CS 3100, Models of Computation, Spring 20, Lec 9

Ganesh Gopalakrishnan
School of Computing
University of Utah
Salt Lake City, UT 84112

bit.ly/3100s20Syllabus

SCHOOL OF COMPUTING
THE UNIVERSITY OF UTAH

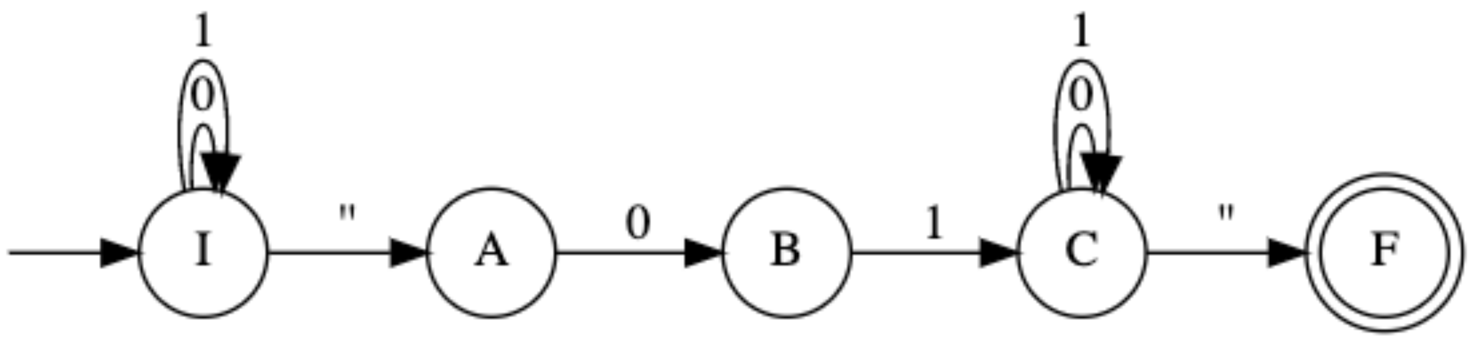
Lecture 9, covering Ch 7,8


Practice using
CH7/CH7.ipynb and then CH8-9/CH8-9.ipynb

Concepts around NFA, DFA, RE, and Applications

- NFA allow regular languages to be specified succinctly

E.g. NFA for “strings that contain 01” (one of many designs)



Concepts around NFA, DFA, RE, and Applications

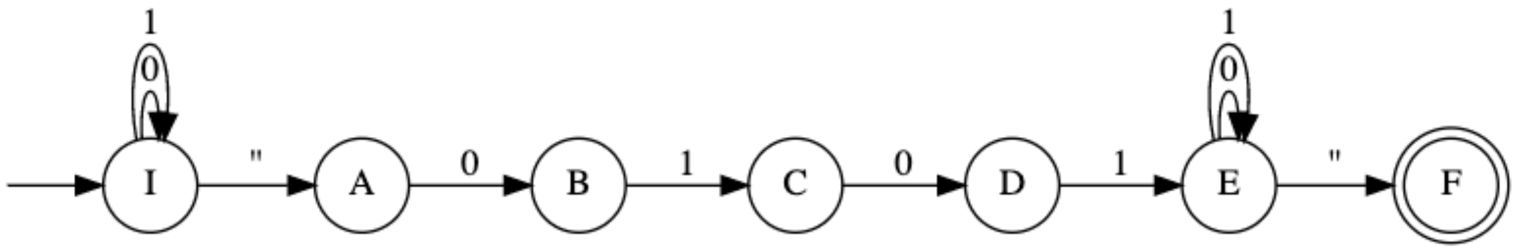
- NFA allow regular languages to be specified succinctly

E.g. NFA for “strings that contain 0101”

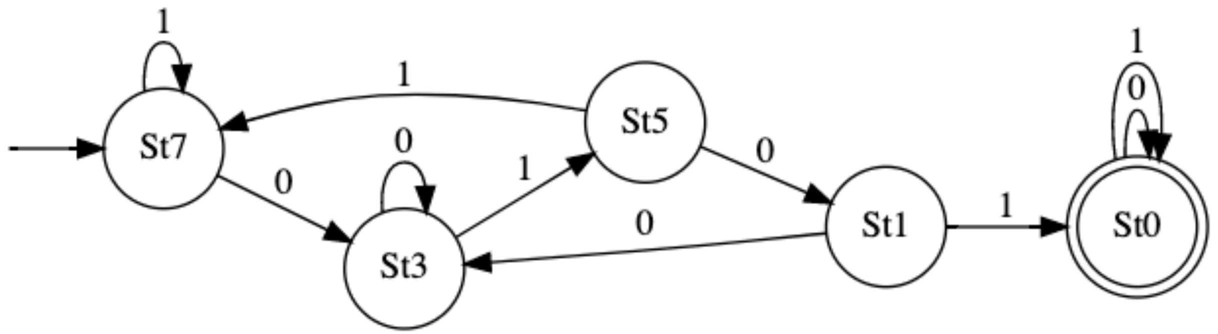
One NFA for “contains 0101”

```
nfahas0101 = md2mc('''
NFA
I : 0 | 1 -> I
I : '' -> A
A : 0  -> B
B : 1  -> C
C : 0  -> D
D : 1  -> E
E : 0 | 1 -> E
E : '' -> F
''')
```

