## Lecture 3: Finite Automata

**Administrivia**

- Everyone should now be registered electronically using the link on our webpage. If you haven't, do so today!
- I'd like to have teams formed by next Monday at the latest.
- Please fill out the background survey linked to on the homework page.
- HW #2 now available (due next Tuesday).
- *Tentative* test dates (in class): 10 Oct, 7 Nov.
- *Tentative* project due dates: 4 Oct, 1 Nov, 3 Dec.



## An Alternative Style for Describing Languages

- Rather than giving a single pattern, we can give a set of rules.
- Each rule has the form

$$A:\ \alpha_1\alpha_2\cdots\alpha_n,\quad n\geq 0,$$

  where

  - $A$ is a symbol that is intended to stand for a language (set of strings)—a *metavariable* or *nonterminal symbol*.
  - Each $\alpha_i$ is either a literal character (like `"a"`) or a nonterminal symbol.

- The interpretation of this rule is

  One way to form a string in $L(A)$ (the language denoted by $A$) is to concatenate one string each from $L(\alpha_1),\ L(\alpha_2),\ldots.$

  (where $L("c")$ is just the language $\{"c"\}$).

- This is *Backus-Naur Form (BNF)*. A set of rules is a *grammar*.
- **Aside:** You'll see that ':' written many different ways, such as '::=', '$\longrightarrow$', etc. We'll just use the same notation our tools use.



## Some Abbreviations

- The basic form from the last slide is good for formal analysis, but not for writing.
- So, we can allow some abbreviations that are obviously exandable into the basic forms:

| Abbreviation | Meaning |
| --- | --- |
| $A:\ \mathcal{R}_1 \mid \cdots \mid \mathcal{R}_n$ | $A:\ \mathcal{R}_1$<br>$\vdots$<br>$A:\ \mathcal{R}_n$ |
| $A:\ \cdots(\mathcal{R})\cdots$ | $B:\ \mathcal{R}$<br>$A: \cdots B \cdots$ |
| $A:\ "c_1" \mid \cdots \mid "c_n"$<br>(likewise other character classes) | $[c_1\cdots c_n]$ |



## Some Technicalities

- From the definition, each nonterminal in a grammar defines a language. Often, we are interested in just one of them (the *start symbol*), and the others are auxiliary definitions.
- The definition of what a rule means ("One way to form a string in $L(A)$ is...") leaves open the possibility that there are other ways to form items in $L(A)$ than covered in the rule.
- We need that freedom in order to allow multiple rules for $A$, but we don't really want to include strings that aren't covered by some rule.
- So precise mathematical definitions throw in sentences like:

  A grammar defines the *minimal* languages that contain all strings that satisfy the rules.

## A Big Restriction (for now)

- For the time being, we'll also add a restriction. In each rule:

$$A:\ \alpha_1\alpha_2\cdots\alpha_n,\ \ n \geq 0,$$

we'll require that if $\alpha_i$ is a nonterminal symbol, then either

- All the rules for that symbol have to occured before all the rules for $A$, or
- $i = n$ (i.e., is the last item) and $\alpha_n$ is $A$.

- We call such a restricted grammar a *Type 3* or *regular* grammar. The languages definable by regular grammars are called *regular languages.*

**Claim:** Regular languages are exactly the ones that can be defined by regular expressions.



## Proof of Claim (I)

- Start with a regular expression, $\mathcal{R}$, and make a (possibly not yet valid) rule,

  R: $\mathcal{R}$

- Create a new (preceding) rule for each parenthesized expression.
- This will leave just the constructs '$X*$', '$X+$', and '$X?$'. What do we do with them?



