CS311 Computational Structures

Review

Lecture 19

Portland State UNIVERSITY

Course Objectives

Upon the successful completion of this course students will be able to:

1. Find regular grammars and context-free grammars for simple languages whose strings are described by given properties.
2. Apply algorithms to: transform regular expressions to NFAs, NFAs to DFAs, and DFAs to minimum-state DFAs; construct regular expressions from NFAs or DFAs; and transform between regular grammars and NFAs.
3. Apply algorithms to transform: between PDAs that accept by final state and those that accept by empty stack; and between context-free grammars and PDAs that accept by empty stack.
4. Describe LL(k) grammars; perform factorization if possible to reduce the size of k; and write recursive descent procedures and parse tables for simple LL(1) grammars.
5. Transform grammars by removing all left recursion and by removing all possible productions that have the empty string on the right side.

6. Apply pumping lemmas to prove that some simple languages are not regular or not context-free.
7. State the Church-Turing Thesis and solve simple problems with some of the following models of computation: Turing machines (single-tape and multi-tape); while-loop programs; partial recursive functions; Markov algorithms; Post algorithms; the lambda calculus; and Post systems.
8. Describe the concepts of unsolvable and partially solvable; state the halting problem and prove that it is unsolvable and partially solvable; and use diagonalization to prove that the set of total computable functions cannot be enumerated.
9. Describe the hierarchy of languages and give examples of languages at each level that do not belong in a lower level.
10. Describe the complexity classes P, NP, and PSPACE.
11. Use an appropriate programming language as an experimental tool for testing properties of computational structures.

Regular Languages

Regular Languages

- What's a Grammar?

Regular Languages

- What's a Grammar?
  - ‹T, V, R, S›, where:

Regular Languages

- What's a Grammar?
  - ‹T, V, R, S›, where:

Regular Languages

- What's a Grammar?
  - ‹T, V, R, S›, where:

- Grammar for L_1= ab*

Regular Languages

- What's a Grammar?
  - ‹T, V, R, S›, where:

- Grammar for L_1= ab*
  - What are the constraints on a regular grammar?

Regular Languages

- What's a Grammar?
  - ‹T, V, R, S›, where:

- Grammar for L_1= ab*
  - What are the constraints on a regular grammar?
    - $S \rightarrow \omega$, or $S \rightarrow \omega V$, where $\omega \in T^*$ (a possibly empty sequence of terminals).

Context-free Languages

- Let C = $\{x\#y \mid x, y \in \{0, 1\}^* \text{ and } x \neq y\}$
  - Design a PDA that accepts C
  - Write a grammar that generates C

Non-deterministic PDA

Start in state 2:

1. Read next input symbol, push 1
2. Non-deterministically go to state 1 or 3
3. If current input is *a*, next state is 4.*a*

4.x Read input symbols until # is read

5.x Read next input, pop

6.x If stack empty, goto 7.x else goto 5.x

7.x Accept if current input is not x, otherwise reject

Grammar

S→RT | R'T'

R→XRX | 1Y#

T→0Y

R'→XR'X | 0Y

T'→1Y

X→0 | 1

Y→YX | ε

Grammar

S→RT | R'T'

R→XRX | 1Y#

T→0Y

R'→XR'X | 0Y

T'→1Y

X→0 | 1

Y→YX | ε

S⇒RT⇒XRXT⇒XX

RXXT⇒* X^nRX^{n}T ⇒

X^n1YX^{n}T⇒

X^n1YX^{n}0Y⇒…

NP-Hard vs NP-Complete

- A problem is NP-hard if all NP problems can be polynomially reduced to it.
- So, the difference between NP-complete and NP-hard is that an NP-complete problem *must be in NP*
  - An NP-hard problem need not be in NP

Example NP-hard problem

- From Sipser Ex 7.33:
  - The problem D = “Does a polynomial p in several variables have integral solutions” is NP-hard.
    - Note: it’s not in NP — in fact, its undecidable
- But we can reduce the known NP-complete problem 3-CNF satisfiability to D

Proof Outline

- Take a formula in 3-CNF and transform it into a Polynomial q as follows:

  variable x → variable x

  ¬ x → (1 – x)

  x ∧ y → xy

  x ∨ y = ¬ (¬ x ∧ ¬ y) → (1 – (1–x)(1–y))

  - So if the 3-CNF formula is satisfiable, the polynomial 1–q has has integral roots.
  - But (1–q) might also have integral roots that do not correspond to a boolean
    - but $(1–q)^2 + (x(1-x))^2 + (y(1-y))^2 + \ldots + (z(1-z))^2$ does not!

