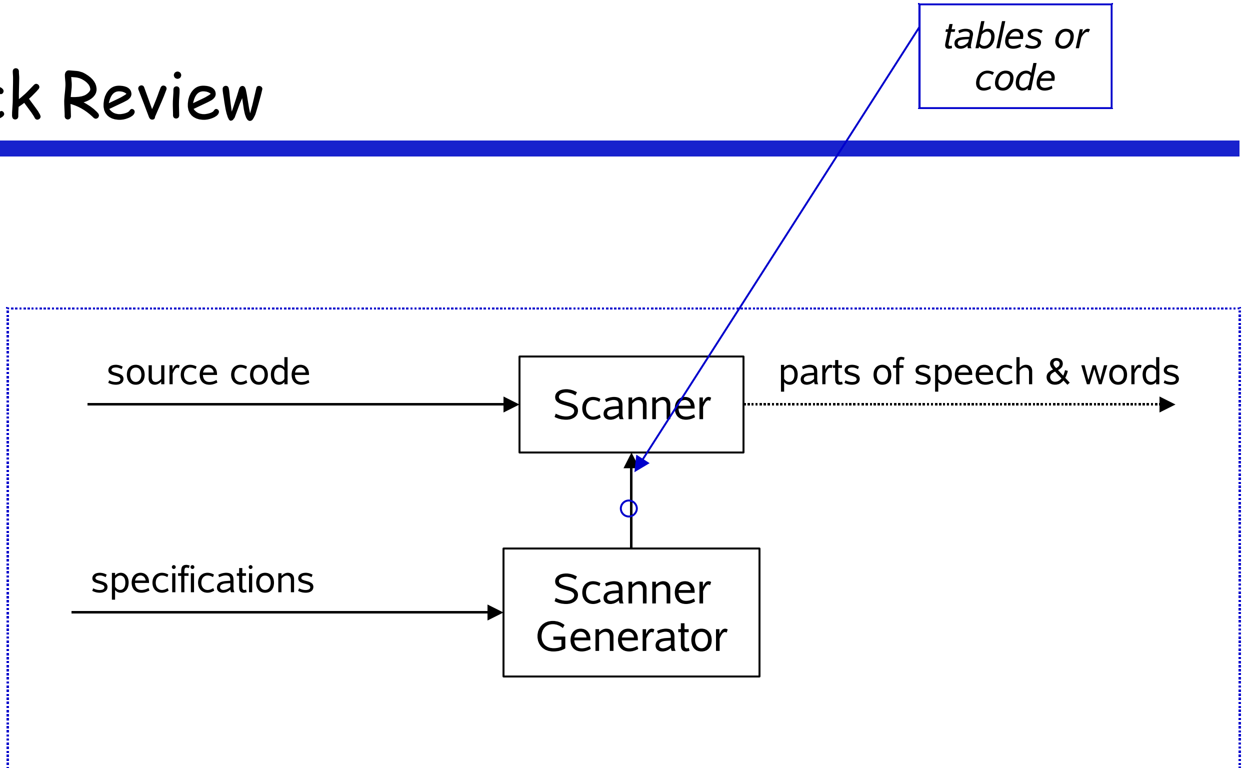
Lexical Analysis — Part II: Constructing a Scanner from Regular Expressions

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

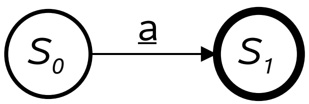
Goal

- We will show how to construct a finite state automaton to recognize any RE
- Overview:
  - → Direct construction of a nondeterministic finite automaton (NFA) to recognize 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
  - → Minimize the number of states
    - Hopcroft state minimization algorithm
  - → Generate the scanner code
    - Additional specifications needed for details

