# CMSC330: Finite State Machines

Chris Kauffman

*Last Updated:*
*Tue Sep 19 09:22:20 AM EDT 2023*

# Logistics

## Assignments

- Project 2 “RNA Transcription” Due 19-Sep
- Project 3 is cooking

## Goals

- ~~Recap of Regexs~~
- Finite State Machines
- Determinism vs Non-Determinism
- Regex to NFA
- NFA to DFA

## Reading

*Introduction to the Theory of Computation by Michael Sipser*

- Chapter 1 covers theory associated with Finite State Machines and their relation to Regular Expresssions
- For the theoretically inclined, treatment is much tighter w/ proofs than our in-class work

*Prof Bakalian’s Notes on FSM*

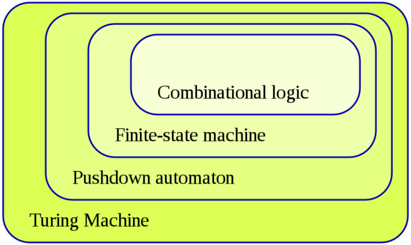
- A good summary of the topics we’ll cover
- Linked on course schedule

# Automata Theory

- Likely you've studied Boolean Logic in a previous class
- Allows the "computation" of certain outcomes based on inputs but has limits in power, does not amount to what a "computer" can do
- Example: cannot **recognize** Regular Expressions with Boolean Logic as Regexes can recognize infinite sets of strings
- **Automata Theory** is the branch of Math / CS that studies what (theoretical) machines with different properties can do
- By introducing notions of state (and time) one can build progressively more powerful machines

# Levels of Computational Power

- A full course on Automata Theory would study each level, comparing, contrasting, formalizing
- Wouldn't leave much time for other fun things like Python, OCaml, Racket...
- In CMSC 330, will study **Finite State Machines (FSM)** also known as **Finite Automata (FA)** as an example of one level of power that is useful in language processing and is connected to Regular Expressions



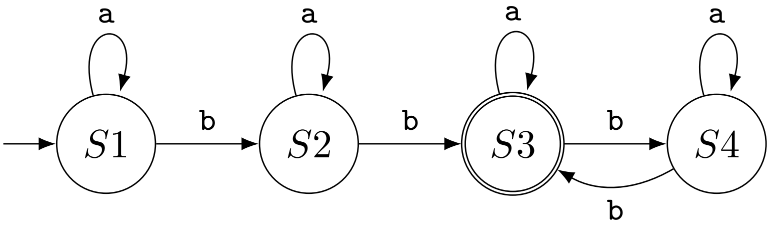

Source: Wikip "Automata Theory"

*The class of problems that can be solved grows with more powerful machines.*

# Even-Bs: A Leading Example

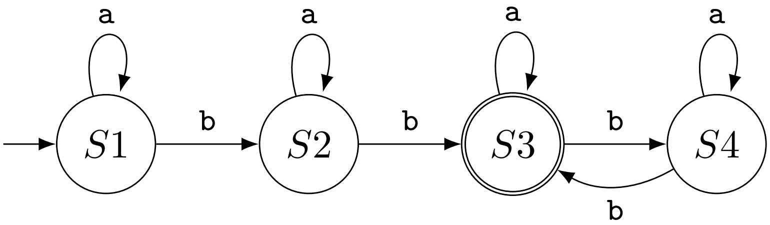
*Let* Even-Bs be the set of all strings composed of `a` and `b` with at least 2 `b`'s and an even number of `b`'s.

- Example members of Even-Bs are `bb, abb, aaababaa, abbabb, abba, babaaa, ...`
- **Regex** matching strings in Even-Bs: `(a*ba*ba*)+`
- **Deterministic Finite Automata (DFA)** recognizing Even-Bs

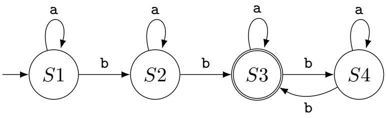


# DFA Diagram Notation

- DFAs are **mathematical graphs** comprised of vertices (circles) and directed edges (arrows between circles)
- Each circle is a **state**; there are a finite number of them
- Each edge / transition is labeled with at least one item from the **input alphabet** like `a` or `b`
- There is one **start state** $S1$ in this case; note the arrow to it
- There are one or more **accept states** which are drawn with 2 circles like $S3$



# Exercise: DFA Example Recognition / Rejection



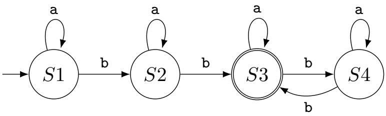

```
       v
input: abbabb
state: S1 a-> S1
        v
input: abbabb
state: S1 b-> S2
         v
input: abbabb
state: S2 b-> S3
          v
input: abbabb
state: S3 a-> S3
           v
input: abbabb
state: S3 b-> S4
            v
input: abbabb
state: S4 b-> S3
             v
input: abbabb
state: S3 ACCEPT
```

```
       v
input: bbaaba
state: S1 b-> S2
        v
input: bbaaba
state: S2 b-> S3
         v
input: bbaaba
state: S3 a-> S3
          v
input: bbaaba
state: S3 a-> S3
           v
input: bbaaba
state: S3 b-> S4
            v
input: bbaaba
state: S4 a-> S4
             v
input: bbaaba
state: S4 REJECT
```

```
       v
input: ababbba
???
???
```

