Two Sides of the Same Coin

Regular Expressions and Finite Automata are Equivalent

Revised Slides
Wednesday/Thursday, October 3/4, 2007
Reading: Beginning of Stoughton 3.11
Sipser 1.2 and 1.3

Imagine a really clever Randyesque graphic here!

CS235 Languages and Automata

Department of Computer Science
Wellesley College

Goals for This Lecture

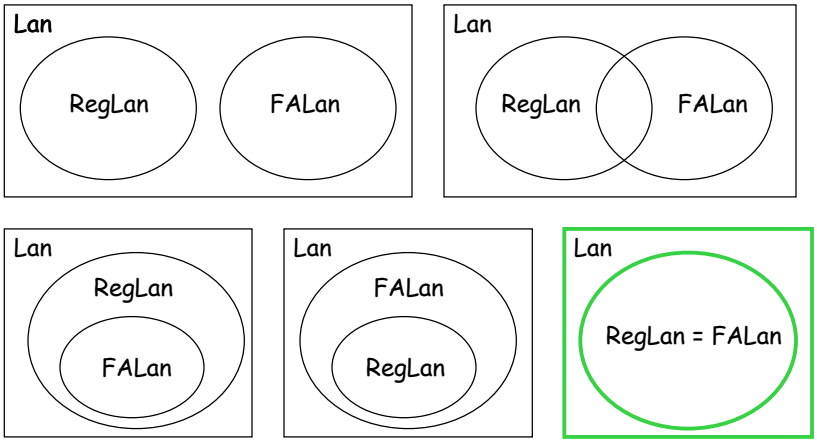
- Discuss relationships between languages described by different formalisms
- Show that regular expressions and finite automata describe the very same set of languages by:
  - Showing how to convert any element of **Reg** to an element of **FA**.
  - Showing how to convert any element of **FA** to an element of **Reg**.
- Summarize Forlan tools for manipulating finite automata and converting them to/from regular expressions.

Languages and Countability

Recall that **Sym** is a fixed universe of symbols and **Str** is the set of all strings over **Sym**.

Set	Countable or Uncountable? Why?
Str	
A **Sym**-language = subset of **Str**	
Lan = *P*(**Str**) = the set of all **Sym**-languages	
Reg = the set of regular expressions.	
RegLan = the set of regular languages (those described by regular expressions.)	
FA = the set of finite automata	
FALan = the set of languages accepted by finite automata.	

Possible Relationships among Lan, RegLan, and FALan



This is the actual relationship!

How Do We Show RegLan = FALan?

We need to describe algorithms for:

1. (Easy) Converting any regular expression to a finite automaton.

   We will use a minor variant of a standard approach used by Stoughton, Sipser, Kozen, etc.

2. (Hard) Converting any finite automaton to a regular expression.

   We will use Sipser's approach (from Sipser 1.3), which is equivalent to the "between languages"/dynamic programming approach used by Stoughton and Kozen, but *much* easier to understand.

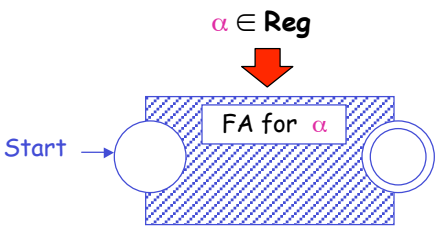
Note: Our conversions don't have to be efficient or lead to pretty results (they often don't). They are primarily to show that it's *possible* to convert from one to the other. However, sometimes we will use them for practical purposes, too.

Conversion 1: **Reg** To **FA**

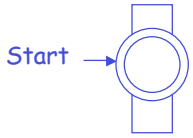
Recall that regular expressions are elements of **Reg**, which is inductively defined by 6 rules:

(empty string)	% ∈ **Reg**
(empty set)	$ ∈ **Reg**
(symbol)	a ∈ **Reg** for all a in **Sym**
(union)	+(α,β) ∈ **Reg** for all α, β ∈ **Reg**
(concatenation)	@(α,β) ∈ **Reg** for all α, β ∈ **Reg**
(Kleene closure)	*(α) ∈ **Reg** for all α ∈ **Reg**

We will show how to inductively construct for each element of **Reg** a finite automaton with exactly one accepting state (which might be the same as its start state).

