

# コンパイラ及び演習

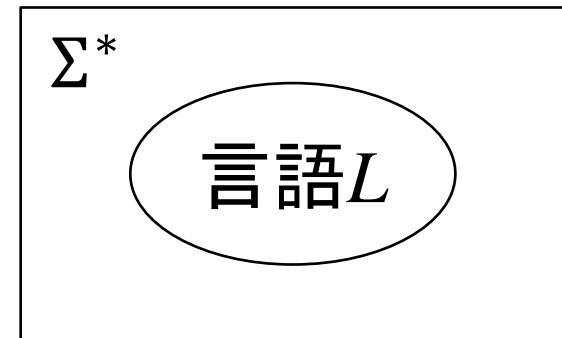
関澤 俊弦

日本大学 工学部 情報工学科

# 復習

## ■ (形式)言語

- アルファベット (記号)  $\Sigma$
- 記号列
- スター閉包  $\Sigma^*$
- $\Sigma$ 上の言語  $L$



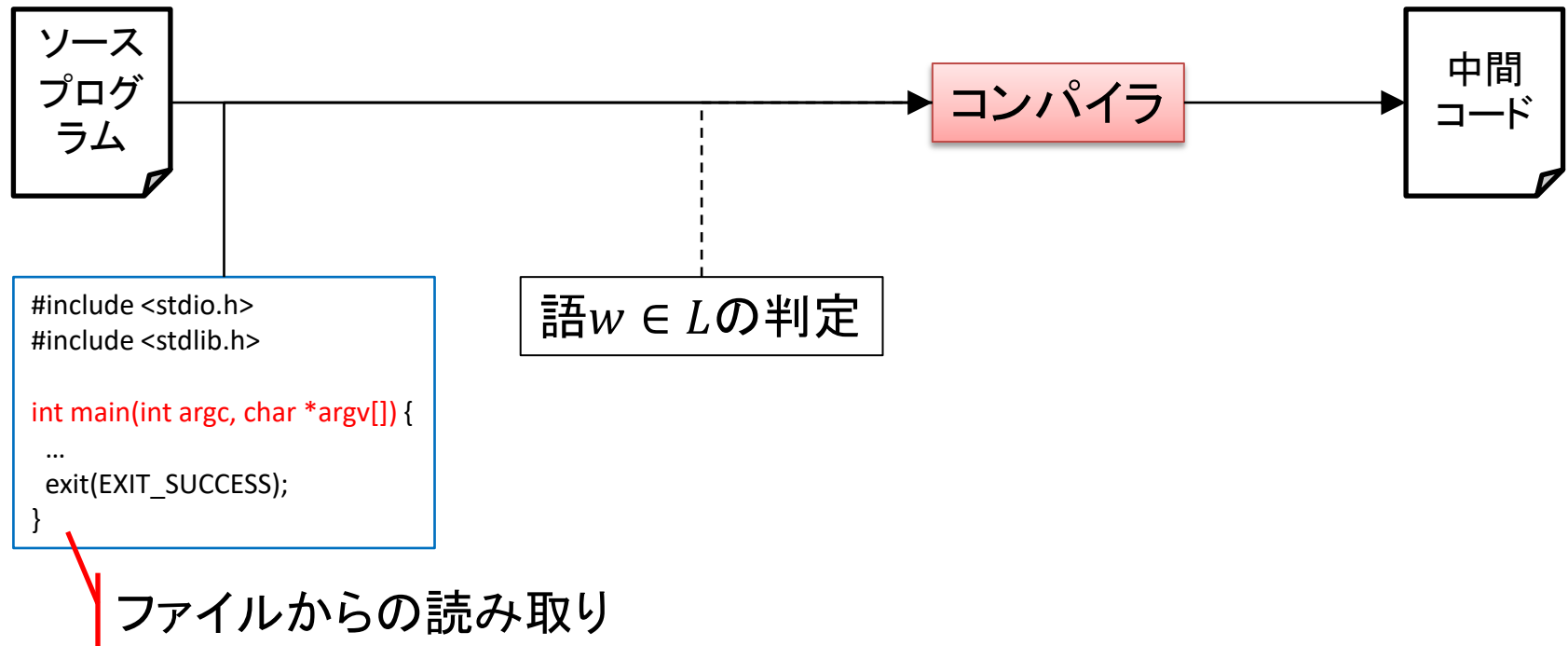
## ■ 形式文法

- 終端記号, 非終端記号, 書き換え規則
- 文法で生成される言語

# 復習

## ■ これまでの実装(演習)

- コマンドラインからのファイル指定
- 語  $w \in L$  の判定



## ■ オートマトン

### □ 有限オートマトン

- 決定性有限オートマトン
- 様相

### □ 正規言語

## ■ プログラミング技法

## ■ オートマトン

### □ 有限オートマトン

- 決定性有限オートマトン
- 様相

### □ 正規言語

## ■ プログラミング技法

# オートマトンとは

---

## ■ オートマトンとは

- 情報科学ではシステムのモデル
- ある「入力」に対して、処理を実行し、「出力」する数学的なモデル

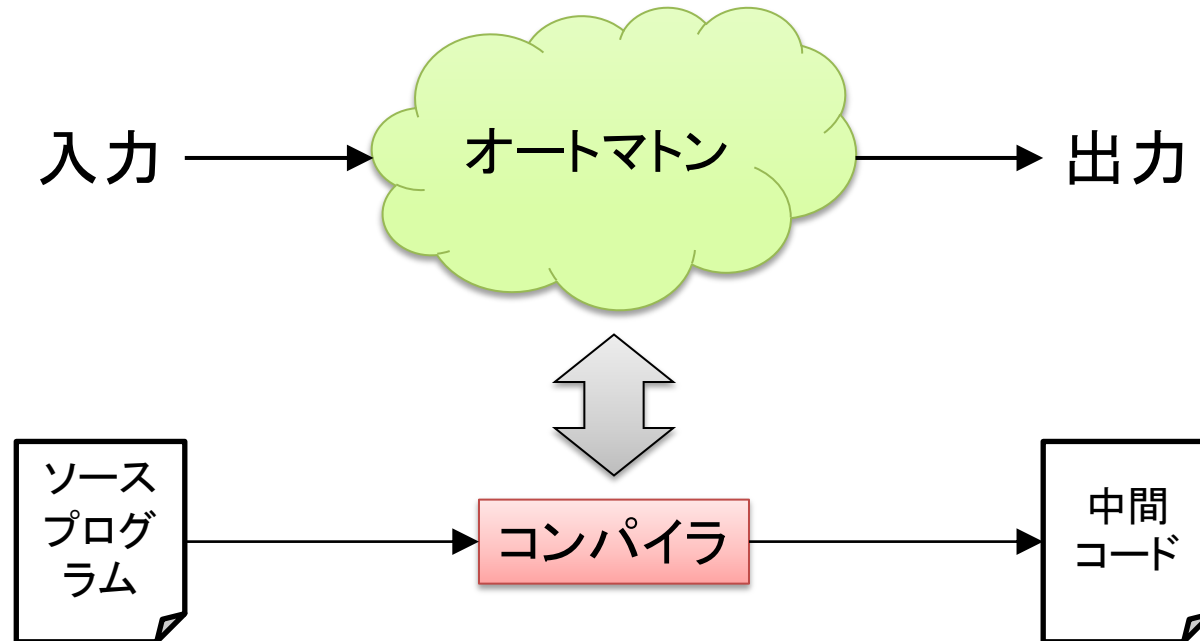


対象となるシステムに応じて、様々なオートマトンがある

# コンパイラとオートマトン

## ■ 直観的な対応

- コンパイラは入力(ソースプログラム)を
- (オートマトンを応用して)処理し,
- 中間コードを出力する



## ■ オートマトン

### □ 有限オートマトン

- 決定性有限オートマトン
- 様相

### □ 正規言語

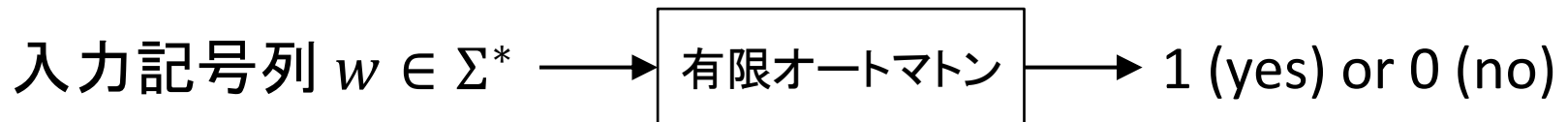
## ■ プログラミング技法



# 有限オートマトン

## ■ 有限オートマトン (FA: Finite Automaton)

- 受け入れ可能な入力記号列を認識し, 受け入れの可否(受理)で 1 (yes) または 0 (no) を出力