Complete the state transitions

# **Answers**: DFA Example Recognition / Rejection



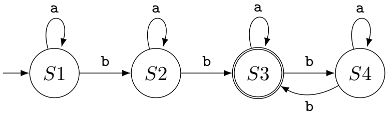

```
       v
input: abbabb
state: S1 a-> S1
        v
input: abbabb
state: S1 b-> S2
         v
input: abbabb
state: S2 b-> S3
          v
input: abbabb
state: S3 a-> S3
           v
input: abbabb
state: S3 b-> S4
            v
input: abbabb
state: S4 b-> S3
             v
input: abbabb
state: S3 ACCEPT
```

```
       v
input: bbaaba
state: S1 b-> S2
        v
input: bbaaba
state: S2 b-> S3
         v
input: bbaaba
state: S3 a-> S3
          v
input: bbaaba
state: S3 a-> S3
           v
input: bbaaba
state: S3 b-> S4
            v
input: bbaaba
state: S4 a-> S4
             v
input: bbaaba
state: S4 REJECT
```

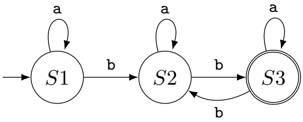
```
       v
input: ababbba
state: S1 a-> S1
        v
input: ababbba
state: S1 b-> S2
         v
input: ababbba
state: S2 a-> S2
          v
input: ababbba
state: S2 b-> S3
           v
input: ababbba
state: S3 b-> S4
            v
input: ababbba
state: S4 b-> S3
             v
input: ababbba
state: S3 a-> S3
              v
input: ababbba
state: S3 ACCEPT
```

# DFAs are Not Unique

## Even-Bs DFA #1




## Even-Bs DFA #2



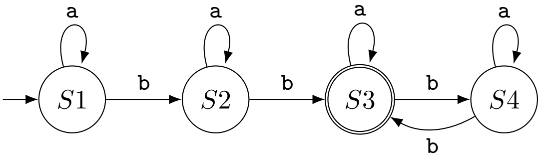

- Both these DFAs recognize the set Even-Bs but are shaped differently
- DFA Minimization finds a DFA which accepts the same input set but has a minimal number of states (subject to caveats)
- Regular Expressions are not unique either:

```
Even-Bs Regex 1: (a*ba*ba*)+
Even-Bs Regex 2: (a*ba*b)+a*
```

# Finite State Machine Formalisms

Formally, a FSM is a 5-tuple (e.g. 5 parts, order matters)

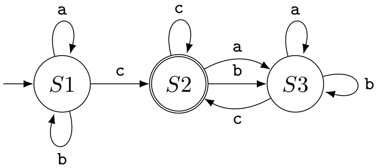
| | Description | Sym | Even-Bs DFA #1 |
|---|---|---|---|
| 1 | Alphabet: set of allowable characters | $\Sigma$ | $\{a,b\}$ |
| 2 | Set of States in FSM | $S$ | $S=\{S1,S2,S3,S4\}$ |
| 3 | Starting state of the FSM | $s_0$ | $S1$ |
| 4 | Set of Final / Accept States | $F$ | $\{S3\}$ |
| 5 | Set of transitions (labeled edges)[1] | $\delta$ | {(S1,a,S1), (S1,b,S2), (S2,a,S2), (S2,b,S3), (S3,a,S3), (S3,b,S4), (S4,a,S4), (S4,b,S3)} |




*Even-Bs DFA #1*

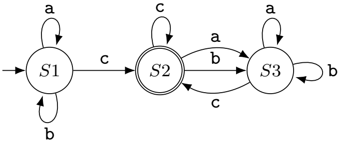
[1]The character $\delta$ is the lower-case Greek letter delta, often used to represent "change" as in a "change of state"; it's capital version is $\Delta$

# Exercise: DFA Practice

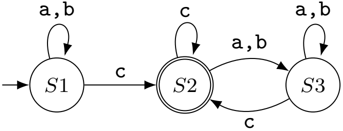



1. Show the formal 5-tuple of parts for this DFA
2. What set of strings does it accept?
3. Find a regular expression that matches that set
4. What set of strings does this Regex match?
   Regex: `[ab]*aab[ab]*`
5. Design a DFA that accepts the same set of strings

# **Answers**: DFA Practice

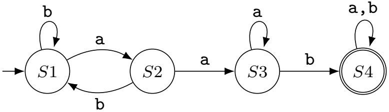



*Ends-C DFA*




*Ends-C DFA with Alt Notation*

1. Show the formal 5-tuple of parts for this DFA
```
1. Alphabet: {a,b,c}
2. States: {S1,S2,S3}
3. Start: S1
4. Accept: {S2}
5. Transitions:
   {(S1,a,S1),(S1,b,S1),(S1,c,S2),
    (S2,a,S3),(S2,b,S3),(S2,c,S2),
    (S3,a,S3),(S3,b,S3),(S3,c,S2)}
```
2. What set of strings does it accept?
   *Strings of `a,b,c` the end with `c`*
3. Find a regular expression that matches that set
   *Regex: `[abc]*c$`*
   *Note use of $ to denote end of input*
4. What set of strings does this Regex match?
   Regex: `[ab]*aab[ab]*`
   *Strings of `a,b` that contain the substring `aab`*
5. Design a DFA that accepts the same set of strings




