有限オートマトンで認識できない言語が存在する

より強力な計算モデルが必要

- **プッシュダウンオートマトン**
- **チューリングマシン**

有限オートマトンで認識できない言語が存在する

⇓

有限オートマトンより強力な計算モデル

⇓

正規言語より広い範囲の言語を扱う

⇓

生成規則による言語の記述（生成文法）

生成規則・生成文法

生成規則を与えることでも
言語を定めることが出来る

$\longrightarrow$ **生成文法** (**generative grammar**)

生成規則による “文法に適っている” 語の生成

- 初期変数を書く
- 今ある文字列中の或る変数を
生成規則のどれかで書換える
- 変数がなくなったら終わり

例： $\{a^{2n}b^{2m+1} | n, m \geq 0\}$

（a が偶数個（0 個も可）続いた後に、b が奇数個続く）

正規表現で表すと、 $(aa)^*b(bb)^*$

- $S \rightarrow aaS$
- $S \rightarrow bB$
- $B \rightarrow bbB$
- $B \rightarrow \varepsilon$

まとめて次のようにも書く

- $S \rightarrow aaS|bB$
- $B \rightarrow bbB|\varepsilon$

生成規則・生成文法

実際の（自然言語を含めた）“文法”では、
或る特定の状況で現われた場合だけ
適用できる規則もあるだろう

そのような生成規則は例えば次の形：

- $u\mathit{A}v \to uwv$

$u, v \in \Sigma^*$：文脈 (context)

変数 A が uAv の形で現われたら、
語 $w \in \Sigma^*$ で書換えることが出来る

生成規則・生成文法

実際の（自然言語を含めた）“文法” では、
或る特定の状況で現われた場合だけ
適用できる規則もあるだろう

そのような生成規則は例えば次の形：

- $u A v \to u w v$

$u, v \in \Sigma^*$：**文脈 (context)**

変数 A が uAv の形で現われたら、
語 $w \in \Sigma^*$ で書換えることが出来る

生成文法の形式的定義

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

- V：有限集合（**変数**の集合）
- Σ：有限集合（**終端記号**の集合）
  ここに $V \cap \Sigma = \emptyset$
- R：有限集合 $\subset (V \cup \Sigma)^* \times (V \cup \Sigma)^*$
  （**規則**の集合）
- $S \in V$：**開始変数**

$(v, w) \in R$ が生成規則 $v \to w$ を表す

文脈自由文法 (context-free grammar)

文脈が全て空列 ε

即ち、規則が全て $A \to w$ $(A \in V)$ の形

文脈自由文法の形式的定義

- V：有限集合（変数の集合）
- Σ：有限集合（終端記号の集合）
  ここに $V \cap \Sigma = \emptyset$
- R：有限集合 $\subset V \times (V \cup \Sigma)^*$ （規則の集合）
- $S \in V$：開始変数

$(A, w) \in R$ が生成規則 $A \to w$ を表す

文脈自由文法 (context-free grammar)

文脈が全て空列 ε

即ち、規則が全て $A \to w$ $(A \in V)$ **の形**

文脈自由文法の形式的定義

- V : **有限集合（変数の集合）**
- Σ : **有限集合（終端記号の集合）**
  ここに $V \cap \Sigma = \emptyset$
- R : **有限集合** $\subset V \times (V \cup \Sigma)^*$ **（規則の集合）**
- $S \in V$: **開始変数**

$(A, w) \in R$ **が生成規則** $A \to w$ **を表す**

例：言語 $A = \{a^n b^n | n \geq 0\}$ は 正規言語ではないが文脈自由言語である

- $S \to aSb|\varepsilon$

従って、

文脈自由言語は正規言語より真に広い!!

さて、正規言語を計算するモデルが
有限オートマトンであった

文脈自由言語を計算するモデル
$\cdots$ プッシュダウンオートマトン

例：言語 $A = \{a^n b^n | n \geq 0\}$ は
正規言語ではないが文脈自由言語である

- $S \to aSb|\varepsilon$

従って、

文脈自由言語は正規言語より真に広い!!