## Proof of Claim (II)

| **Replace construct. . .** | **. . . with $Q$, where** |
|---|---|
| $R*$ | `Q :`<br>`Q : R Q` |
| $R+$ | `Q : R`<br>`Q : R Q` |
| $R?$ | `Q :`<br>`Q : R` |



## Example

- Consider the regular expression `("+"|"-")?("0"|"1")+`

1. R: `("+"|"-")?("0"|"1")+` *replace with . . .*

2. $Q_1$: `"+" | "-"`
   $Q_2$: `"0" | "1"`
   R: $Q_1$? $Q_2$+ *replace with . . .*

3. $Q_3$: $\epsilon$ | $Q_1$
   $Q_4$: $Q_2$ | $Q_2$ $Q_4$
   R: $Q_3$ $Q_4$

## Classical Pattern-Matching Implementation

- For compilers, can generally make do with "classical" regular expressions.
- Implementable using *finite(-state) automata* or *FAs.* ("Finite state" = "finite memory").
- Classical construction:

  regular expression $\Rightarrow$ nondeterministic FA (NFA)
  $\Rightarrow$ deterministic FA (DFA) $\Rightarrow$ table-driven program.

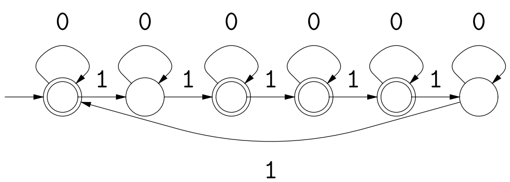


## Review: FA operation

- A FA is a graph whose nodes are states (of memory) and whose edges are state transitions. There are a finite number of nodes.
- One state is the designated start state.
- Some subset of the nodes are final states.
- Each transition is labeled with a set of symbols (characters, etc.) or $\epsilon$.
- A FA recognizes a string $c_1 c_2 \cdots c_n$ if there is a path (sequence of edges) from the start state to a final state such that the labels of the edges in sequence, aside from $\epsilon$ edges, respectively contain $c_1, c_2, \ldots, c_n$.
- If the edges leaving any node have disjoint sets of characters and if there are no $\epsilon$ nodes, FA is a DFA, else an NFA.

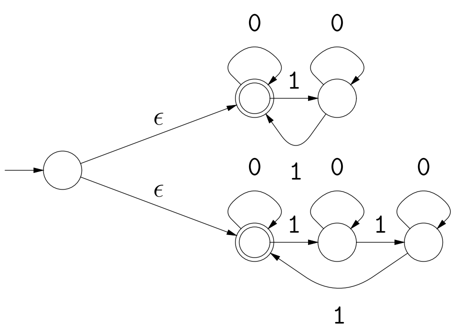


## Example: What does this DFA recognize?



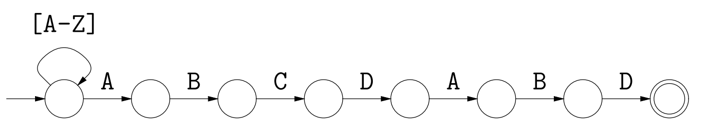

Bit strings with # of 1's divisible by 2 or 3.

What is the simplest equivalent NFA you can think of?



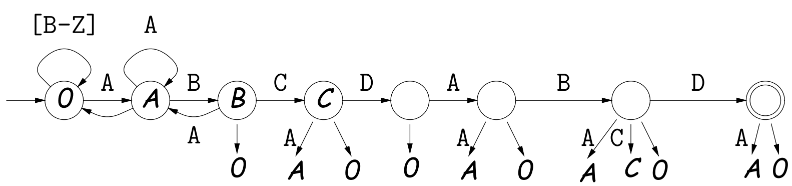



## Example: What does this NFA recognize?



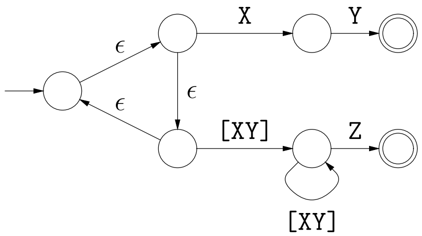

Strings of capitals ending in ABCDABD.

What is the simplest equivalent DFA you can think of?