λ-calculus

- Recall:

  #0 = λ z . λ s . z
  #1 = λ z . λ s . s z
  #2 = λ z . λ s . s (s z)
  #3 = λ z . λ s . s (s (s z))
  add = λ x . λ y . λ z . λ s . x (y z s) s

- Reduce:

  add #1 #2

- add #1 #2
- (λ x . λ y . λ z . λ s . x (y z s) s) #1 #2
- (λ y . λ z . λ s . #1 (y z s) s) #2
- λ z . λ s . #1 (#2 z s) s
- λ z . λ s . (λ z0 . λ s0 . s0 z0) (#2 z s) s
- λ z . λ s . (λ s0 . s0 (#2 z s)) s
- λ z . λ s . s (#2 z s)
- λ z . λ s . s ((λ z . λ s0 . s0 (s0 z)) z s)
- λ z . λ s . s ((λ s0 . s0 (s0 z)) s)
- λ z . λ s . s (s (s z)

Busy Beavers

- Define: a Turing machine is a “Beaver” if
  - it is deterministic,
  - accepts the empty string,
  - writes only 1s to its tape, and
  - eventually halts
- A “Busy Beaver” writes as many 1s as any other Beaver with the same number of states.
- Let b(n) be the number of 1 that can be written by a Busy Beaver with n states (+ a halt state)

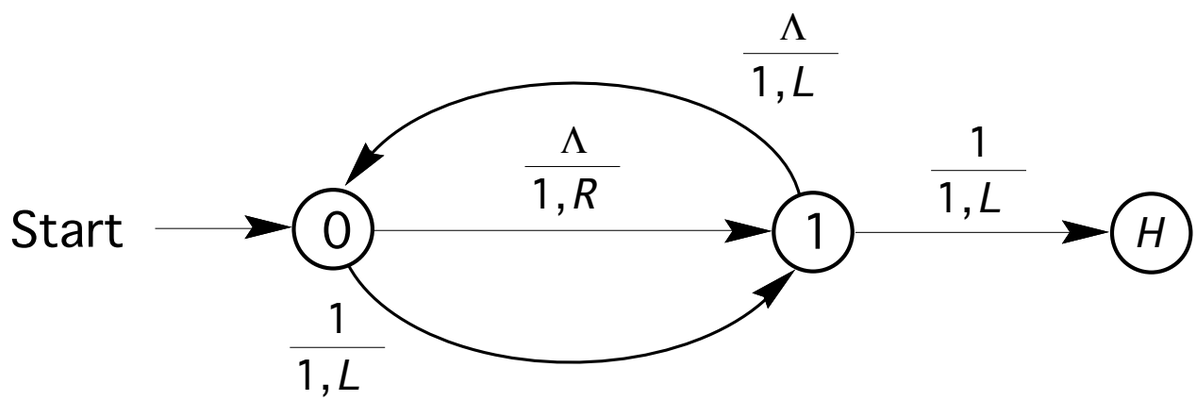
b(n)

b(1) = 1

b(2) = 4

b(3) = 6

b(4) = 13



Busy beaver with 2 states

- These particular values of b have been computed. But we can still ask:
- Is b(n) computable?

Lemma: b(n+1) > b(n)

Lemma: b(n+1) > b(n)

- To Prove: ∀n > 0, b(n+1) > b(n)

Lemma: b(n+1) > b(n)

- To Prove: ∀n > 0, b(n+1) > b(n)
- Proof:

Lemma: b(n+1) > b(n)

- To Prove: $\forall n > 0$, b(n+1) > b(n)
- Proof:
  - let T_n be a busy beaver with n states, n > 0.

Lemma: b(n+1) > b(n)

- To Prove: $\forall n > 0$, b(n+1) > b(n)
- Proof:
  - let T_n be a busy beaver with n states, n > 0.
  - Construct T_{n+1} as follows:

Lemma: b(n+1) > b(n)

- To Prove: $\forall n > 0$, b(n+1) > b(n)
- Proof:
  - let T_n be a busy beaver with n states, n > 0.
  - Construct T_{n+1} as follows:
    - replace the halt state in T_n by a state that skips to the right so long as it reads a 1, and when it finds a ␣, writes a 1 and transfers to the halt state.

Lemma: b(n+1) > b(n)

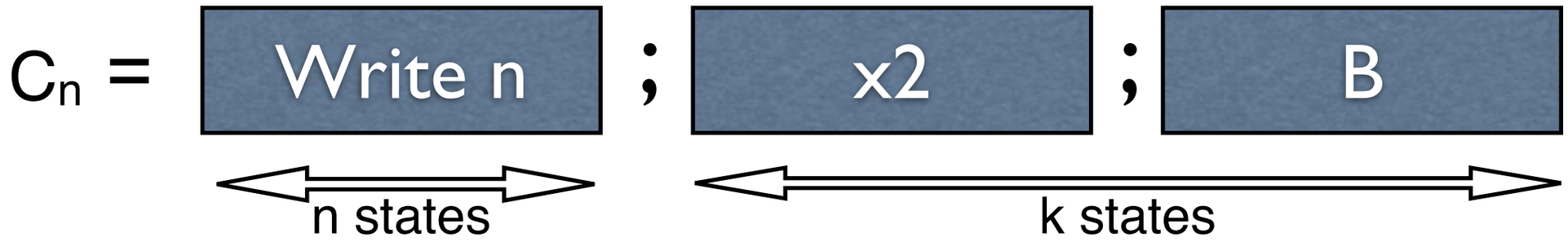
- To Prove: $\forall n > 0$, b(n+1) > b(n)
- Proof:
  - let T_n be a busy beaver with n states, n > 0.
  - Construct T_{n+1} as follows:
    - replace the halt state in T_n by a state that skips to the right so long as it reads a 1, and when it finds a ␣, writes a 1 and transfers to the halt state.
  - Clearly, T_{n+1} has n+1 states, is a beaver, and writes b(n)+1 1s