*Has-AAB DFA*
*Adapted from Sipser Figure 1.13*

# DFAs in Code as Data Structures

```
# even_Bs_dfa.py:
even_Bs_dfa = {
  "alphabet":{"a","b"},
  "nstates":4,
  "start":1,
  "accept":{3},
  "trans":[{},
           {"a":1,"b":2},
           {"a":2,"b":3},
           {"a":3,"b":4},
           {"a":4,"b":3}],
}

def dfa_match(dfa,instr):
  state = dfa["start"]
  trans = dfa["trans"]
  for i in instr:
    if not i in dfa["alphabet"]:
      return "Error"
    state = trans[state][i]
  if state in dfa["accept"]:
    return "Accept!"
  else:
    return "Reject"
```

- Encode the 5 parts of the DFA in some sort of data structure
- Python's built-in Lists, Dictionaries, Sets make this pleasant
- `dfa_match(dfa,instr)` will return Accept / Reject string using DFAs encoded as the example above
- The general goal of compiling a regular expression is to produce this kind of data structure
- Study the data structure and explain its parts

# DFAs as Code

```c
// even_Bs_dfa.c:
int even_Bs_dfa(char *input){
  int pos=-1;
 S1:
  pos++;
  switch(input[pos]){
    case 'a':  goto S1;
    case 'b':  goto S2;
    case '\0': goto REJECT;
    default:   goto ERROR;
  }
 S2:
  pos++;
  switch(input[pos]){
    case 'a':  goto S2;
    case 'b':  goto S3;
    case '\0': goto REJECT;
    default:   goto ERROR;
  }
 S3:
  pos++;
  switch(input[pos]){
    case 'a':  goto S3;
    case 'b':  goto S4;
    case '\0': goto ACCEPT;
    default:   goto ERROR;
  }
 S4:
  pos++;
  switch(input[pos]){
```

- A common output option for parsing tools like Lex and Yacc is to encode state machines as positions in code
- Instruction Pointer is “state”
- Tools process a Regex or more complex language **Grammar** then generates C code that represents the state machine
- Generated C code is nigh impenetrable BUT compiles to much faster recognition routines than alternatives
- With all those `goto`'s, you know… *Here be Dragons*

# Formal Regular Expressions

- Introduced Regexs in code somewhat informally as a pattern matching device
- Formally, Regular Expressions are
  1. $\epsilon$: the Empty String (zero-length) (Greek Letter "epsilon")
  2. $\emptyset$: the empty set of no regexs
  3. Single item: like a from an alphabet $\Sigma = a, b$
  4. $R_1 R_2$: concatenation of two regexs
  5. $R_1 | R_2$: union / alternation of two regexs
  6. $R_1^*$: zero-or-more of a regex, its **Kleene Closure**[2]
- These 6 parts are minimal, allow construction of all the regex convenience mechanisms we've seen so far, and limit the cases of in formal proofs

  Ex: Shorthand: `[ab]+` Formal: $(a|b)(a|b)^*$

  Ex: Shorthand: `a?b+aa` Formal: $(a|\epsilon)bb^*aa$

[2] Named for Stephen Kleene who studied under Alonzo Church and contributed to the development of Church's Lambda Calculus

# Equivalence of FSM and Regular Expressions

**Definition:** A language is **Regular** if some Finite State Machine accepts it. The FSM may be either Deterministic or Non-deterministic.

Using a series of proofs one can show the following:

1. A language is Regular if and only if some **Regular Expression** describes it; *shown by giving a procedure to convert a Regular Expression to a Non-deterministic Finite Automata (NFA)*
2. Regular Expressions are closed under the 3 **regular operations** of concatenation, union, and star (Kleene closure) *e.g. all regexs that can exist can be built from simpler regexs with these ops*
3. Every NFA has an equivalent DFA; *procedures exist to convert NFAs to DFAs that accept the same language; we'll study this*

**Conclusion:** Regular Expressions and Finite State Machines are equivalent in power, allow recognition of identical sets

*If you want to see those proofs, grab a copy of Sipser's Introduction to the Theory of Computation*

# Nonregular Languages and the Limits Regexes/FSMs

- Before moving forward, note that Regexs / FSMs hit practical limits in power quickly and in cases we'd want to overcome
- Example: Let Equal-ABs be the set of all strings start some number $n$ of `a` characters and are followed immediately by $n$ `b` characters.
  - Equal-ABs $= \{a^n b^n | n > 0\}$
  - Equal-ABs = `{ab, aabb, aaabbb, aaaabbbb, ...}`
- Fool's Errands:
  - Construct a DFA to accept Equal-ABs
  - Write a Regex matching Equal-ABs
  - **No such DFA or Regex Exists**
- Why do we care? Well, a similar set is **Balanced-Paren**, the set of all strings that have properly balanced parentheses
  - Balanced-Paren = `{(), (()), ((())), ...}`
- One needs a more powerful machine than FSMs / Regexs to properly recognize Equal-ABs and Balanced-Parens which is crucial for processing programming languages

# Flow of "Compiling" Regexs

Given a Regular Expression $R$, the notion of "Compiling" it usual boils down to...

1. Use a procedure to convert it to a **Non-deterministic Finite Automata** $N$
2. Use $N$ for matching input directly OR
3. Use a procedure to convert[3] $N$ to a Deterministic Finite Automata $D$
4. Then match the input with $D$

Will examine each items and overview the procedures mentioned BUT an upcoming assignment will have you **code some of these procedures** to a get a feel for them

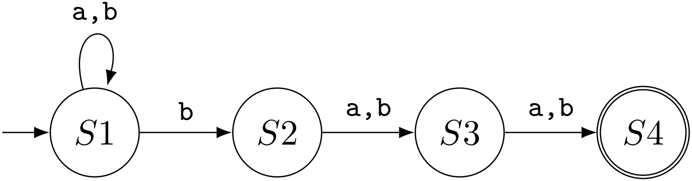
[3]There are also procedures to convert DFAs and NFAs into equivalent Regexs. Not so useful in computing practice but useful to prove the equivalence of FSMs and Regexs. They are covered in Sipser's textbook.