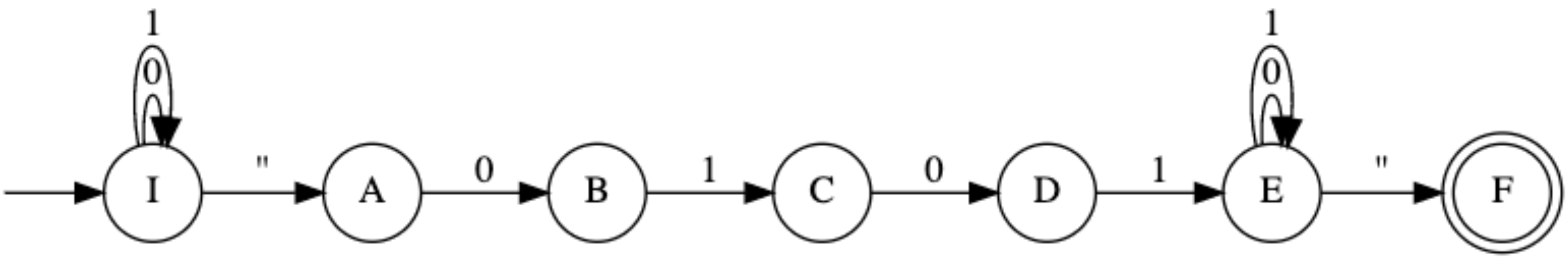
```
dotObj_nfa(nfahas0101)
```



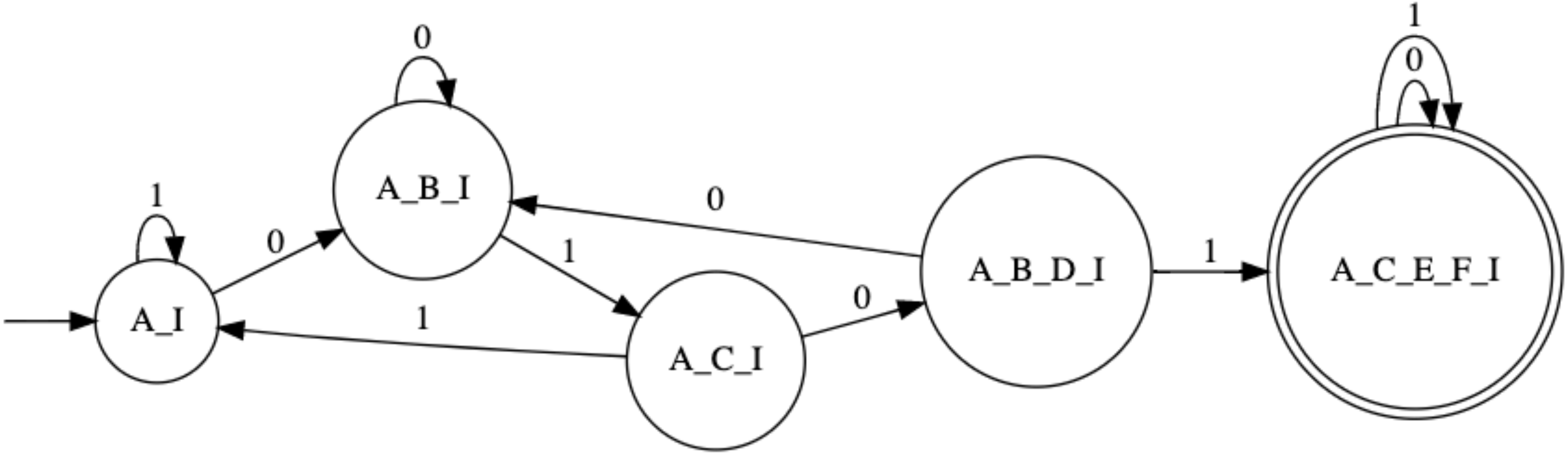

```
dotObj_dfa(min_dfa(nfa2dfa(nfahas0101)))
```



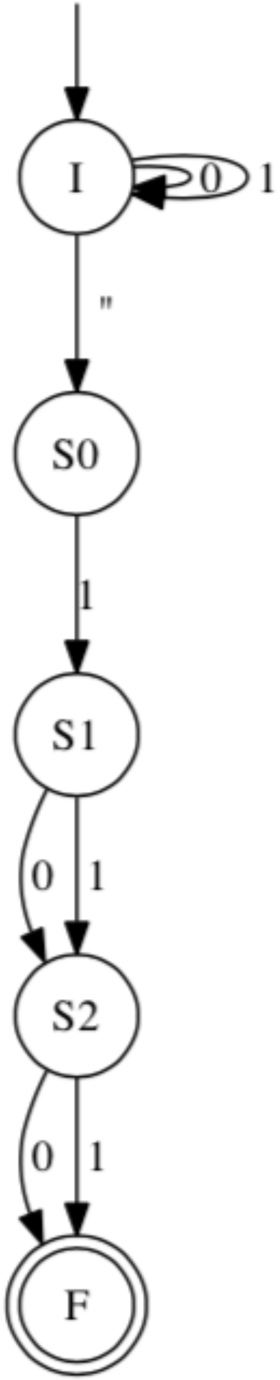
```
dotObj_nfa(nfahas0101)
```




```
dotObj_dfa(min_dfa(nfa2dfa(nfahas0101, STATENAME_MAXSIZE = 50)), STATENAME_MAXSIZE = 50)
```



What is an NFA formally?



Let Σ_ε stand for $(\Sigma \cup \{\varepsilon\})$. An NFA N is a structure $(Q, \Sigma, \delta, Q_0, F)$, where:

- Q is a *finite non-empty* set of states (as with DFA);
- Σ is a *finite non-empty* alphabet (as with DFA);
- $\delta : Q \times \Sigma_\varepsilon \rightarrow \mathscr{P}(Q)$, is a transition function. An NFA's δ function takes a state in Q and a symbol or ε and returns a *set of states* (which is a member of $\mathscr{P}(Q)$, the *Powerset* of Q). See Figure 7.4 for the state transition table 'δ' for the example NFA.
- $Q_0 \subseteq Q$ is a *set of initial states*; and
- $F \subseteq Q$, is a *finite, possibly empty* set of final states.

