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

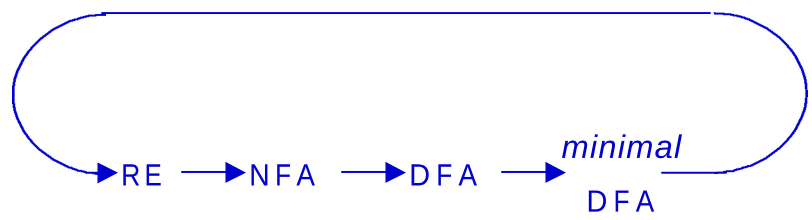
DFA →Minimal DFA *(today)*

- Hopcroft's algorithm

DFA →RE

- 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

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)
- transitions to distinct sets $\Rightarrow$ states must be in distinct sets

A partition *P* of *S*

- Each *s* Î *S* is in exactly one set p_i Î *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, as needed
- States that remain grouped together are equivalent

Initial partition, P_0 , has two sets *{F}* & *{Q-F}* *(D =(Q,S,d,q_0,F))*

Splitting a set

- Assume q_a, q_b, & $q_c \in s$, and
- $\delta(q_a,\underline{a}) = q_x$, $\delta(q_b,\underline{a}) = q_y$, & $\delta(q_a,\underline{a}) = q_z$
- If q_x, q_y, & q_z are not in the same set, then s must be split
- One state in the final DFA cannot have two transitions on $\underline{a}$

DFA Minimization

The algorithm

P ¬ { F, {Q-F}}

while (P is still changing)

  T ¬ { }

  for each set s Î P

    for each a Î S

      partition s by a

        into s_1, s_2, …, s_k

      T ¬ T È s_1, s_2, …, s_k

  if T ¹ P then

    P ¬ T

This is a fixed-point algorithm!

Why does this work?

- Partition *P Î* 2^Q
- Start off with 2 subsets of *Q*
  {F} and {Q-F}
- *While* loop takes P_i®P_{i+1} by splitting 1 or more sets
- P_{i+1} is at least one step closer to the partition with |*Q* | sets
- Maximum of |*Q* | splits

Note that

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

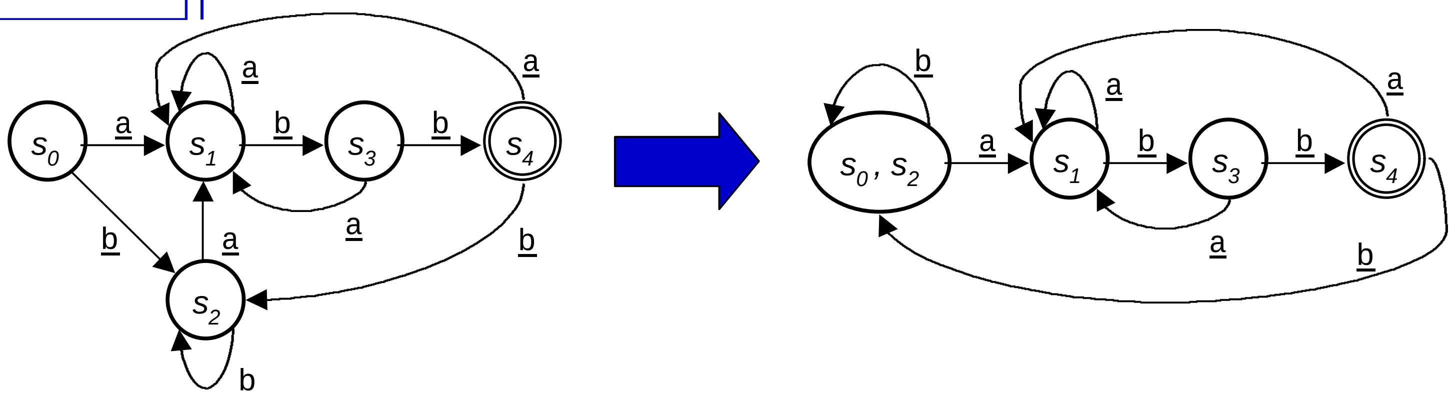
DFA Minimization

Enough theory, does this stuff work?