さて、正規言語を計算するモデルが
有限オートマトンであった

文脈自由言語を計算するモデル
$\cdots$ プッシュダウンオートマトン

例：言語 $A = \{a^n b^n | n \geq 0\}$ は
正規言語ではないが文脈自由言語である

- $S \to aSb|\varepsilon$

従って、

文脈自由言語は正規言語より真に広い!!

さて、正規言語を計算するモデルが
有限オートマトンであった

文脈自由言語を計算するモデル
$\cdots$ プッシュダウンオートマトン

例：言語 $A = \{a^n b^n | n \geq 0\}$ は
正規言語ではないが文脈自由言語である

- $S \to aSb|\varepsilon$

従って、

文脈自由言語は正規言語より真に広い!!

さて、正規言語を計算するモデルが
有限オートマトンであった

文脈自由言語を計算するモデル
… プッシュダウンオートマトン

例：言語 $A = \{a^n b^n | n \geq 0\}$ は
正規言語ではないが文脈自由言語である

- $S \to aSb|\varepsilon$

従って、

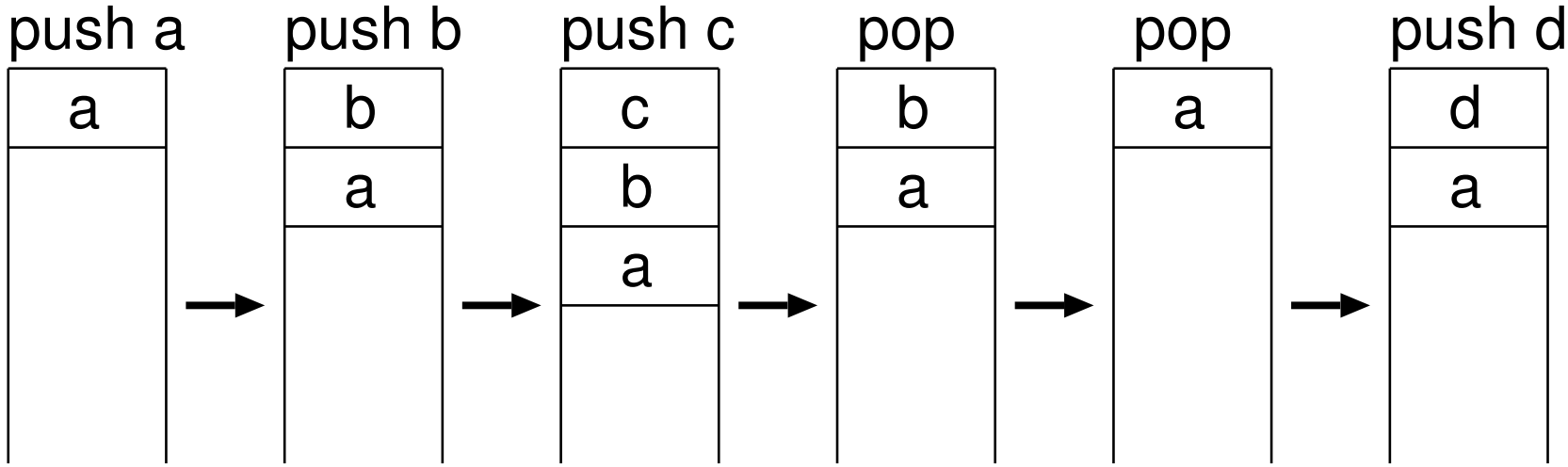
文脈自由言語は正規言語より真に広い!!

