有限オートマトンで認識できない言語が存在する

$\Downarrow$

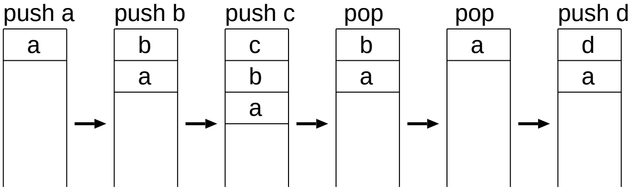
より強力な計算モデルが必要

$\Downarrow$

- **プッシュダウンオートマトン**
- **チューリングマシン**

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)**

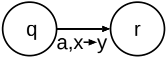
プッシュダウンオートマトンの形式的定義

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

- 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 (`pop; 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**

スタックマシン

このように

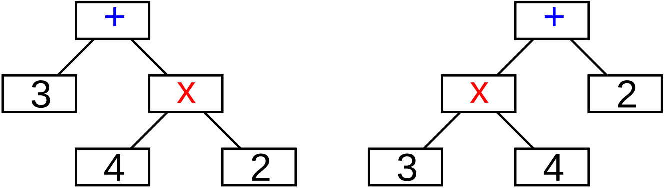
記憶場所としてプッシュダウンスタックを備えた計算モデルや仮想機械・処理系を

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
$3+4\times 2$	$+\ 3\ \times\ 4\ 2$	$3\ 4\ 2\ \times\ +$
$(3+4)\times 2$	$\times\ +\ 3\ 4\ 2$	$3\ 4\ +\ 2\ \times$
$3\times 4+2$	$+\ \times\ 3\ 4\ 2$	$3\ 4\ \times\ 2\ +$

後置記法（逆ポーランド記法）

後置	日本語
$3\ 4\ 2\ \times\ +$	3 に 4 に 2 を**掛**けたものを**足**したもの
$3\ 4\ +\ 2\ \times$	3 に 4 を**足**したものに 2 を**掛**けたもの
$3\ 4\ \times\ 2\ +$	3 に 4 を**掛**けたものに 2 を**足**したもの

$\Downarrow$

スタックを用いた計算に便利

後置記法の演算式のスタックを用いた計算（逆ポーランド電卓）

- 数値 $\Longrightarrow$ push
- 演算子 $\Longrightarrow$ 被演算子を（所定の個数だけ）pop $\longrightarrow$ 演算を施し、結果を push
- 入力終了 $\Longrightarrow$ pop $\longrightarrow$ スタックが丁度空になったらその値が答え

問：後置記法（逆ポーランド記法）の式に対しスタックを用いて値を計算するアルゴリズムを実装せよ

後置記法の有利性

後置記法の演算式が簡明に計算できるのは、

（各演算子に対して
被演算子の個数が決まっていれば）

括弧が必要ない（優先順位を考慮しなくてよい）

ことが大きく効いている

- 式 :: 定数 ‖ 変数 ‖ 式 式 二項演算子
（$+$ も $\times$ も区別なし）

中置記法と演算子の優先順位

中置記法の演算式には括弧が必要
（演算子の優先順位を定めておく必要あり）

$$3 \times 4 + 2 \qquad\qquad 3 + 4 \times 2$$

計算する際には優先順位を考慮する必要がある

- **式** $::$ **項** $\|$ **項** $+$ **式**
- **項** $::$ **因子** $\|$ **因子** $\times$ **項**
- **因子** $::$ **定数** $\|$ **変数** $\|$ $($**式**$)$

（$+$ と $\times$ とで純然たる区別あり）

スタックマシンの例：PostScript

ページ記述言語の一つ

- **Adobe Systems が開発**
- **PDF (Portable Document Format) の元になった言語**
- **レーザプリンタなどで実装**
- **オープンソースなインタプリタとして Ghostscript が良く利用されている**

- **図形を描いたりフォントを置いたりする**
- **逆ポーランド記法**

スタックマシンの例：PostScript

逆ポーランド記法

- データを push
- 命令（演算子, operator）が
  所定数のデータ（被演算子, operand）を
  pop して処理

例：$(100, 200)$ **から** $(300 + 50, 400)$ **へ、**
引続き $(200, 600 - 50)$ **へ線を引く**

```
100 200 moveto
300 50 add 400 lineto
200 600 50 sub lineto
stroke
```

定理：

L：**正規言語**

$\Updownarrow$

L **が或る有限オートマトンで認識される**

定理：

L：**文脈自由言語**

$\Updownarrow$

L **が或るプッシュダウンオートマトンで認識される**

本質的な違いは？

文脈自由言語は再帰 (recursion) を記述できる

定理：

L：**正規言語**

$\Updownarrow$

L **が或る有限オートマトンで認識される**

定理：

L：**文脈自由言語**

$\Updownarrow$

L **が或るプッシュダウンオートマトンで認識される**

本質的な違いは？

文脈自由言語は再帰 (recursion) を記述できる

文脈自由言語と再帰

- $S \longrightarrow aSb|\varepsilon$

```
S(){
  either
    "";
  or
    { "a"; S() ; "b"; }
}

main(){
  S();
}
```

