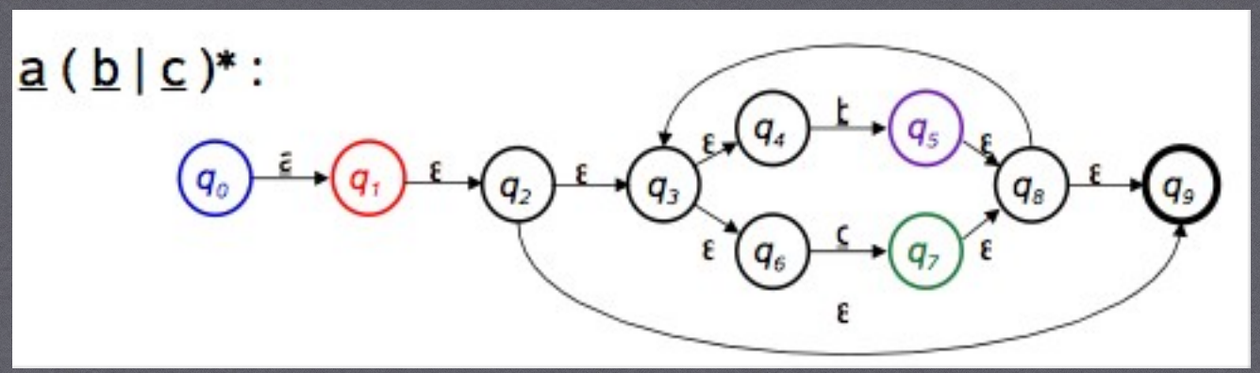


Compiling Techniques

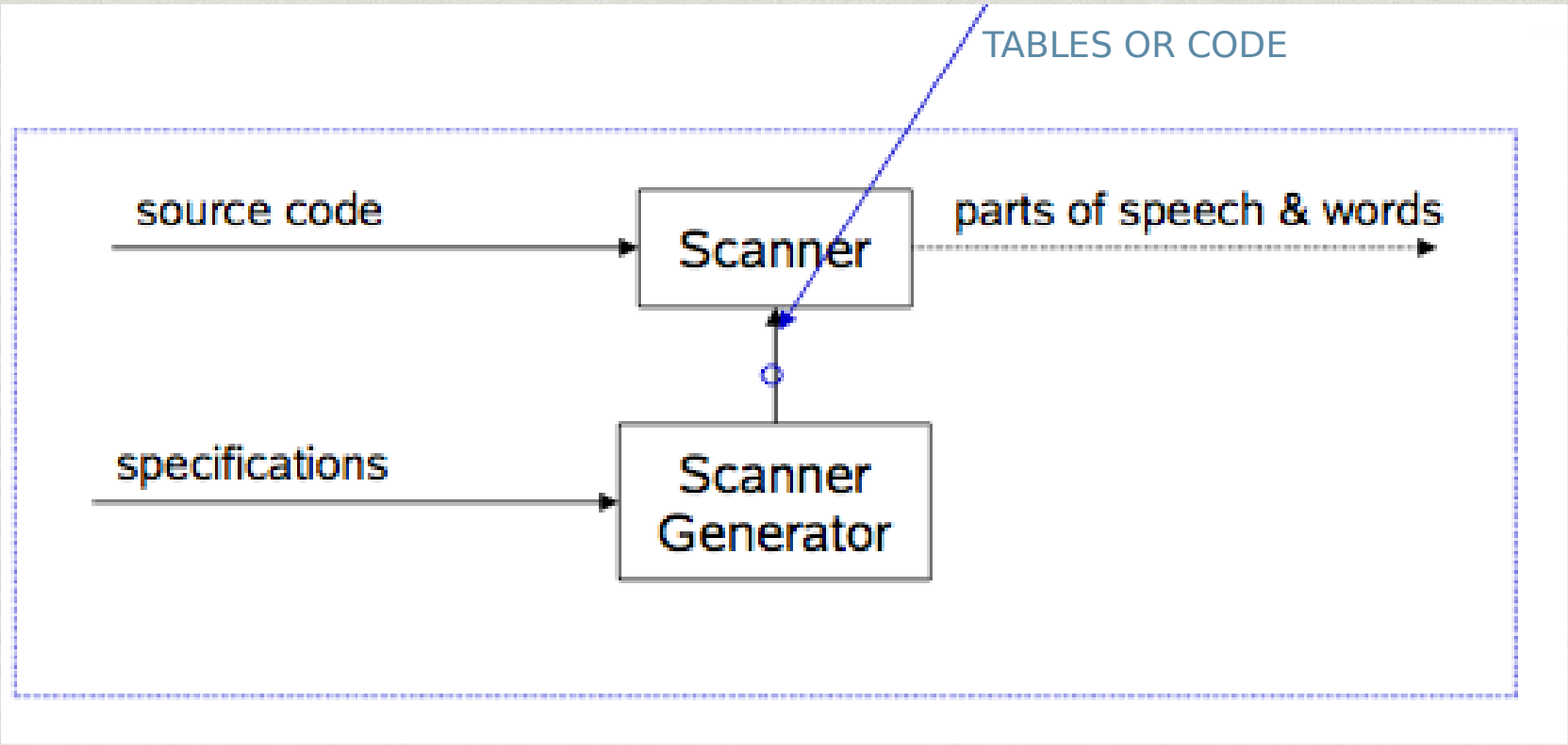
Lecture 4: Constructing a Scanner from Regular Expressions

