Lexical Analysis: DFA Minimization & Wrap Up

COMP 412
Fall 2005

Copyright 2005, Keith D. Cooper, Ken Kennedy & Linda Torczon, all rights reserved.
Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.

Automating Scanner Construction

RE→NFA *(Thompson's construction)* ✓
- Build an NFA for each term
- Combine them with ε-moves

NFA →DFA *(subset construction)* ✓
- Build the simulation

DFA →Minimal DFA *(today)*
- Hopcroft's algorithm

DFA →RE *(not really part of scanner construction)*
- All pairs, all paths problem
- Union together paths from s_0 to a final state

The Cycle of Constructions

DFA Minimization

The Big Picture

- Discover sets of equivalent states
- Represent each such set with just one state

DFA Minimization

The Big Picture

- Discover sets of equivalent states in the DFA
- Represent each such set with a single state

Two states are equivalent if and only if:

- The set of paths leading to them are equivalent
- $\forall\ \alpha \in \Sigma$, transitions on α lead to equivalent states (DFA)
- α-transitions to distinct sets $\Rightarrow$ states must be in distinct sets

DFA Minimization

The Big Picture

- Discover sets of equivalent states
- Represent each such set with just one state

Two states are equivalent if and only if:

- The set of paths leading to them are equivalent
- $\forall\ \alpha \in \Sigma$, transitions on α lead to equivalent states (DFA)
- α-transitions to distinct sets $\Rightarrow$ states must be in distinct sets

A partition *P* of *S*

- A collection of sets *P* s.t. each $s \in S$ is in exactly one $p_i \in P$
- The algorithm iteratively partitions the DFA's states

DFA Minimization

Details of the algorithm

- Group states into maximal size sets, *optimistically*
- Iteratively subdivide those sets, based on transition graph
- States that remain grouped together are equivalent

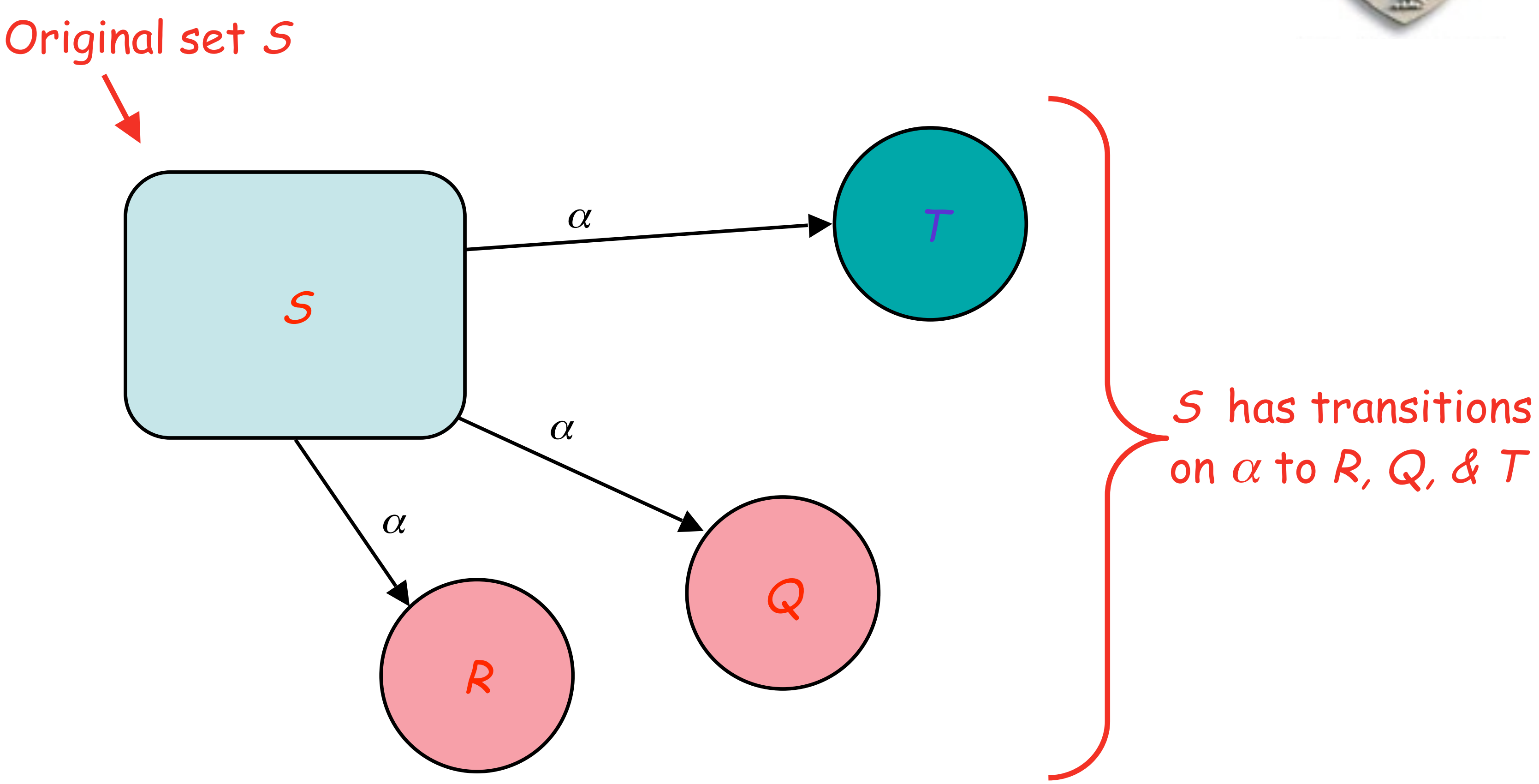
Initial partition, P_0 , has two sets: $\{F\}$ & $\{S-F\}$ $(D = (S,\Sigma,\delta,s_0,F))$