*(Edges without labels mean "any character not covered by another edge.")*

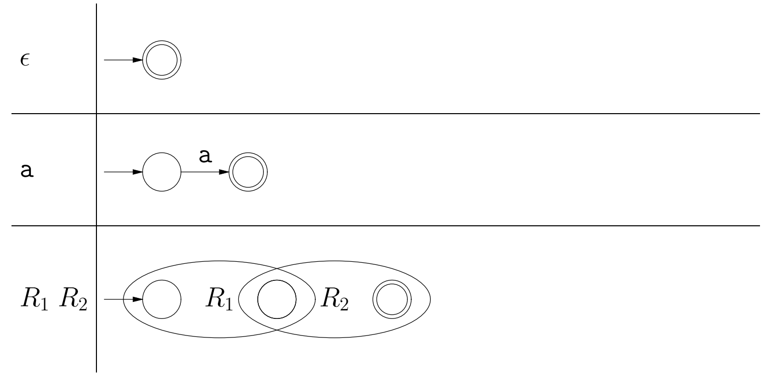
## Example: What does this NFA recognize?




What is the simplest equivalent DFA you can think of?

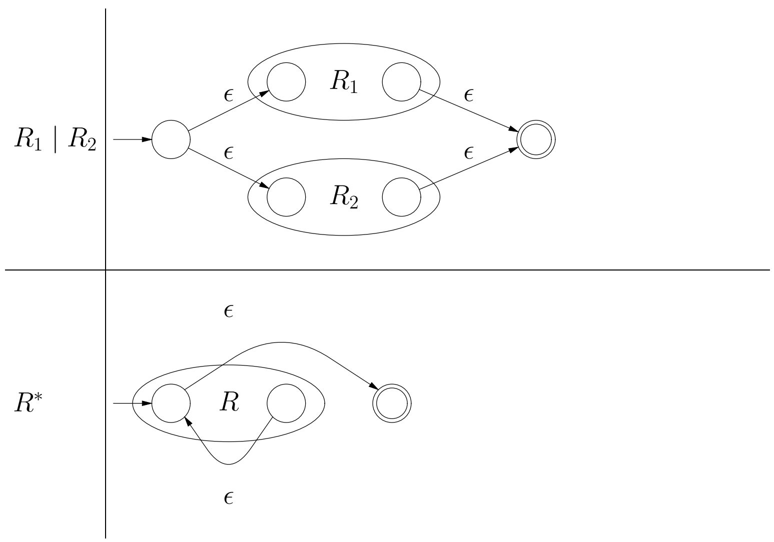


## Review: Classical Regular Expressions to NFAs (I)






## Review: Classical Regular Expressions to NFAs (II)



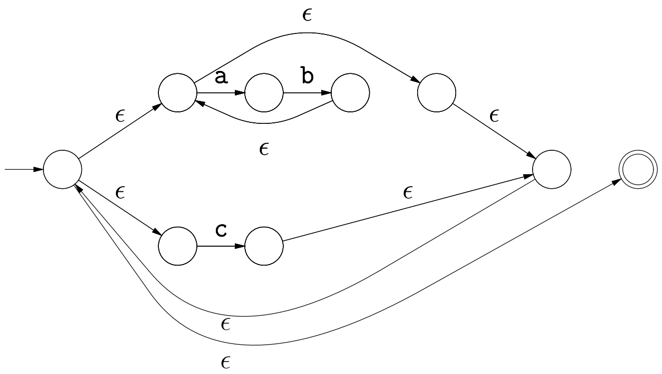



## Extensions?

- How would you translate $\phi$ (the empty language, containing no strings) into an FA?
- How could you translate 'R?' into an NFA?
- How could you translate 'R+' into an NFA?
- How could you translate '$R_1|R_2|\cdots|R_n$' into an NFA?