Christophe Dubach

Overview

- Regular Expressions to NFAs
- NFAs to DFAs
- Minimisation of DFAs
- Limits of Regular Languages

Quick Review

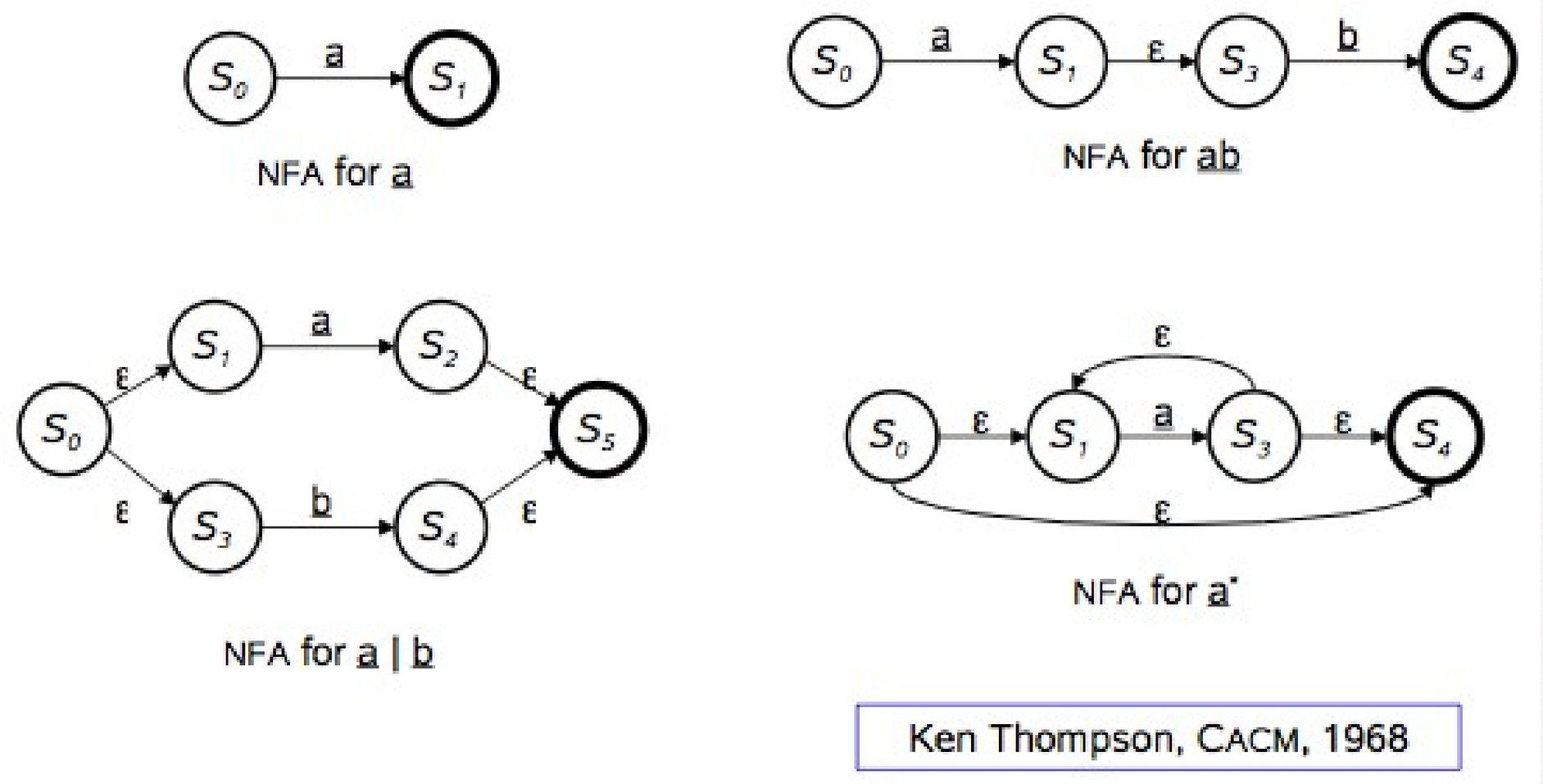


- Previous class:
  - The scanner is the first stage in the front end
  - Specifications can be expressed using regular expressions
  - Build tables and code from a DFA
  - Regular expressions, NFAs and DFAs

Goal

- Construct a finite state automaton to recognise any RE
- Overview:
  - Direct construction of a nondeterministic finite automaton (NFA) to recognise a given RE
    - Requires ε-transitions to combine regular subexpressions
  - Construct a deterministic finite automaton (DFA) to simulate the NFA
    - Use a set-of-states construction
  - Minimise the number of states
    - Hopcroft state minimisation algorithm
  - Generate the scanner code
    - Additional specifications needed for details

RE→NFA



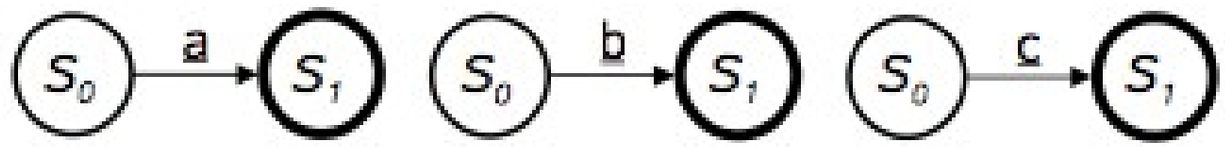
Key idea:

- NFA pattern for each symbol & each operator
- Join them with ε moves in precedence order