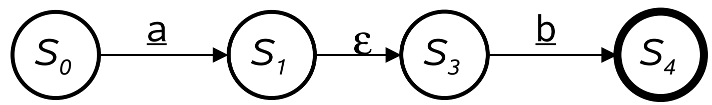
RE →NFA using Thompson's Construction

Key idea

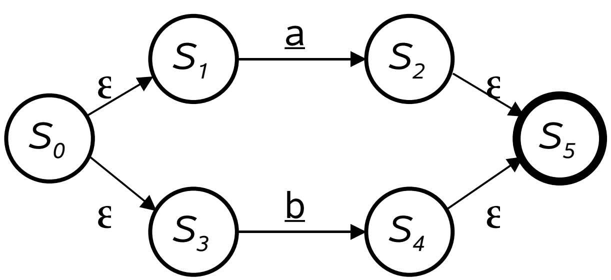
- NFA pattern for each symbol & each operator
- Join them with ε moves in precedence order



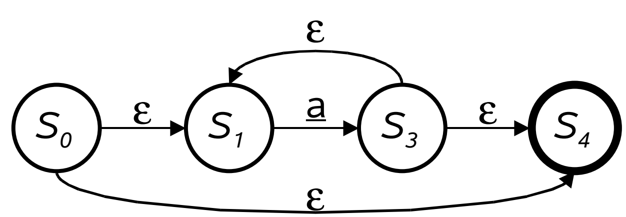
NFA for a



NFA for ab



NFA for a | b



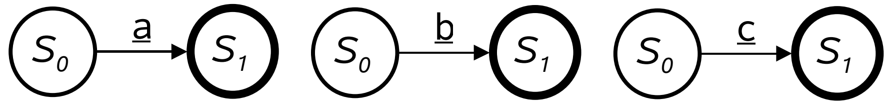
NFA for a^*

Ken Thompson, CACM, 1968

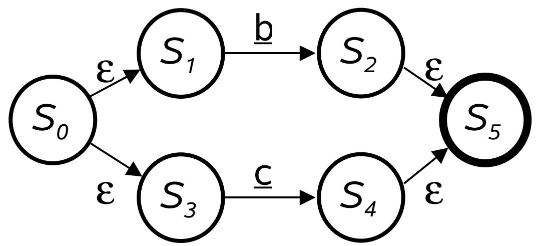
Example of Thompson's Construction

Let's try a (b | c)*

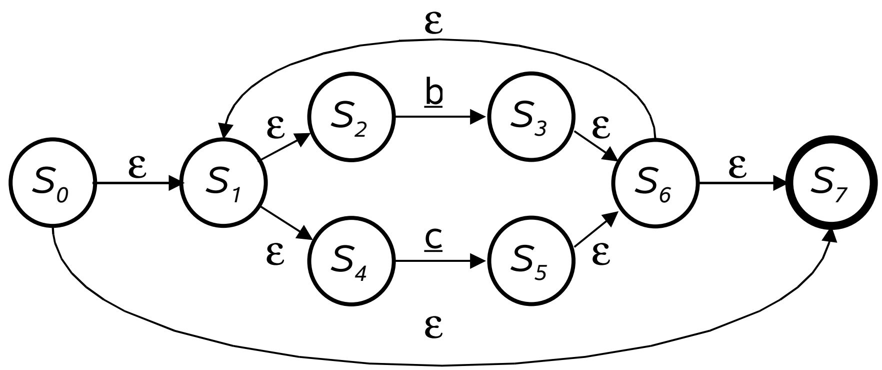
1. a, b, & c



2. b | c

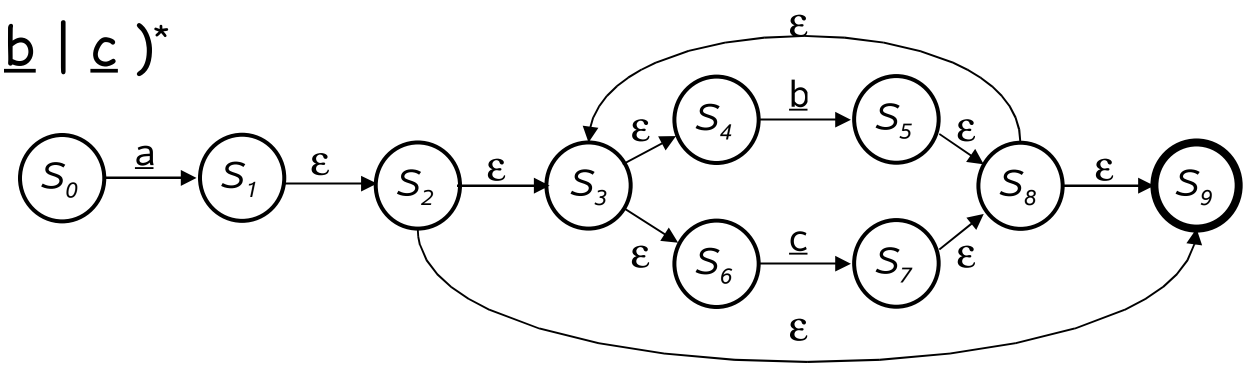


3. (b | c)*

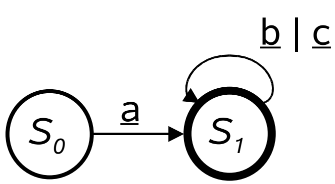


Example of Thompson's Construction (*con't*)

4. a (b | c)*



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



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

NFA →DFA with Subset Construction

Need to build a simulation of the NFA

Two key functions