State	Next state upon inputs		
	0	1	ε
I	$\{I\}$	$\{I\}$	$\{S0\}$
S0	{}	$\{S1\}$	{}
S1	$\{S2\}$	$\{S2\}$	{}
S2	$\{F\}$	$\{F\}$	{}
F	{}	{}	{}

```
{'Q'    : {'F', 'I',
           'S0','S1','S2'},
 'Sigma': {'0', '1'},
 'Delta':
 {('I', '0')  : {'I'},
  ('I', '1')  : {'I'},
  ('I', '')   : {'S0'},
  ('S0', '1') : {'S1'},
  ('S1', '0') : {'S2'},
  ('S1', '1') : {'S2'},
  ('S2', '0') : {'F'},
  ('S2', '1') : {'F'}},
  'q0': {'I'},
  'F' : {'F'}}
```

NFA to DFA Conversion

Algorithm for Subset Construction:

- Input: An NFA $N = (Q, \Sigma, \delta, Q_0, F)$
- Output: A language-equivalent DFA D
- Method: **Subset Construction**
  - Add the Eclosure of the initial state of the NFA as an unexpanded state of the DFA D being built. This would also be the **initial state of the DFA being built.**

    Repeat

    Choose a state S of D that has not been expanded

    Expand(S)

    Until there are no more unexpanded states in D
  - **Expand**(S):

    Mark S as expanded;

    If $S \cap F \neq \emptyset$, **record** S **to be a final state of the DFA**

    For each symbol c in Σ

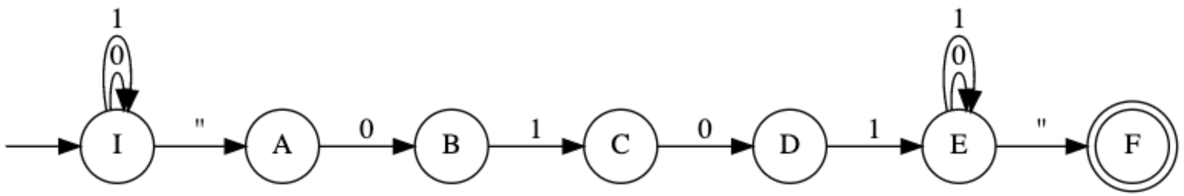
    For each state $s \in S$ do

    Let $s_c = \delta(s, c)$;

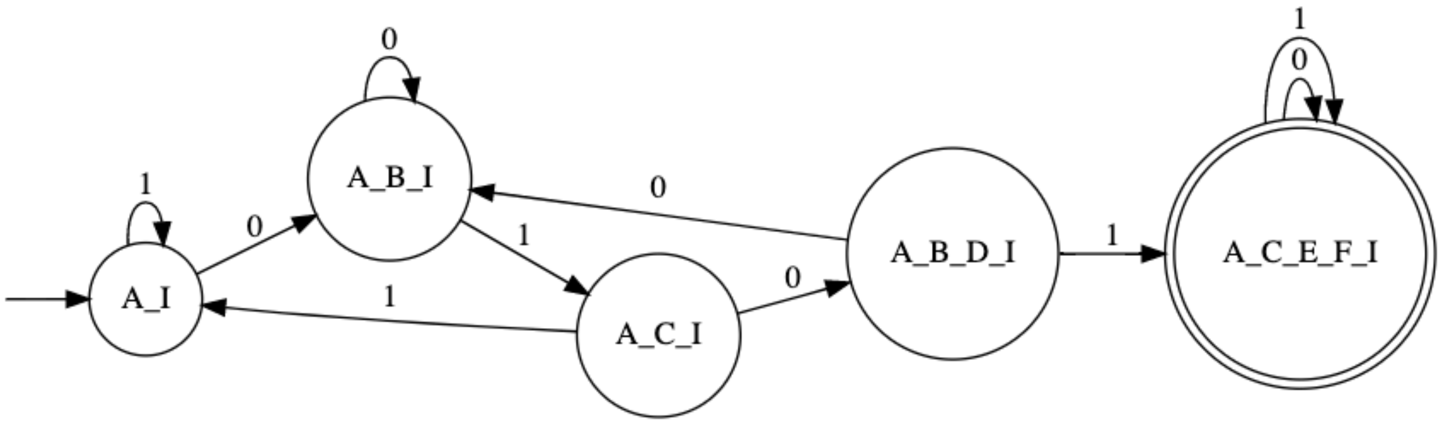
    Let $S_c = Eclosure((\cup_{s \in S}\, s_c))$;

Subset construction illustrated

```
dotObj_nfa(nfahas0101)
```




```
dotObj_dfa(min_dfa(nfa2dfa(nfahas0101, STATENAME_MAXSIZE = 50)), STATENAME_MAXSIZE = 50)
```