# Non-Deterministic Finite Automata: Differences 1

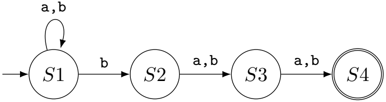
- First difference from DFAs: relax constraint of "every state has one edge for every member of the alphabet"
  - Input chars may appear on multiple edges: choices
  - Some states may not transition from every input
- Input is accepted if **some path exists** for the input to an accept state for the entire input
- When there are two transitions with `a` on it, try both: e.g. **search for an accepting path**

Consider the Regex `(a|b)*b(a|b)(a|b)`: strings of `a,b` with `b` in the third to last position; name that set of strings B-Third-Last.

## NFA Recognizing B-Third-Last: `[ab]*b[ab]{2}`



# NFA Example Recognition of B-Third-Last: Search Tree

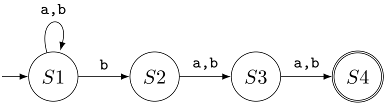



```
       V
input: ababbaa
state: S1 a-> S1
        V
input: ababbaa
state: S1 b-> S1,S2
PICK S1--------------------------------------------------------------PICK S2
         V                                                                    V
input: ababbaa                                                       input: ababbaa
state: S1 a-> S1                                                     state: S2 a-> S3
          V                                                                    V
input: ababbaa                                                       input: ababbaa
state: S1 a-> S1,S2                                                  state: S3 b-> S4
PICK S1-----------------------------------------PICK S2                        V
           V                                               V         input: ababbaa
input: ababbaa                                  input: ababbaa       state: S4 b-> REJECT
state: S1 b-> S1,S2                             state: S2 b-> S3     No b-trans for S4
PICK S1--------------------PICK S2                          V
            V                          V                input: ababbaa
input: ababbaa             input: ababbaa       state: S3 a-> S4
state: S1 a-> S1           state: S2 a-> S3                  V
             V                          V       input: ababbaa
input: ababbaa             input: ababbaa       state: S4 a-> REJECT
state: S1 a-> S1           state: S3 a-> S4     No a-trans for S4
              V                          V
input: ababbaa             input: ababbaa
state: S1 REJECT           state: S4 ACCEPT!
```

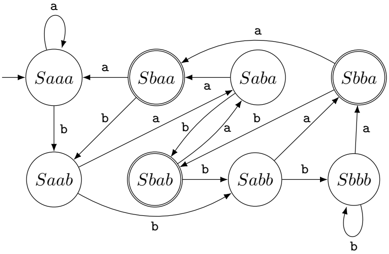
# Why DFA vs NFA?

- DFAs involve no choices as they check input, computational benefits, may have a large number of states, more difficult to convert Regex directly to a DFA
- NFAs allow **choices** which induces the need to **search**, computationally more cumbersome, easier to convert Regexs to NFAs, can be converted to DFAs

## NFA Accepting B-Third-Last



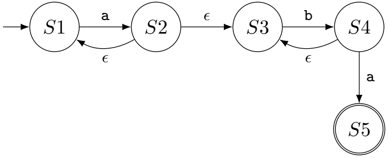

## DFA Accepting B-Third-Last



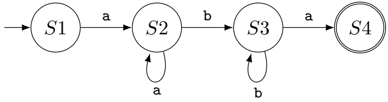
# NFA Differences 2: Epsilon Transitions

- Recall $\epsilon$ is the empty string, a Regex itself and a sort of "special" character
- Second difference of NFAs from DFAs: allow epsilon transitions ($\epsilon$-transitions) between states along $\epsilon$-edges
  - Consumes no input
  - Change state without affecting input position
- Example: Consider the Regex `a+b+a` (formal $aa^*bb^*a$)
- Here are two NFAs which accept the same Regex

## With $\epsilon$-Transitions

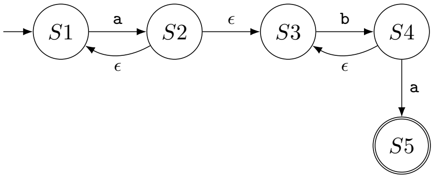



## No $\epsilon$-Transitions



# NFA Recognition with Epsilon Transitions

```
       V
input: aaabba
state: S1 a-> S2
        V
input: aaabba
state: S2 a-> REJECT
       S2 eps-> S1
        V
input: aaabba
state: S1 a-> S2
         V
input: aaabba
state: S2 a-> REJECT
state: S2 eps-> S1
         V
input: aaabba
state: S1 a-> S2
          V
input: aaabba
state: S2 eps-> S3
           V
input: aaabba
state: S3 b-> S4
           V
input: aaabba
state: S4 b-> REJECT
state: S4 eps-> S3
           V
input: aaabba
state: S3 b-> S4
            V                   V
input: aaabba      input: aaabba
state: S4 a-> S5   state: S5 ACCEPT
```