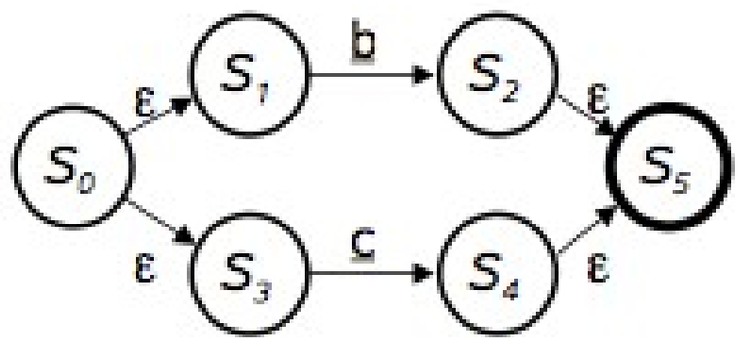
Example

a (b | c)*

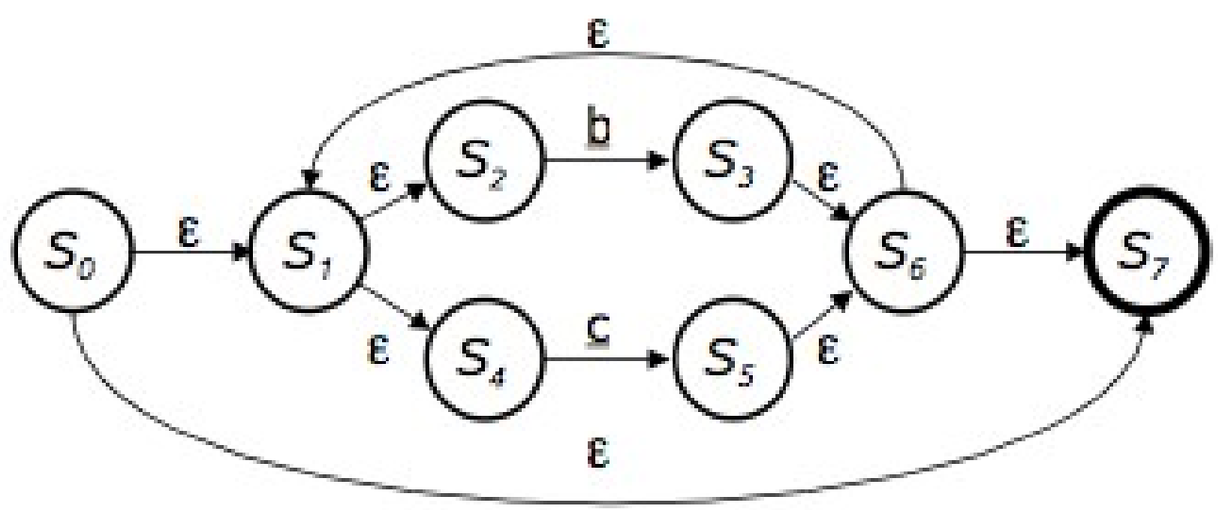
1. a, b, & c



2. b | c

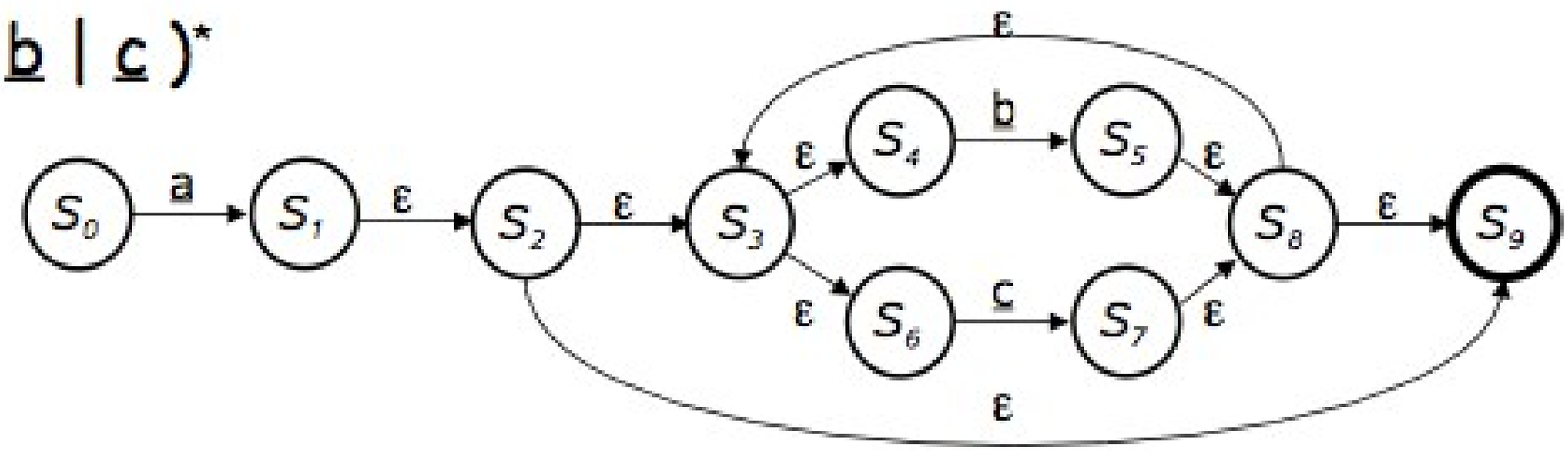


3. (b | c)*

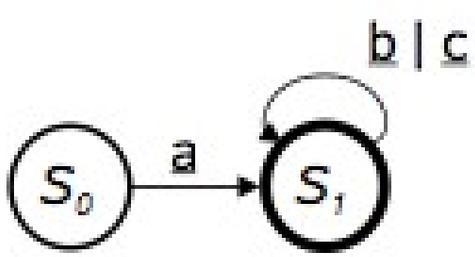


Example continued

4. a (b | c)*



Of course, a human would design something simpler ...