  - 認識機械 (リコグナイザ(recognizer))
- 記憶の種類(状態の数)は有限
- 最終状態(受理状態)をもつ
  - 1 (yes) を出力する状態



# 有限オートマトン

---

## ■ 有限オートマトン $M = (Q, \Sigma, \delta, q_0, F)$

- $Q$ : 状態の有限集合
- $\Sigma$ : 入力記号の有限集合
- $\delta$ : 状態遷移関数
- $q_0 \in Q$ : 初期状態
- $F \subseteq Q$ : 最終状態(受理状態)の集合



有限オートマトンでは、受理状態を区別するため、出力関数は明示しない

# オートマトンの入力(一部再掲)

---

## ■ 入力記号 $\Sigma$

□ オートマトンの入力は記号で表わす

•  $0, 1, 2, \dots$  や  $a, b, c, \dots$  など

□ 個々のオートマトンについては, 入力記号を有限に限定し, 有限集合で表わす

•  $\Sigma = \{0, 1\}$  や  $\Sigma = \{a, b, c\}$  など

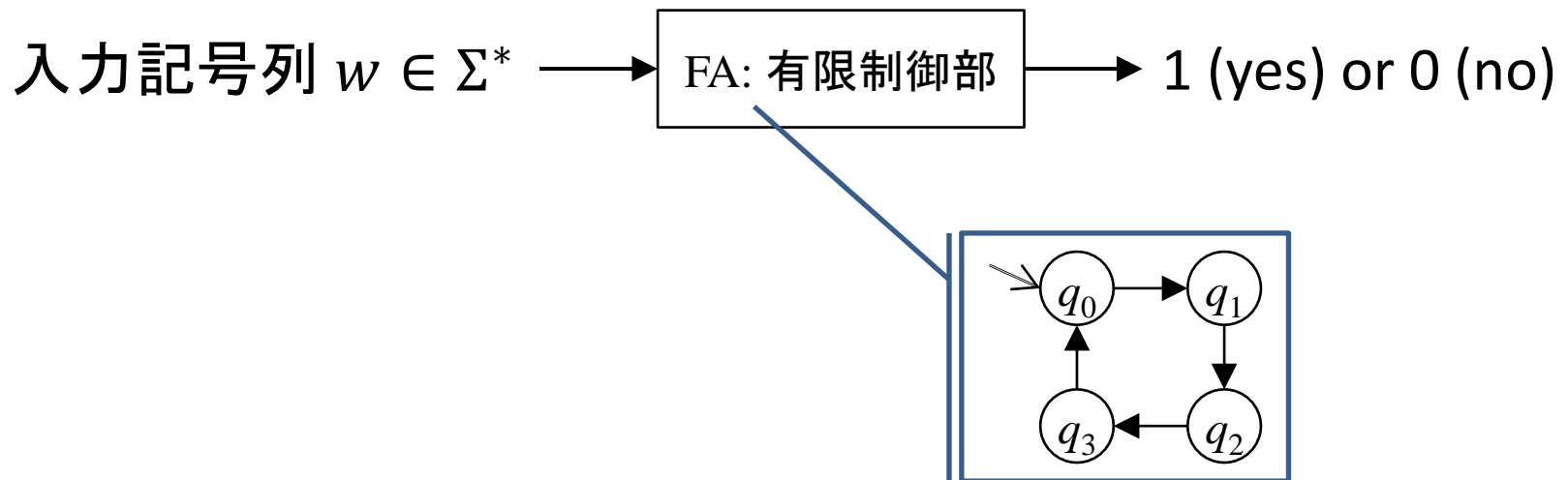
## ■ 入力記号列, 入力系列

□ 入力記号を並べてできる系列

•  $010, 010110$  など

# FAの概念

- FAは入力記号列の受理を判定する
  - ヘッドで入力記号列を読み込む
  - 入力記号により有限制御部の状態を変化させる
  - 動作を停止したときの状態で受理を判定する



# FAの定義と概念

## ■ 有限オートマトン $M = (Q, \Sigma, \delta, q_0, F)$

- $Q$ : 状態の有限集合
- $\Sigma$ : 入力記号の有限集合
- $\delta: Q \times \Sigma \rightarrow Q$ , 状態遷移関数
- $q_0 \in Q$ : 初期状態
- $F \subseteq Q$ : 受理状態の集合

