# Lecture 9 - Oct. 6

## Lexical Analysis, Syntactic Analysis

***Minimizing DFA***
***Implementing a Scanner***
***Context-Free Grammar (CFG): Basics***

# Announcements

- Reading week study item: ANTLR tutorial
  + RE
  + CFG
  + OOP and Composite & visitor design patterns
- Assignment 1 due tomorrow (Friday) at 2pm
- Programming Test date reminder:
  + 2:00pm to 3:20pm on Saturday, October 29
  + Venue to be confirmed
- Quiz 1 to be returned in class on October 17
- Quiz 2 postponed to Thursday, October 19

# Minimizing DFA: Algorithm

① What if $M' = M \Rightarrow$ no optimization can be done
② Is $|Q(M')| > |Q(M)|$ possible? ⇒ algo. not achieving what it's supposed to.

```
ALGORITHM: MinimizeDFAStates
  INPUT: DFA M = (Q, Σ, δ, q0, F)
  OUTPUT: M′ s.t. minimum |Q| and equivalent behaviour as M
PROCEDURE:
  P := ∅ /* refined partition so far */
  T := { F, Q − F } /* last refined partition */
  while (P ≠ T):
    P := T
    T := ∅
    for(p ∈ P):
      find the maximal S ⊂ p s.t. splittable(p, S)
      if S ≠ ∅ then
        T := T ∪ {S, p − S}
      else
        T := T ∪ {p}
      end
```

partition #1 (accepting states)

partition #2 (non-accepting states)

$P = T$ means no more optimization can be done

fixed point.

***splittable***$(p, S)$ holds iff there is $c \in \Sigma$ s.t.

1. $S \subset p$ (or equivalently: $p - S \neq \varnothing$)
2. Transitions via $c$ lead all $s \in S$ to states in ***same partition*** $p1$ ($p1 \neq p$).

# Partitions of States

input

e.g., $Q = \{s0, s1, s2, s3\}$

- Smallest number of partitions.
- Largest number of partitions.
- Partitions somewhere in-between
- Analogy from Software Testing: Equivalent Classes