But, we can automate production of the more complex one ...

NFA→DFA (Subset Constr.)

- Need to build a simulation of the NFA
- Two key functions
  - Move(s_i,a) is set of states reachable from s_i by a
  - ε-closure(s_i) is set of states reachable from s_i by ε
- The algorithm:
  - Start state derived from s_0 of the NFA
  - Take its ε-closure S_0 = ε-closure(s_0)
  - Take the image of S_0, Move(S_0,α) for each α ∈ Σ, and take its ε-closure
  - Iterate until no more states are added

NFA→DFA (Subset Constr.)

The algorithm:

$s_0 \leftarrow \varepsilon\text{-closure}(q_{0n})$

while (S is still changing)
  for each $s_i \in S$
    for each $\alpha \in \Sigma$
      $s_? \leftarrow \varepsilon\text{-closure}(Move(s_i, \alpha))$
      if ($s_? \notin S$) *then*
        add $s_?$ *to S as* s_j
        $T[s_i, \alpha] \leftarrow s_j$

Let's think about why this works

The algorithm halts:

1. *S* contains no duplicates (test before adding)
2. 2^{Qn} is finite
3. while loop adds to *S*, but does not remove from *S* *(monotone)*

⇒ the loop halts

S contains all the reachable NFA states

It tries each character in each s_i.

It builds every possible NFA configuration.

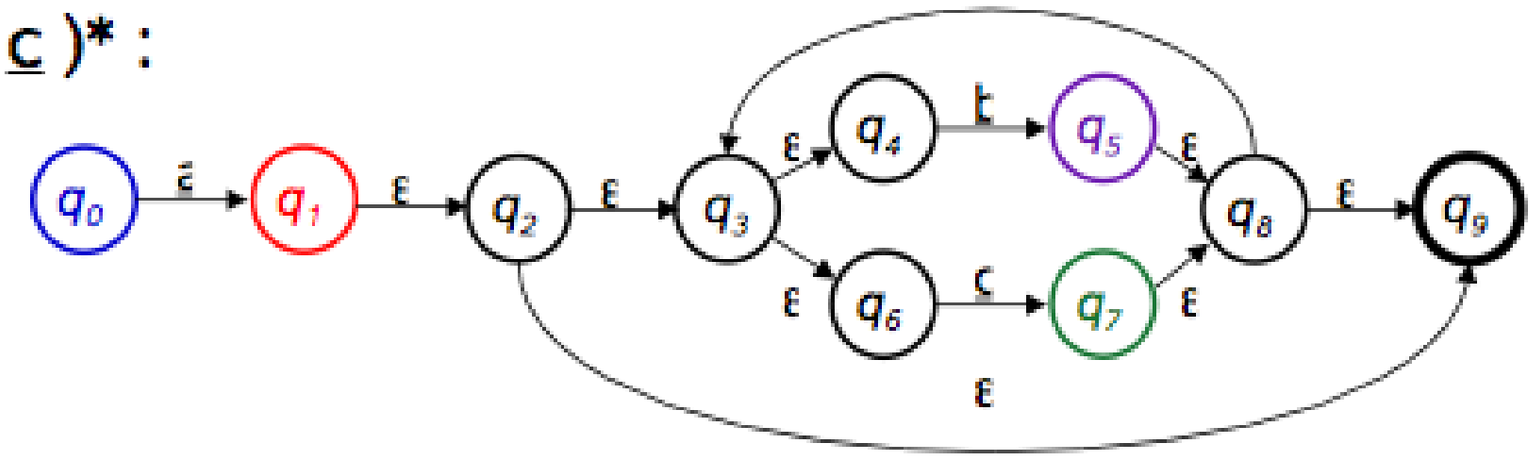
⇒ *S and T form the DFA*

NFA→DFA (Subset Constr.)