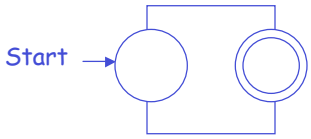


Reg To FA : The Three Leaf Cases



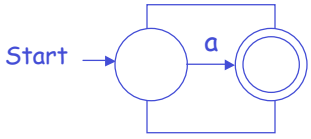
FA for %

$Q_{\%} = \{A\}$; states
$s_{\%} = A$; starting state
$A_{\%} = \{A\}$; accepting states
$T_{\%} = \{\}$; transitions



FA for $

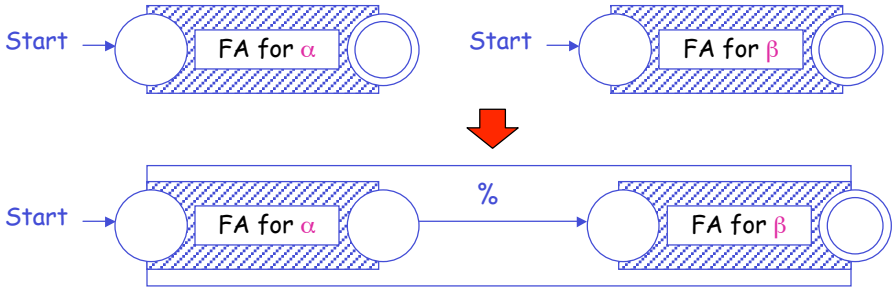
$Q_{\$} = \{A,B\}$
$S_{\$} = A$
$A_{\$} = \{B\}$
$T_{\$} = \{\}$



FA for a

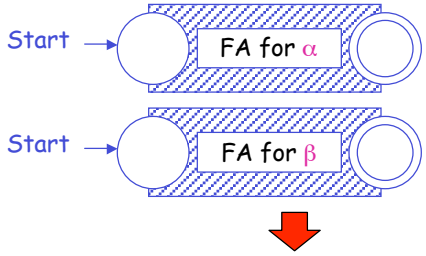
$Q_{\$} = \{A,B\}$
$S_{\$} = A$
$A_{\$} = \{B\}$
$T_{\$} = \{(A,a,B)\}$

Reg To FA : Concatenation @(α, β)

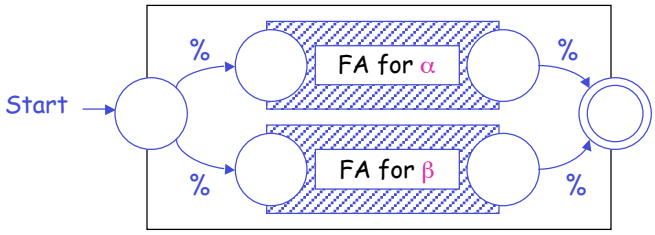


$Q_{@(\alpha,\beta)} = \{\langle 1,q\rangle \mid q \in Q_\alpha\} \cup \{\langle 2,q\rangle \mid q \in Q_\beta\}$
$S_{@(\alpha,\beta)} = \langle 1,s_\alpha\rangle$
$A_{@(\alpha,\beta)} = \{\langle 2,f_\beta\rangle\}$, where $A_\beta = \{f_\beta\}$
$T_{@(\alpha,\beta)} = T_\alpha \cup T_\beta \cup \{ (\langle 1,f_\alpha\rangle, \%, \langle 2,s_\beta\rangle)\}$, where $A_\alpha = \{f_\alpha\}$