再帰：関数 S() **の中で、自分自身を呼び出す**

計算機での関数呼出・再帰の実現

関数呼出は原理的には次の仕組みで行なっている

- **現在の実行番地（戻る場所）を覚えておく**
- **関数を実行する**
- **関数を実行し終えたら、
覚えていた実行番地に戻って呼出側の実行再開**

再帰呼出では呼出す度に覚えておく番地が増える

**$\longrightarrow$ スタックに積んで覚えておく
（関数呼出の際に番地を push、戻ったら pop）**

計算機での関数呼出・再帰の実現

関数呼出は原理的には次の仕組みで行なっている

- 現在の実行番地（戻る場所）を覚えておく
- 関数を実行する
- 関数を実行し終えたら、
  覚えていた実行番地に戻って呼出側の実行再開

再帰呼出では呼出す度に覚えておく番地が増える

$\longrightarrow$ スタックに積んで覚えておく
（関数呼出の際に番地を push、戻ったら pop）

計算機での関数呼出・再帰の実現

関数呼出は原理的には次の仕組みで行なっている

- 現在の実行番地（戻る場所）を覚えておく
- 関数を実行する
- 関数を実行し終えたら、
  覚えていた実行番地に戻って呼出側の実行再開

再帰呼出では呼出す度に覚えておく番地が増える

$\longrightarrow$ スタックに積んで覚えておく
（関数呼出の際に番地を push、戻ったら pop）

正規言語における再帰

正規表現：$(aa)^*$

- $S \to aaS|\varepsilon$

```
S(){
  either
    "";
  or
    { "aa"; S() ; }
}

main(){
  S();
}
```

$\longrightarrow$ **末尾再帰の除去**

```
main(){
  loop {
    "aa";
  }
}
```

繰返しで記述可能
（再帰は不要）

正規言語における再帰

正規表現：$(aa)^*$

- $S \to aaS | \varepsilon$

```
S(){
  either
    "";
  or
    { "aa"; S() ; }
}

main(){
  S();
}
```

$\longrightarrow$ **末尾再帰の除去**

```
main(){
  loop {
    "aa";
  }
}
```

繰返しで記述可能
（再帰は不要）

正規言語・文脈自由言語と再帰

- **正規言語は繰返しを記述できる**
- **文脈自由言語は再帰を記述できる**
- **再帰の実装にはスタックを要す**
- **文脈自由言語の生成規則は次の形に出来る**
  - $\star$ $X \longrightarrow YZ \ (X, Y, Z \in V)$
  - $\star$ $X \longrightarrow x \ (X \in V, x \in \Sigma_\varepsilon)$

  （Chomsky の標準形）
- **正規言語の生成規則は次の形に出来る**
  - $\star$ $X \longrightarrow xY \ (X, Y \in V, x \in \Sigma)$
  - $\star$ $X \longrightarrow x \ (X \in V, x \in \Sigma_\varepsilon)$

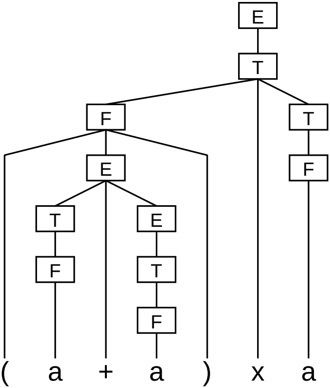
  特に、末尾再帰であり再帰の除去可能

構文解析木

生成規則の適用過程を木で表したもの

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

- $V = \{E, T, F\}$
- $\Sigma = \{a, +, \times, \ (,)\ \}$
- R :
  - $\star\ E \longrightarrow T \mid T + E$
  - $\star\ T \longrightarrow F \mid F \times T$
  - $\star\ F \longrightarrow a \mid (E)$



文脈自由言語の Pumping Lemma

文脈自由言語 A に対し、

$\exists n \in \mathbb{N}:$

$\forall w \in A, |w| \geq n:$

$\exists u, v, x, y, z \in \Sigma^* : w = uvxyz$

(1) $vy \neq \varepsilon$ **(即ち** $v \neq \varepsilon$ **または** $y \neq \varepsilon$**)**
(2) $|vxy| \leq n$
(3) $\forall k \geq 0 : uv^k xy^k z \in A$

文脈自由言語の例

回文全体の成す言語は文脈自由

- $S \rightarrow aSa|bSb|a|b|\varepsilon$

問：回文全体の成す言語を認識する
プッシュダウンオートマトンを構成せよ

文脈自由言語の例

回文全体の成す言語は文脈自由

- $S \to aSa \mid bSb \mid a \mid b \mid \varepsilon$

問：回文全体の成す言語を認識する
プッシュダウンオートマトンを構成せよ

文脈自由言語の例

回文全体の成す言語は文脈自由

- $S \rightarrow aSa|bSb|a|b|\varepsilon$

問：回文全体の成す言語を認識する
プッシュダウンオートマトンを構成せよ

プッシュダウンオートマトンでは認識できない言語の例

同じ文字列 2 回の繰返しから成る文字列全体

$$A = \{ww | w \in \Sigma^*\}$$

入力を読み直せないのが弱点

$\longrightarrow$ **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では認識できない言語の例

同じ文字列 2 回の繰返しから成る文字列全体

$$A = \{ww | w \in \Sigma^*\}$$

入力を読み直せないのが弱点

$\longrightarrow$ 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では認識できない言語の例

同じ文字列 2 回の繰返しから成る文字列全体

$$A = \{ww | w \in \Sigma^*\}$$

入力を読み直せないのが弱点

$\longrightarrow$ **より強力な計算モデルが必要**