final states others

Splitting a set ("partitioning a set by $\underline{a}$")

- Assume s_a & $s_b \in p_i$, and $\delta(s_a,\underline{a}) = s_x$, & $\delta(s_b,\underline{a}) = s_y$
- If s_x & s_y are not in the same set, then p_i must be split
  - s_a has transition on a, s_b does not $\Rightarrow$ $\underline{a}$ splits p_i
- One state in the final DFA cannot have two transitions on $\underline{a}$

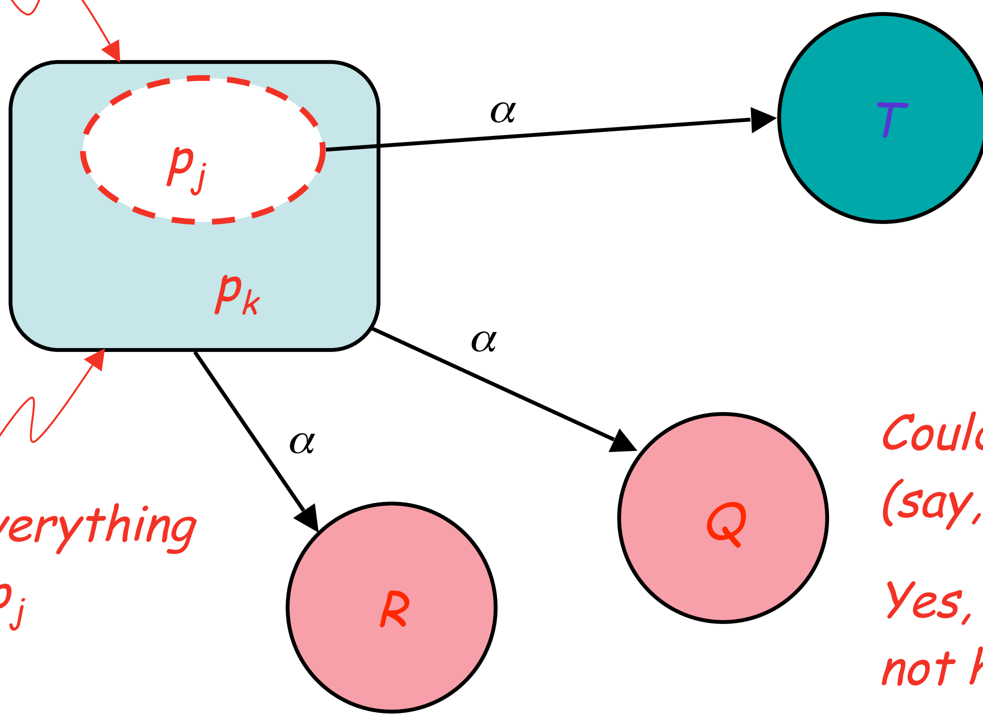
Key Idea: Splitting S around α



The algorithm partitions S around α

Key Idea: Splitting p_i around α

Original set p_i



p_k is everything in p_i - p_j

Could we split p_k further? (say, between Q & R?)

Yes, but doing so does not help asymptotically.

The algorithm will split p_k in a future iteration.

<u>*This is a fixed-point algorithm!*</u>

DFA Minimization

The algorithm

$P \leftarrow \{ F, \{S\text{-}F\}\}$

while (P is still changing)
    $T \leftarrow \{ \}$
    for each $p_i \in P$
        for each $\alpha \in \Sigma$
            partition p_i *by* α
                into p_j, *and* p_k
            if p_i *splits*
                $T \leftarrow T \cup p_j \cup p_k$
            else
                $T \leftarrow T \cup p_i$
    if $T \neq P$ *then*
        $P \leftarrow T$

Why does this work?