- Recall our example: (a | b)* abb

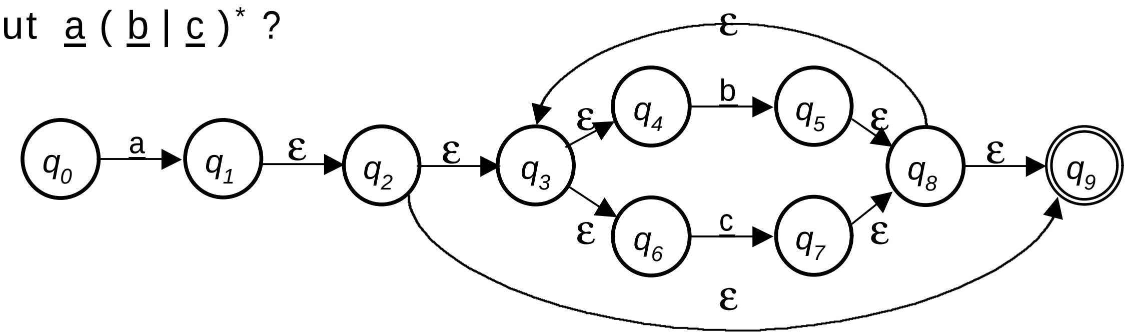
	Current Partition	*Split on* a	*Split on* b
P_0	$\{s_4\}$ $\{s_0, s_1, s_2, s_3\}$	none	$\{s_0, s_1, s_2\}$ $\{s_3\}$
P_1	$\{s_4\}\{s_3\}\{s_0, s_1, s_2\}$	none	$\{s_0, s_2\}\{s_1\}$
P_2	$\{s_4\}\{s_3\}\{s_1\}\{s_0, s_2\}$	none	None

final state



DFA Minimization

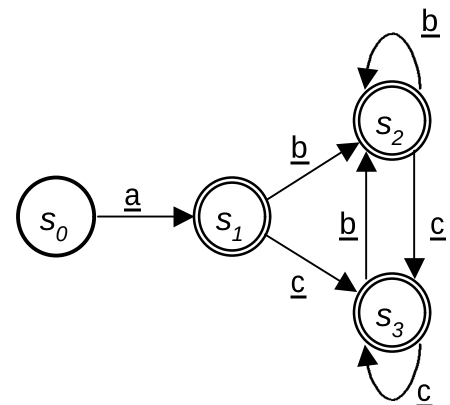
What about <u>a</u> (<u>b</u> | <u>c</u>)* ?



First, the subset construction:

		ε-closure(move(s,*))		
	NFA states	<u>a</u>	<u>b</u>	<u>c</u>
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

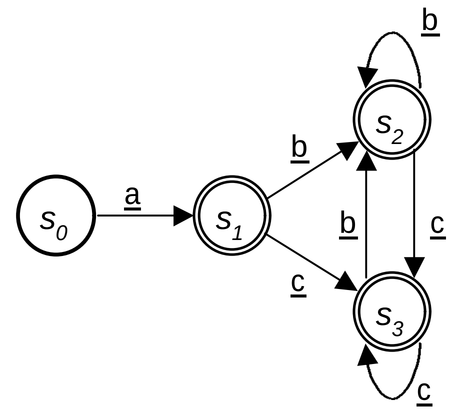


DFA Minimization

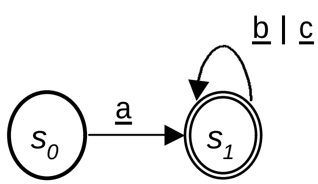
Then, apply the minimization algorithm

	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 6, I said that a human would design a simpler automaton than Thompson's construction did.

The algorithms produce that same DFA!

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

Term ® [a-zA-Z] ([a-zA-z] | [0-9])*

Op ® + | - | * | /

Expr ® (*Term Op*)* *Term*

Of course, this would generate a DFA ...

If REs are so useful ...

Why not use them for everything?

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

- Alternating 0's and 1's

  $(\varepsilon \mid 1)(0 \mid 1) (0 \mid \varepsilon)$

- Sets of pairs of 0's and 1's

  $(01 \mid 10) (01 \mid 10)^*$

RE's can count bounded sets and bounded differences

What can be so hard?

Poor language design can complicate scanning

- Reserved words are important

  if then then then = else; else else = then (PL/I)

- Significant blanks (Fortran & Algol68)

  do 10 i = 1,25

  do 10 i = 1.25

- String constants with special characters (C, others)

  newline, tab, quote, comment delimiters, ...

- Finite closures

  > Limited identifier length

  > Adds states to count length

What can be so hard? *(Fortran 66/77)*

```
          INT E G E R FUNC  TIO NA
          P ARA  ME TE R(A  =6 , B=2  )
          IMPL IC IT  CHARA   C TE R* (A-B)  (A-B)
          INT E G E R  F O RMA  T(1 0), I F(1 0),  D O 9 E1
 1 0 0    FOR  MAT (4H)  =(3 )
 2 0 0    FOR  MAT (4    )= (3 )
          D O9  E 1 = 1
          D O9  E 1 = 1 , 2
   9      IF (X)= 1
          IF (X)H= 1
          IF (X)3 0 0 , 2 0 0
 3 0 0    C O NT INU E
          E ND
C         TH IS  IS  A “C O MME NT  C ARD ”
     $    F ILE (1 )
          E ND
```

How does a compiler do this?

- First pass finds & inserts blanks
- Can add extra words or tags to create a scanable language
- Second pass is normal scanner

Example due to Dr. F.K. Zadeck