightarrow \mathtt{c}X\mathtt{a}Z \Rightarrow \mathtt{c}X\mathtt{aa}$)
Generally: $X \Longrightarrow^* \mathtt{cc}X\mathtt{aaaa} \Longrightarrow^* \mathtt{ccc}X\mathtt{aaaaaa}\ldots$

The substrings c and aa can be pumped, because a derivation of cabaab can go as follows:

$$\begin{aligned} S &\Longrightarrow^* X\mathtt{b} \\ &\Longrightarrow^* \mathtt{c}\underline{X}\mathtt{aab} \\ &\Longrightarrow^* \mathtt{cabaab} \end{aligned}$$

but the underlined X could have been used to generate extra c's and aa's on each side of it.

Given a grammar, it's not hard to see that any sufficiently long string will have a derivation tree in which a path down to a leaf must contain a repeated variable.
If the grammar is in Chomsky normal form and has v variables, any string of length $> 2^{v+1}$ will necessarily have such a derivation tree.

Examples

The language $\{1^n : n \text{ is a square number}\}$ is not a CFL.

Prove the following is not a CFL:
$\{$ 1, 101, 101001, 1010010001, 101001000100001, $\ldots\}$
Proof by contradiction: suppose string s (in above set) has substrings x and y that can be pumped.
x and y must contain at least one 1, or else all new strings generated by repeating x and y would have same number of 1's, a contradiction.
But if x contains a 1, then any string that contains x^3 as a substring must have 2 pairs of 1's with the same number of 0's between them, also a contradiction.

Another example

Prove the following language is not a CFL: Let L be words of the form ww (where w is any word over $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$)
(e.g. $\mathtt{aa}$, $\mathtt{abcabc}$, $\mathtt{baaabaaa}$, ...)
Let p be "sufficiently large" word length promised by pumping lemma.
Choose $w = \mathtt{a}^{p+1}\mathtt{b}^{p+1}\mathtt{a}^{p+1}\mathtt{b}^{p+1}$, so $w \in L$
We can argue that there is no subword of w of length p which contains any pair of subwords which, if repeated once, give another member of L.

Problems associated with analysis of CFLs

- Ideally, we'd have **efficient algorithms** to decide properties of CFL or PDA
  - basic question: decide whether string w is in given CFL L
    - e.g.programming language often described by CFG. Determine if string is valid program. This is called a **membership test** (related to **parsing**)
  - Another question: **emptiness test** – is the language given by a PDA/CFG empty?
  - Equivalence: do 2 given CFGs define the same language?
- If CFL recognized by **deterministic PDA**, just simulate the PDA to check membership $\rightarrow$ linear time algorithm
  - but not all CFLs are
- For NFA, how difficult is it to do a membership test?
- For NPDA or CFG, not clear how to do it at all (for an NPDA, infinitely many runs on a single input).

We consider this in the exercises.

Summary

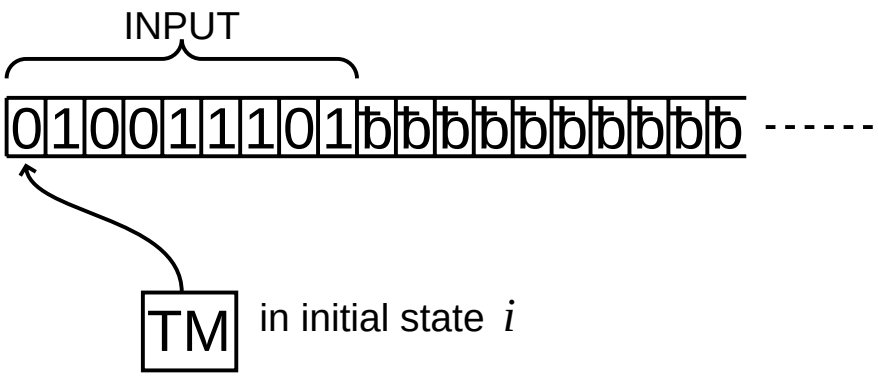
Nondeterministic Pushdown Automata (NPDA)

- **Positive:**
  - closure under regular operations
  - Context-Free Grammars (CFGs) describe Context-Free Languages (CFLs)
  - efficient membership test
- **Negative:**
  - Pumping Lemma can be used to show languages are not recognized by NPDA
  - Can be used to show lack of closure under intersection, complement