- *Move*(s_i , $\underline{a}$) is set of states reachable from s_i by $\underline{a}$
- 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 $S_0 = \varepsilon\text{-closure}(s_0)$
- Take the image of S_0, Move(S_0, α) for each $\alpha \in \Sigma$, and take its ε-closure
- Iterate until no more states are added

Sounds more complex than it is…

NFA $\rightarrow$DFA with Subset Construction

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)*

$\Rightarrow$ the loop halts

S contains all the reachable NFA states

It tries each character in each s_i.

It builds every possible NFA configuration.

$\Rightarrow$ *S and T form the DFA*

NFA $\rightarrow$DFA with Subset Construction

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

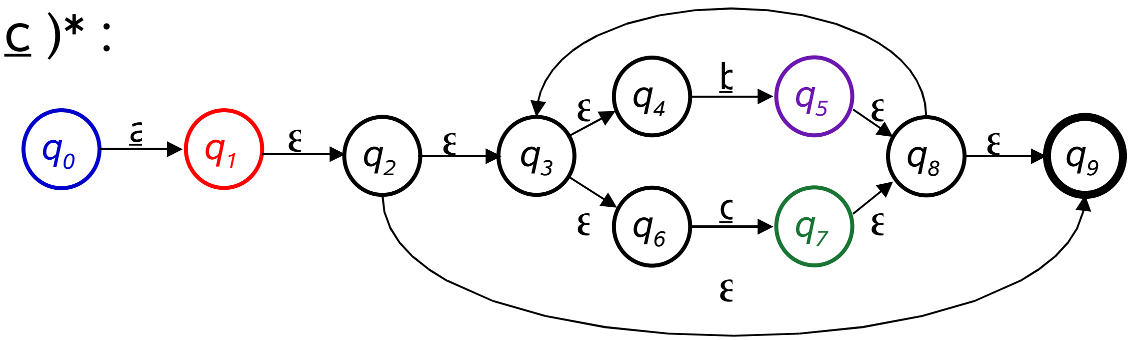
Other fixed-point computations

- 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

We will see many more fixed-point computations

NFA →DFA with Subset Construction

a (b | c)* :