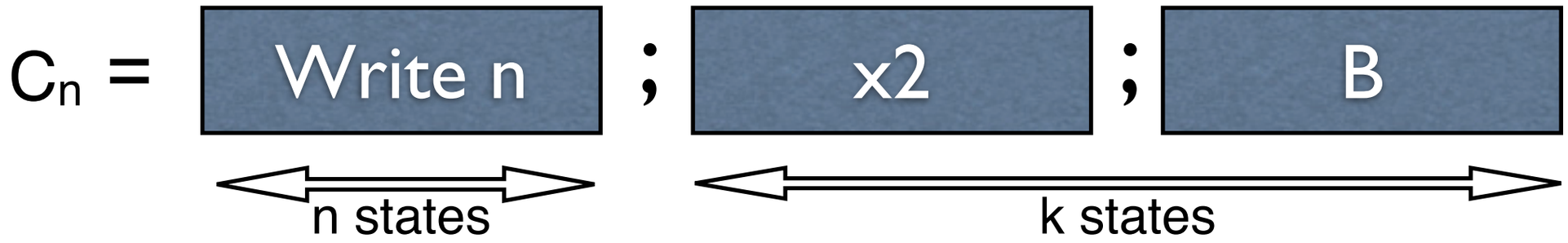
Lemma: b(n+1) > b(n)

- To Prove: $\forall n > 0$, $b(n+1) > b(n)$
- Proof:
  - let T_n be a busy beaver with n states, $n > 0$.
  - Construct T_{n+1} as follows:
    - replace the halt state in T_n by a state that skips to the right so long as it reads a 1, and when it finds a ␣, writes a 1 and transfers to the halt state.
  - Clearly, T_{n+1} has n+1 states, is a beaver, and writes b(n)+1 1s
  - Hence, $b(n+1) \geq b(n)+1 > b(n)$

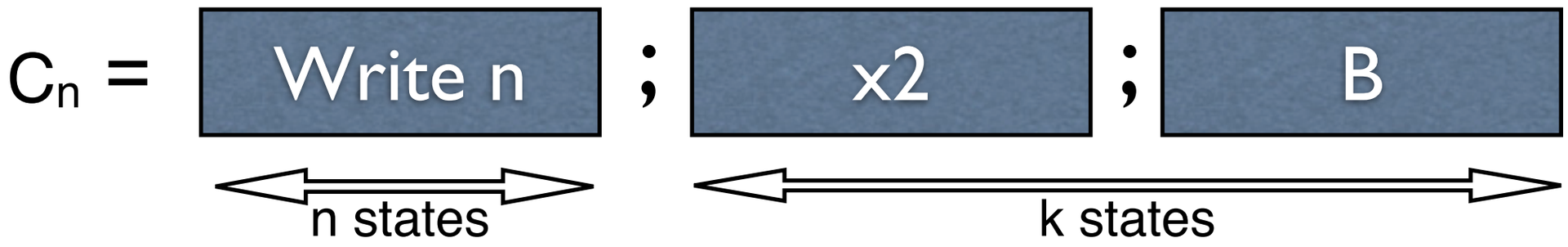
Proof: b(n) is not computable

- Suppose, by way of contradiction, that b(n) is computable.
- Then there is a TM B that computes b(n) in unary, starting with a tape containing n in unary.
  - There is also a TM TwoB that computes b(2n), starting with n on the tape; suppose that TwoB has k states
- Construct a family of TMs C_n with (k+n) states as follows:
  - start with an empty tape
  - uses n states to write n on the tape in unary
  - behaves like TwoB, using k states to compute b(2n)