A more powerful machine

- limitation of NPDA related to fact that their memory is stack-based (last in, first out)
- What is the **simplest** alteration that adds general-purpose "memory" to our machine?
- Should be able to recognize, e.g., $\{a^n b^n c^n \ : \ n \geq 0\}$

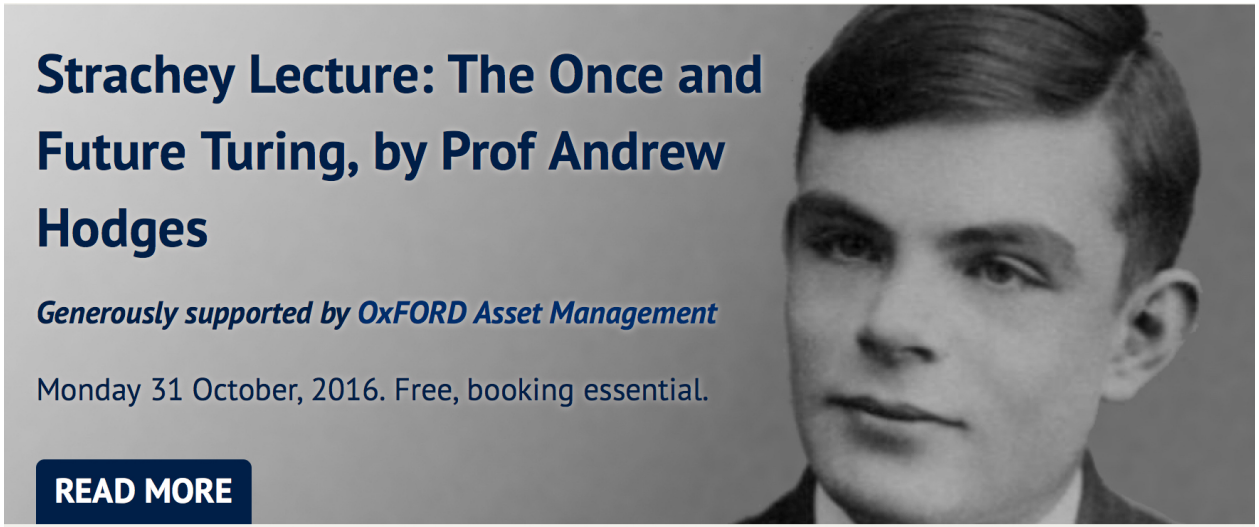
Turing machines



New capabilities:

- infinite tape
- can read OR write to tape
- read/write head can move **left** and right

(so input is not nec. "consumed" when read)



Turing Machine

Informal description:

- input written on left-most squares of tape
- rest of squares are blank
- at each point, take a step determined by
  - **current symbol being read**
  - **current state of finite control**
- a step consists of
  - **writing new symbol**
  - **moving read/write head left or right**
  - **changing state**

TM formal definition

A TM is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ where:

- Q is a finite set called the **states**
- Σ is a finite set called the **input alphabet**
- Γ is a finite set called the **tape alphabet**
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is a function called the **transition function**
- q_0 is an element of Q called the **start state**
- q_{accept}, q_{reject} are the **accept** and **reject** states

Examples on web page; **many** variants of definition exist

TM configurations

- At every step in a computation, a TM is in a **configuration** determined by:
  - the contents of the tape
  - the state
  - the location of the read/write head
- next step completely determined by current configuration
- shorthand: string $u; q; v$ with $u, v \in \Gamma^*, q \in Q$ meaning:
  - tape contents: uv followed by blanks
  - in state q
  - head on first symbol of v

Formal definition of TM execution

configuration C_1 **yields** configuration C_2 if TM can legally move from C_1 to C_2 in 1 step

- notation: $C_1 \Rightarrow C_2$
- also: **"yields in 1 step"** notation: $C_1 \Rightarrow^1 C_2$
- **"yields in k steps"** notation: $C_1 \Rightarrow^k C_2$
  if there exists configurations $D_1, D_2, \ldots, D_{k-1}$ for which
  $C_1 \Rightarrow D_1 \Rightarrow D_2 \Rightarrow \ldots \Rightarrow D_{k-1} \Rightarrow C_2$
- also: **"yields in some number of steps"** $(C_1 \Rightarrow^* C_2)$

Formal definition of TM execution

notation

$u, v \in \Gamma^*$; $a, b, c \in \Gamma$;
$q_i, q_j \in Q$

(examples may be more helpful to get the idea...)