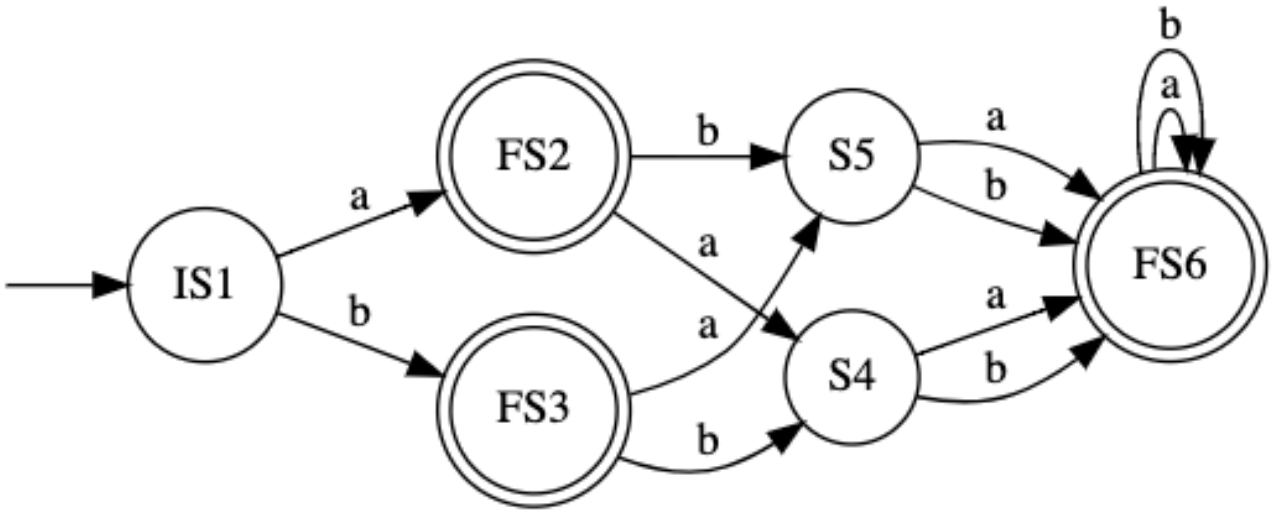


Review of concepts so far

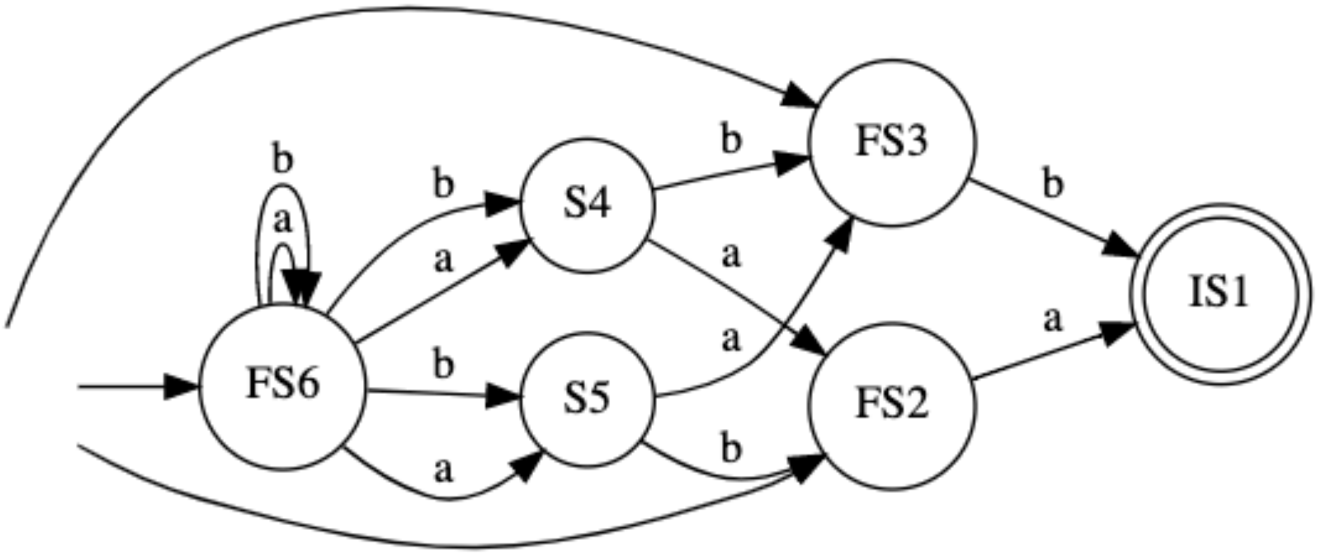
- NFA allow regular languages to be specified succinctly
  - No direct NFA minimization!
  - But they are often quite succinct
  - NFA can never be larger than DFA
    - DFA are essentially NFA
      - No epsilon moves
      - Next SET of states is to a singleton set
- NFA can be converted to a DFA with a potential exp blowup
  - Exp blowup is apparent when we convert the “Nth-last is a 1” NFA to a DFA
  - Algorithm is called subset construction

Reversal of DFA produce NFA

```
dotObj_dfa(FBloat)
```




```
dotObj_nfa(rev_dfa(FBloat))
```



What’s the language of FBloat and its reverse?

- Language of FBloat via language operations

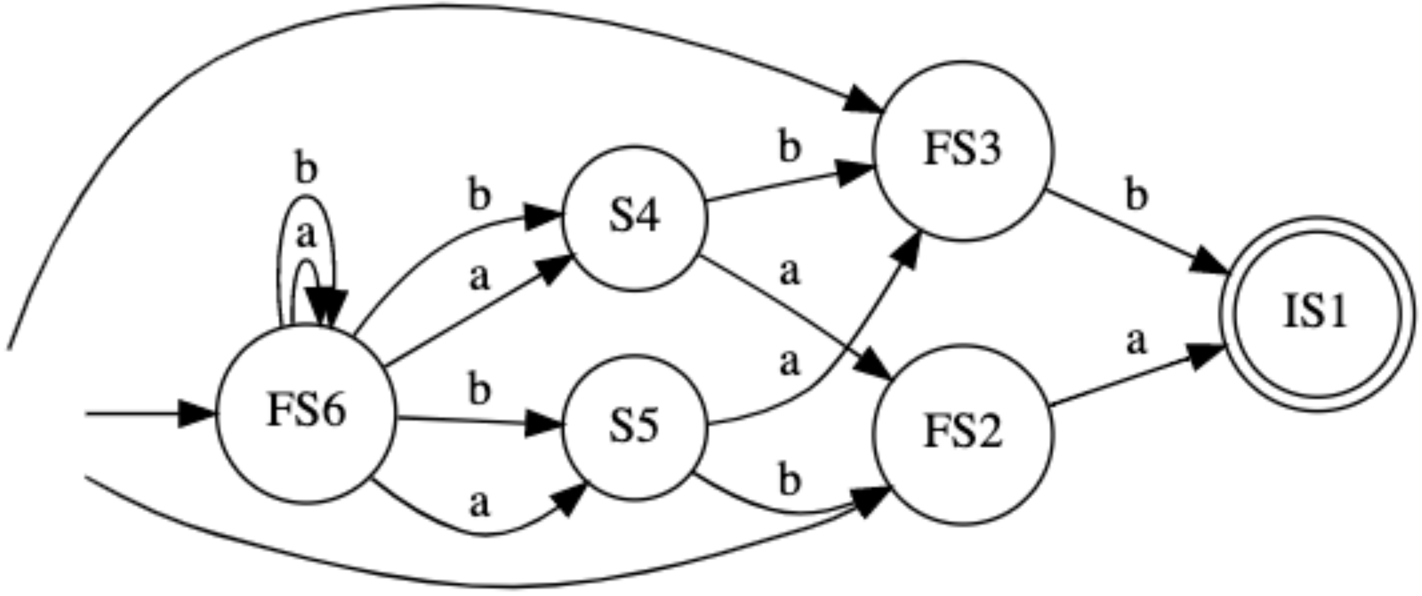
- Language of rev_dfa(Fbloat)

Reversal followed by nfa2dfa

i.e.

R ; D
so far

Do subset constrn.