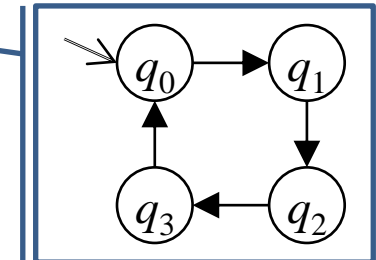


FA: 有限制御部



受理状態による判定

1 (yes) or 0 (no)



# FAの動作(1)

## ■ 入力テープ

- コマに分かれている
- 各コマには入力記号が1つ入っている
- 右端には終わりを示すエンドマーカ\$がある
  - エンドマーカ: どの入力記号とも異なる記号  
 $\Sigma \cap \{\$ \} = \emptyset$

## ■ 入力テープの準備

- 入力したい記号列  $w = a_1 a_2 \dots a_n$  をセットする
  - $a_i \in \Sigma, 1 \leq i \leq n$

入力テープ

$a_1$	$a_2$	...	$a_i$	...	$a_n$	\$
-------	-------	-----	-------	-----	-------	----

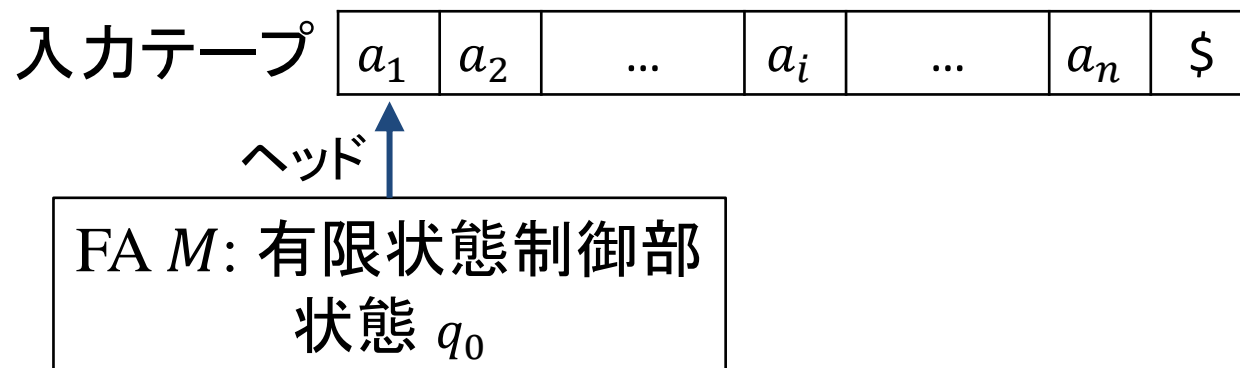
# FAの動作(2)

## ■ ヘッド

- 入力テープを1記号(1コマ)ずつ読み込む

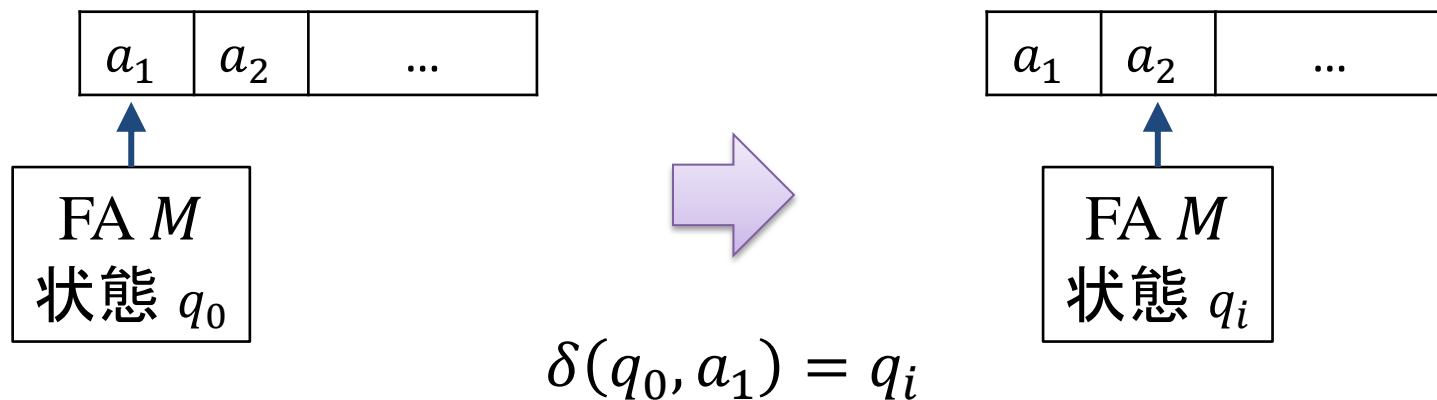
## ■ 入力テープ読み込みの準備

- ヘッドを入力テープの左端に置く
- $M$ の内部状態を初期状態 $q_0$ とする



# FAの動作(3)

- 有限オートマトン $M$ の1回の動作
  - ヘッドの位置にある記号を読み込む
  - 状態遷移関数 $\delta$ で定まる状態に遷移する
  - ヘッドを1コマ右に移動する





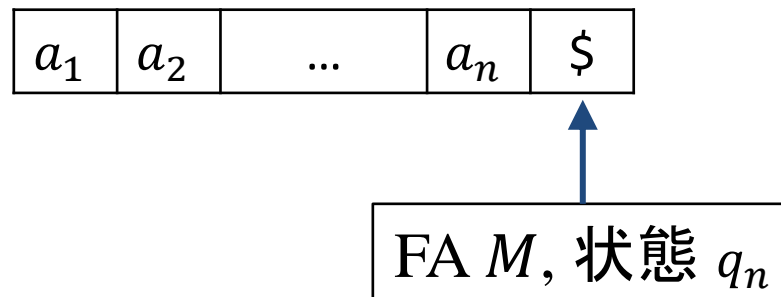
# FAの動作(4)

## ■ 動作の終了

- ヘッドがエンドマーカ的位置に来たら終了する

## ■ 記号列の受理判定

- 動作を停止したときの状態を $q_n$ とする
- $q_n$ が受理状態の1つならば $M$ は $w$ を受理する
  - $q_n \in F$
- そうでなければ,  $M$ は $w$ を拒否(非受理)する