- Formal definition of "yields":

$$ua; q_i; bv \Rightarrow u; q_j; acv$$

if $\delta(q_i, b) = (q_j, c, L)$ (overwrite b with c and move left)

$$ua; q_i; bv \Rightarrow uac; q_j; v$$

if $\delta(q_i, b) = (q_j, c, R)$ (overwrite b with c, move right)

- two special cases:
  - left end: $q_i; bv \Rightarrow q_j; cv$ if $\delta(q_i, b) = (q_j, c, L)$ **(move Left stays put)**
  - "right end" $ua; q_i$ – replace with $ua; qi; \overline{b}$ (since this is what it "really is") and handle right moves as above

TM acceptance

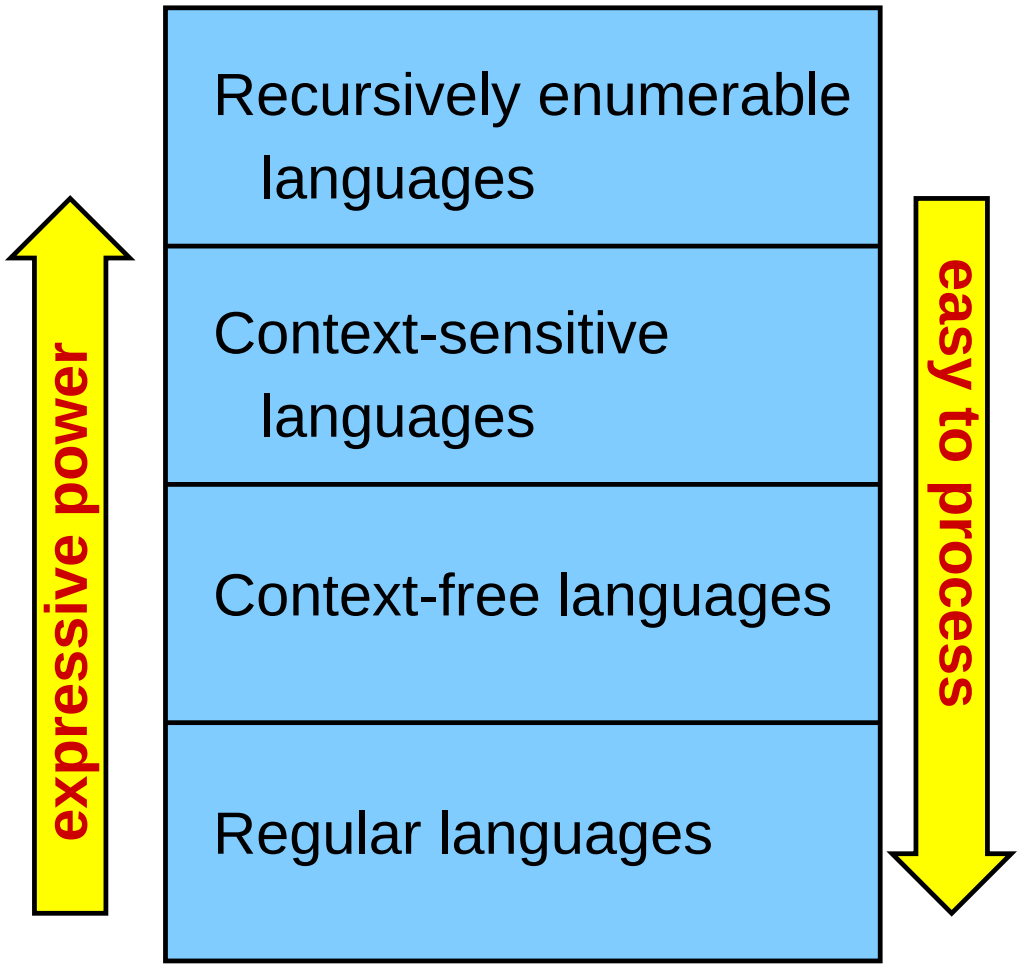
Computation ends when some combination of symbol and state is reached for which no transition is defined, or if TM runs off the LHS of tape.
TM is deemed to have accepted if the state it's in is one of the accepting states.