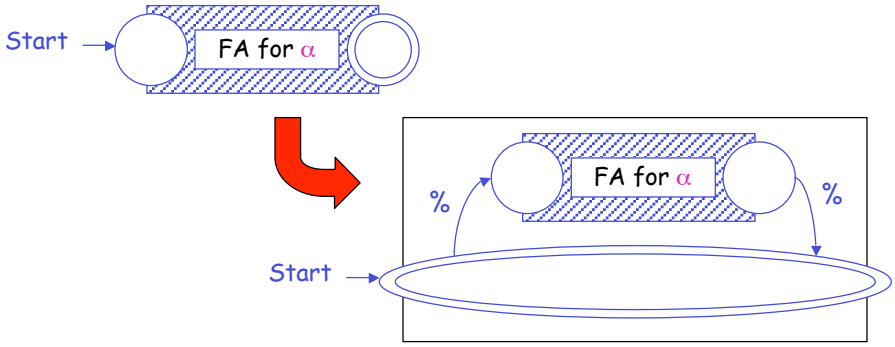
Reg To FA : Union +(α, β)



$Q_{+(\alpha,\beta)} = \{s,f\} \cup \{\langle 1,q\rangle \mid q \in Q_\alpha\}$
$\cup \{\langle 2,q\rangle \mid q \in Q_\beta\}$
$S_{+(\alpha,\beta)} = s$
$A_{+(\alpha,\beta)} = \{f\}$
$T_{+(\alpha,\beta)} = T_\alpha \cup T_\beta \cup \{(s,\%,\langle 1,s_\alpha\rangle),(s,\%,\langle 2,s_\beta\rangle)\}$
$\cup \{(\langle 1,f_\alpha\rangle,\%,f), (\langle 2,f_\beta\rangle,\%,f)\}$,
where $A_\alpha = \{f_\alpha\}$ and $A_\beta = \{f_\beta\}$



Reg To FA : Kleene Closure *(α)



$Q_{*(\alpha)} = \{s\} \cup \{\langle 1,q\rangle \mid q \in Q_\alpha\}$
$S_{*(\alpha)} = s$
$A_{*(\alpha)} = \{s\}$
$T_{*(\alpha)} = T_\alpha \cup \{(s,\%,\langle 1,s_\alpha\rangle), (\langle 1,f_\alpha\rangle,\%,s)\}$, where $A_\alpha = \{f_\alpha\}$

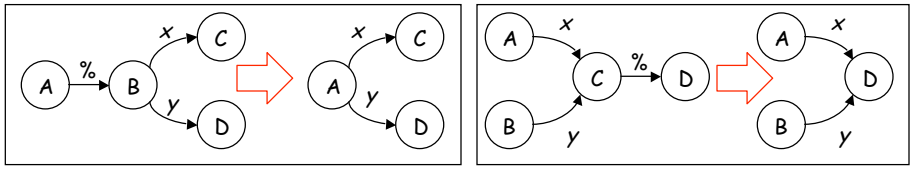
Reg To **FA** : Examples

(a + b)*c

(% + a)(b + c$)*

Simplification of Finite Automata

1. Can delete any **unreachable state** = a state to which there is no path from the start state.
2. Can delete any **dead state** = state from which there is no path to a final state.
3. Can delete a state with a %-labeled single input edge or a %-labeled single output edge.

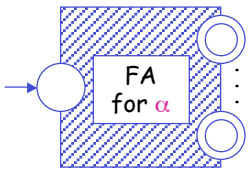


Note: Forlan's FA simplifier uses #1 and #2 but not #3.

Alternative Formulation of Reg to FA

Stoughton and Sipser give an alternative formulation of the Reg to FA conversion in which the FA may have any number of final states.

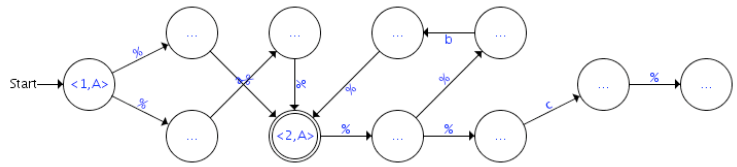
For practice, we'll work out some of the details.