# 有限オートマトンの記法: 状態遷移図

## ■ 状態

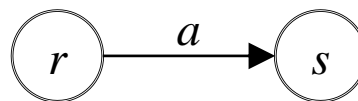
□ 状態は記号を丸で囲ったもので表わす

- 初期状態は矢印を付加する
- 最終状態は二重丸で表す



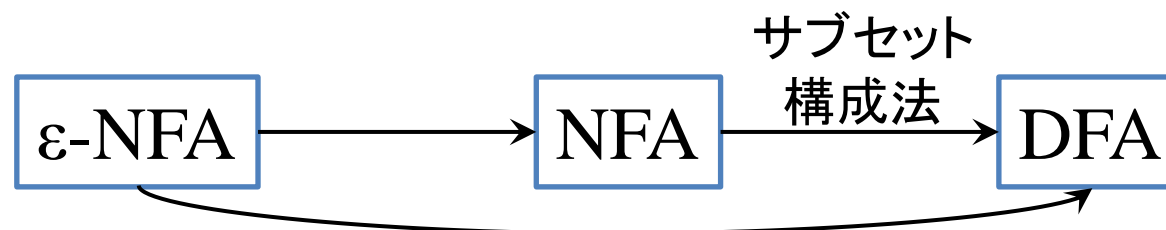
## ■ 遷移

□  $\delta(r, a) = s$  のとき, 状態  $r$  から状態  $s$  に向かう矢印を描き, 矢印に入力記号  $a$  をラベルとして付加する



# 様々な有限オートマトン

- 決定性有限オートマトン (DFA)
  - 同時に2つ以上の状態にいることはできない
- 非決定性有限オートマトン (NFA)
  - 同時に複数の状態にいることができる
- $\varepsilon$ -動作を含む非決定性有限オートマトン ( $\varepsilon$ -NFA)
  - 空記号 $\varepsilon$ を使える有限オートマトン



## ■ オートマトン

### □ 有限オートマトン

- 決定性有限オートマトン
- 様相

### □ 正規言語

## ■ プログラミング技法

# 決定性有限オートマトン

- 決定性有限オートマトン  $M = (Q, \Sigma, \delta, q_0, F)$ 
  - $Q$ : 状態の有限集合
  - $\Sigma$ : 入力記号の有限集合
  - $\delta: Q \times \Sigma \rightarrow Q$ , 状態遷移関数
    - 現在の状態  $p \in Q$  と入力  $a \in \Sigma$  に対して, 遷移先状態  $q \in Q$  を,  $\delta(p, a) = q$  により一意に定める関数.
  - $q_0 \in Q$ : 初期状態
  - $F \subseteq Q$ : 最終状態 (受理状態) の集合



DFA (Deterministic Finite Automaton) と呼ばれる

# DFAの処理と受理判定


## ■ 与えられた入力記号列の受理判定

1. 入力記号列  $a = a_1 a_2 a_3 \dots a_n$  が与えられたとする
2. 状態を初期状態  $q_0$  とする
3. 遷移関数  $\delta(q_0, a_1)$  を用いて, 遷移先を調べる
4. 同様に,  $\delta(q_{i-1}, a_i) = q_i$  の関係にある状態を見つける
5. 最終的な状態  $q_n$  が  $F$  の要素 ( $q_n \in F$ ) ならば, 入力記号列は"受理"される. そうでなければ"非受理"となる.




DFA  $M$  が受理する記号列の全体を  $M$  の"言語"と呼ぶ

## 例: DFA $M_{21}$

$\Sigma = \{a, b\}, L_{21} = \{a^m b^n \mid m, n > 0\}$ とする  p.19

■  $\Sigma$ 上の語 $w$ が与えられたとき,  $w$ が $L_{21}$ の要素かどうか判定する

□  $w \in L_{21}$ ならばyes(受理),  
 $w \notin L_{21}$ ならばno(非受理)と判定する

 p.23, 例2.7

# 例: DFA $M_{21}$

---

## ■ 判定方法

- 入力 $w$ を先頭から1文字ずつ読む
- $a$ で始まり, いくつか $a$ が続くことを確認する
- $b$ が入力された後は, いくつか $b$ が続き,  $a$ が入力されないことを確認する
- 入力を最後まで読んだとき,  $w$ が $L_{21}$ の要素ならばyes, そうでないときはnoと表示する



p.23, 例2.7



# 例: DFA $M_{21}$

## ■ DFA $M_{21} = (Q, \Sigma, \delta, q_0, F)$ を考える

□  $Q = \{p, q, r, d\}$

□  $\Sigma = \{a, b\}$

□  $\delta: Q \times \Sigma \rightarrow Q$

- $\delta(p, a) = q, \delta(p, b) = d,$
- $\delta(q, a) = q, \delta(q, b) = r,$
- $\delta(r, a) = d, \delta(r, b) = r,$
- $\delta(d, a) = d, \delta(d, b) = d$