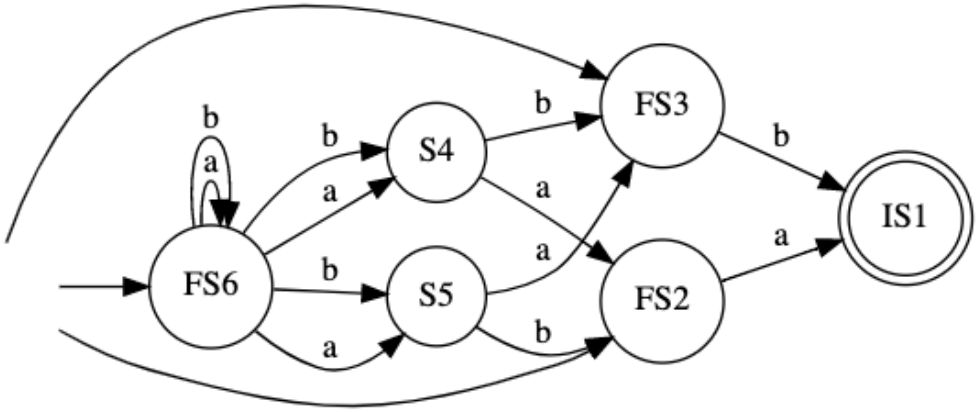


Reversal followed by nfa2dfa

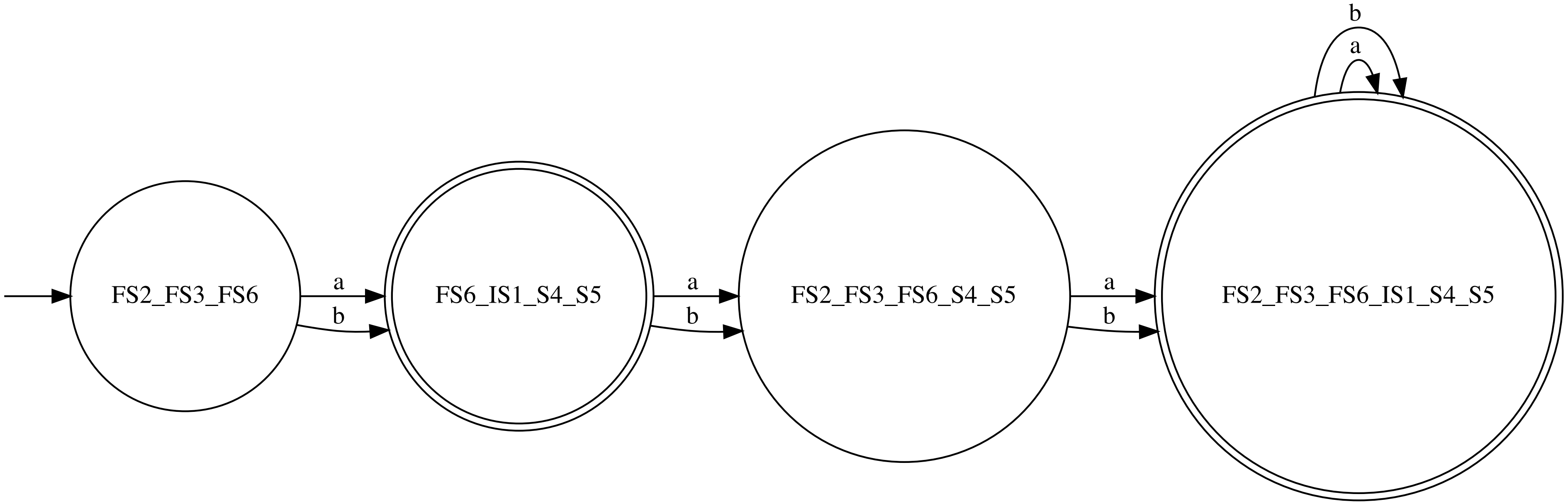
i.e.

R ; D
so far

Do subset constrn.



dotObj_dfa(nfa2dfa(rev_dfa(FBloat), STATENAME_MAXSIZE=50), STATENAME_MAXSIZE=50).render('/private/tmp/rdbloat')