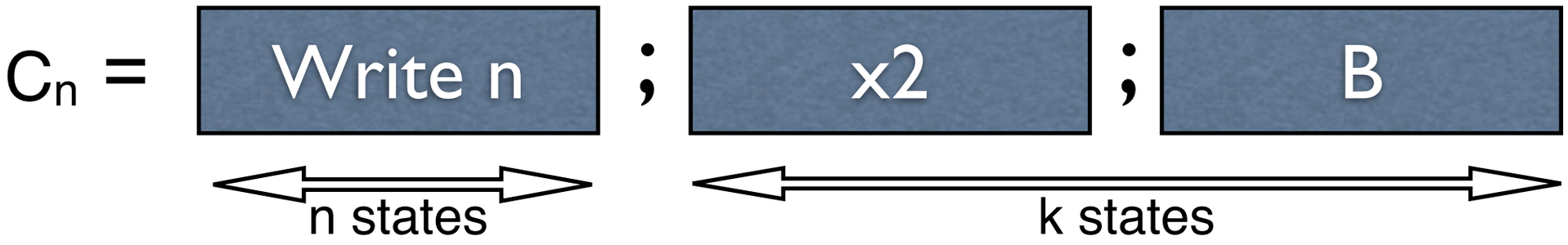

C_n =
Write n
;
x2
;
B
n states
k states



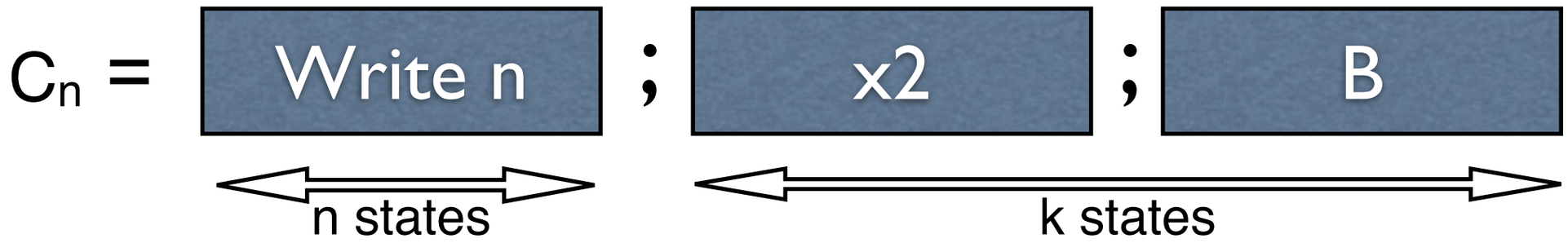
- Note that C_n is a Beaver



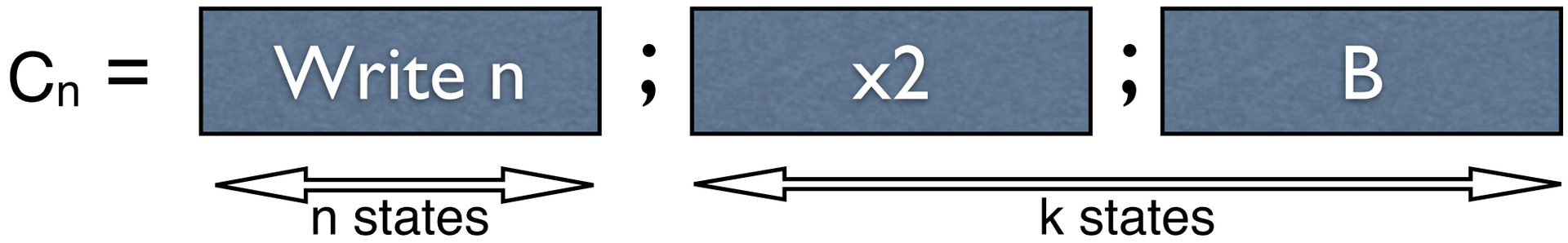
- Note that C_n is a Beaver
  - C_n computes b(2n) and has (k + n) states



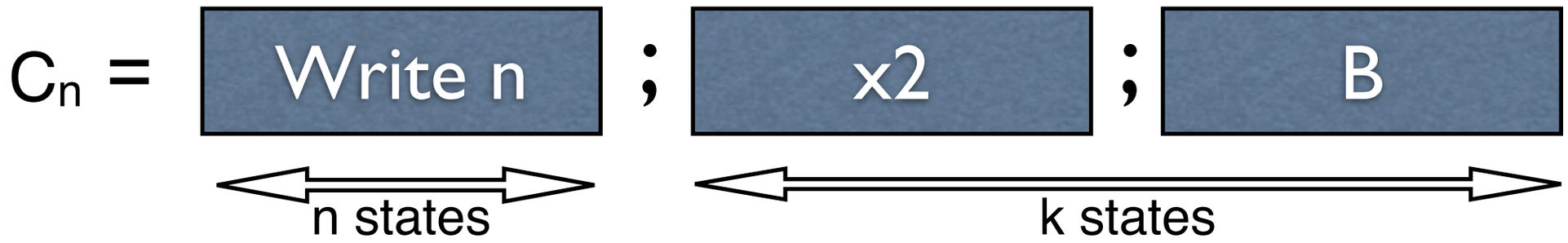
- Note that C_n is a Beaver
  - C_n computes b(2n) and has (k + n) states
  - C_n writes b(2n) 1s and has (k + n) states



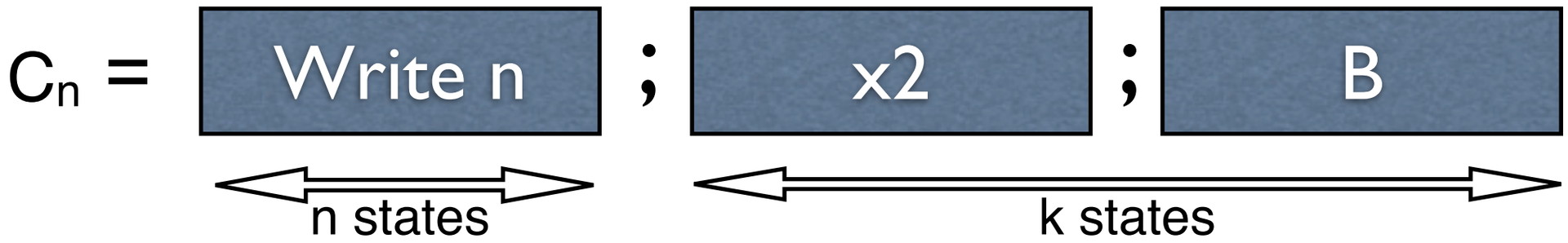
- Note that C_n is a Beaver
  - C_n computes b(2n) and has (k + n) states
  - C_n writes b(2n) 1s and has (k + n) states
  - C_{k+1} writes b(2(k+1)) 1s and has (k + (k+1)) states