□  $q_0 = p$

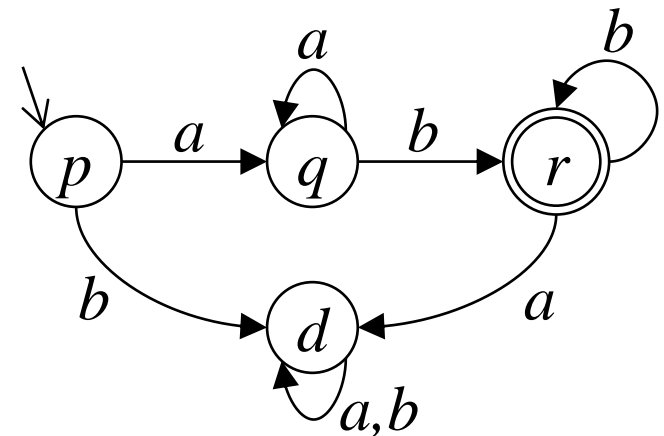
□  $F = \{r\}$



p.25, 例2.8



p.26, 問2.6

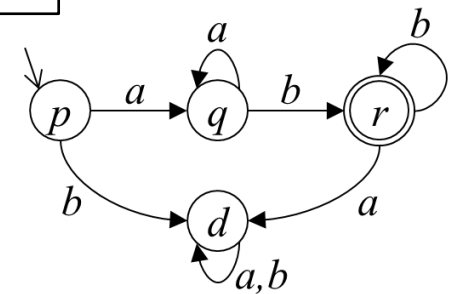
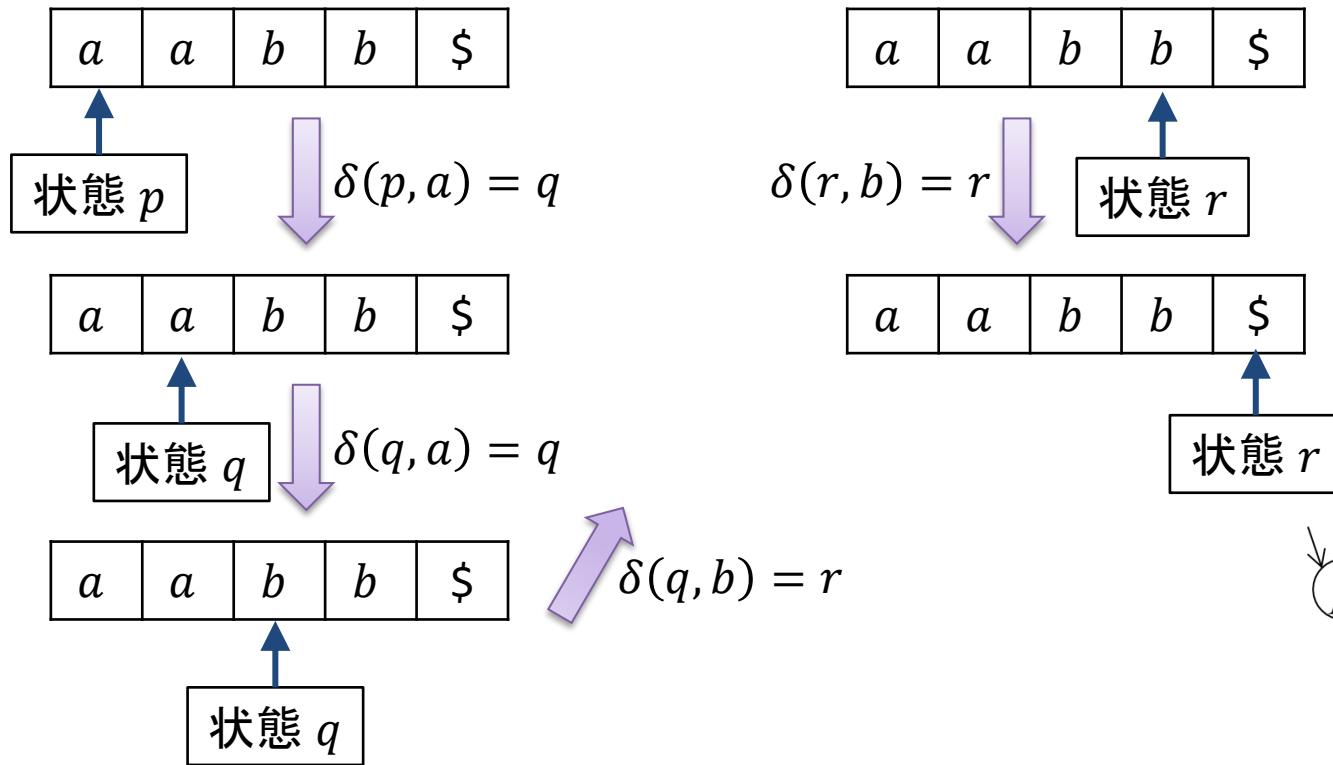


# 例: DFA $M_{21}$

## ■ DFA $M_{21}$ に入力 $aabb$ を与えたときの動作

□ 動作を停止したときの状態  $r \in F$  である.

ゆえに, DFA  $M_{21}$  は入力  $aabb$  を受理する.

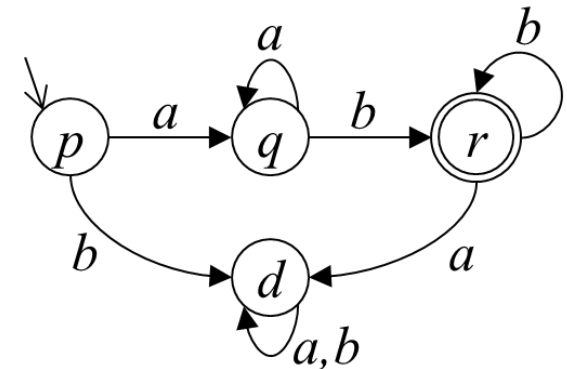
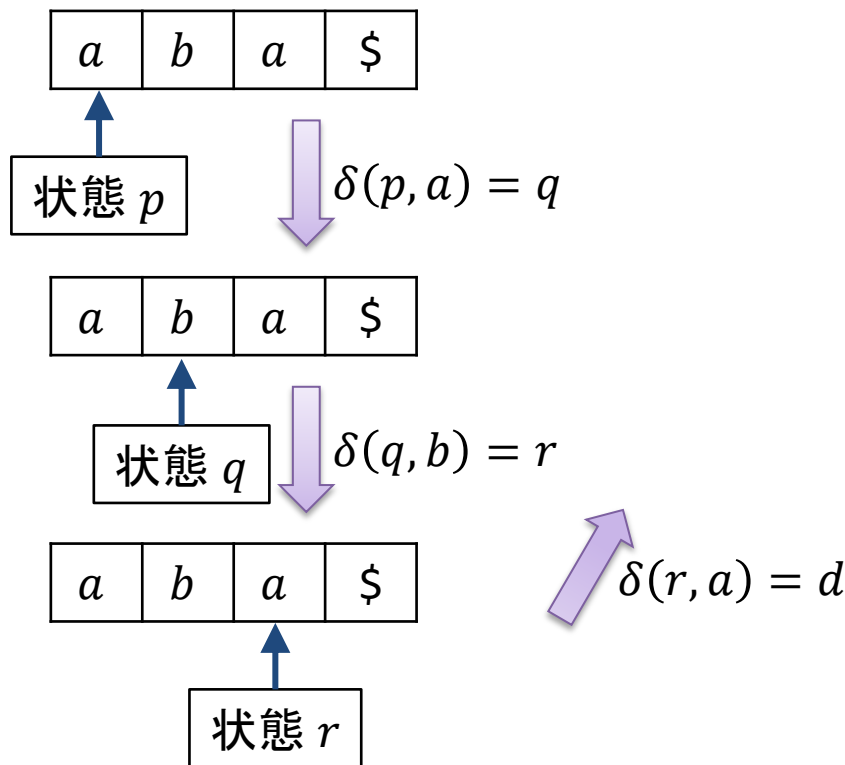


# 例: DFA $M_{21}$

## ■ DFA $M_{21}$ に入力 $aba$ を与えたときの動作

□ 動作を停止したときの状態  $d \notin F$  である.

ゆえに, DFA  $M_{21}$  は入力  $aba$  を受理しない.



## ■ オートマトン

### □ 有限オートマトン

- 決定性有限オートマトン
- 様相

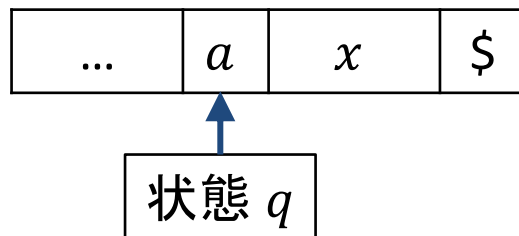
### □ 正規言語

## ■ プログラミング技法

# 様相

- 様相 (コンフィギュレーション (configuration)) とは
  - オートマトン  $M$  の動作の途中の状況
- 様相の表記  $(q, y)$ 
  - $q \in Q$ : 現在の状態
  - $y \in \Sigma^*$ : 処理されていない入力記号列

□ 例: 様相  $(q, ax)$



# 様相を用いた動作の表記

## ■ オートマトン $M$ の1回の動作を $\vdash_M$ と表わす

□  $M$ が明らかなきときは省略可.  $\vdash$ と表わす.