- Partition $P \in 2^S$
- Start off with 2 subsets of S: *{F} and {S-F}*
- The *while* loop takes $P^i \rightarrow P^{i+1}$ by splitting 1 or more sets
- P^{i+1} is at least one step closer to the partition with $|S|$ sets
- Maximum of $|S|$ splits

Note that

- Partitions are <u>never</u> combined
- Initial partition ensures that final states remain final states

DFA Minimization

Refining the algorithm

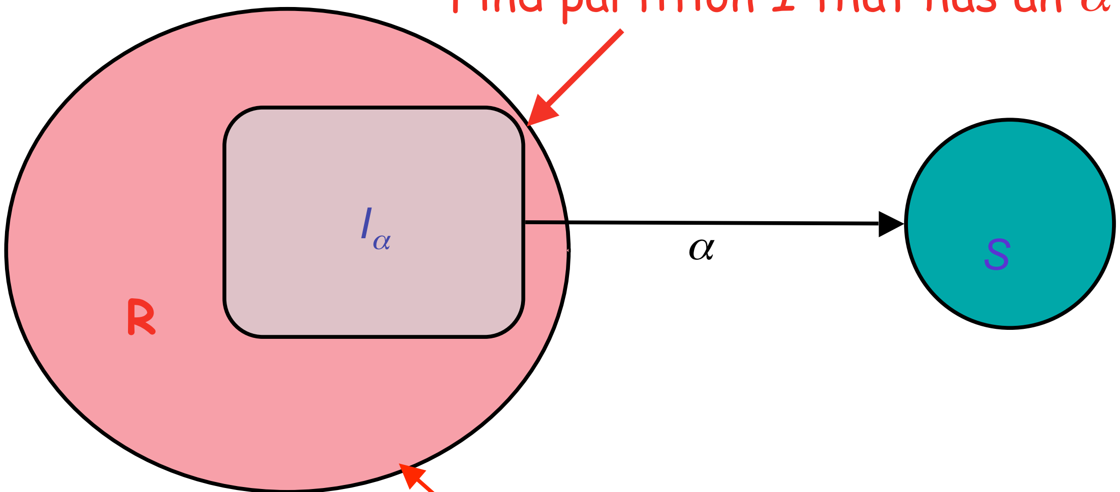
- As written, it examines every $p_i \in P$ on each iteration
  - This strategy entails a lot of unnecessary work
  - Only need to examine p_i if some T, reachable from p_i, has split
- Reformulate the algorithm using a "worklist"
  - Start worklist with initial partition, F and $\{S-F\}$
  - When it splits p_i into p_1 and p_2, place p_2 on worklist

This version looks at each $p_i \in P$ many fewer times

- Well-known, widely used algorithm due to John Hopcroft

Key Idea: Splitting S around α

Find partition I that has an α-transition into S



This part must have an α-transition to one or more other states in one or more other partitions.

Otherwise, it does not split!

Hopcroft's Algorithm

$W \leftarrow \{F, S\text{-}F\}$; $P \leftarrow \{F, S\text{-}F\}$; *// W is the worklist, P the current partition*

while (W is not empty) **do begin**

  select and remove s from W; // s is a set of states

  for each α in Σ **do begin**

   let $I_\alpha \leftarrow \delta_\alpha^{-1}(s)$; // I_α is set of all states that can reach s on α

   for each *R in P such that* $R \cap I_\alpha$ is not empty

    and R is not contained in I_α **do begin**

     partition R into R_1 and R_2 such that $R_1 \leftarrow R \cap I_\alpha$; $R_2 \leftarrow R - R_1$;

     replace R in P with R_1 and R_2;

     if $R \in W$ **then** replace R with R_1 in W and add R_2 to W;

     else if $|R_1| \leq |R_2|$

      then add add R_1 to W;