- Example of a fixed-point computation
  - Monotone construction of some finite set
  - Halts when it stops adding to the set
  - Proofs of halting & correctness are similar
  - These computations arise in many contexts
- Other fixed-point computations
  - Canonical construction of sets of LR(1) items
    - Quite similar to the subset construction
  - Classic data-flow analysis (& Gaussian Elimination)
    - Solving sets of simultaneous set equations

NFA→DFA (Subset Constr.)

a (b | c)* :

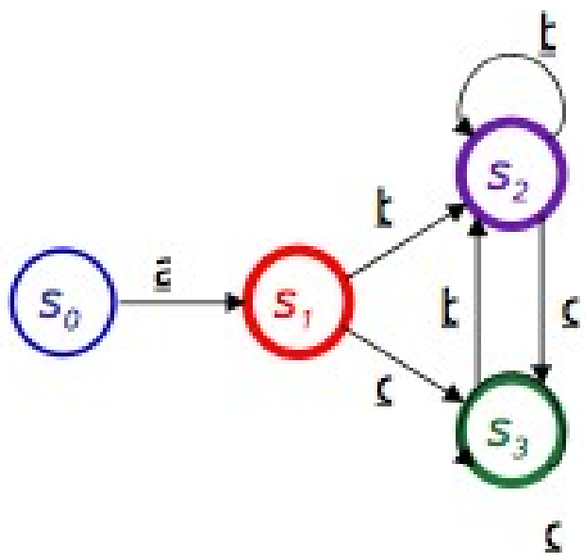


Applying the subset construction:

		ε-closure(move(s,*))		
	NFA states	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

NFA→DFA (Subset Constr.)



δ	a	b	c
s_0	s_1	-	-
s_1	-	s_2	s_3
s_2	-	s_2	s_3
s_3	-	s_2	s_3

- DFA for a (b | c)*
  - Ends up smaller than the NFA
  - All transitions are deterministic
  - Use same code skeleton as before

Where are we?

- 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
  - Hopcroft's algorithm

DFA Minimisation

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

DFA Minimisation

- 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

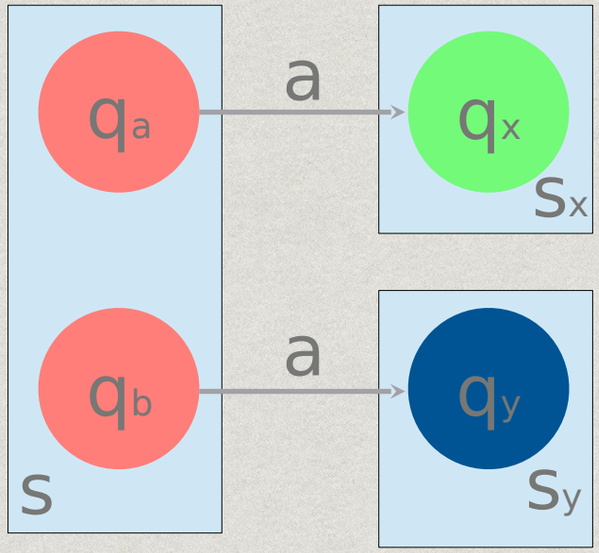
DFA Minimisation

- A partition P of S
  - Each $s \in S$ is in exactly one set $p_i \in P$
  - The algorithm iteratively partitions the DFA's states

DFA Minimisation

- Key idea of the algorithm
  - Group states into maximal size sets, optimistically
  - Iteratively subdivide those sets, as needed
  - States that remain grouped together are equivalent

DFA Minimisation



- Initial partition, P_0, has two sets:
  - {F} & {Q-F} (D =(Q,Σ,δ,q0,F))
- Splitting a set ("partitioning a set by a")
  - Assume q_a & $q_b \in s$, and $\delta(q_a,a)=q_x$ & $\delta(q_b,a)=q_y$
  - If q_x &q_y are not in the same set, then s must be split
    - one state in the final DFA cannot have two transitions on a
  - If q_a has transition on a and q_b does not ⇒ a splits s

DFA Minimisation

The algorithm

```
P ← { F, {Q-F}}
while ( P is still changing)
    T ← { }
    for each set S ∈ P
        for each α ∈ Σ
            partition S by α
                into S1, and S2
            T ← T ∪ S1 ∪ S2
    if T ≠ P then
        P ← T
```

This is a fixed-point algorithm!