*NFA which accepts* `a+b+a` *using $\epsilon$-transitions*

- In this simple example, only choices are `REJECT` or take the $\epsilon$-transitions
- Taking $\epsilon$-transitions change states without affecting input
- In more complex NFAs, a state may have valid input character transitions and $\epsilon$-transitions; requires searching all possible paths for an `ACCEPT` sequence

# Why Allow $\epsilon$-Transitions?

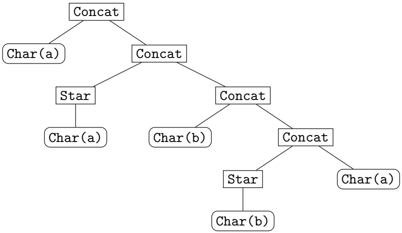
- $\epsilon$-transitions don't add any additional power to NFAs BUT...
- They make it much easier to convert Regexs to NFAs
- Recall the 3 operators that construct a larger Regex from a smaller ones
  - $R_1 R_2$: Concatenation
  - $R_1 | R_2$: Union
  - $R_1^*$: Star (Kleene Closure)
- Each uses $\epsilon$-transitions during Regex to NFA conversion

# Regex to NFA Conversion: Parse Trees

- Idea behind conversion procedure is easier to understand with a **parse tree** for a regular expression
- Is implied by the formal definition of a Regular Expression but enlightening to look examples explicitly
- Shown are both Drawings and a Code-like constructions

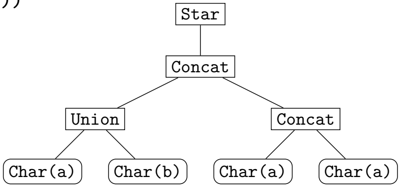
## Parse Tree for $aa^*bb^*a$

```
Concat( Char(a),
        Concat( Star( Char(a) ),
                Concat( Char(b),
                        Concat( Star( Char(b) ),
                                Char(a)))))
```




## Parse Tree for $((a|b)aa)^*$

```
Star( Concat( Union( Char(a),
                     Char(b))
              Concat( Char(a),
                      Char(a))))
```



# Principls of Regex to NFA Conversions

- Each of the constructs comprising Regular Expressions has an NFA equivalent
- Typically work bottom up on the the Regex parse tree converting leaves to small NFAs, then combining those on the way up through interior nodes
  - **Recursion** helps a lot with this
  - Convert all child trees to NFAs recursively, combine/alter the child NFAs according to the interior node's operation
- Operations like Union, Concatenation, and Star may introduce additional states and use $\epsilon$-transitions to "glue" smaller NFAs together
- When the Root of the parse tree is finished, have a single NFA which will Accept all strings the Regex matches
- This process is the basis for the *constructive* proof that Regexs and FSMs are equivalent

# Example Regex to NFA Conversion

This is somewhat involved and is shown in a separate linked handout which looks like the nearby miniaturized version. It outlines the process on a specific example describing how `Char(x)`, `Union(x,y)`, `Concat(x,y)`, `Star(x)` are converted to NFAs. The handout is near to where this slide is located.

***A Sample Regex to NFA Conversion***

*UMD CMSC330 - Kauffman*

The parse tree for following formal regex is shown nearby.

`((a|b)aa)*`

In a program, it would likely be written with some shorthand conventions like this:

`([ab]aa)*`

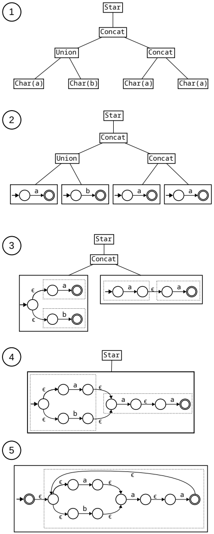
In a bottom up conversion, the leaf nodes which are Char() parts of the Regex can be converted to 2-state NFAs which Accept after reading the single input character indicated

(Left branch) The Union of two NFAs is constructed by introducing a new start state with ϵ-edges to the two other NFA start states. Accept states for both sub-NFAs become accept states in the union.

(Right branch) Concatenation switches all of the first NFA's accept states non-accepting, then connects them to the second NFA's start state with an ϵ-edge.

A second concatenation follows.

Star (Kleen Closure) introduces a new Start state which is also an Accept state. This is connected to the sub-NFA's start state with an ϵ-edge. Finally, all Accept states are connected to the original Start state with an ϵ-edge.



# Parse Trees are Handy

- Parse Tree shows a graphical structure for the Regex
- Makes the order of what to convert when more obvious
- Parse Trees or **Abstract Syntax Trees** will be handy elsewhere in the course

## But where to parse trees come from?

- **Construct them explicitly** using construction functions like

  `Concat(Star(Char(a)), Char(b))`

  Useful in beginner projects like one we are cooking for you now
- **Process the Regex language** to construct the tree, more difficult as need to establish the allowable syntax, semantics of your Regex language, parse them, etc. Regexs are often used in language processing...

*But if I'm building a Regex language processor and need a Regex processor to do it, aren't I stuck?*

- This is the same problem as writing a C compiler in the C language: the first C compiler was written in something else.