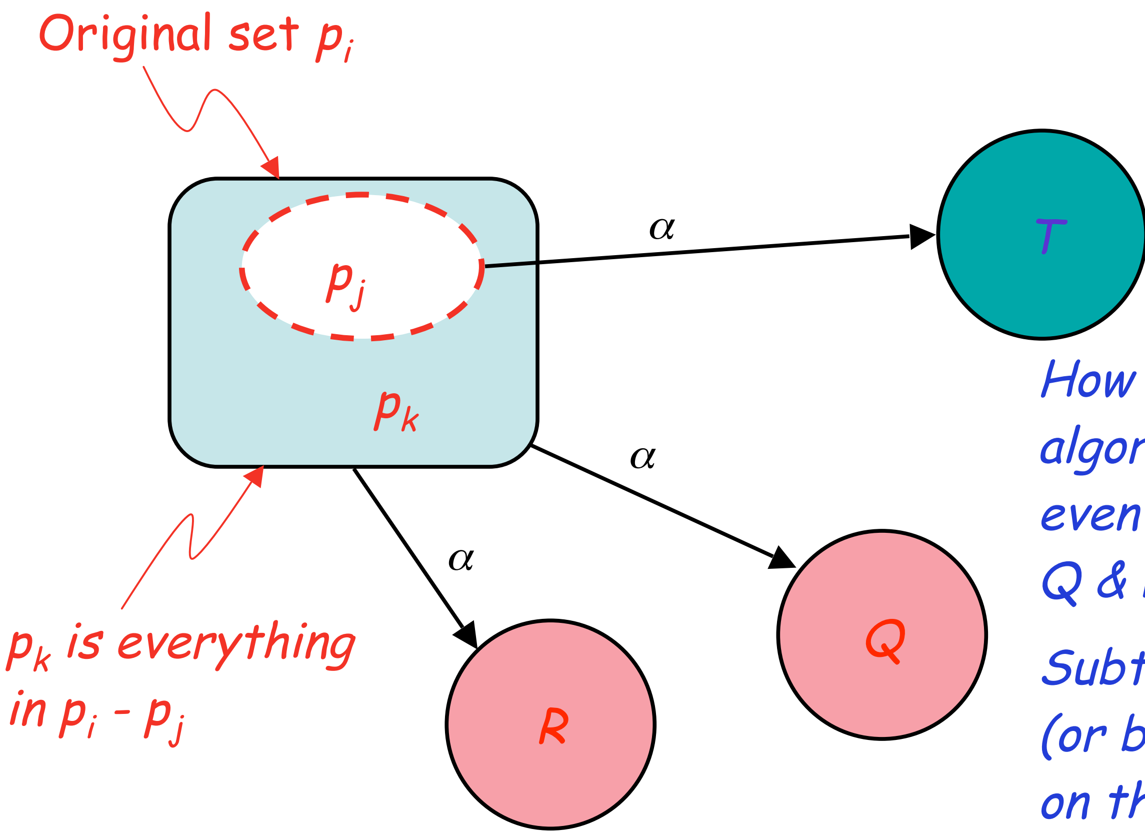
      else add R_2 to W;

   end

  end

end

Key Idea: Splitting p_i around α



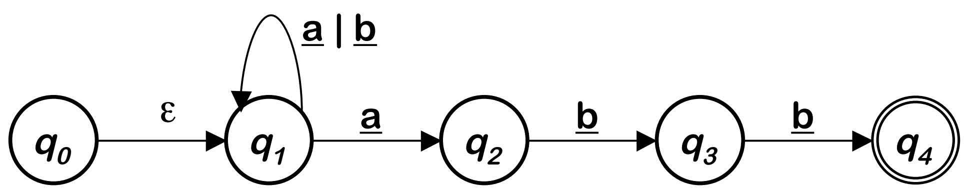
How does the worklist algorithm ensure that p_k eventually splits around Q & R ?

Subtle point: either Q or R (or both) must already be on the worklist. (Q & R have split from {S-F}.)

Thus, it can split p_i around one state (T) & add either p_j or p_k to the worklist.

A Detailed Example

Remember (a | b)* abb ? *(from last lecture)*



Our first NFA

Applying the subset construction:

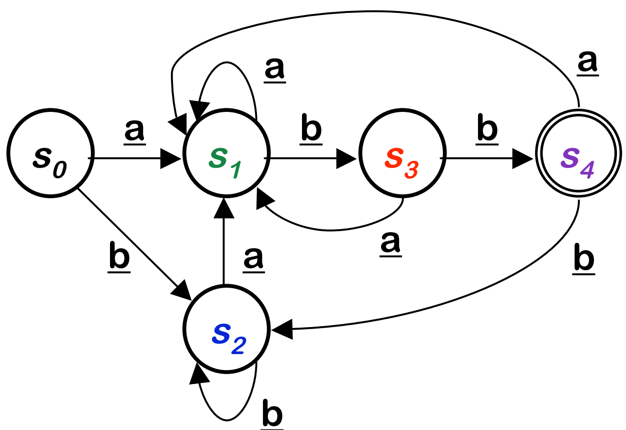
Iter.	State	Contains	ε-closure(move(s_i,a))	ε-closure(move(s_i,b))
0	s_0	q_0, q_1	q_1, q_2	q_1
1	s_1	q_1, q_2	q_1, q_2	q_1, q_3
	s_2	q_1	q_1, q_2	q_1
2	s_3	q_1, q_3	q_1, q_2	q_1, q_4
3	s_4	q_1, q_4	q_1, q_2	q_1

contains q_4 (final state)