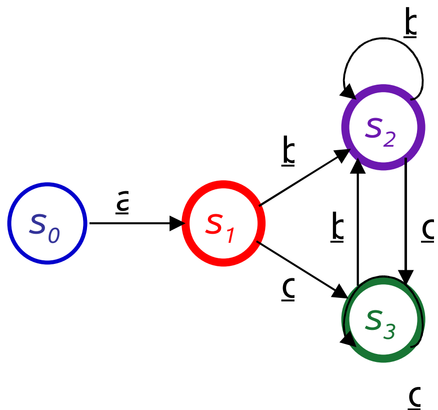
Applying the subset construction:

		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 with Subset Construction

The DFA for a (b | c)*



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

- Ends up smaller than the NFA
- *All transitions are deterministic*
- Use same code skeleton as before

Where are we? Why are we doing this?

RE $\rightarrow$ NFA *(Thompson's construction)* √

- Build an NFA for each term
- Combine them with ε-moves

NFA $\rightarrow$ DFA *(subset construction)* √

- Build the simulation

DFA $\rightarrow$ Minimal DFA

- Hopcroft's algorithm

DFA $\rightarrow$ RE

- All pairs, all paths problem
- Union together paths from s_0 to a final state

The Cycle of Constructions

$\rightarrow$ RE $\rightarrow$ NFA $\rightarrow$ DFA $\rightarrow$ *minimal* DFA

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
- 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

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

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

Initial partition, P_0, has two sets: *{F}* & *{Q-F}* *(D =(Q,Σ,δ,q_0,F))*

Splitting a set ("partitioning a set by a")

- Assume q_a, & $q_b \in s$, and $\delta(q_a,\underline{a}) = q_x$, & $\delta(q_b,\underline{a}) = q_y$
- If q_x & q_y are not in the same set, then s must be split
  - $\rightarrow$ q_a has transition on a, q_b does not $\Rightarrow$ <u>a</u> splits s
- One state in the final DFA cannot have two transitions on <u>a</u>

DFA Minimization

The algorithm

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

while (P is still changing)

$\quad T \leftarrow \{\ \}$

$\quad$ *for each set* $S \in P$

$\quad\quad$ *for each* $\alpha \in \Sigma$

$\quad\quad\quad$ *partition* S *by* α

$\quad\quad\quad\quad$ *into* S_1, *and* S_2

$\quad\quad\quad T \leftarrow T \cup S_1 \cup S_2$

$\quad$ *if* $T \neq P$ *then*

$\quad\quad P \leftarrow T$

This is a fixed-point algorithm!