So, now I've switched definition to one that allows more than one accepting state...

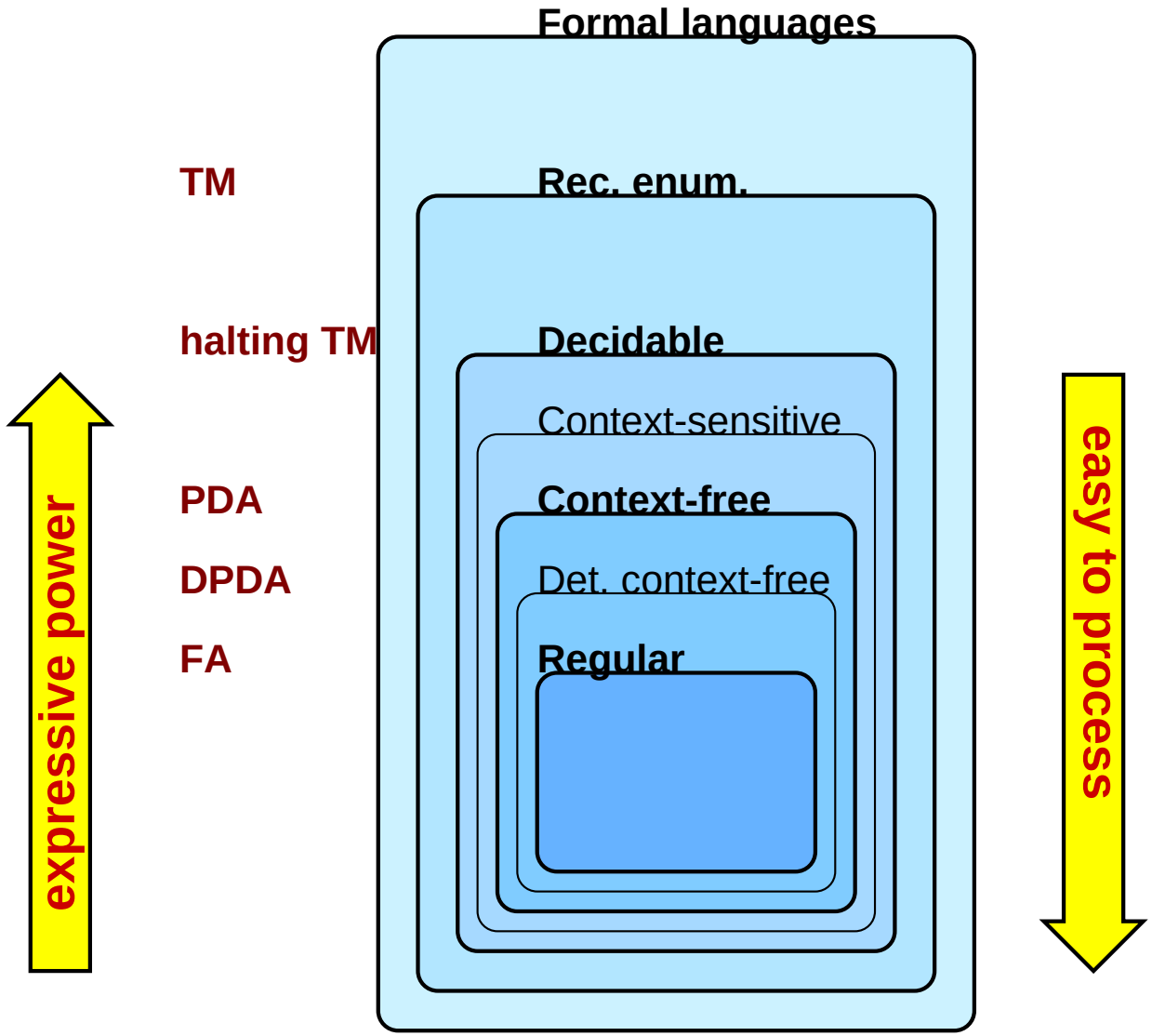
Deciding and Recognising

- $L(M)$ is set of strings accepted TM M, the language that M **recognises**.
- If M rejects every string $x \notin L$ then M **decides** L.
- Note that FAs, PDAs always accept/reject any string, not so TMs!
- It's easy to see that TMs can simulate FAs; in fact they can also simulate PDAs
- A language recognised by a TM is said to be **semi-decidable** or **recursively enumerable**. A language decided by some TM is said to be **decidable** or **recursive**.