□ 例:  $(q, ax) \vdash_M (p, x)$

- $M$ の様相が $(q, ax)$ であり,  $\delta(q, a) = p$ のとき,  $M$ は動作して様相 $(p, x)$ となる.

□  $p_i \in Q, a_i \in \Sigma, 1 \leq i \leq n$ として,  
 $(p_1, a_1 a_2 a_3 \dots a_n) \vdash (p_2, a_2 a_3 \dots a_n),$   
 $(p_2, a_2 a_3 \dots a_n) \vdash (p_3, a_3 \dots a_n), \dots,$   
 $(p_{n-1}, a_{n-1} a_n) \vdash (p_n, a_n)$

のとき,

$(p_1, a_1 a_2 a_3 \dots a_n) \vdash (p_2, a_2 a_3 \dots a_n) \vdash \dots \vdash (p_n, a_n)$   
と書く.

# 様相を用いた動作の表記

- $M$ の複数回の動作を $\vdash_M^*$ （または $\vdash^*$ ）と書く
  - 直感的には $\vdash_M$ の何回かの繰り返し
  - $M$ の一連の動作を簡潔に記述できる
    - 例: 最初と最後の様相にのみ関心があるとき
$$(q_0, a_1 a_2 \dots a_n) \vdash_M^* (q_n, \varepsilon)$$



$\vdash_M^*$ は、 $\vdash_M$ の推移的閉包であるという

# 例: 様相を用いた $M_{21}$ の動作の表記

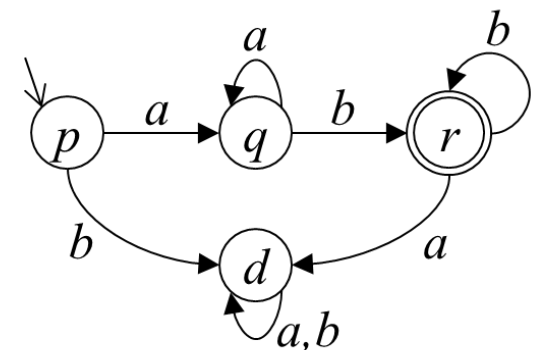
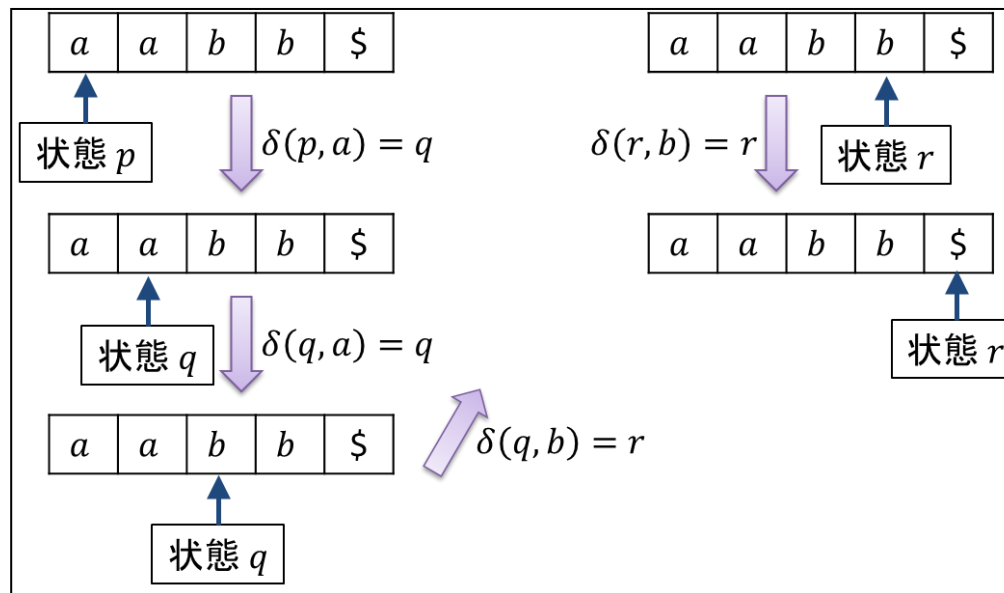
## ■ DFA $M_{21}$ に入力 $aabb$ を与えたときの動作

□  $(p, aabb) \vdash (q, abb) \vdash (q, bb) \vdash (r, b) \vdash (r, \varepsilon)$

- 動作を停止したときの状態  $r \in F$  である.

ゆえに, DFA  $M_{21}$  は入力  $aabb$  を受理する.

- 受理を示すならば,  $(p, aabb) \vdash^* (r, \varepsilon)$  と書ける





## ■ オートマトン

### □ 有限オートマトン

- 決定性有限オートマトン
- 様相

### □ 正規言語

## ■ プログラミング技法

# 正規言語 (RL: Regular Language)

- DFA  $M = (Q, \Sigma, \delta, q_0, F)$  の言語  $L(M)$  とは
  - 初期状態  $q_0$  を, 受理状態の一つに導く入力記号列の集合
  - $L(M) = \{w \mid w \in \Sigma^*, (q_0, w) \vdash_M^* (q_n, \varepsilon), q_n \in F\}$ 
    - 初期様相  $(q_0, w)$  から動作を始め,  $w$  を読み終わって動作を停止したときの様相を  $(q_n, \varepsilon)$  とする
- 正規言語 (正則言語) (RL)
  - ある言語  $L$  が, DFA  $M$  の言語  $L(M)$  と一致するとき,  $L$  を正規言語と呼ぶ
    - オートマトンが受理する言語