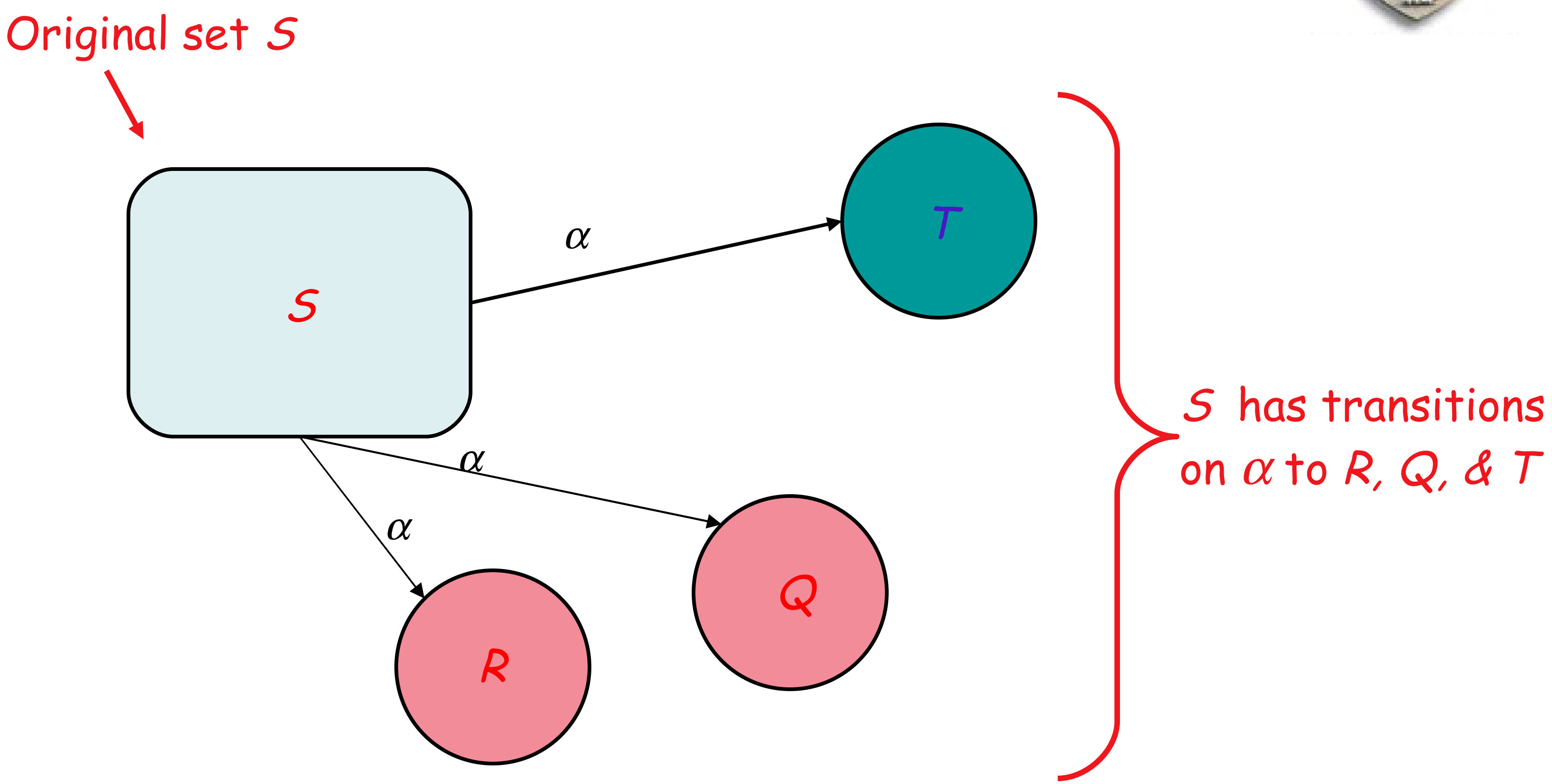
Why does this work?

- Partition $P \in 2^Q$
- Start off with 2 subsets of Q $\{F\}$ and $\{Q\text{-}F\}$
- *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