Chomsky hierarchy



Chomsky hierarchy + more detail



Deciding and Recognizing

$$input \xrightarrow{\textbf{machine}} \begin{cases} \bullet \text{ accept} \\ \bullet \text{ reject} \\ \bullet \text{ loop forever.} \end{cases}$$

- TM M:
  - $L(M)$ = set of accepted strings, is the language M recognizes.
  - if M rejects every $x \notin L(M)$ it decides L
  - set of languages recognised by some TM is called semi-decidable or computably enumerable (CE) or recursively enumerable (RE).
  - set of languages decided by some TM is called Turing-decidable or decidable or computable or recursive.

Sometimes use Computable more generally for functions computed by Turing Machines (definition is pretty obvious). Decidable is just for decision problems/languages.

Coming up next:

- Convince ourselves that TMs really can carry out "generic computation". ("Church-Turing thesis")
- Then: decidable versus undecidable languages. Limitations on what program analysis can achieve (important!)
- start with a mention (next slide) that TMs can indeed recognise CFLs

(later: computational complexity, logic)

Membership test for CFLs

An algorithm: **IsGenerated(x,A)**

if $|x| = 1$, then return YES if $A \to x$ is a production,
  else return NO
for all $n - 1$ ways of splitting $x = yz$
  for all $\leq m$ productions of the form $A \to BC$
    if IsGenerated(y,B) and IsGenerated(z,C),
      return YES
return NO

Can implement this with a TM! (assuming that TMs can indeed simulate any well-defined algorithm...)

Positive Results

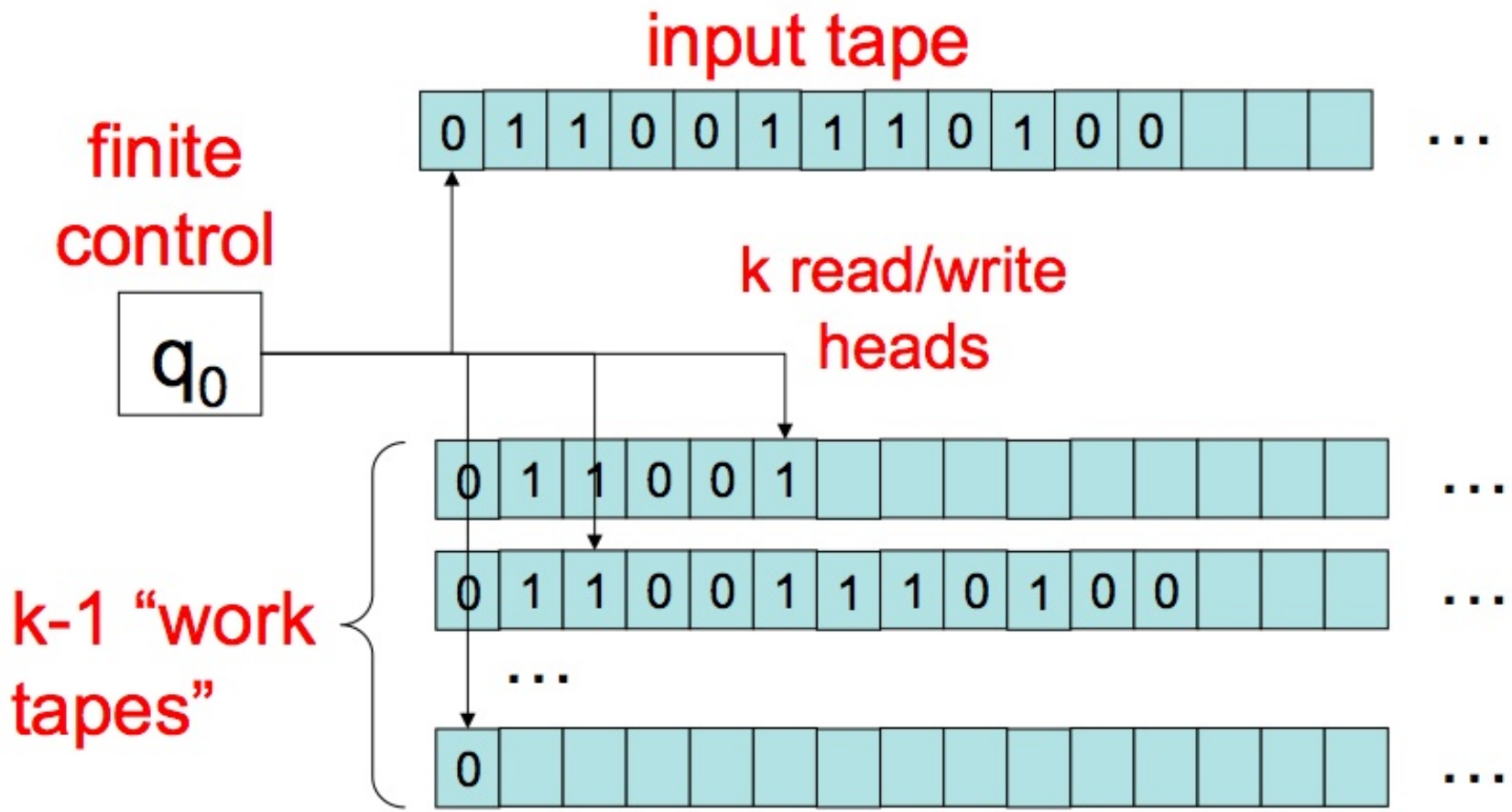
Lots of variations of TMs turn out to define the same set of languages. This is how we convince ourselves of Church-Turing thesis!