Iteration 3 adds nothing to *S*, so the algorithm halts

A Detailed Example

The DFA for (a | b)* abb



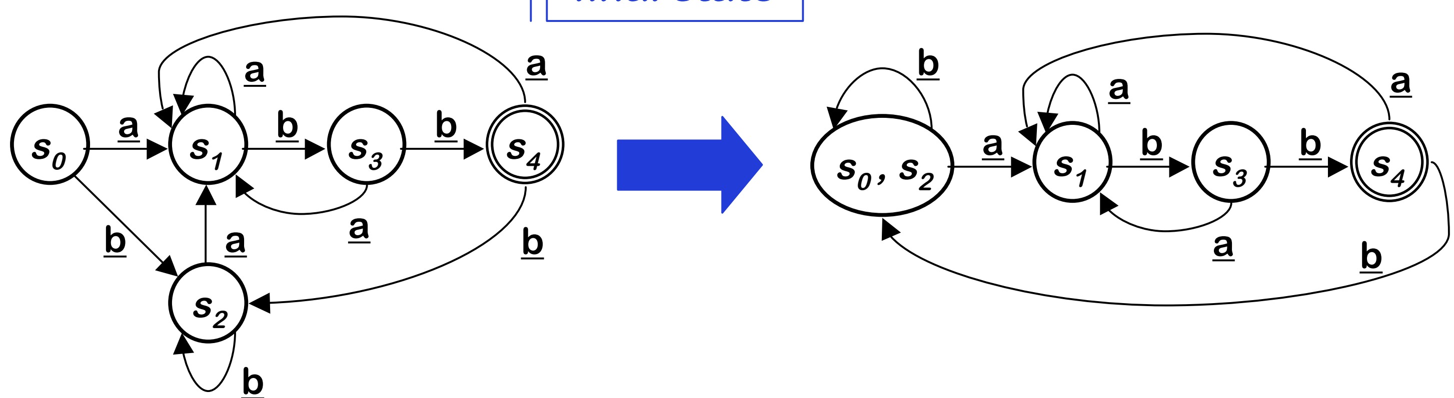
δ	a	b
s_0	s_1	s_2
s_1	s_1	s_3
s_2	s_1	s_2
s_3	s_1	s_4
s_4	s_1	s_2

- Not much bigger than the original NFA
- All transitions are deterministic
- Use same code skeleton as before

A Detailed Example (DFA Minimization)

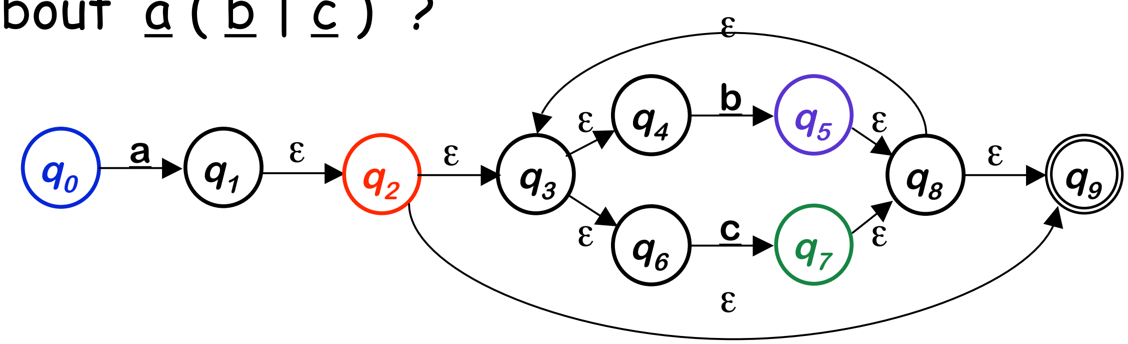
	Current Partition	*Worklist*	*s*	*Split on* a	*Split on* b
P_0	$\{s_4\}\ \{s_0,s_1,s_2,s_3\}$	$\{s_4\}$ $\{s_0,s_1,s_2,s_3\}$	$\{s_4\}$	none	none
P_0	$\{s_4\}\{s_0,s_1,s_2,s_3\}$	$\{s_0,s_1,s_2,s_3\}$	$\{s_0,s_1,s_2,s_3\}$	none	$\{s_0, s_1, s_2\}$ $\{s_3\}$
P_1	$\{s_4\}\{s_3\}\{s_0,s_1,s_2\}$	$\{s_3\}$	$\{s_3\}$	none	$\{s_0, s_2\}\{s_1\}$
P_2	$\{s_4\}\{s_3\}\{s_1\}\{s_0,s_2\}$	$\{s_1\}$	$\{s_1\}$	none	none