さて、正規言語を計算するモデルが
有限オートマトンであった

文脈自由言語を計算するモデル
$\cdots$ プッシュダウンオートマトン

プッシュダウンオートマトン

（非決定性）有限オートマトンに
プッシュダウンスタックを取り付けたもの



無限（非有界）の情報を保持できるが、
読み書きは先頭だけ
$\cdots$ LIFO (Last In First Out)

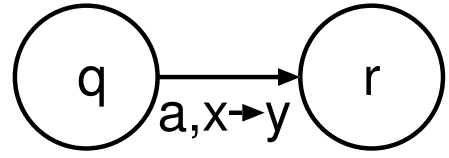
プッシュダウンオートマトンの形式的定義

$$M = (Q, \Sigma, \Gamma, \delta, s, F)$$

- Q：**有限集合** $\cdots$ **状態の集合**
- Σ：**有限集合** $\cdots$ **alphabet**
- Γ：**有限集合** $\cdots$ **stack alphabet**
  $\Sigma_\varepsilon := \Sigma \cup \{\varepsilon\}$, $\Gamma_\varepsilon := \Gamma \cup \{\varepsilon\}$ **と置く**
- $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$
  ：遷移関数（非決定的） $\cdots$ **可能な遷移先全体**
- $s \in Q$ $\cdots$ **初期状態**
- $F \subset Q$ $\cdots$ **受理状態の集合**

$$\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$$

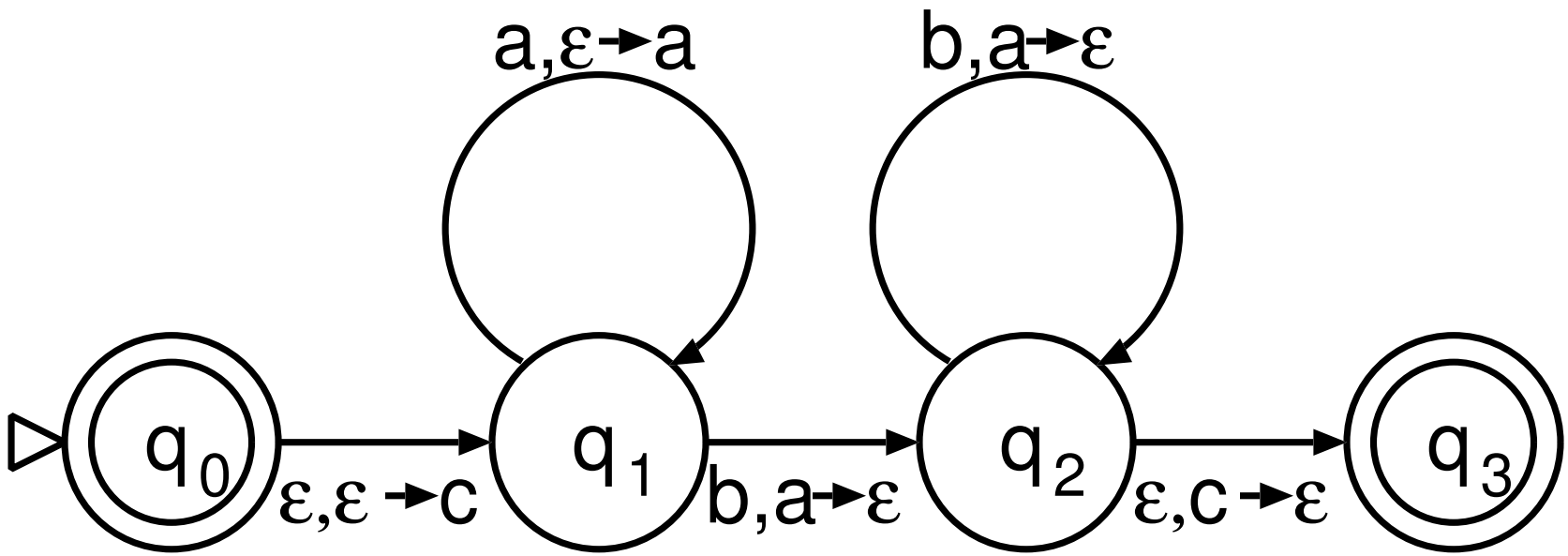
- $(r, y) \in \delta(q, a, x)$ **とは、**
  「入力 a **を読んだとき、**
  状態 q **でスタックの先頭が** x **なら、**
  スタックの先頭を y **に書換えて、**
  状態 r **に移って良い」**
  ということ $(\mathtt{pop};\ \mathtt{push}\ y)$