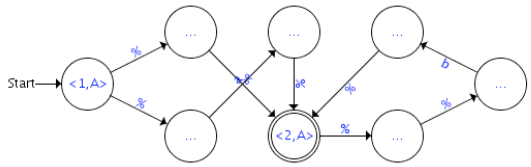


Reg to FA conversion in Forlan

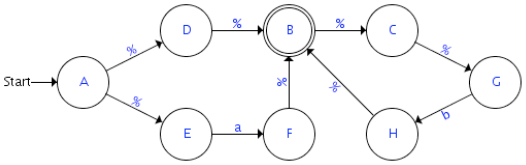
```
- val ex2_fa = FA.fromReg (Reg.fromString "(% + a)(b + c$)*");
val ex2_fa = - : fa

- print (FA.toString ex2_fa);
{states}
<1,A>, <2,A>, <2,<A>>, <1,<1,A>>, <1,<2,A>>, <1,<2,B>>, <2,<<1,A>>>,
<2,<<1,B>>>, <2,<<2,<1,A>>>>, <2,<<2,<1,B>>>>, <2,<<2,<2,A>>>>
{start state}
<1,A>
{accepting states}
<2,A>
{transitions}
<1,A>, % -> <1,<1,A>> | <1,<2,A>>; <2,A>, % -> <2,<A>>;
<2,<A>>, % -> <2,<<1,A>>> | <2,<<2,<1,A>>>>; <1,<1,A>>, % -> <2,A>;
<1,<2,A>>, a -> <1,<2,B>>; <1,<2,B>>, % -> <2,A>;
<2,<<1,A>>>, b -> <2,<<1,B>>>; <2,<<1,B>>>, % -> <2,A>;
<2,<<2,<1,A>>>>, c -> <2,<<2,<1,B>>>>; <2,<<2,<1,B>>>>, % -> <2,<<2,<2,A>>>>

- output("ex2.fa", ex2_fa);
val it = () : unit
```

Manipulating FAs via JFA





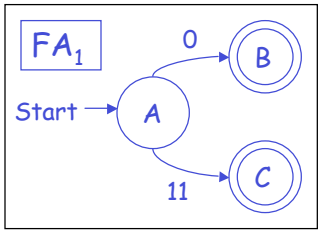
result of FA.simplify



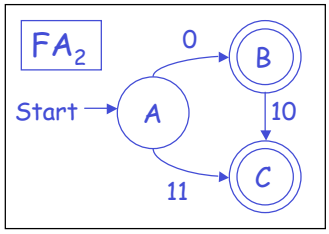
result of FA.renameStatesCanonically

Conversion 2: **FA** to **Reg** (Easy Cases)

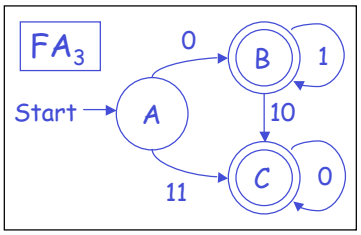
For FAs that are trees or DAGs with looping limited to self-loops, can do the conversion by inspection:



FA_1 is a tree; Reg_1 =



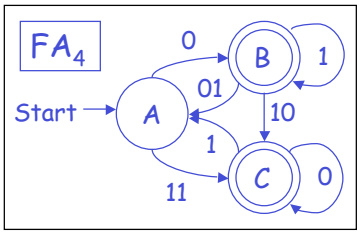
FA_2 is a DAG; Reg_2 =



FA_3 is a DAG with self-loops; Reg_3 =

Conversion 2: **FA** to **Reg** (Hard Case)

The hard case is when loops span multiple nodes.
What the heck to do here?



FA_4 has nontrivial cycles.
Reg_4 = ???

Stoughton describes a classic conversion algorithm based on "between languages", but it is hard to explain and visualize.