Some trivial ones: can change the policy on moving left, can allow moves that write multiple symbols (analogy with variation of PDA def.).

Next: a more interesting variant: multi-tape TM

Multitape TMs

A useful variant: k-tape TM



Multitape TMs

Informal description of k-tape TM:

- input written on left-most squares of tape 1 (other tapes are "work tapes")
- rest of squares are blank on all tapes
- at each point, take a step determined by
  - current k symbols being read on k tapes
  - current state of finite control
- a step consists of
  - writing k new symbols on k tapes
  - moving each of k read/write heads left or right
  - changing state

Multitape TM formal definition

A Multi-Tape TM is a 7-tuple

$(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ where:

everything is the same as a TM except the transition function:

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R\}^k$$

$\delta(q_i, a_1, a_2, \ldots, a_k) = (q_j, b_1, b_2, \ldots, b_k, L, R, \ldots, L) =$

"in state q_i reading $a_1 a_2 \ldots a_k$ on k tapes, move to state q_j, write $b_1 b_2, \ldots, b_k$ on k tapes, move L, R on k tapes as specified."

Multitape TMs

Theorem

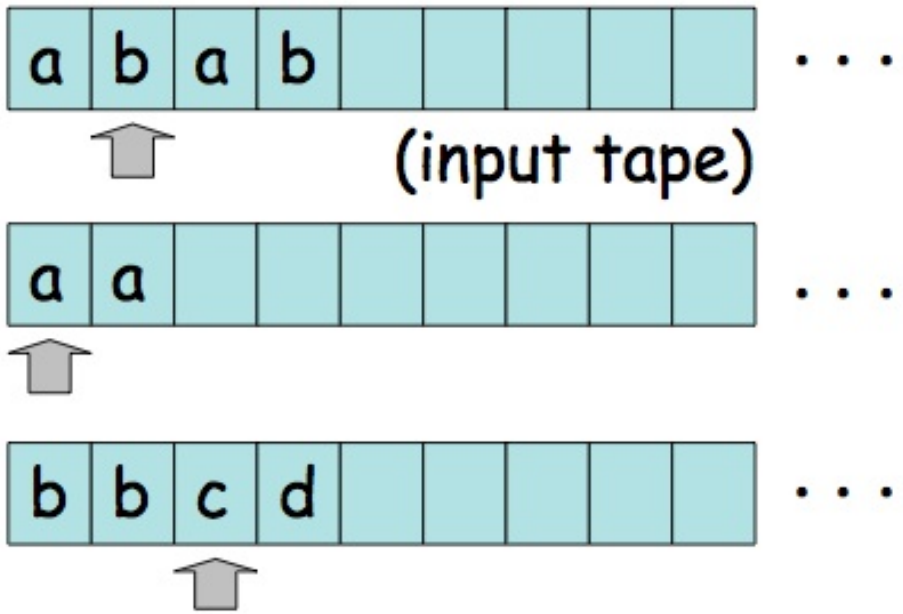
every k-tape TM has an equivalent single-tape TM.