Why does this work?

- Partition $P \in 2^Q$
- Start off with 2 subsets of Q *{F} and {Q-F}*
- *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 $|Q|$ sets
- Maximum of $|Q|$ splits

Note that

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

DFA Minimisation

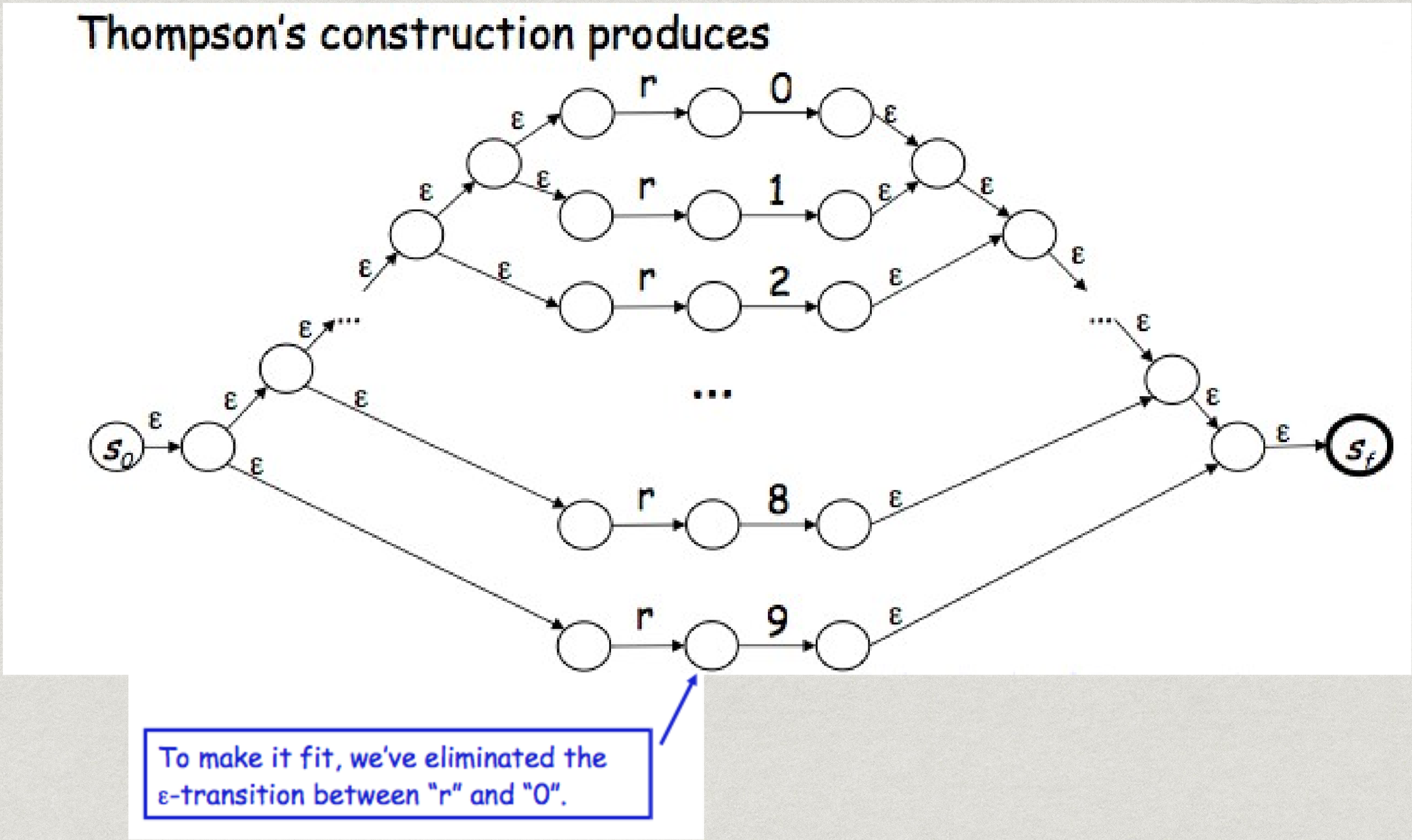
- Refining the algorithm
  - As written, it examines every $S \in P$ on each iteration
    - This does a lot of unnecessary work
    - Only examine S if some T, reachable from S, has split
- Reformulate the algorithm using a “worklist”
  - Start worklist with initial partition, F and {Q-F}
  - When it splits S into S_1 and S_2, place S_2 on worklist
- This version looks at each $S \in P$ many fewer times ⇒ Well-known, widely used algorithm due to John Hopcroft