# Conversion from NFA to DFA

- Can work with NFA's to do Regex matching but this requires a more complex matching routine that supports search
- Likely upcoming project: Regex to NFA convesion + NFA matching routine - "good enough"
- In many cases it is worthwhile to convert the NFA to a DFA for more efficient matching
- There is a "standard" way to convert NFAs to DFAs along with slightly optimized "lazy" procedure; will discuss both

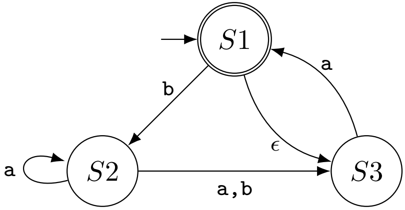
# Standard NFA to DFA Conversion

Standard / "Dumb" Conversion of NFA to a DFA proceeds in these steps

1. Create one state in the DFA for each element of the **Power Set** of NFA states (Subset Construction)
2. DFA Starts at the state $\epsilon$-Closure of NFA's start state
3. DFA Accept states are any that contain a DFA end state
4. DFA transitions are the $\epsilon$**-closure** of transitions between NFA states

## NFA "N4" to Convert

Regex: `((ba*[ab]a)|a)*`

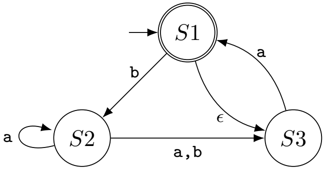



1. Alphabet: $\{a, b\}$
2. States: $\{S1, S2, S3\}$
3. Start: $S1$
4. Accept: $\{S2\}$
5. Transitions:
$$\{(S1, \epsilon, S3), (S1, b, S2),$$
$$(S2, a, S2), (S2, a, S3),$$

# NFA to DFA: States via Power Set

- The **Power Set** of a set is the set of all possible subsets
- Has $2^n$ elements in it
- Initial DFA states are labeled with power set of NFA states

## NFA "N4" to Convert




$$
\begin{aligned}
States(N4) &= \{S1, S2, S3\} \\
States(D4) &= \; Pow(States(N4)) \\
&= \{\emptyset, \{S1\}, \{S2\}, \{S3\}, \\
&\quad \{S1, S2\}, \{S1, S3\}, \\
&\quad \{S2, S3\}, \{S1, S2, S3\}\}
\end{aligned}
$$

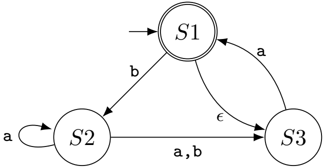
## D4 States: Power Set of N4 States

| | | | |
|---|---|---|---|
| $T_\emptyset = \emptyset$ | $T_1 = \{S1\}$ | $T_2 = \{S2\}$ | $T_{12} = \{S1, S2\}$ |
| $T_3 = \{S3\}$ | $T_{13} = \{S1, S3\}$ | $T_{23} = \{S2, S3\}$ | $T_{123} = \{S1, S2, S3\}$ |

# DFA to NFA: Epsilon-Closure of a Transition

- The $\epsilon$-Closure of a state $S_x$ is the set of states that can be reached from $S_x$ using only $\epsilon$-transitions including $S_x$ itself
- $\epsilon$-Closure of a set of states is the set which can be reached via only $\epsilon$-edges from any of them
- An important concept to complete DFA to NFA conversion
- In N4, the only significant $\epsilon$-Closure is for $S1$ which can transition to $S3$ on an $\epsilon$-edge

## NFA N4



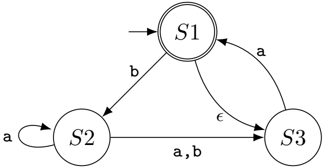

## Epsilon Closure Examples

$$\begin{aligned}
\epsilon_{clos}(S1) =& \{S1, S3\} \\
\epsilon_{clos}(S2) =& \{S2\} \\
\epsilon_{clos}(S3) =& \{S3\} \\
\epsilon_{clos}(\{S1, S2\}) =& \{S1, S2, S3\} \\
\epsilon_{clos}(\{S1, S3\}) =& \{S1, S3\} \\
\epsilon_{clos}(\{S1, S2, S3\}) =& \{S1, S2, S3\}
\end{aligned}$$

# NFA to DFA: Initial and Final States

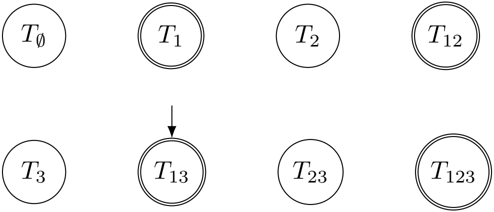
- DFA Initial State: state labeled as $\epsilon$-Closure of NFA start state
- DFA Accept States: any with label containing NFA accept

## NFA "N4" to Convert




$$Start(N4) = S1$$
$$Start(D4) = \{S1\} = T_1$$
$$Accept(N4) = \{S1\}$$
$$Accept(D4) = \{T_1, T_{12}, T_{13}, T_{123}\}$$

## D4 Initial and Final States Assigned



# NFA to DFA: Transitions in DFA

To determine the transition for DFA $D$'s state $T_z = \{S_i, S_j, ...\}$ for alphabet letter $x$

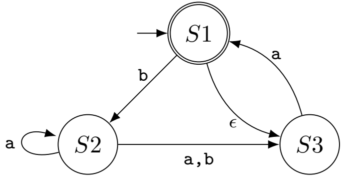
- Initialize an empty destination set: $dest \leftarrow \{\}$
- Consider $S_i$ which is associated with $T_z$
- In the NFA $N$, find all states $R_x$ connected to $S_i$ via an x-edge, e.g. all states of the form $(S_i, x, R_x)$
- Let this set be $R$
- Add the epsilon closure of $R$ to $dest$; $dest \leftarrow dest \cup \epsilon_{clos}(R)$
- Then consider $S_j$ associated with $T_z$ and do the same
- Quit when through with all of $S_i, S_j, \ldots$
- $dest$ is now a set of states like $\{S1, S5, S7, S8\}$
- Add the edge $(T_z, x, T_{1578})$ to the transitions for $D$
- If $dest$ is empty, add the edge $(T_z, x, T_\emptyset)$

Repeat this process for every state / alphabet pair in $D$ to complete the transitions.