single partition

$Q' = \{ \boxed{\{S_0, S_1, S_2, S_3\}} \}$

$Q' = \{ \{S_0\}, \{S_1\}, \{S_2\}, \{S_3\} \}$

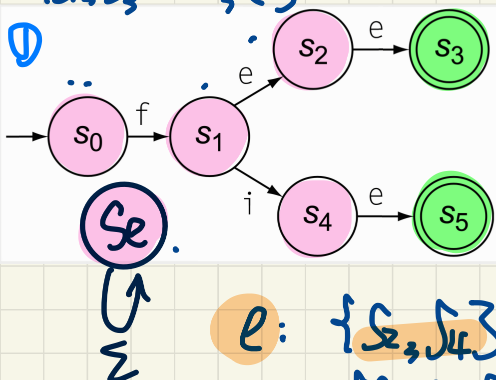
no optimization

# Minimizing DFA: Example (1)

$\Sigma = \{a, b, \dots, z\}$

fee | fie

① 

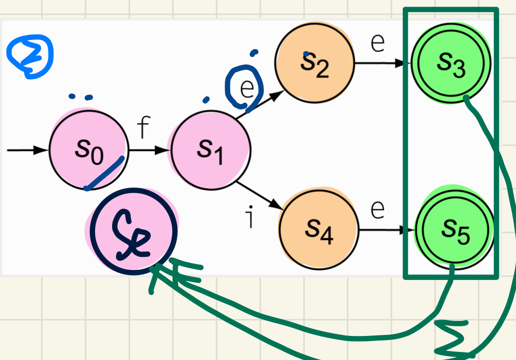



f: $\{S_0, S_1, S_2, S_4, S_e\} \xrightarrow{f}$

e: $\{S_2, S_4\} \xrightarrow{e}$

$\{S_e, S_0, S_1\} \xrightarrow{e}$

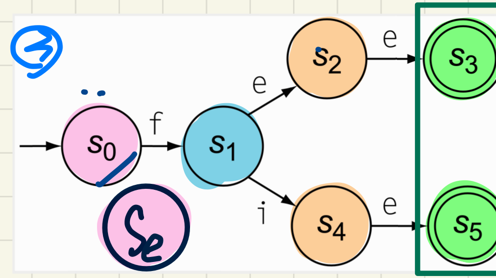
② 




e: $\{S_1\} \rightarrow$

$\{S_0, S_e\} \rightarrow$

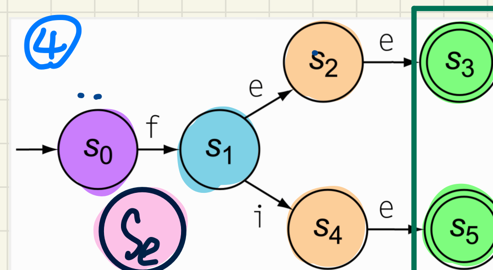
③ 

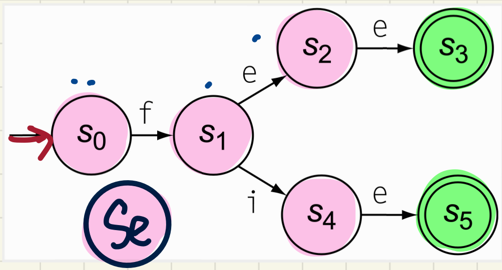



f: $\{S_0\} \xrightarrow{f}$

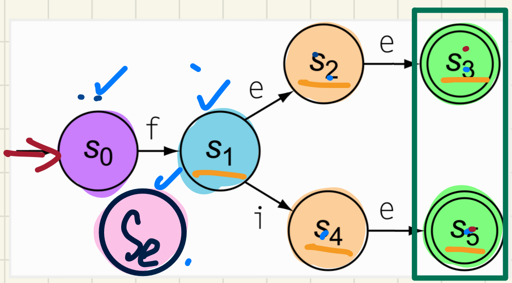
$\{S_e\} \xrightarrow{f}$

④ 






input : 7 states

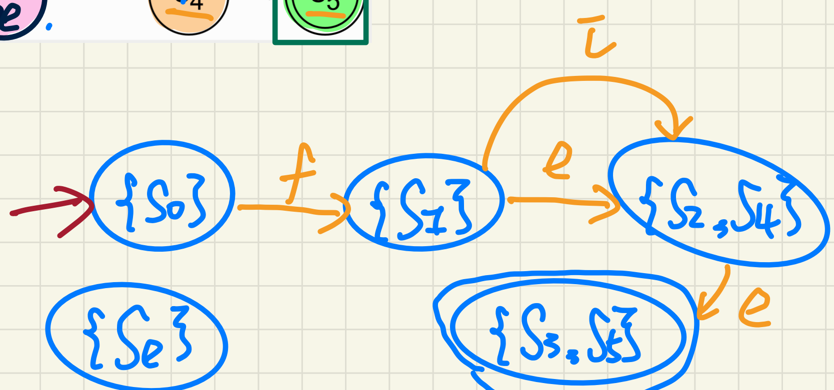



output : 5 partitions

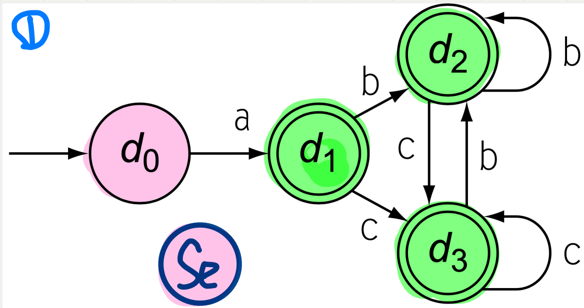
7 states

↓

5 states.



# Minimizing DFA: Example (2)




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

partition

a:

$d_0 \xrightarrow{a} \{d_1, d_2, d_3\}$

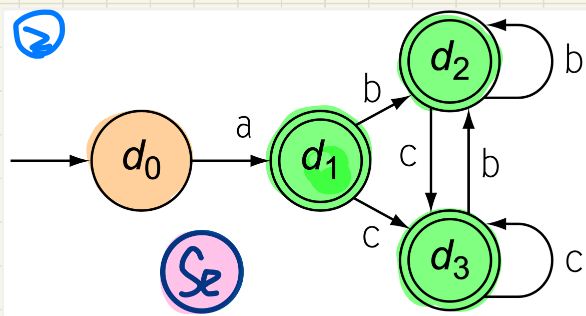
$s_e \xrightarrow{a} \{d_0, s_e\}$

5 states

$\downarrow$

3 partitions

$\boxed{\text{Exercise: draw DFA}}$



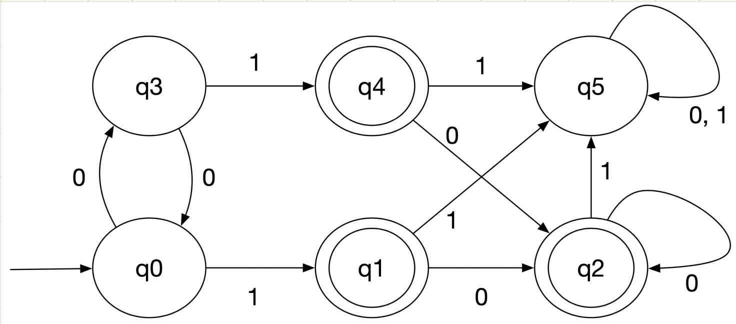

a:

$\{d_1, d_2, d_3\} \xrightarrow{a} \{s_e\}$

$\{d_1, d_2, d_3\} \xrightarrow{b} \{d_1, d_2, d_3\}$

$\{d_1, d_2, d_3\} \xrightarrow{c} \{d_1, d_2, d_3\}$

# Minimizing DFA: Example (3)




(Exercise)

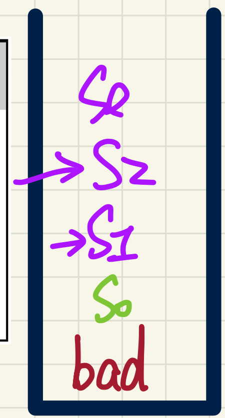
# From RE to Scanner (1)

## Token Type (CharCat)

| r | 0, 1, 2, ..., 9 | EOF | **Other** |
|---|---|---|---|
| *Register* | *Digit* | *Other* | *Other* |

## Transition

| | *Register* | *Digit* | *Other* |
|---|---|---|---|
| $s_0$ | $s_1$ | $s_e$ | $s_e$ |
| $s_1$ | $s_e$ | $s_2$ | $s_e$ |
| $s_2$ | $s_e$ | $s_2$ | $s_e$ |
| $s_e$ | $s_e$ | $s_e$ | $s_e$ |




## Token Type (Type)

| $s_0$ | $s_1$ | $s_2$ | $s_e$ |
|---|---|---|---|
| *invalid* | *invalid* | *register* | *invalid* |

## Regular Expression: r[0..9]+

```
NextWord()
  -- Stage 1:  Initialization
  state := s0 ; word := ε
  initialize an empty stack s ; s.push(bad)
  -- Stage 2:  Scanning Loop
  while (state ≠ se)
    NextChar(char) ; word := word + char
    if state ∈ F then reset stack s end
    s.push(state)
    cat := CharCat[char]
    state := δ[state, cat]
  -- Stage 3:  Rollback Loop
  while (state ∉ F ∧ state ≠ bad)
    state := s.pop()
    truncate word
  -- Stage 4:  Interpret and Report
  if state ∈ F then return Type[state]
  else return invalid
  end
```

S0 S1

Example input: r2X

EOF

word: ~~ε~~ r2

state: ~~S0~~ ~~S1~~ S2

cat: ~~Register~~ Digit

# From RE to Scanner (1)

## Token Type (CharCat)

| r | 0, 1, 2, ..., 9 | EOF | **Other** |
|---|---|---|---|
| *Register* | *Digit* | *Other* | *Other* |

## Transition

| | ***Register*** | ***Digit*** | ***Other*** |
|---|---|---|---|
| $\boldsymbol{s_0}$ | $s_1$ | $s_e$ | $s_e$ |
| $\boldsymbol{s_1}$ | $s_e$ | $s_2$ | $s_e$ |
| $\boldsymbol{s_2}$ | $s_e$ | $s_2$ | $s_e$ |
| $\boldsymbol{s_e}$ | $s_e$ | $s_e$ | $s_e$ |

## Token Type (Type)

| $\boldsymbol{s_0}$ | $\boldsymbol{s_1}$ | $\boldsymbol{s_2}$ | $\boldsymbol{s_e}$ |
|---|---|---|---|
| *invalid* | *invalid* | *register* | *invalid* |

## Regular Expression: r[0..9]+

```
NextWord()
  -- Stage 1:  Initialization
  state := s0 ; word := ϵ
  initialize an empty stack s ; s.push(bad)
  -- Stage 2:  Scanning Loop
  while (state ≠ se)
    NextChar(char) ; word := word + char
    if state ∈ F then reset stack s end
    s.push(state)
    cat := CharCat[char]
    state := δ[state, cat]
  -- Stage 3:  Rollback Loop
  while (state ∉ F ∧ state ≠ bad)
    state := s.pop()
    truncate word
  -- Stage 4:  Interpret and Report
  if state ∈ F then return Type[state]
  else return invalid
  end
```

Example input: r24*3

EOF.

(Exercise)

word:

state:

cat:

# Context-Free Grammar (CFG): Terminology

The following language that is ***non-regular***

$$\{0^n\#1^n \mid n \overset{>}{\cancel{\geq}} 0\}$$

can be described using a ***context-free grammar (CFG)***:

$$
\begin{aligned}
A &\rightarrow 0A1 \\
A &\rightarrow B \\
B &\rightarrow \#
\end{aligned}
$$

start variable

non-terminals (variables)

terminals

production rule

# Visualization Derivations from CFG

$A \xrightarrow{①} 0A1$

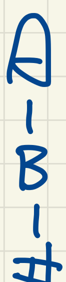
$A \xrightarrow{②} B$

$B \xrightarrow{③} \#$

- Shortest Derivation? #
- (A) 000#111?
- (B) 010#101? No.

Exercise: Modify/extend the grammar to allow it.

$A \overset{②}{\Rightarrow} B \overset{③}{\Rightarrow} \#$ (derivation result)



(A)