Example RE to Min-DFA

Start with a regular expression

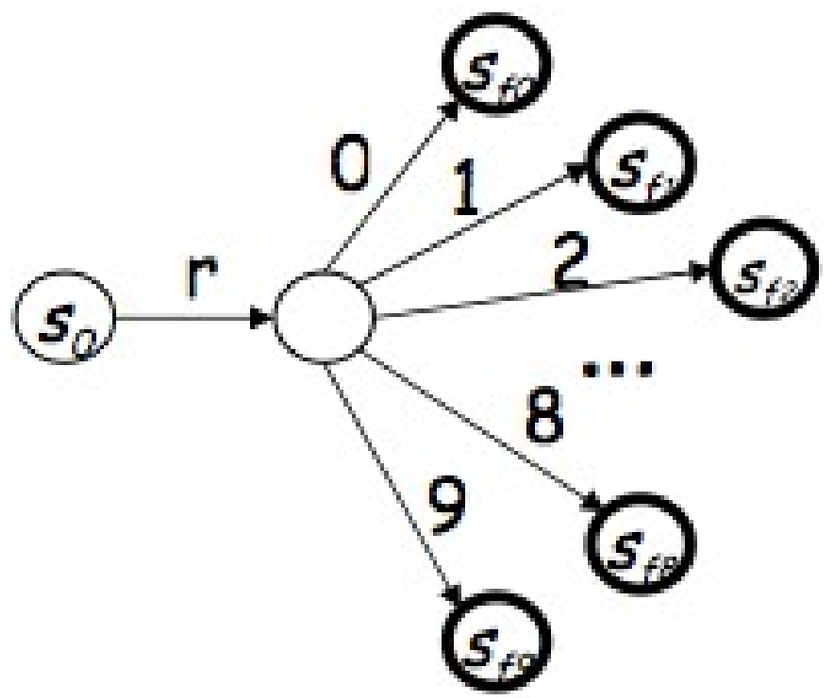
r0 | r1 | r2 | r3 | r4 | r5 | r6 | r7 | r8 | r9

Example RE to Min-DFA



Example RE to Min-DFA

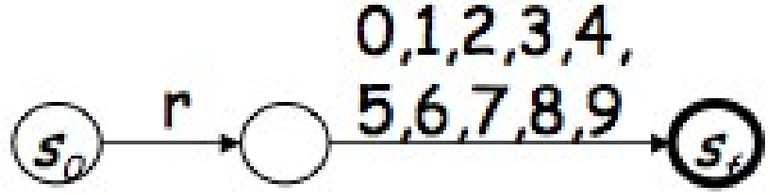
The subset construction builds



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

Example RE to Min-DFA

The DFA minimization algorithm builds



This looks like what a skilled compiler writer would do!

Limits of Regular Languages

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

Limits of Regular Languages

- Not all languages are regular RL's ⊂ CFL's ⊂ CSL's (R=regular, CF=context-free, CS=context-sensitive)
- Cannot construct DFA's to recognise:
  - $L = \{wcw \mid w \in \Sigma^*\}$
  - This is not a regular language (cannot be expressed with RE)
- But, this is a little subtle. You can construct DFA's for
  - Strings with alternating 0's and 1's: (ε | 1) (01)* (ε | 0)
  - Strings with even number of 0's or 1's

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)

- Insignificant blanks (Fortran & Algol68)

  do 10 i = 1,25 DO 10 I = 1,25 (LOOP)

  do 10 i = 1.25 DO10I = 1.25 (ASSIGNMENT)

- String constants with special characters (C, C++, Java, ...)

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

- Finite closures (Fortran 66 & Basic)

  → Limited identifier length

  → Adds states to count length

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

Preview

- Context-Free Grammars
- Introduction to Parsing