CMSC330: Finite State Machines

Chris Kauffman

Last Updated:
Wed Sep 13 07:10:01 PM EDT 2023

Logistics

Assignments

- ▶ Project 1 "Intro Python" Due Sun 10-Sep
- ▶ First Discussion Quiz during Discussion on Fri 15-Sep
  - ▶ 20min at beginning of discussion sections
  - ▶ Paper quiz, write answers, hand it in

Goals

- ▶ Recap of Regexs
- ▶ Finite State Machines
- ▶ Determinism vs Non-Determinism

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 soon

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
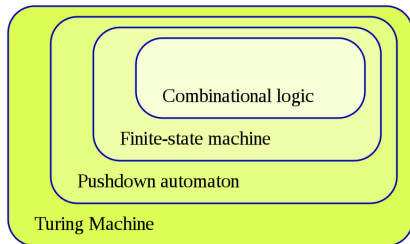

Source: Wikip "Automata Theory"

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
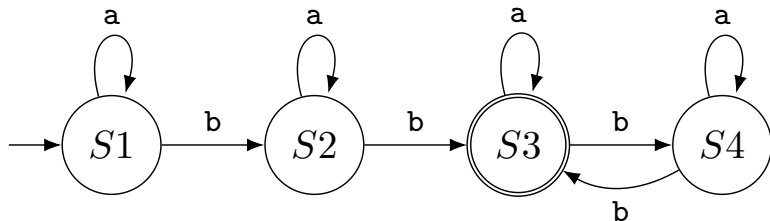
DFA Diagram Notation

- ▶ 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 $S3$
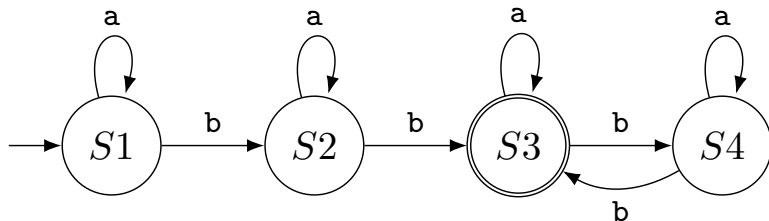
Exercise: DFA Example Recognition / Rejection
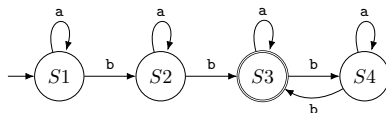

```
          v
input: abbabb
state: S1 a-> S1
          v
input: abbabb
state: S1 b-> S2
          v
input: abbabb
state: S2 b-> S3
          v
input: abbabb
state: S3 a-> S3
          v
input: abbabb
state: S3 b-> S4
          v
input: abbabb
state: S4 b-> S3
          v
input: abbabb
state: S3 ACCEPT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

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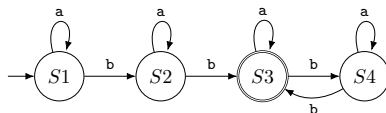
```
          v
input: ababbba
???
???
```

Complete the state transitions

Answers: DFA Example Recognition / Rejection