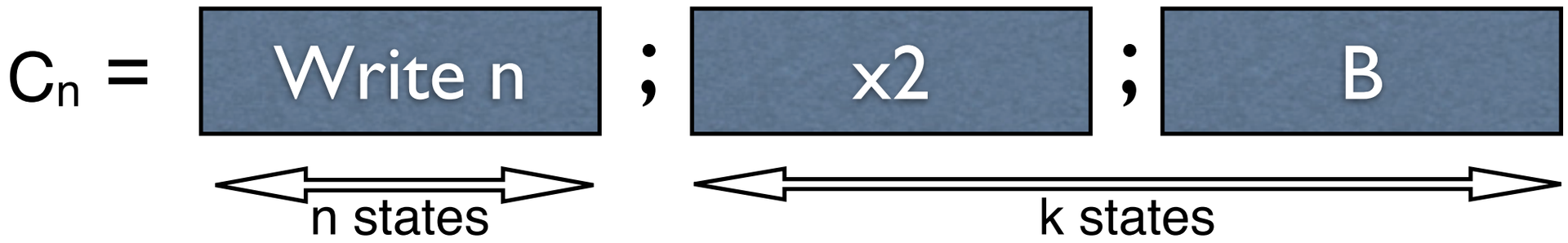
- Note that C_n is a Beaver
  - C_n computes b(2n) and has (k + n) states
  - C_n writes b(2n) 1s and has (k + n) states
  - C_{k+1} writes b(2(k+1)) 1s and has (k + (k+1)) states
  - C_{k+1} writes b(2k+2) 1s and has (2k+1) states



- Note that C_n is a Beaver
  - C_n computes b(2n) and has (k + n) states
  - C_n writes b(2n) 1s and has (k + n) states
  - C_{k+1} writes b(2(k+1)) 1s and has (k + (k+1)) states
    - C_{k+1} writes b(2k+2) 1s and has (2k+1) states
- C_{k+1} is a Beaver with (2k+1) states and writes b(2k+2) 1s



- Note that C_n is a Beaver
  - C_n computes b(2n) and has (k + n) states
  - C_n writes b(2n) 1s and has (k + n) states
  - C_{k+1} writes b(2(k+1)) 1s and has (k + (k+1)) states
    - C_{k+1} writes b(2k+2) 1s and has (2k+1) states
- C_{k+1} is a Beaver with (2k+1) states and writes b(2k+2) 1s
- But a *Busy* Beaver with (2k+1) states can write only b(2k+1) 1s, and b(2k+2) > b(2k+1) by the lemma



- Note that C_n is a Beaver
  - C_n computes b(2n) and has (k + n) states
  - C_n writes b(2n) 1s and has (k + n) states
  - C_{k+1} writes b(2(k+1)) 1s and has (k + (k+1)) states
    - C_{k+1} writes b(2k+2) 1s and has (2k+1) states
- C_{k+1} is a Beaver with (2k+1) states and writes b(2k+2) 1s
- But a *Busy* Beaver with (2k+1) states can write only b(2k+1) 1s, and b(2k+2) > b(2k+1) by the lemma
- So C_{k+1} cannot exist — by definition of "*Busy* Beaver"

Pumping Lemma for Regular languages

Pumping Lemma (Regular Languages) (11.13)

Let L be an infinite regular language over the alphabet A. Then there is an integer $m > 0$ (m is the number of states in a DFA to recognize L) such that for any string $s \in L$ where $|s| \geq m$ there exist strings x, y, $z \in A^*$, where $y \neq \Lambda$, such that $s = xyz$, $|xy| \leq m$ and $xy^k z \in L$ for all $k \geq 0$. The last property tells us that $\{xz, xyz, xy^2z, ..., xy^kz, ...\} \subset L$.

- How did we prove this lemma?

What is a PDA?

- Review the definition of a PDA
- Formal definition was in my Context-free languages lecture (lecture 8)
  - Be clear what happens on each transition!
  - Is the top of the stack “popped”?
  - What symbol(s) are “pushed”?

Pushdown Automata (PDA)

- A pushdown automaton M is defined as a 7-tuple: $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where:
  - Q is a set of states, $q_0 \in Q$ is the start state
  - Σ is the input alphabet,
  - Γ is the stack alphabet, $Z_0 \in \Gamma$ is the initial stack symbol
  - $\delta : (Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon) \rightarrow \mathcal{P}\{Q \times \Gamma^*\}$ is the transition function
  - $F \subseteq Q$ is a set of final states, and
  - $X_\varepsilon = X \cup \{\varepsilon\}$, the set X augmented with ε

Transitions

- We defined $\delta : (Q \times A_\varepsilon \times \Gamma_\varepsilon) \rightarrow \mathcal{P}\{Q \times \Gamma^*\}$
  - The transitions $\delta(q, a, \gamma)$ are applicable iff
    - q is the current state,
    - $a = \varepsilon$, or the next character on the input tape is a , and
    - $\gamma = \varepsilon$, or the top of the stack is γ
  - If you select a transition (q', ω) , then
    - The new state is q'
    - if $\gamma \neq \varepsilon$, γ is popped off of the stack, and
    - the (possibly empty) sequence of symbols ω is pushed onto the stack

Acceptance by Final State

A run of PDA $M = (Q, A, \Gamma, \delta, q_0, \gamma_0, F)$ is a sequence

$$(q_0, \gamma_0) \stackrel{a_0}{\to} (q_1, s_1) \stackrel{a_1}{\to} \cdots \stackrel{a_{n-1}}{\to} (q_n, s_n)$$

with $q_0, \ldots, q_n \in Q$, $s_1, \ldots, s_n \in \Gamma^*$, and $a_0, \ldots, a_{n-1} \in A$ such that:

for all $i \in [0 \,..\, n-1]$. $(q_{i+1}, \gamma_{i+1}) \in \delta(q_i, a_i, \gamma_i)$ and

$s_i = \gamma_i t_i$ and $s_{i+1} = \gamma_{i+1} t_i$ for some $t_i \in \Gamma^*$, and

$w = a_0 a_1 a_2 \ldots a_{n-1}$ is the input.