Proof:

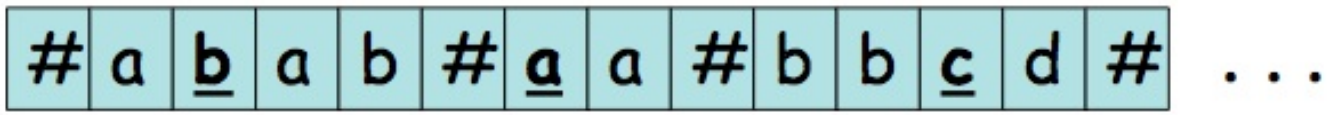
- Idea: simulate k-tape TM on a 1-tape TM.

Multitape TMs

simulation of k-tape TM M by single-tape TM M':

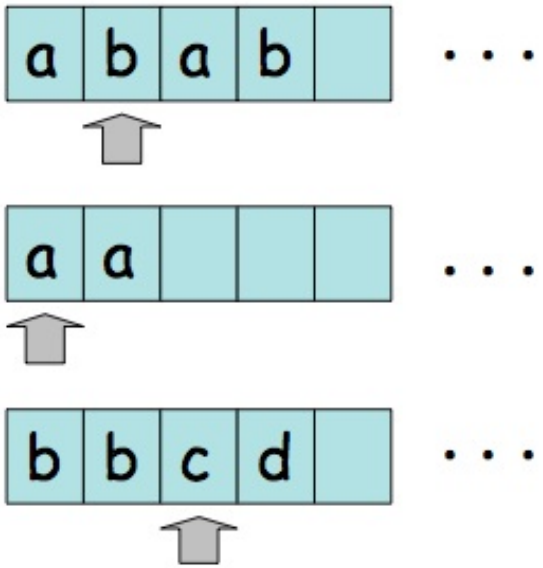


- add new symbol **x** for each old x
- marks location of "virtual heads"



Single move to M simulated by many moves of M'.
After each "macro move" (sequence in M' mimicking one move of M), head returning to the beginning.

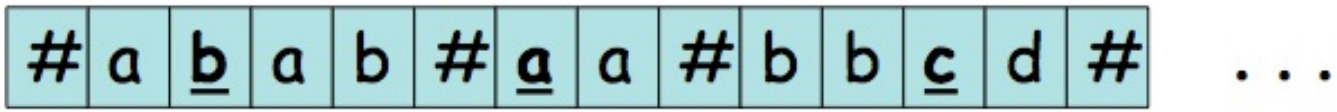
Multitape TMs



In control state of M', can store state of M, and also which part of M' tape we are looking at.

Macro step:

- scan tape, as you pass each virtual head, remember the symbol on each using the control state
- make changes to reflect each head move of M
- simulating is easy, except when a head of M moves into blank space — need to shift everything to make room. But can do this by remembering the previous symbol in M''s control state.



Programming TMs: basic operations

Convince yourself that the following types of operations are easy to implement as part of TM "program"
(but perhaps tedious to write out)

- copying
- moving
- incrementing/decrementing
- arithmetic operations $+, -, *, /$

Encodings

- the input to a TM is always a string in Σ^*
- often we want to interpret the input as representing another object
- examples:
  - tuple of strings (x, y, z)
  - 0/1 matrix
  - graph in adjacency-list format
  - Context-free Grammar

Using these encodings, you can show that various "algorithms" that we talked about before (Membership for CFL, emptiness of FA, equivalence of FA) can be performed by a TM on the encodings (I use $\langle O \rangle$ for string coding object O), and hence the encoded problems are all decidable.

Church-Turing Thesis

- many other models of computation
  - multitape TM,
  - others don't resemble TM at all (e.g. Random Access Machines)
  - common features
    - unrestricted access to unlimited memory
    - finite amount of work in a single step
- every single one can be simulated by a TM
- many are equivalent to a TM
- problems that can be solved by computer does not depend on details of the model!

Church-Turing Thesis

the belief that TMs formalise our intuitive notion of an algorithm is:

The Church-Turing Thesis
everything we can compute on a physical computer can be computed on a Turing machine.

Note: this is a belief, not a theorem.

Restricting to languages/decision problems, this says: if L is a language where membership can be determined by a computer, then L is decidable