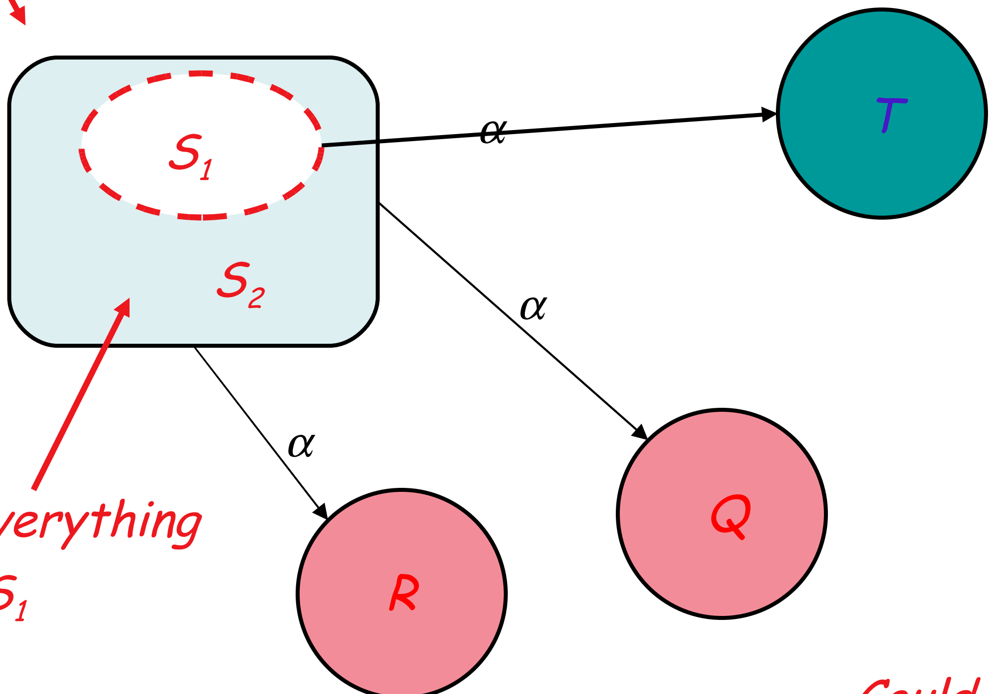
Key Idea: Splitting S around α



The algorithm partitions S around α

Key Idea: Splitting S around α

Original set S



S_2 is everything in $S - S_1$

Could we split S_2 further?

Yes, but it does not help asymptotically

DFA Minimization

Refining the algorithm

- As written, it examines every $S \in P$ on each iteration
  - → This does a lot of unnecessary work
  - → Only need to 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

Abbreviated Register Specification

Start with a regular expression

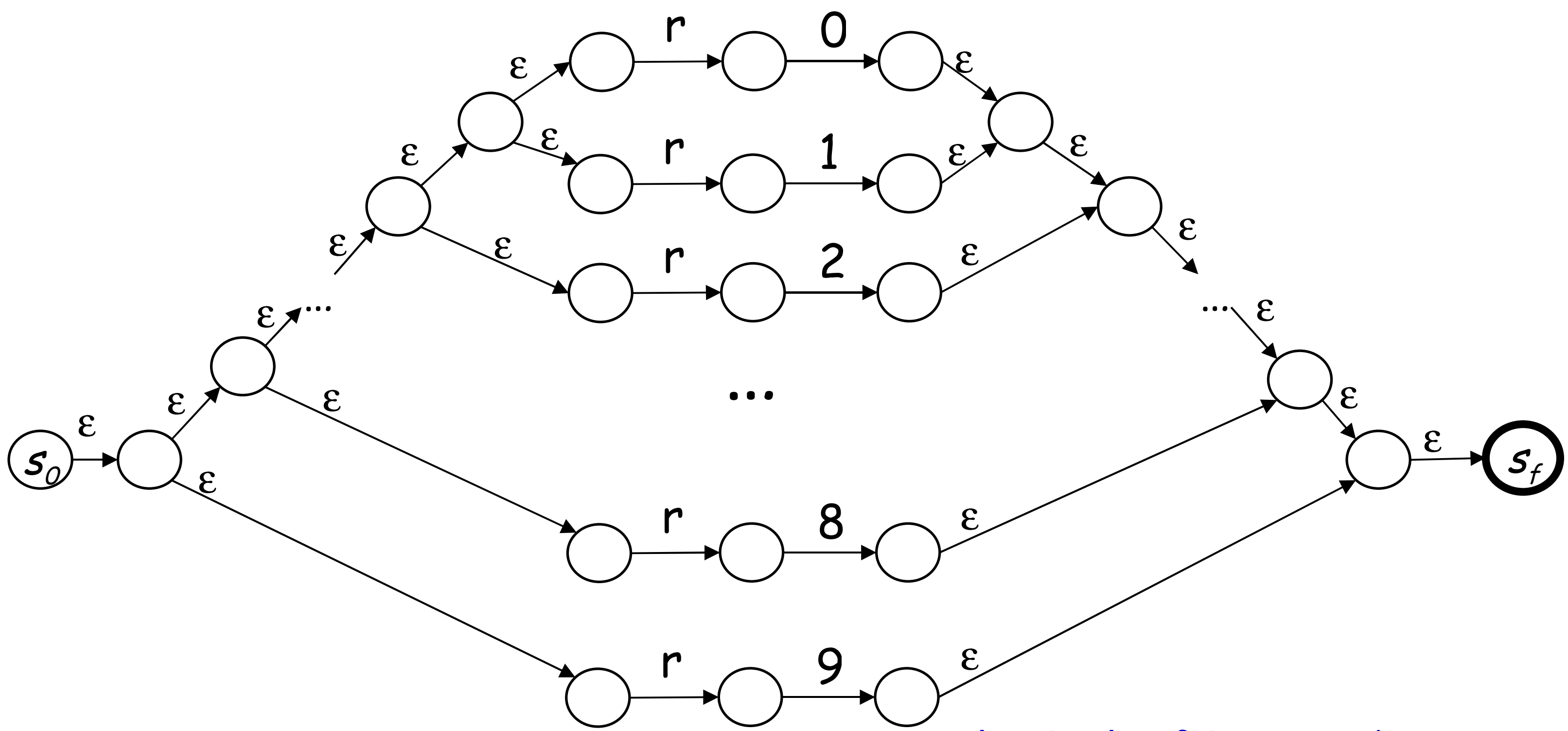
r0 | r1 | r2 | r3 | r4 | r5 | r6 | r7 | r8 | r9