final state



DFA Minimization

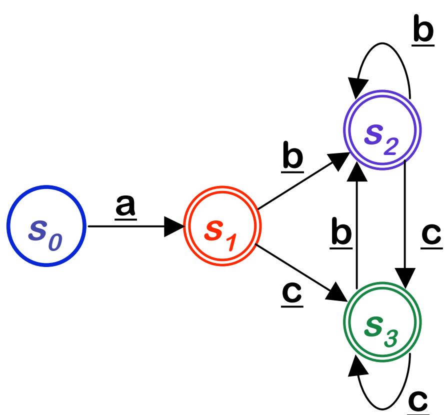
What about a (b | c)* ?



First, the subset construction:

	NFA states	ε-closure(move(s,*)) a	b	c
s_0	q_0	$q_1, q_2, q_3, q_4, q_6, q_9$	*none*	*none*
s_1	$q_1, q_2, q_3, q_4, q_6, q_9$	*none*	$q_5, q_8, q_9, q_3, q_4, q_6$	$q_7, q_8, q_9, q_3, q_4, q_6$
s_2	$q_5, q_8, q_9, q_3, q_4, q_6$	*none*	s_2	s_3
s_3	$q_7, q_8, q_9, q_3, q_4, q_6$	*none*	s_2	s_3

Final states

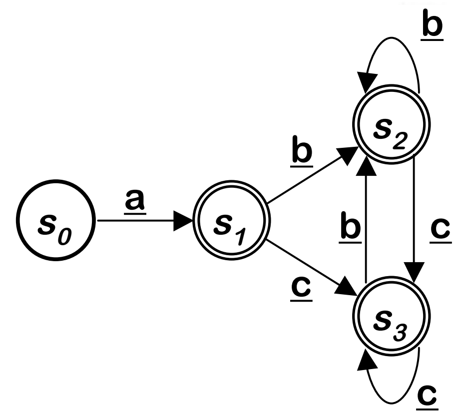


DFA Minimization

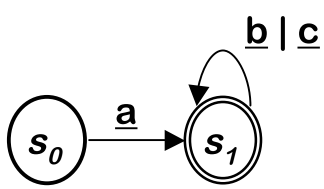
Then, apply the minimization algorithm

	Current Partition	*Split on*		
		a	b	c
P_0	$\{s_1, s_2, s_3\}$ $\{s_0\}$	*none*	*none*	*none*

final states



To produce the minimal DFA



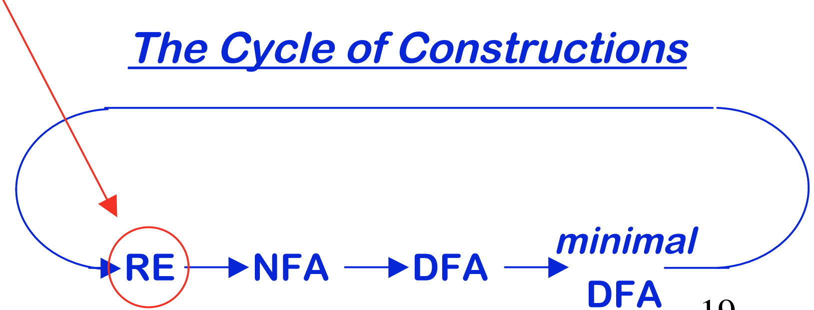
In lecture 5, we observed that a human would design a simpler automaton than Thompson's construction & the subset construction did.

Minimizing that DFA produces the one that a human would design!