The run accepts w if $q_n \in F$.

The language of M, $L(M)$ is given by

$$L(M) = \{w \in A^* | \; w \text{ is accepted by some run of } M\}$$

Acceptance by Empty Stack

A run of PDA $M = (Q, A, \Gamma, \delta, q_0, \gamma_0, \emptyset)$ is a sequence

$$(q_0, \gamma_0) \stackrel{a_0}{\to} (q_1, s_1) \stackrel{a_1}{\to} \cdots \stackrel{a_{n-1}}{\to} (q_n, s_n)$$

with $q_0, \ldots, q_n \in Q$, $s_1, \ldots, s_n \in \Gamma^*$, and $a_0, \ldots, a_{n-1} \in A$ such that:

for all $i \in [0 \,..\, n-1]$. $(q_{i+1}, \gamma_{i+1}) \in \delta(q_i, a_i, \gamma_i)$ and

$s_i = \gamma_i t_i$ and $s_{i+1} = \gamma_{i+1} t_i$ for some $t_i \in \Gamma^*$, and

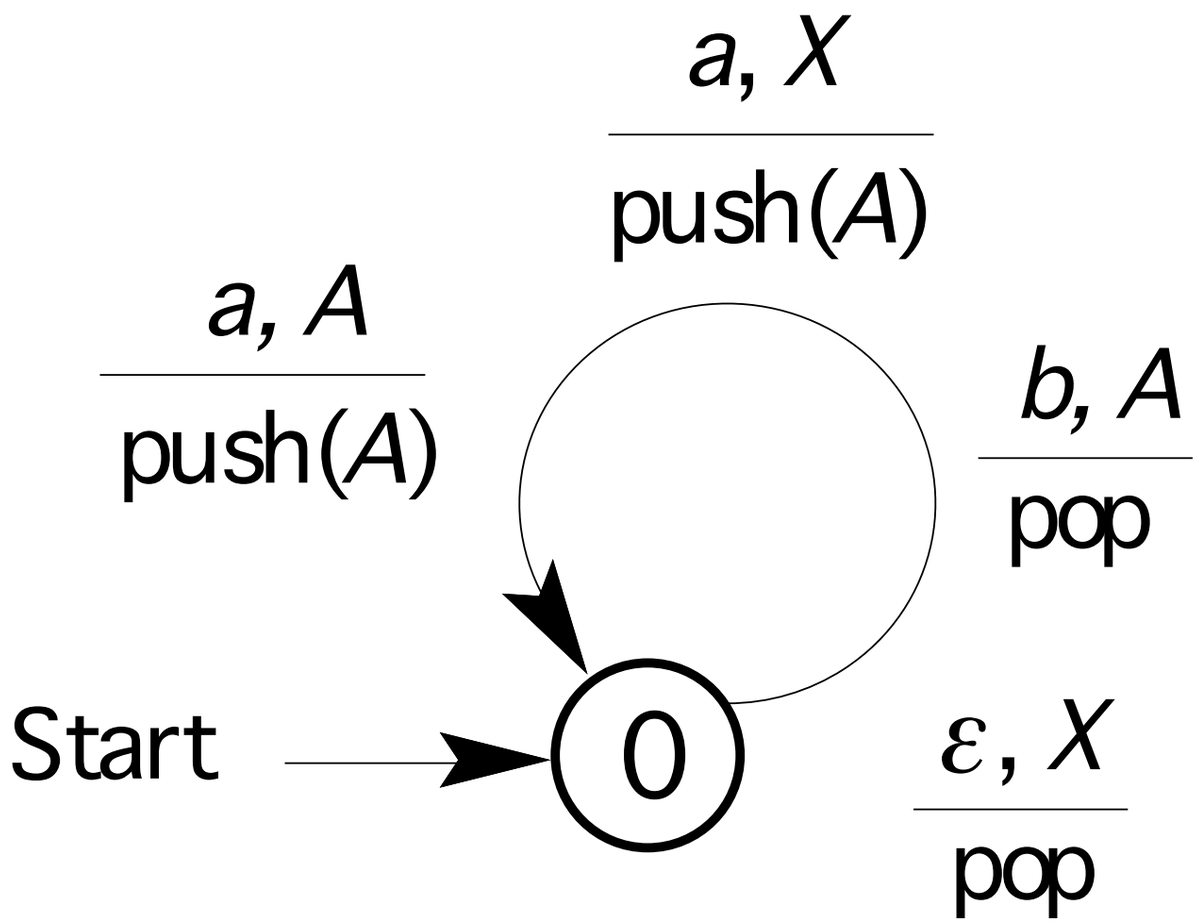
$w = a_0 a_1 a_2 \ldots a_{n-1}$ is the input.

The run accepts w if $s_n = \varepsilon$.

The language of M, $L(M)$ is given by

$$L(M) = \{w \in A^* | \; w \text{ is accepted by some run of } M\}$$

Problem: what language is accepted by this PDA?