  状態遷移図では 

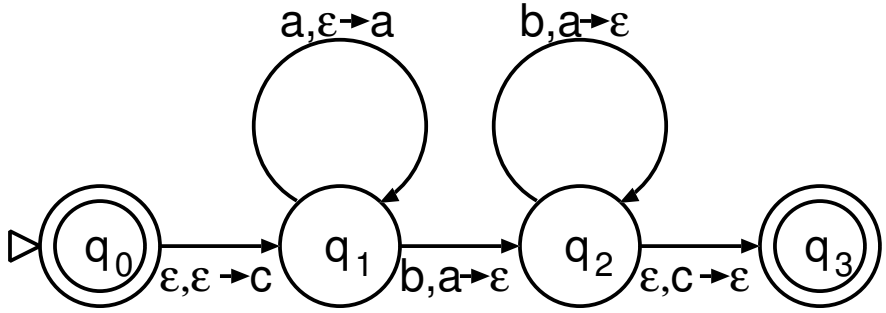
  で表す
- $x = y$ **は書き換え無し**
- $x = \varepsilon$ **は（スタックの先頭を見ずに）push のみ**
- $y = \varepsilon$ **は pop のみ**
- $a = \varepsilon$ **は入力を読まずに遷移**

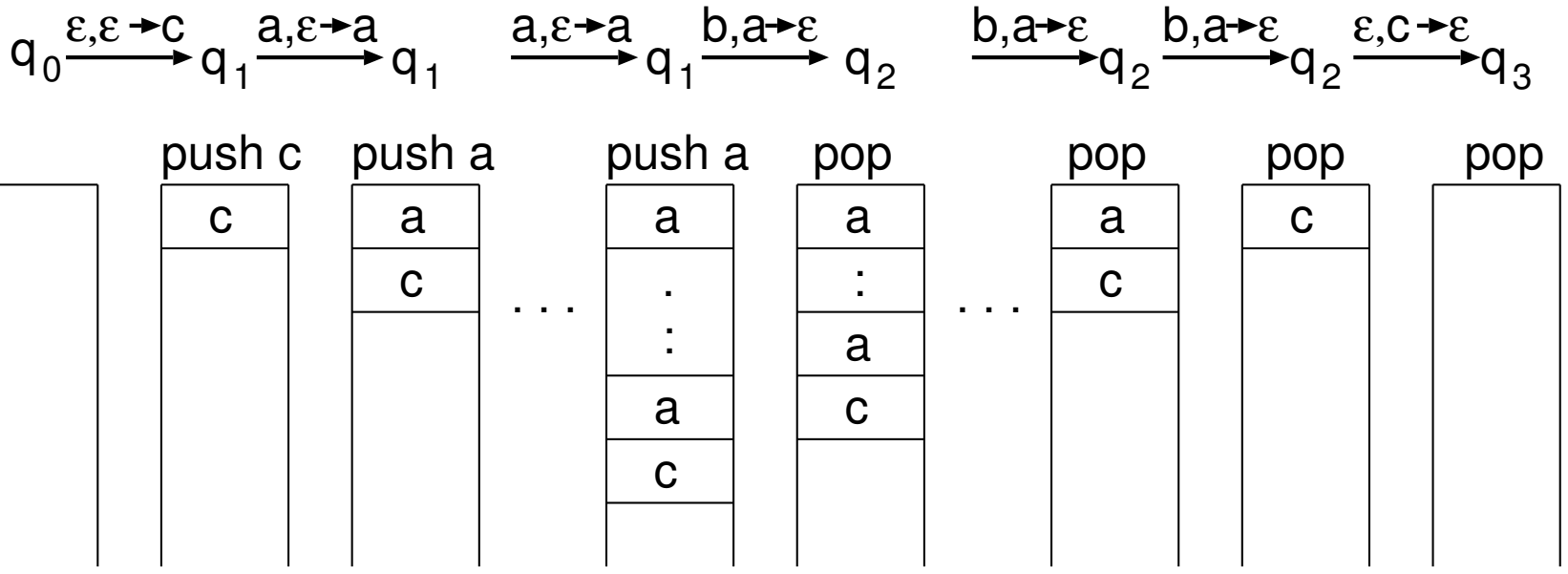
例：言語 $A = \{a^n b^n | n \geq 0\}$ を認識する プッシュダウンオートマトン