The Cycle of Constructions

→ RE → NFA → DFA → *minimal* DFA

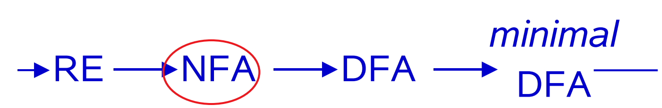
Abbreviated Register Specification

Thompson's construction produces



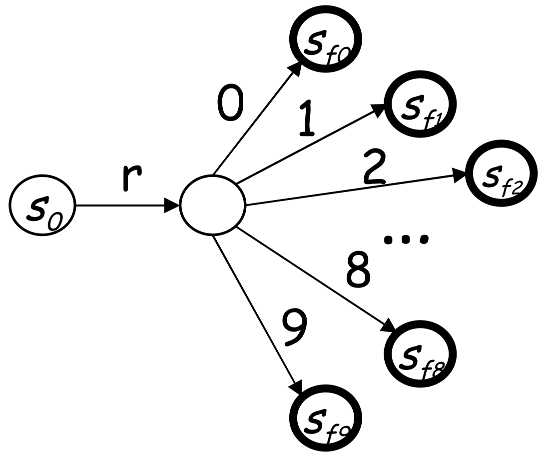
To make it fit, we've eliminated the ε-transition between "r" and "0".

The Cycle of Constructions



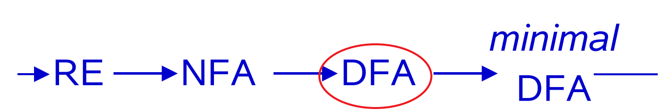
Abbreviated Register Specification

The subset construction builds



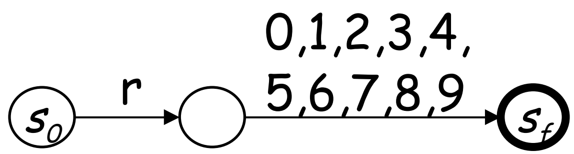
This is a DFA, but it has a lot of states …

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

→ RE → NFA → DFA → *minimal* 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

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

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

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

Of course, this would generate a DFA …

If REs are so useful …

Why not use them for everything?

Limits of Regular Languages

Not all languages are regular

RL's ⊂ CFL's ⊂ 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 can 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

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)

- Insignificant blanks (Fortran & Algol68)

  do 10 i = 1,25

  do 10 i = 1.25

- 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