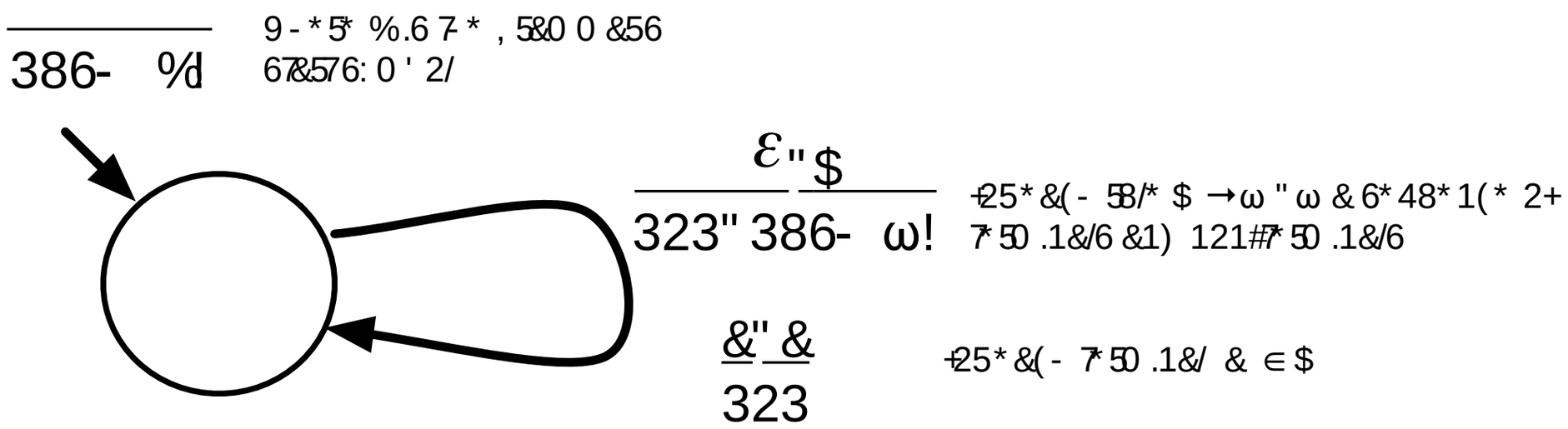


- Assume that *X* is initially on the stack

CFG to PDA Construction

- It's easy to build a PDA give a Context-free grammar:
  - The PDA has one state; label it 0
  - The alphabet A consist of the terminal symbols of the grammar
  - The stack alphabet Γ consists of {non-terminals of the grammar} $\cup$ A
  - The initial symbol on the stack is the start symbol

- The PDA's transitions are as follows:
  - For each terminal symbol a, define the transition $\delta(0, a, a) = (\varepsilon, 0)$
  - For each production A → ω, where ω is a (possibly empty) sequence of terminals and non-terminals, define the transition $\delta(0, \varepsilon, A) = (\omega, 0)$
- Key idea: each transition in the PDA corresponds to a derivation step in the grammar



Try this example:

- Simple arithmetic expressions

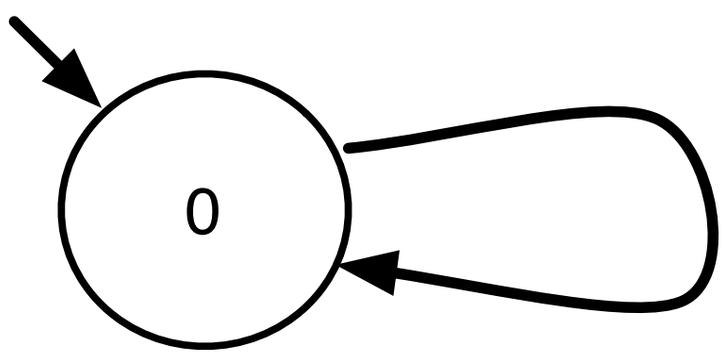
$$E \rightarrow V \mid V + E$$
$$V \rightarrow a \mid 1$$

for each rule $A \rightarrow \omega$, ω a sequence of terminals and non-terminals

$$\frac{}{\text{push (E)}}$$ because E is the grammar's start symbol

$$\frac{\varepsilon,\ E}{\text{pop, push(V + E)}} \qquad \frac{\varepsilon,\ V}{\text{pop, push(a)}}$$

$$\frac{\varepsilon,\ E}{\text{pop, push(V)}} \qquad \frac{\varepsilon,\ V}{\text{pop, push(1)}}$$



$$\frac{+,\ +}{\text{pop}} \qquad \frac{1,\ 1}{\text{pop}} \qquad \frac{a,\ a}{\text{pop}}$$

for each terminal $\in$ A

Proving things Uncomputable

- Is there an effective enumeration of the total functions $\mathbb{N} \to \mathbb{N}$?
- Is there an algorithm to decide if an arbitrary computable function $\mathbb{N} \to \mathbb{N}$ is total?

Homework 9, Problem 3

3. Suppose we have the following effective enumeration of all the computable functions that take a single argument:

$$f_0, f_1, f_2, \ldots, f_n, \ldots.$$

For each of the following functions g, explain what is *wrong* with the following diagonalization argument claiming to show that g is a computable function that isn't in the list. "Since the enumeration is effective, there is an algorithm to transform each n into the function f_n. Since each f_n is computable, it follows that g is computable. It is easy to see that g is not in the list. Therefore g is a computable function that isn't in the list."

a. $g(n) = f_n(n) + 1$.

b. $g(n) =$ if $f_n(n) = 4$ then 3 else 4.

c. $g(n) =$ if $f_n(n)$ halts and $f_n(n) = 4$ then 3 else 4.

d. $g(n) =$ if $f_n(n)$ halts and $f_n(n) = 4$ then 3 else loop forever.