```
        v                    v                    v
input: abbabb        input: bbaaba        input: ababbba
state: S1 a-> S1     state: S1 b-> S2     state: S1 a-> S1
        v                    v                    v
input: abbabb        input: bbaaba        input: ababbba
state: S1 b-> S2     state: S2 b-> S3     state: S1 b-> S2
         v                    v                    v
input: abbabb        input: bbaaba        input: ababbba
state: S2 b-> S3     state: S3 a-> S3     state: S2 a-> S2
          v                    v                    v
input: abbabb        input: bbaaba        input: ababbba
state: S3 a-> S3     state: S3 a-> S3     state: S2 b-> S3
           v                    v                    v
input: abbabb        input: bbaaba        input: ababbba
state: S3 b-> S4     state: S3 b-> S4     state: S3 b-> S4
            v                    v                    v
input: abbabb        input: bbaaba        input: ababbba
state: S4 b-> S3     state: S4 a-> S4     state: S4 b-> S3
             v                    v                    v                    v
input: abbabb        input: bbaaba        input: ababbba        input: ababbba
state: S3 ACCEPT     state: S4 REJECT     state: S3 a-> S3      state: S3 ACCEPT
```
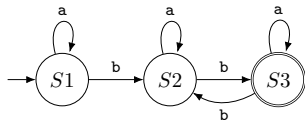
8

DFAs are Not Unique

Even-Bs DFA #1
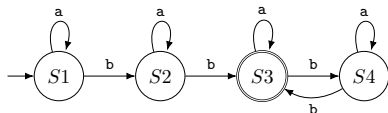
Even-Bs DFA #2

- ▶ 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	Σ	$\{a, b\}$
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]	δ	$\{$(S1,a,S1), (S1,b,S2), (S2,a,S2), (S2,b,S3), (S3,a,S3), (S3,b,S4), (S4,a,S4), (S4,b,S3)$\}$
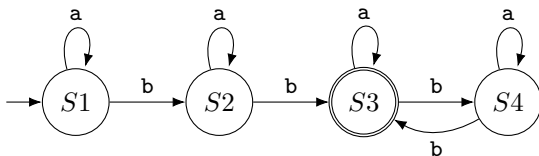
Even-Bs DFA #1

[1]The character δ is the lower-case Greek letter delta, often used to represent "change" as in a "change of state"; it's capital version is Δ
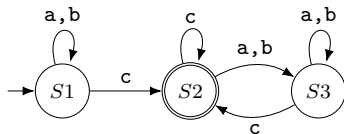
Exercise: DFA Practice
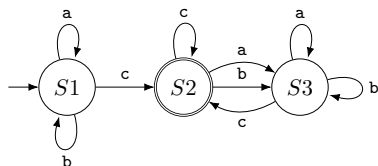
1. Show the formal 5-tuple of parts for this DFA

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
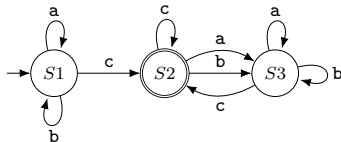
Ends-C DFA

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
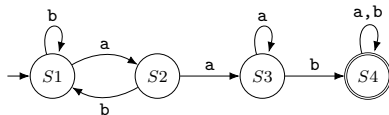
Has-AAB DFA
Adapted from Sipser Figure 1.13

12

DFAs in Code as Data Structures

```python
1  # even_Bs_dfa.py:
2  even_Bs_dfa = {
3    "alphabet":{"a","b"},
4    "nstates":4,
5    "start":1,
6    "accept":{3},
7    "trans":[{},
8             {"a":1,"b":2},
9             {"a":2,"b":3},
10            {"a":3,"b":4},
11            {"a":4,"b":3}],
12 }
13
14 def dfa_match(dfa,instr):
15   state = dfa["start"]
16   trans = dfa["trans"]
17   for i in instr:
18     if not i in dfa["alphabet"]:
19       return "Error"
20     state = trans[state][i]
21   if state in dfa["accept"]:
22     return "Accept!"
23   else:
24     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. ϵ: 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}
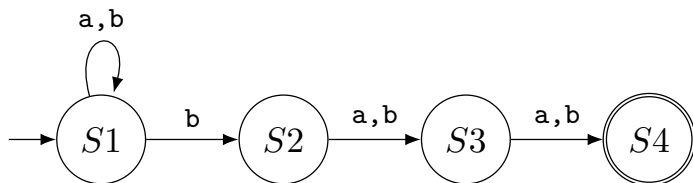
NFA Example Recognition of B-Third-Last: Search Tree
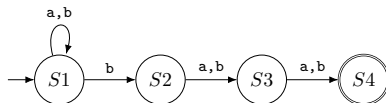

```
        V
input: abatbaa
input: ababbaa
state: S1 a-> S1
        V
input: ababbaa
state: S1 b-> S1,S2
PICK S1---------------------------------------------------------PICK S2
        V                                                            V
input: ababbaa                                          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!
```
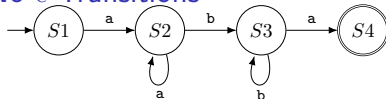
NFA Differences 2: Epsilon Transitions