# 有限オートマトンと正規言語

---

## ■ 有限オートマトンの言語

□ DFA, NFA,  $\varepsilon$ -NFAは同じ言語を規定する

$$L(\varepsilon - NFA) = L(NFA) = L(DFA)$$

## ■ 正規言語 (RL) と有限オートマトン

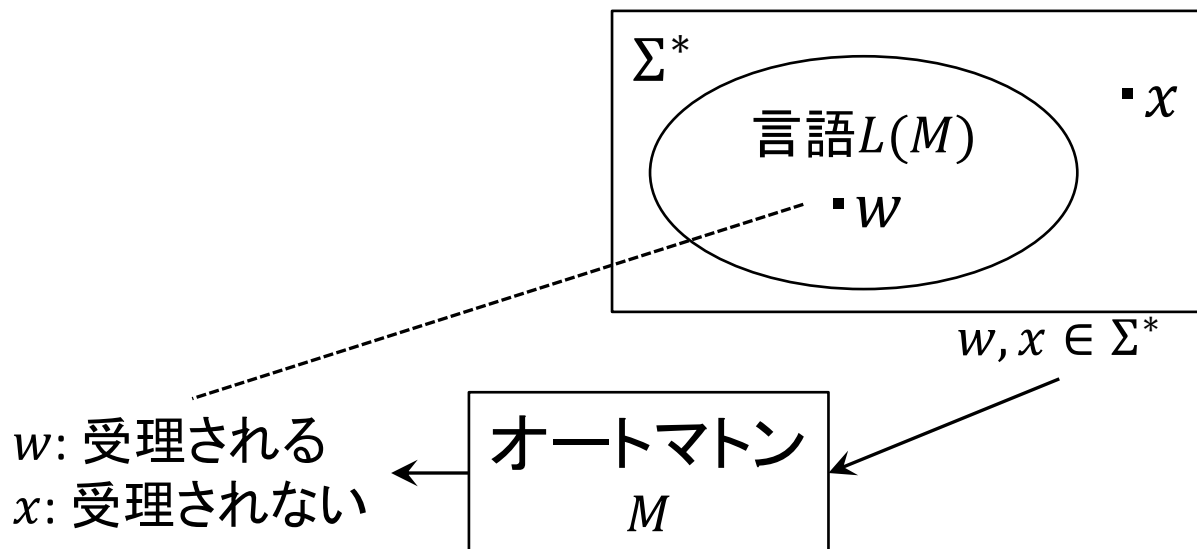
$$RL = L(\varepsilon - NFA) = L(NFA) = L(DFA)$$

# オートマトンから言語を生成する

## ■ 有限オートマトン $M$

□  $M$ を言語 $L(M)$ を生成するシステムとみなす

- 受理される語を考え、動作により記号が確定すると捉える
- $L(M) = \{w \mid w \in \Sigma^*, (q_0, w) \vdash_M^* (q_n, \varepsilon), q_n \in F\}$



# 他の代表的なオートマトン

---

- プッシュダウンオートマトン  
(pushdown automaton: PDA)
  - 有限オートマトンに, プッシュダウンスタックと呼ばれる補助記憶を付け加えたオートマトン
- 線形拘束オートマトン  
(linear bounded automaton: LBA)
  - 1本のテープを持つ非決定性TM
- チューリング機械  
(Turing machine: TM)
  - 有限オートマトンに, 作業用テープを加えたもの

## ■ オートマトン

### □ 有限オートマトン

- 決定性有限オートマトン
- 様相

### □ 正規言語

## ■ プログラミング技法

# 列挙型

---

## ■ 列挙型 (enum)

### □ データ型の一つ

- C言語では整数定数に名前を付けて実現される

### □ typedefと共に使用することで型を定義できる

- プログラムの意味を明確にできる

# 列挙型

## ■ 例

- 列挙型を用いた状態の定義.

Sp, Sq, Sr, SdをメンバにもつSTATE型をtypedefを用いて定義し, プログラム中使用する

```
typedef enum {  
    Sp, Sq, Sr, Sd  
} STATE;
```

enumを用いたSTATE型の定義

```
int main(void) {  
    STATE s = Sp;  
    :  
}
```

STATE型の変数sの定義と初期化



# まとめ

---

## ■ オートマトン

- 有限オートマトン (FA)
  - 決定性有限オートマトン (DFA)
  - 様相
- 正規言語 (RL)

# 演習

# 本演習のポイント

---

## ■ C言語

### □ 列挙型(enum)

- enumは今後も多用します

## ■ 処理

### □ 関数nextChar

- nextCharは今後も使用する重要な関数

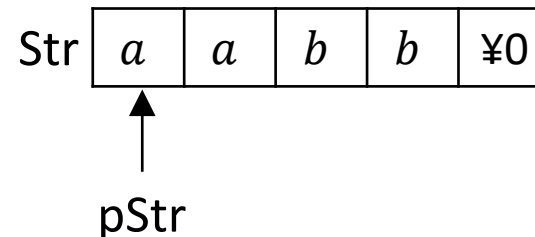
### □ 文字(記号)による処理の振り分け

# 関数nextCharの実装例

## ■ 関数nextChar, Lesson02, 演習Step2, サンプルコードの意図

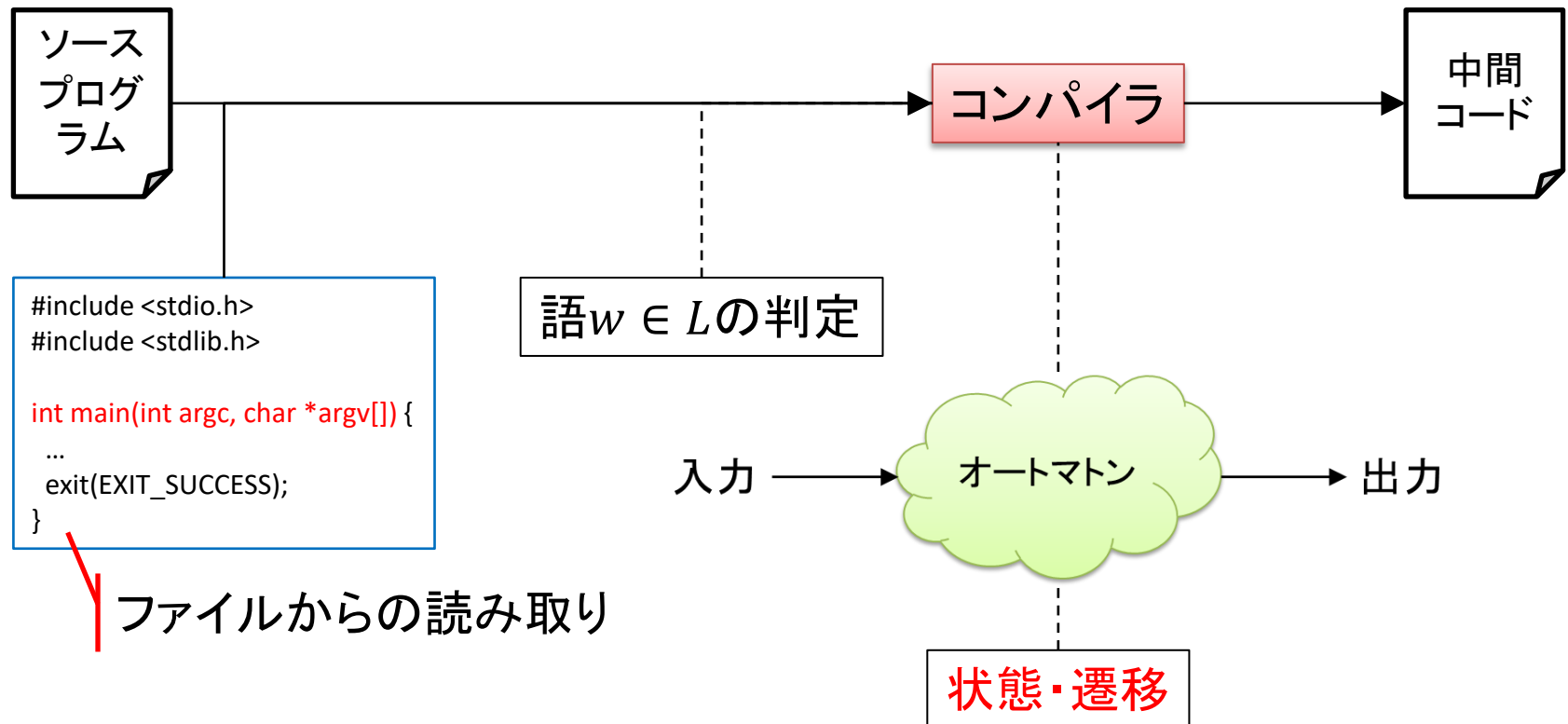
- オートマトンの入力記号列を文字列Strで表わす
  - static char \*pStr で文字列Strを参照する
- オートマトンが読み込んだ文字を文字cで表わす
  - ポインタpStrが指している文字をcとして取得する

```
char nextChar(void) {  
    static char *pStr = Str;  
    char c;  
  
    /* Step2をここに実装する */  
  
    return c;  
}
```