For all $x$ in alphabet, add edges $(T_\emptyset, x, T_\emptyset)$ e.g. "garbage state"

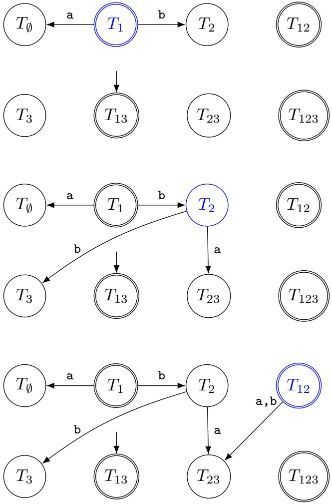
# NFA to DFA: Transitions Example 1 / 3

## NFA4 being Converted



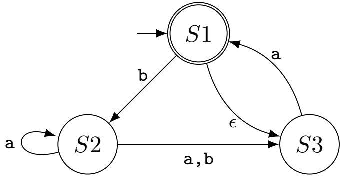

- $S1$ has no a-edge in NFA4, $T_1$ to $T_0$ in DFA4
- $S2$ transitions to either $S2$ or $S3$ on an a-edge: $dest = \{S2, S3\}$ so $(T_2, a, T_{23})$ in DFA4
- $T_{12}$ for alphabet letters is
  - $a$: $\emptyset$ for $S1$, $\{S2, S3\}$ for $S2$; so $dest = \{S2, S3\}$
  - $b$: $S2$ for $S1$, $S3$ for $S2$, so $dest = \{S2, S3\}$

## DFA4 Adding Transitions

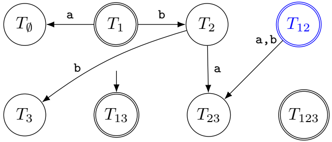


# Exercise NFA to DFA: Transitions Example 2 / 3

## NFA4 being Converted




## DFA4 Adding Transitions



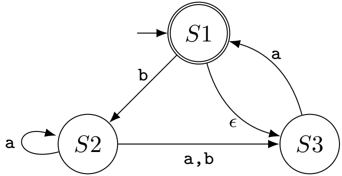

Determine where the following transitions should be added to DFA4 states:

1. $(T3, a, ??)$
2. $(T3, b, ??)$
3. $(T_{13}, a, ??)$
4. $(T_{13}, b, ??)$

Explain why how the destination was determined in each case

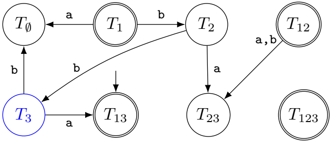
# **Solution**: NFA to DFA: Transitions Example 2 / 3

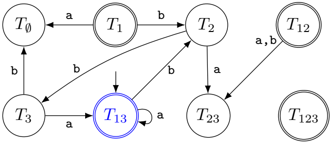
## NFA4 being Converted




- $T_3, a$: $S3$ a-edge to $S1$ PLUS an $\epsilon$-edge back to $S3$; so $dest = \{S1, S3\}$
- $T_3, b$: $S3$ has no b-edge $dest = \emptyset$
- $T_{13}, a$: No a-edge from S1, $(S3, a, S1)$ with $\epsilon_{clos}(S1) = \{S1, S3\} = dest$
- $T_{13}, b$: $(S1, b, S2)$, no $S3$ b-edge, $dest = \{S2\}$

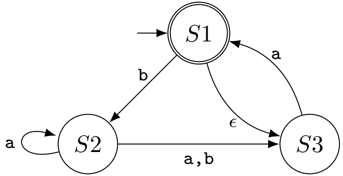
## DFA4 Adding Transitions



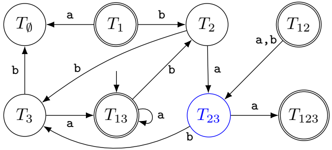
# NFA to DFA: Transitions Example 3 / 3

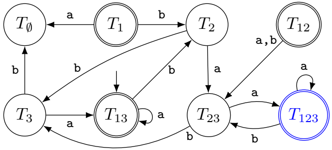
## NFA4 being Converted

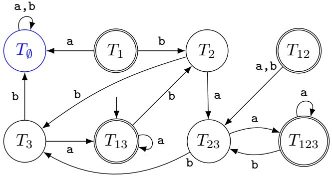



- Similar reasoning for $T_{23}, T_{123}$
- Loop on $T_\emptyset$ for all alphabet chars; represents failure from DFA not having a valid transition

## DFA4 Adding Transitions



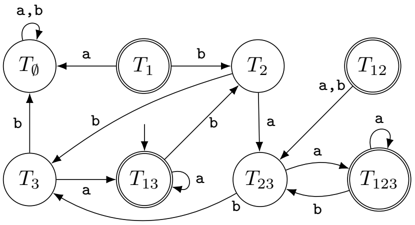




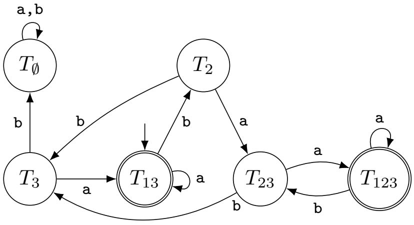
# NFA to DFA: State Elimination

- Some states are **unreachable** from the start state for any possible input so do not have any practical effect
- Example: $T_1, T_{12}$ have no incoming edges
- Can be detected via directed **graph traversal** from start state
- Eliminate unreachable “dead states” and their transitions