Abbreviated Register Specification

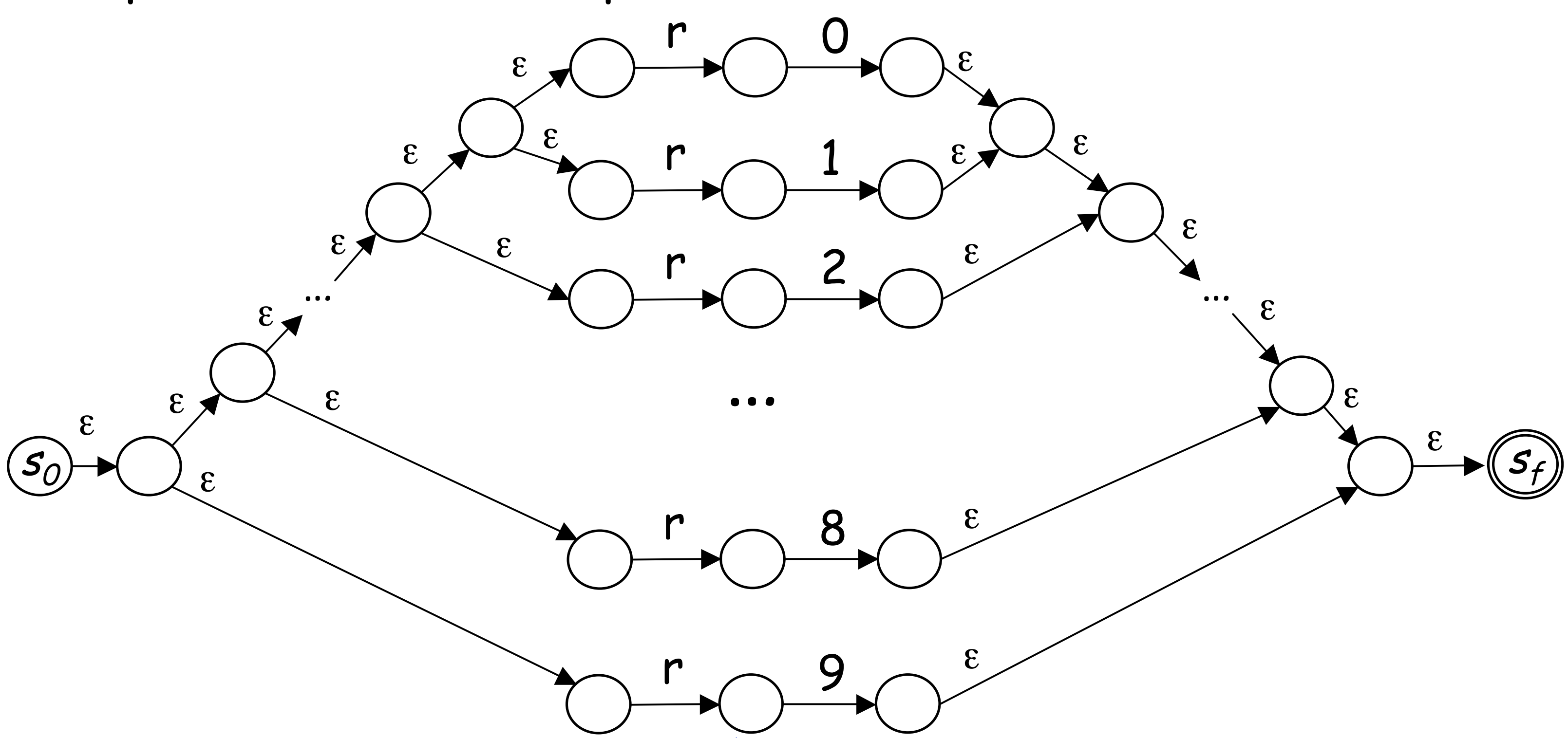
Start with a regular expression

r0 | r1 | r2 | r3 | r4 | r5 | r6 | r7 | r8 | r9



Abbreviated Register Specification

Thompson's construction produces

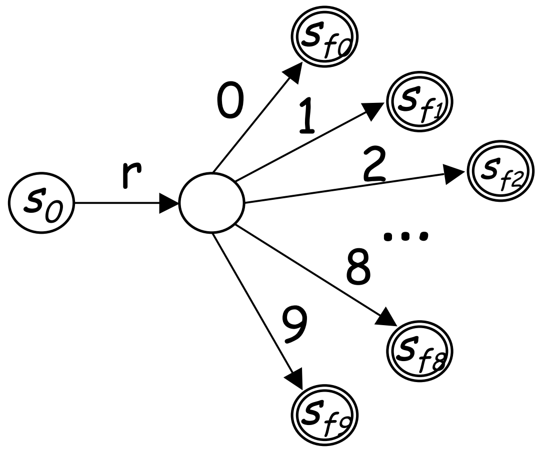


To make it fit, we've eliminated the ε -transition between "r" and "0".