FA to **Reg** : Sipser to the Rescue

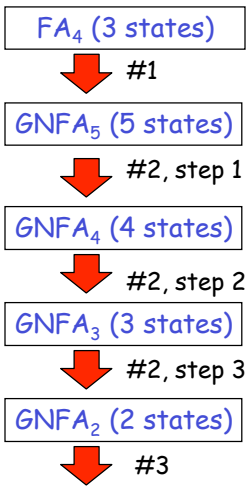
Sipser 1.3 describes an elegant algorithm for transforming an arbitrary finite automaton to a regular expression:

#1 Convert an *n*-state FA to a *(n+2)*-state Generalized Nondeterministic Finite Automaton (GNFA). This is basically an FA with a new start state and single final state in which edges are labeled by regular expressions rather than strings.

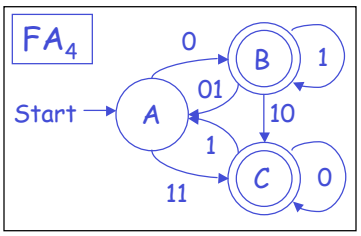
#2 Iterate *n* steps a state-ripping process that removes one state and its associated edges updates the regular expression labels of the remaining edges appropriately.

#3 The result is a 2-state GNFA having a single edge with the desired regular expression as its label.

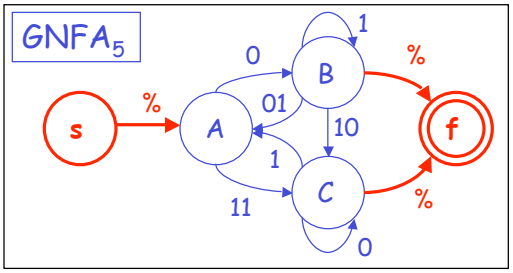
Sipser's algorithm is effectively the same as Stoughton's but presented in a way that's easier to explain and visualize.



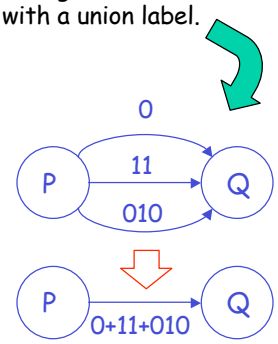
Sipser #1: Convert the FA to a GNFA



#1

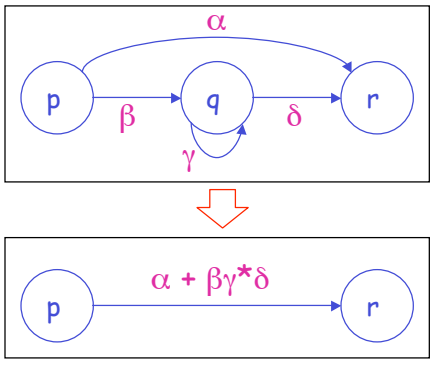


- Add a new GNFA start state **s** connected to the FA start state by %.
- Add a new GNFA final state **f** and connect every FA final state to it by %. The FA final states are not final in the GNFA.
- Replace multiple edges between two nodes by a single node with a union label.



Sipser #2: State-Ripping on GNFA

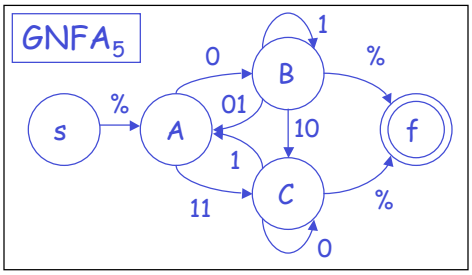
Pick any* state *q* (except s or f) and rip it out of GNFA, updating every labeled path p ⇒ q ⇒ r through q as follows:



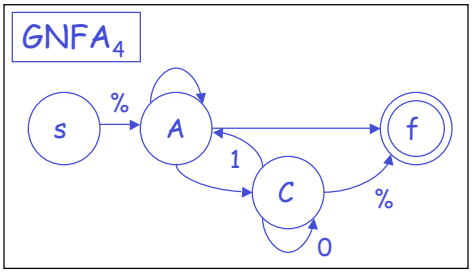
Can simplify this label manually, or automatically via weakSimplify or (simplify weakSubst)

* It is a good strategy to pick a state q participating as a middle element in the fewest labeled paths p ⇒ q ⇒ r.