Turing Machine Construction

- Hein §13.1 Ex 6

6. Construct a Turing machine to test for equality of two strings over the alphabet $\{a, b\}$, where the strings are separated by a cell containing #. Output a 0 if the strings are not equal and a 1 if they are equal.

Grammar Transformations

Grammar Transformations

- What is Chomsky Normal Form?

Grammar Transformations

- What is Chomsky Normal Form?
  - Productions have the form A→a or A→BC.

Grammar Transformations

- What is Chomsky Normal Form?
  - Productions have the form A→a or A→BC.
  - If the language contains ε, then A→ε is also allowed if A does not appear on the rhs of any production.

Transforming a Grammar to Chomsky Normal Form

1. If there is a production $A \to \Lambda$, where A is not the start symbol S, then use the preceding algorithm to remove all productions that contain Λ. If this process removes $S \to \Lambda$, then add it back.

2. This step removes all *unit* productions $A \to B$, where A and B are nonterminals. For each pair of nonterminals A and B, if $A \to B$ is a unit production or if there is a derivation $A \Rightarrow^+ B$, then add all productions of the form $A \to w$, where $B \to w$ is not a unit production. Now remove all the unit productions.

3. For each production whose right side has two or more symbols, replace all occurrences of each terminal a with a new nonterminal A, and also add the new production $A \to a$.

4. For each production of the form $B \to C_1 C_2 \ldots C_n$, where $n > 2$, replace it with the following two productions, where D is a new nonterminal:

$$B \to C_1 D \quad \text{and} \quad D \to C_2 \ldots C_n.$$

Continue this step until all productions with nonterminal strings on the right side have length 2.

Algorithm to Remove Lambda Productions

1. Find the set of all nonterminals N such that N derives Λ.
2. For each production of the form $A \rightarrow w$, create all possible productions of the form $A \rightarrow w'$, where w' is obtained from w by removing one or more occurrences of the nonterminals found in Step 1.
3. The desired grammar consists of the original productions together with the productions constructed in Step 2, minus any productions of the form $A \rightarrow \Lambda$.

Example Problem

- Hein §12.4 Ex 2

2. Find a Chomsky normal form for each of the following grammars.

a. $S \to aSa \mid bSb \mid c.$

b. $S \to abC \mid babS \mid de$
$C \to aCa \mid b.$

c. $S \to aSa \mid R$
$R \to S \mid b.$

Properties of CFLs

Properties of Context-Free Languages (12.22)

1. The union of two context-free languages is context-free.
2. The language product of two context-free languages is context-free.
3. The closure of a context-free language is context-free.
4. The intersection of a regular language with a context-free language is context-free.

Context-Free Language Morphisms (12.23)

Let $f : A^* \to A^*$ be a language morphism. In other words, $f(\varepsilon) = \varepsilon$ and $f(uv) = f(u)f(v)$ for all strings u and v. Let L be a language over A.

1. If L is context-free, then $f(L)$ is context-free.
2. If L is context-free, then $f^{-1}(L)$ is context-free.

Example Problem

- Hein §12.4 Ex 5:

5. Show that the language $\{a^n b^n a^n \mid n \in \mathrm{N}\}$ is not context-free by performing the following tasks:

a. Given the morphism $f : \{a, b, c\}^* \to \{a, b, c\}^*$ defined by $f(a) = a$, $f(b) = b$, and $f(c) = a$, describe $f^{-1}(\{a^n b^n a^n \mid n \in \mathrm{N}\})$.

b. Show that

$$f^{-1}(\{a^n b^n a^n \mid n \in \mathrm{N}\}) \cap \{a^k b^m c^n \mid k, m, n \in \mathrm{N}\} = \{a^n b^n c^n \mid n \in \mathrm{N}\}.$$

c. Argue that $\{a^n b^n a^n \mid n \in \mathrm{N}\}$ is not context-free by using parts (a) and (b) together with (12.22) and (12.23).

Pumping Lemma for Context-free languages

- Hein §12.4.2

Pumping Lemma for Context-Free Languages (12.19)

Let L be an infinite context-free language. Then there is a positive integer m such that for all strings $z \in L$ with $|z| \geq m$, z can be written in the form $z = uvwxy$, where the following properties hold:

$$|vx| \geq 1,$$

$$|vwx| \leq m,$$

$$uv^k wx^k y \in L \text{ for all } k \geq 0.$$

  - m is called the pumping length for L

Pumping Lemma

Pumping Lemma

- Show that L = $\{a^n b^n a^n \mid n \geq 0\}$ is not Context-free using the pumping lemma.

Pumping Lemma

- Show that L = $\{a^n b^n a^n \mid n \geq 0\}$ is not Context-free using the pumping lemma.
- Let's suppose that L is CF
  - Then by the pumping lemma, ∃ $z = a^m b^m a^m$ in L where m is the pumping length and $z = uvwxy$ where

    | vx | ≥ 1

    | vwx | ≤ m

    $uv^k wx^k y \in$ L for all $k \geq 0$

- What can vwx be?