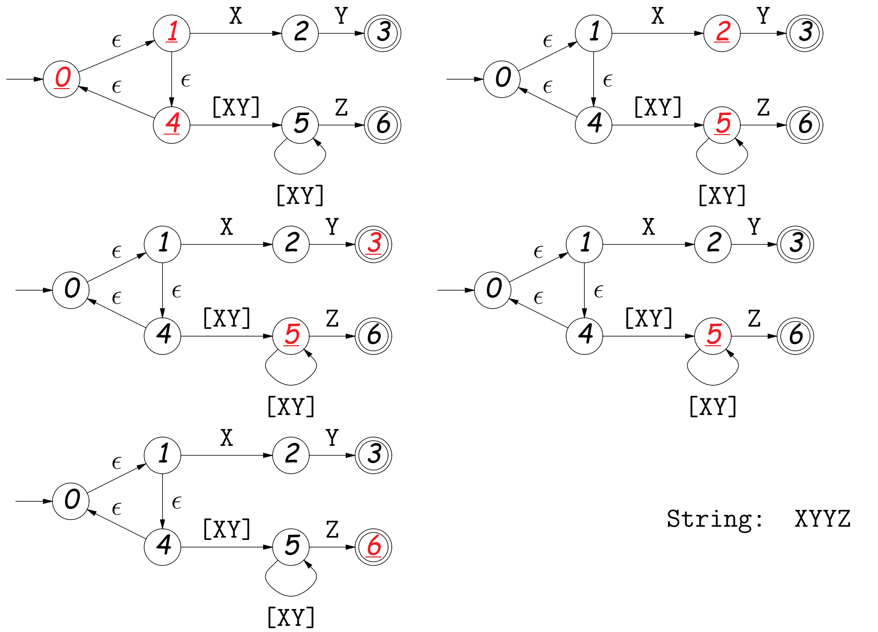
# Example of Conversion

How would you translate `((ab)*|c)*` into an NFA (using the construction above)?





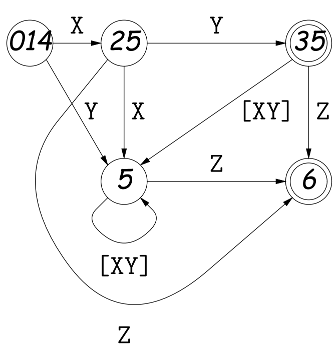

# Abstract Implementation of NFAs






# Review: Converting to DFAs

- **OBSERVATION:** The set of states that are marked (colored red) changes with each character in a way that depends only on the set and the character.
- In other words, machine on previous slide acted like this DFA:






# DFAs as Programs

- Can realize DFA in program with control structure:

```
state = INITIAL;
for (s = input; *s != '\0'; s += 1) {
   switch (state):
   case INITIAL:
      if (*s == 'a') state = A_STATE; break;
   case A_STATE:
      if (*s == 'b') state = B_STATE; else state = INITIAL; break;
   ...
   }
}
return state == FINAL1 || state == FINAL2;
```

- Or with data structure (table driven):

```
state = INITIAL;
for (s = input; *s != '\0'; s += 1)
   state = transition[state][s];
return isfinal[state];
```

# What Flex Does

- Flex program specification is giant regular expression of the form $R_1|R_2|\cdots|R_n$, where none of the $R_i$ match $\epsilon$.
- Each final state labeled with some action.
- Converted, by previous methods, into a table-driven DFA.
- But, this particular DFA is used to recognize *prefixes* of the (remaining) input: initial portions that put machine in a final state.
- Which final state(s) we end up in determine action. To deal with multiple actions:
  - Match *longest* prefix ("maximum munch").
  - If there are multiple matches, apply *first* rule in order.



# How Do They Do It?

- How can we use a DFA to recognize longest match?
- How can we use DFA to act on first of equal-length matches?
- How can we use a DFA to handle the $R_1/R_2$ pattern (matches just $R_1$ but only if followed by $R_2$, like $R_1$`(?=`$R_2$`)` in Python)?