$$\Sigma = \{a, b\}, \qquad \Gamma = \{a, b, c\}$$



PDA による 文字列 $a^n b^n$ の受理





スタックマシン

このように

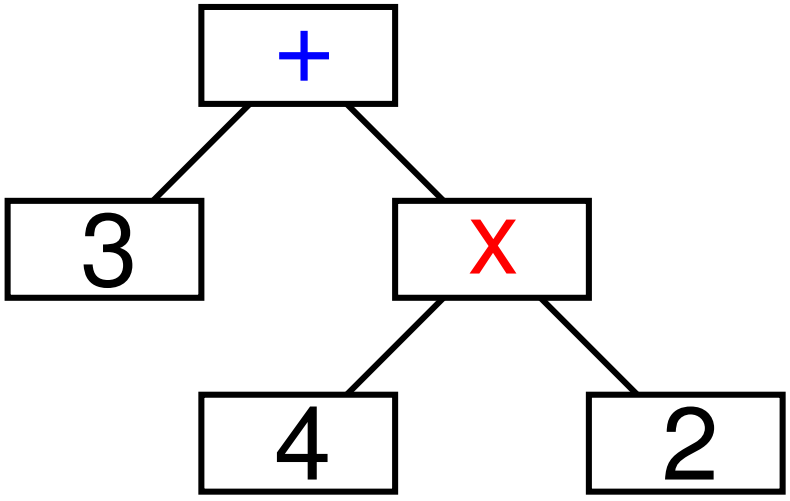
記憶場所としてプッシュダウンスタックを備えた
計算モデルや仮想機械・処理系を

一般にスタックマシンという

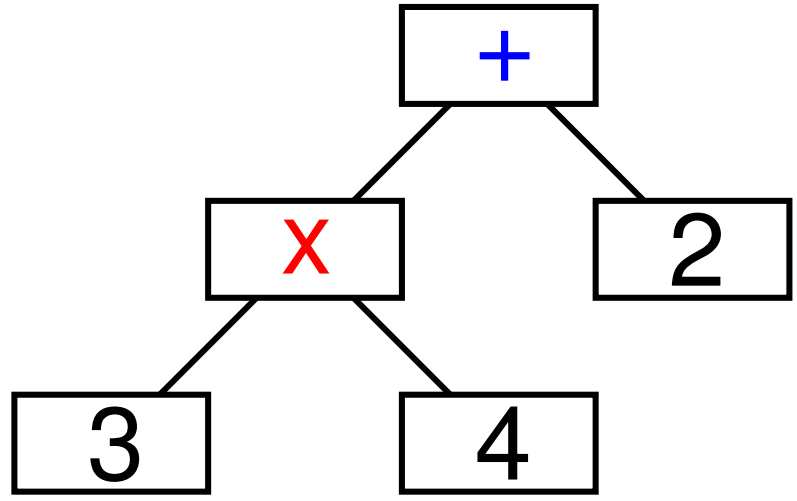
例：

- 逆ポーランド電卓
- PostScript

式と演算木



3+4x2



3x4+2

Mathematica などの
数式処理（計算機代数）ソフトウェアでは、

通常、内部的に数式の木構造を保持

演算木の表記

演算子を置く場所により、中置・前置・後置がある

中置	前置	後置
$3 + 4 \times 2$	$+\ 3\ \times\ 4\ 2$	$3\ 4\ 2\ \times\ +$
$(3 + 4) \times 2$	$\times\ +\ 3\ 4\ 2$	$3\ 4\ +\ 2\ \times$
$3 \times 4 + 2$	$+\ \times\ 3\ 4\ 2$	$3\ 4\ \times\ 2\ +$