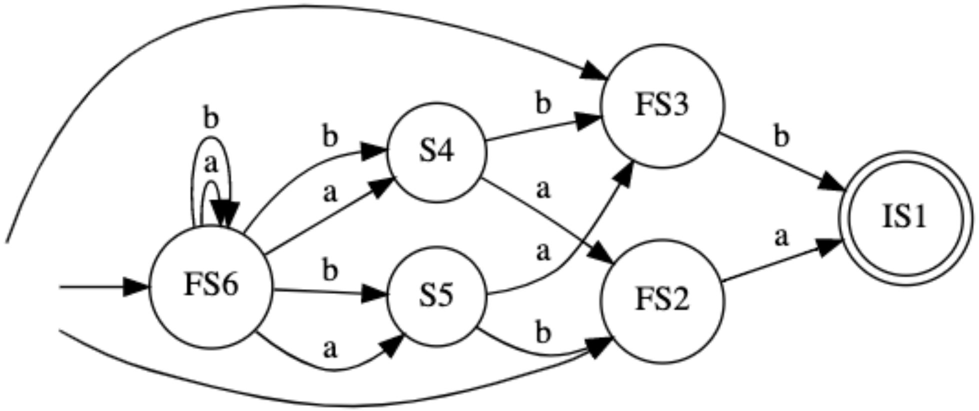


Reversal followed by nfa2dfa

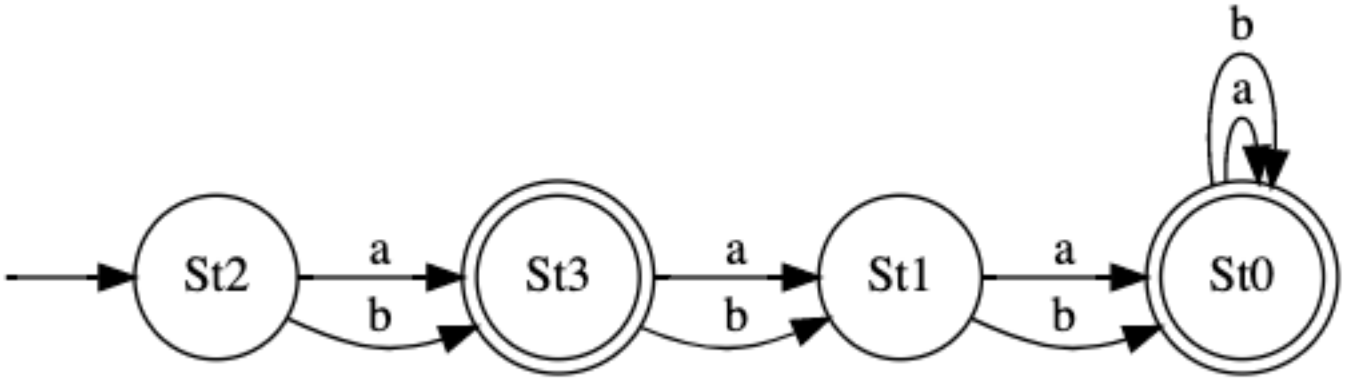
i.e.

R ; D
so far

Do subset constrn.

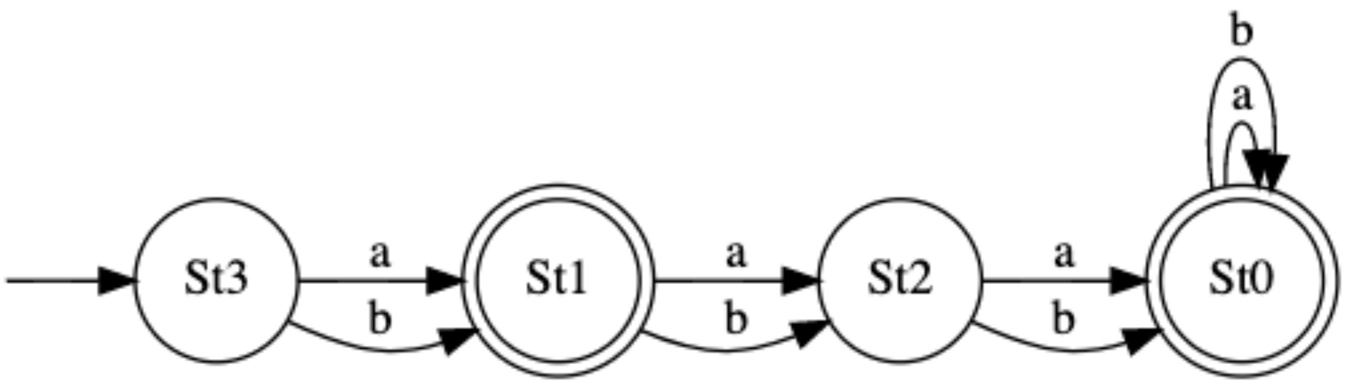



```
dotObj_dfa(nfa2dfa(rev_dfa(FBloat)))
```

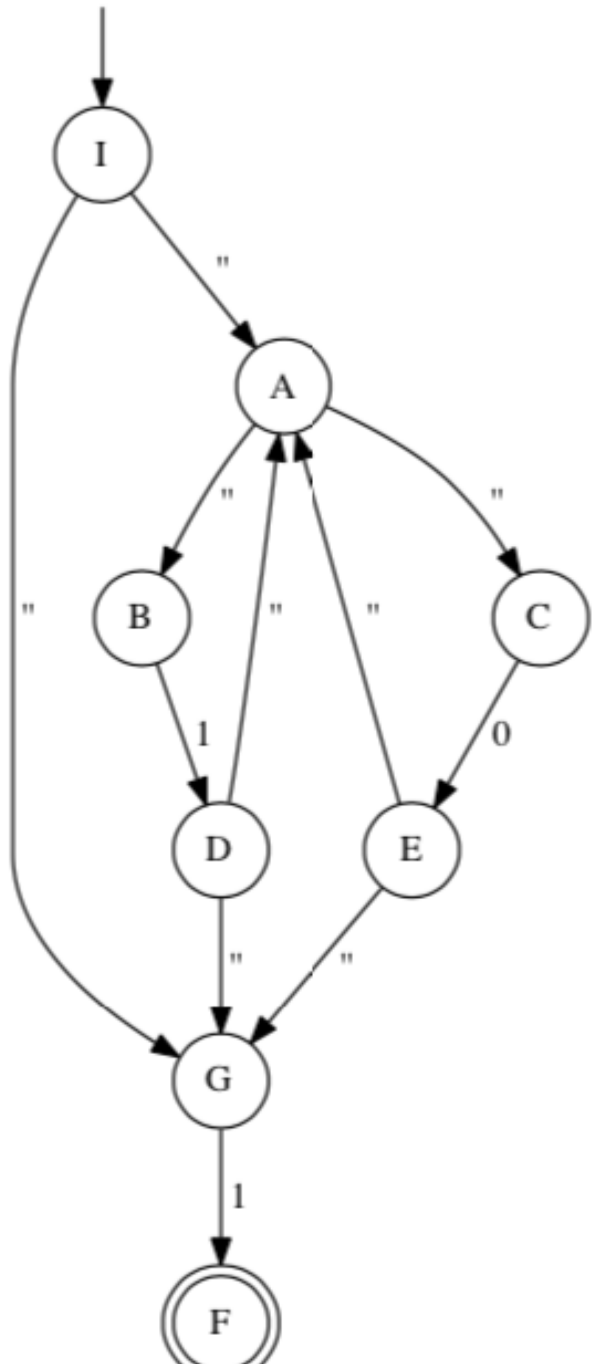


R ; D ; R ; D is Brzozowski's minimization!