- ▶ Second difference for DFAs: allow epsilon transitions between states along ϵ edges
  - ▶ Consumes no input
  - ▶ Change state without affecting input position
- ▶ Recall: ϵ is the empty string: change states without consuming input
- ▶ Example: Consider the Regex a+b+a (formal aa^*bb^*a)
- ▶ Here are two NFAs which accept the same Regex

With ϵ-Transitions
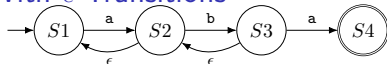
No ϵ-Transitions

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 b-> S3
            V
input: aaabba
state: S3 b-> REJECT
state: S3 eps-> S2
            V
input: aaabba
state: S2 b-> S3
             V
input: aaabba
state: S3 a-> S4
              V
input: aaabba
state: S4 Accept
```
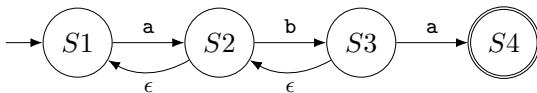

- ▶ In this simple example, only choices are REJECT or take the ϵ-transitions
- ▶ Taking the ϵ-transitions do not change the input position: change states without affecting input
- ▶ In more complex NFAs, may be several transition choices from a state, both input character transitions and ϵ-transitions; requires searching all possible paths for an ACCEPT sequence

Why Allow ϵ-Transitions?

- ϵ-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 ϵ-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)))))
```
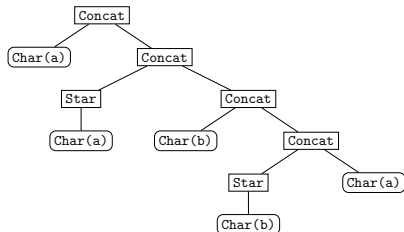

Parse Tree for $((a|b)aa)^*$

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


24
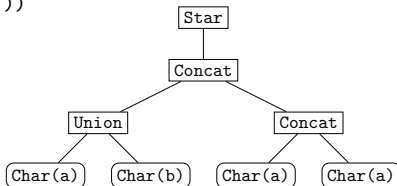
Principls of Regex to NFA Conversions

▶ Each of the constructs comprising Regular Expressions has an NFA equivalent

▶ Typically 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 ϵ-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

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
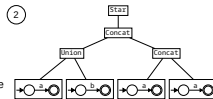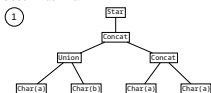UMD CMSC330 - Kauffman

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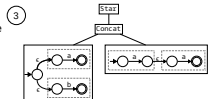
(Right branch) Concatenation switches all of the first NFA's accept states non-accepting, then connects them to the second NFA's start state with an ε-edge.
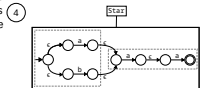
A second concatenation follows.

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.
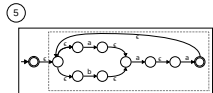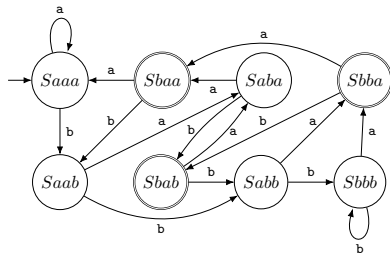
Why DFA vs NFA?

▶ 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
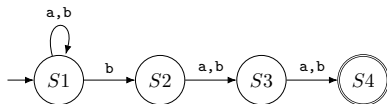
DFA Accepting B-Third-Last

–END Thu 13-Sep Topics–

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
- ▶ What follows is a an outline of the procedure to do that conversion

Other Uses for Finite State Machines

Regexs in Other Languages