The Cycle of Constructions

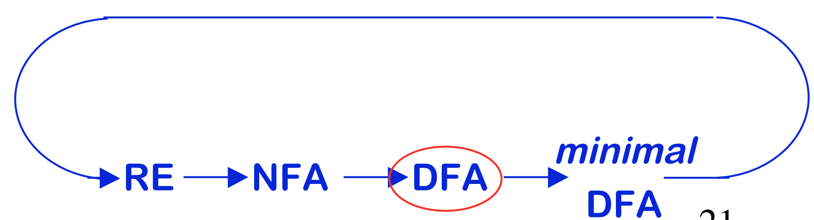
Abbreviated Register Specification

The subset construction builds



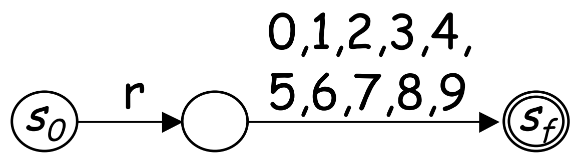
This is a DFA, but it has a lot of states …

The Cycle of Constructions



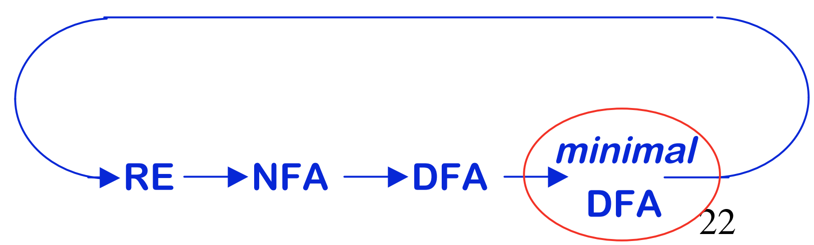
Abbreviated Register Specification

The DFA minimization algorithm builds



This looks like what a skilled compiler writer would do!

The Cycle of Constructions



Limits of Regular Languages

Not all languages are regular

RL's $\subset$ CFL's $\subset$ CSL's

You cannot construct DFA's to recognize these languages

- $L = \{ p^k q^k \}$ *(parenthesis languages)*
- $L = \{ wcw^r \mid w \in \Sigma^* \}$

Neither of these is a regular language *(nor an RE)*

But, this is a little subtle. You <u>can</u> construct DFA's for

- Strings with alternating 0's and 1's
  $(\varepsilon \mid 1) (01)^* (\varepsilon \mid 0)$
- Strings with and even number of 0's and 1's
  See Homework 1!

RE's can count bounded sets and bounded differences

Limits of Regular Languages

Advantages of Regular Expressions

- Simple & powerful notation for specifying patterns
- Automatic construction of fast recognizers
- Many kinds of syntax can be specified with REs

Example — an expression grammar

$$\mathit{Term} \rightarrow [\text{a-zA-Z}]\,([\text{a-zA-Z}] \mid [\underline{0}\text{-}\underline{9}])^{*}$$

$$\mathit{Op} \rightarrow \underline{+} \mid \underline{-} \mid \underline{*} \mid \underline{/}$$

$$\mathit{Expr} \rightarrow (\ \mathit{Term}\ \mathit{Op}\ )^{*}\ \mathit{Term}$$

Of course, this would generate a DFA …

If REs are so useful …

Why not use them for everything?

Table-Driven Versus Direct-Coded Scanners

Table-driven recognizers use indexing

- *index* — Read (& classify) the next character
- *index* — Select the case using *action()*
- *index* — Find the next state
- *register* — Assign to the state variable
- Branch back to the top

```
state ← s0 ;
while (state ≠ exit)
  state ← d(state,char);
  perform (action(state,char));
  char ← next character;
```

Alternative strategy: direct coding

- Encode state in the program counter
  - Each state is a separate piece of code
- Do transition tests locally and directly branch
- Generate ugly, spaghetti-like code
- More efficient than table driven strategy
  - Fewer memory operations, might have more branches

Building Faster Scanners from the DFA

A direct-coded recognizer for r̲ *Digit Digit**

goto s_0;

s_0: *word* $\leftarrow \emptyset$;
char $\leftarrow$ *next character*;
if (char = 'r')
then goto s_1;
else goto s_e;

s_1: *word* $\leftarrow$ *word + char*;
char $\leftarrow$ *next character*;
if ('0' $\leq$ *char* $\leq$ *'9')*
then goto s_2;
else goto s_e;

s2: *word* $\leftarrow$ *word + char*;
char $\leftarrow$ *next character*;
if ('0' $\leq$ *char* $\leq$ *'9')*
then goto s_2;
else if (char = eof)
then report success;
else goto s_e;

s_e: *print error message*;
return failure;

- Many fewer operations per character
- Almost no memory operations
- Even faster with careful use of fall-through cases

Building Scanners

The point

- All this technology lets us automate scanner construction
- Implementer writes down the regular expressions
- Scanner generator builds NFA, DFA, minimal DFA, and then writes out the (table-driven or direct-coded) code
- This reliably produces fast, robust scanners

For most modern language features, this works

- You should think twice before introducing a feature that defeats a DFA-based scanner
- The ones we've seen (e.g., insignificant blanks, non-reserved keywords) have not proven particularly useful or long lasting