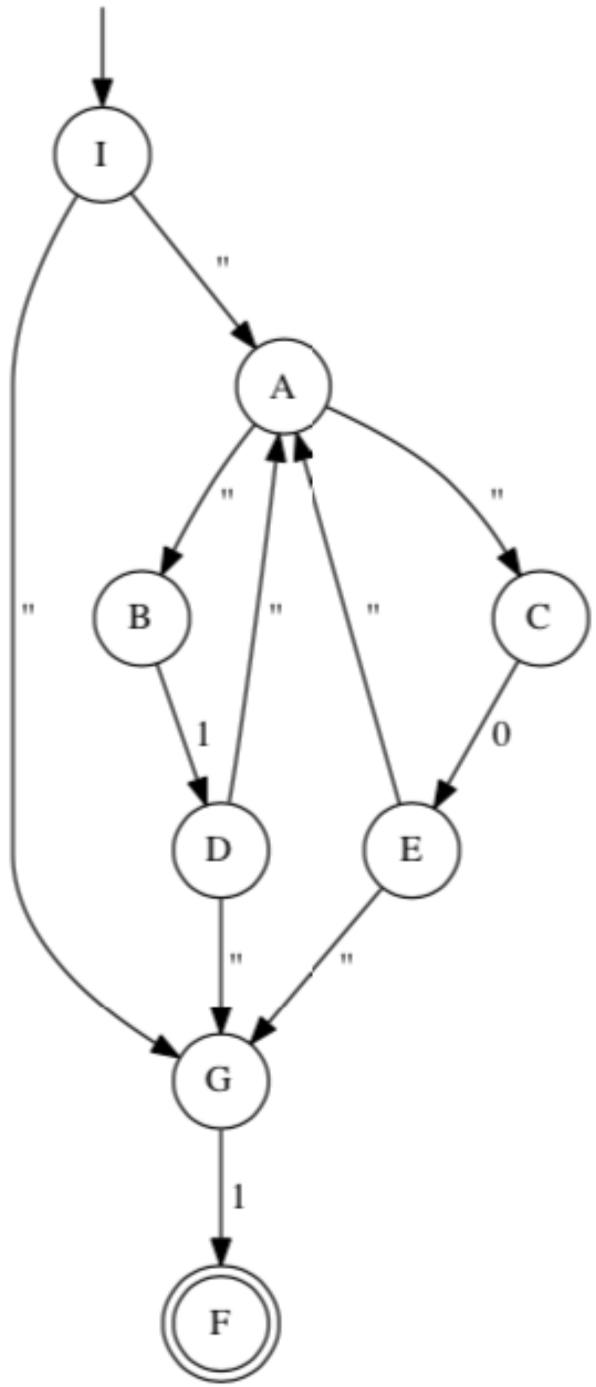
```
dotObj_dfa(nfa2dfa(rev_dfa(nfa2dfa(rev_dfa(FBloat)))))
```

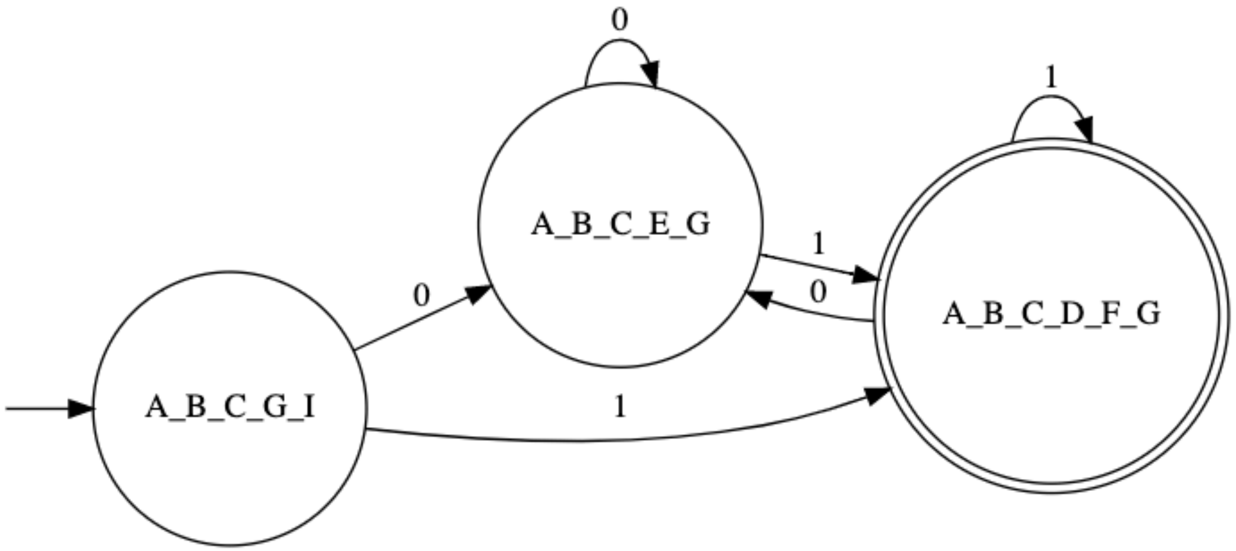


NFA2DFA for NFA with epsilons



NFA2DFA for NFA with epsilons





Summary

- DFA minimization can be done via Rev;Det;Rev;Det
  - This is Brzozowski's algorithm

Regular Expressions

- RE are textual short-hands for regular languages
  - Languages put together using Union, Concat, Star, and basic languages

- In general, we won't ask you to design complicated NFA
  - We will ask you to write REs instead

Regular Expressions: Examples

User syntax	Mathematical Syntax	Language Denoted
"	ε	$\{\varepsilon\}$
1	1	$\{1\}$
a	a	$\{a\}$
aa	aa	$\{a\}\{a\} = \{aa\}$
a+b	$a+b$	$\{a\} \cup \{b\} = \{a,b\}$
(a+b)(a+c)	$(a+b)(a+c)$	$\{a,b\}\{a,c\} = \{aa,ac,ba,bc\}$
(ab)+(ac)	$(ab)+(ac)$	$\{ab\} \cup \{ac\}$
a*	a^*	$\{a\}^*$
nothing	$\emptyset$	$\{\}$

Regular Expressions: General rules

The General Syntax for Regular Expressions (RE): REs can be defined over an alphabet Σ as follows:

1. ε is a RE denoting the regular language $\{\varepsilon\}$;

2. $a \in \Sigma$ is a RE denoting the regular language $\{a\}$;