## Original Complete DFA4




## Dead States Eliminated



# Exercise NFA to DFA: Pseudocode for Transitions

- Loose Pythonic pseudocode for the "standard" DFA algorithm is given below
- What is the big-O complexity (approximately) of each loop?
- Of the code overall?

```
for every T in DFA.states:                # O(??)
  for every x in DFA.alphabet:            # O(??)
    dest = set()
    for every S in T:                     # O(??)
      R = NFA.trans[S].get(x,set())
      dest.union(eclosure(R))             # O(??)
    DFA.trans[T][x] = DFA.state_names[dest]
eliminate_dead_states(DFA)
```

# **Answers** NFA to DFA: Pseudocode for Transitions

- Loose Pythonic pseudocode for the “standard” DFA algorithm is given below
- Note its complexity is high in this “standard” approach

```
for every T in DFA.states:                # 2^n states
  for every x in DFA.alphabet:            # len(DFA.alphabet)
    dest = set()
    for every S in T:                     # could be n states
      R = NFA.trans[S].get(x,set())
      dest.union(eclosure(R))             # union is not O(1)
    DFA.trans[T][x] = DFA.state_names[dest]
eliminate_dead_states(DFA)
```

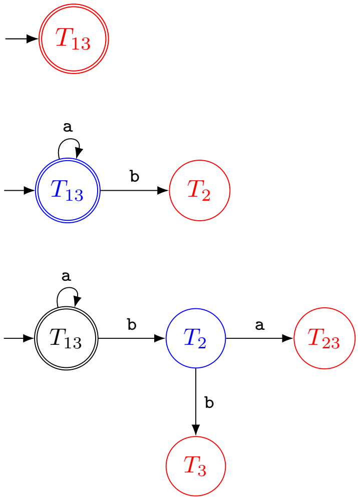
- Algorithm works but has HIGH complexity: $O(2^n * len(alphabet))$
- Leads to alternative “on demand” algorithm…

# NFA to DFA: Algorithmic Improvements

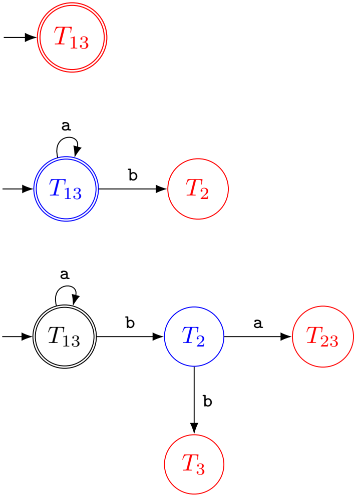
- Rather than immediately add all possible DFA states, add them only “as needed” or “as discovered” or “on demand”
- Avoids the immediate cost of adding $2^n$ states
- Won’t add dead states as no edges connect them
- Generally more practical than the “standard” method

# NFA to DFA: On Demand Algorithm 1 / 2

- Track two collections of states
  - **Completed (black)**
  - Todo (red)
- Start by adding only the start state as a Todo state
- Each iteration, select one Active (blue) state from the Todo states
- Determine Active state's transitions for all alphabet letters
- Any transition to a state not already seen adds to Todo
  - $T_{13}$: $b$ goes to $T_2$ which is added to Todo
  - $T_2$: transitions add $T_{23}$ and $T_3$ to Todo

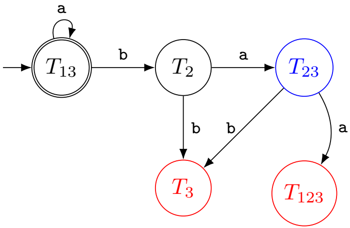


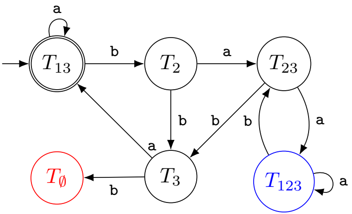
# Exercise NFA to DFA: On Demand Algorithm 2 / 2

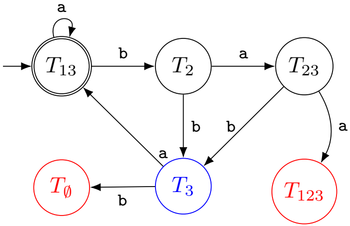


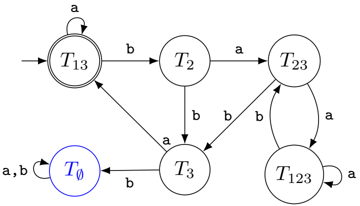

- Complete the execution of the on-demand algorithm adding states transitions for a Todo state and adding states as they are "discovered"
- Start with $T_{23}$ as the Active state

# **Answers** NFA to DFA: On Demand Algorithm 2 / 2



# NFA to DFA: On Demand Final

- While slightly trickier to implement, the On-Demand method is much more practical
- Resulting DFA shown nearby is equivalent to that constructed via Standard method after dead-state elimination
- You may implement the On-Demand conversion procedure in a future project