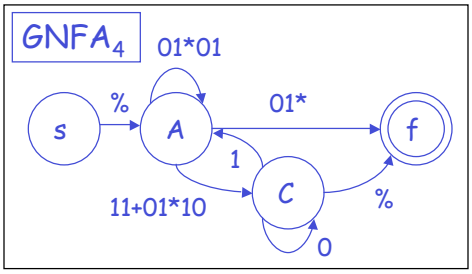
Sipser #2, Step 1: Rip out B



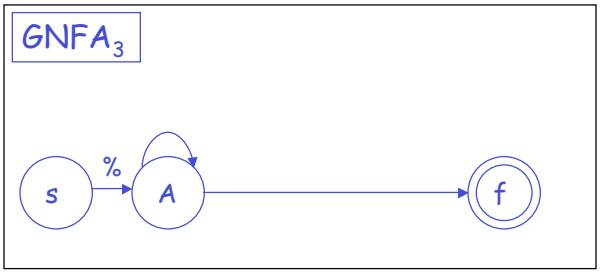
#2, step 1



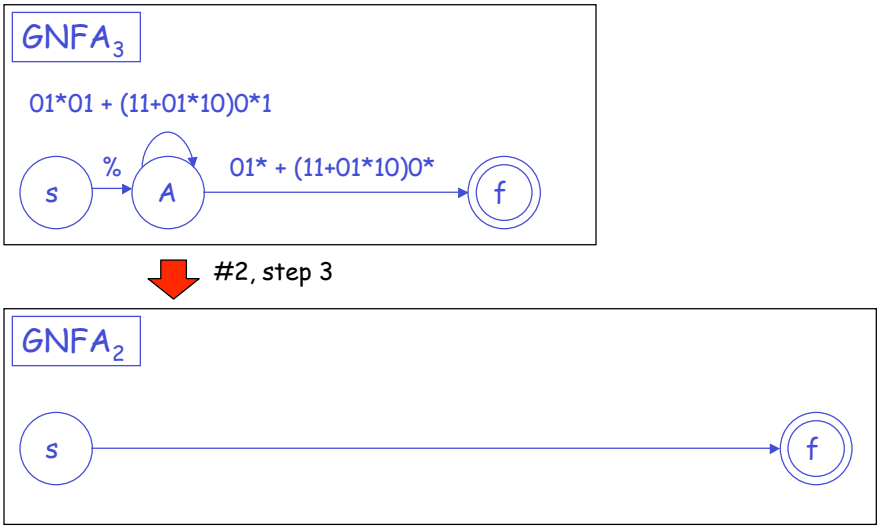
Sipser #2, Step 2: Rip out C



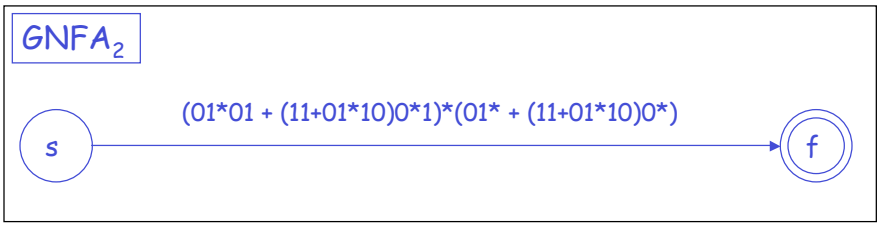
#2, step 2



Sipser #2, Step 3: Rip out A



Sipser #3: Extract Reg. Exp. from 2-state GNFA



RegExp(FA_4)
= (01*01 + (11+01*10)0*1)*(01* + (11+01*10)0*)