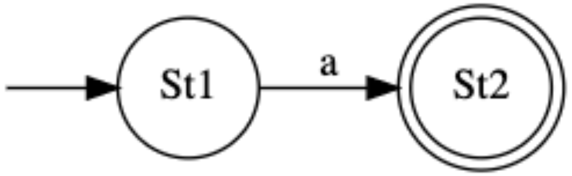
3. if r is a RE, so is r^* as well as (r); the former denotes the regular language $(\mathscr{L}(r))^*$ and the latter[2] denotes $\mathscr{L}(r)$, the language of r;

4. if r_1 and r_2 are REs, so are $r_1 + r_2$, and $r_1 r_2$. These expressions denote $(\mathscr{L}(r_1)) \cup (\mathscr{L}(r_2))$ and $(\mathscr{L}(r_1))(\mathscr{L}(r_2))$ respectively.[3]

re2nfa

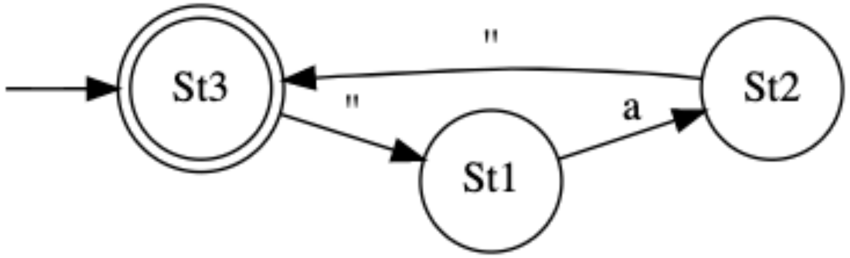
```
dotObj_nfa(re2nfa("a"))
```

```
Generating LALR tables
```



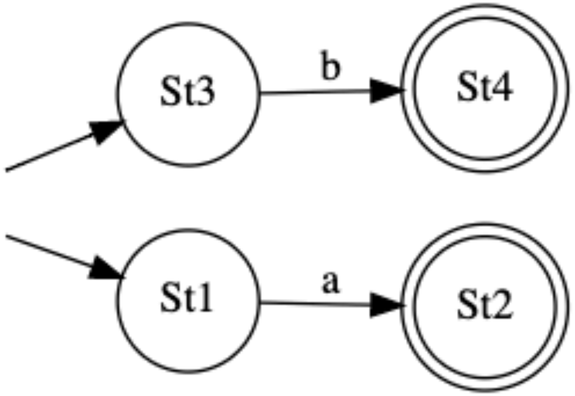

```
dotObj_nfa(re2nfa("a*"))
```

```
Generating LALR tables
```




```
dotObj_nfa(re2nfa("a+b"))
```

```
Generating LALR tables
```



Example: All words with 0101 with a 1-bit error

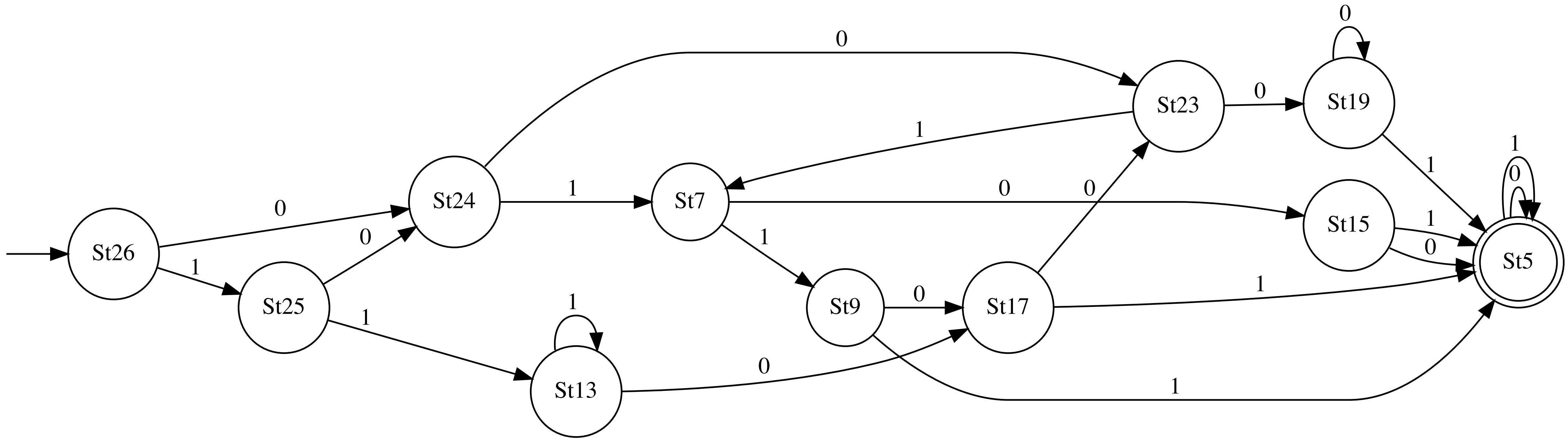
- ….0101….
- Here either the 0 or the 1 or the 0 or the 1 could be flipped
- We must still accept

Idioms for REs

- … is (0+1)*
- One-bit errors can be captured by a (0+1) pattern
- That is,
  - 0101
    - Versus
  - (0+1)101
- Build the whole RE
- Experiment in Jove

...0101... with a 1-bit error (Hamming dist).

dotObj_dfa(min_dfa(nfa2dfa(re2nfa("(0+1)* ((0+1)101 + 0 (0+1) 01 + 01 (0+1) 1 + 010 (0+1)) (0+1)*")))).
  render('/private/tmp/0101-one-bit-error')



Find the strings in the language of these RE

- (00*1 + 11*01)*
- ((00*1)* + 11*01)*
- (00*1 + (11*01)*)*

```
iso_dfa(
    min_dfa(
        nfa2dfa(
            re2nfa( " (00*1 + 11*01)* " ))),
    min_dfa(
        nfa2dfa(
            re2nfa( " ( (00*1)* + 11*01)* " )))
)
```

```
Generating LALR tables
Generating LALR tables
```

```
True
```

- Find out by developing a min DFA
  - Use iso_dfa

Compare these RE pairwise

- (00*1)*

- (0 (0+1)* 1)*

Compare these RE pairwise

- $(00^*1 + 11^*01)^*$
- $(0\,(0+1)^*\,1 + 11^*01)^*$