# 演習3-1: DFA $M_{31}$ の実装

- オートマトン  $M_{31}$  を, C言語で実装する
  - 状態・遷移・判定を実現する



# $M_{31}$ の仕様

## ■ 仕様

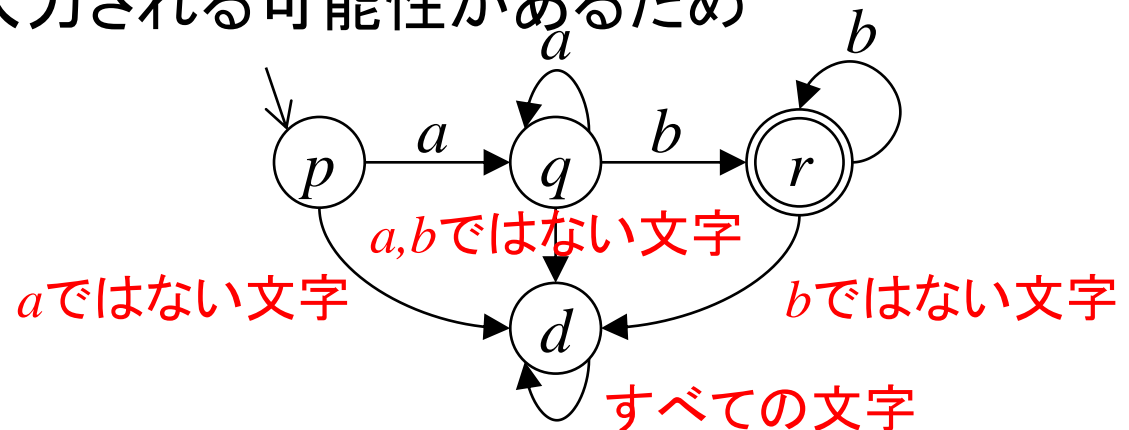
□ 実装言語: C言語

□ 入力記号列: 標準入力より, 1文として入力する

- 入力の最大文字数は256文字と仮定する

□  $M_{31}$ のエラー状態 $d$ への遷移の拡張

- 標準入力からは, 入力記号の有限集合 $\Sigma$ に含まれない文字が入力される可能性があるため



# $M_{31}$ の直観的な実装(1)

---

- オートマトン $M_{31}$ を;
  - 状態は, ラベルとして記述する
  - 遷移は, ラベルへのgoto文で記述する



問題点:

一般に, goto文の使用は推奨されない

# $M_{31}$ の直観的な実装(2)

## ■ compiler03\_1\_goto.c(一部)

### □ ラベルによる状態と, goto文を用いた状態遷移

Q:

```
ch = nextChar();  
if (ch == 'a') { printf("d(q, %c) = q\n", ch); goto Q; }  
if (ch == 'b') { printf("d(q, %c) = q\n", ch); goto R; }  
if (ch == '¥0') { goto REJECT; }  
goto D;
```

R:

```
ch = nextChar();  
if (ch == 'b') { printf("d(r, %c) = r\n", ch); goto R; }  
if (ch == '¥0') { goto ACCEPT; }  
goto D;
```

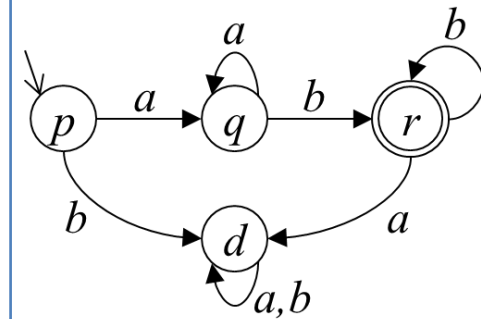
/\* (略) \*/

ACCEPT:

```
printf("yes\n");  
exit(EXIT_SUCCESS);
```

REJECT:

```
printf("no\n");  
exit(EXIT_SUCCESS);
```





# $M_{31}$ の実装

---

## ■ オートマトン $M_{31}$ を

- gotoを用いないで実装する
- 状態は列挙型(enum)で定義する
- 遷移は状態を表わす変数の変化で表わす

## ■ 仕様・制約・実行例は、演習課題提出システムを参照のこと

# 演習3-1: 実行例

## ■ 入力記号列と期待される判定結果

1. aabb – yes
2. aba – no
3. abc – no

```
gw.cse.ce.nihon-u.ac.jp - PuTTY
sekizawa@cse-ssh[38]: ./3-1
w = aabb
d(p, a) = q
d(q, a) = q
d(q, b) = r
d(r, b) = r
yes
sekizawa@cse-ssh[39]: ./3-1
w = aba
d(p, a) = q
d(q, b) = r
d(r, a) = d
no
sekizawa@cse-ssh[40]: ./3-1
w = abc
d(p, a) = q
d(q, b) = r
d(r, c) = d
no
sekizawa@cse-ssh[41]: █
```