RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

CS415 Compilers

Lexical Analysis

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University

Review: Automating Scanner Construction

RE→ NFA *(Thompson's construction)*

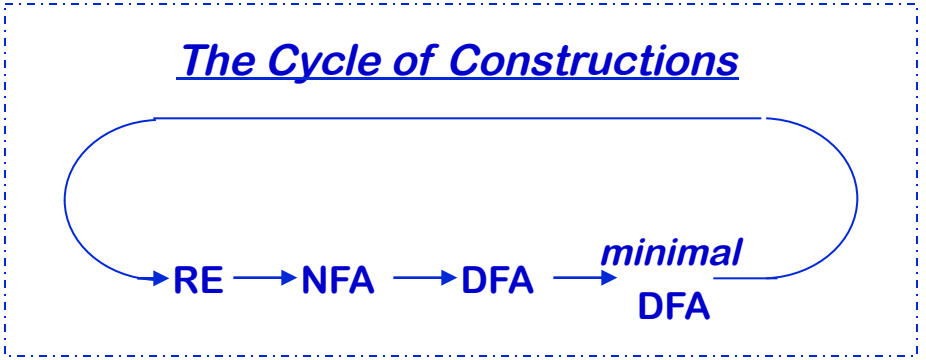
- Build an NFA for each term
- Combine them with ε-moves

NFA → DFA *(subset construction)*

- Build the simulation

DFA → Minimal DFA

- Hopcroft's algorithm



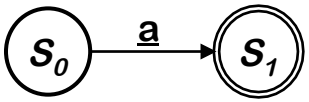
DFA →RE *(Not part of the scanner construction)*

- All pairs, all paths problem
- Take the union of all paths from s_0 to an accepting state

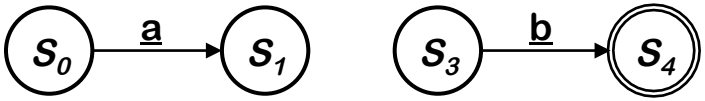
RE →NFA using Thompson's Construction

Key idea

- NFA pattern for each symbol and each operator
- Each NFA has a single start and accept state
- Join them with ε moves in precedence order



NFA for a



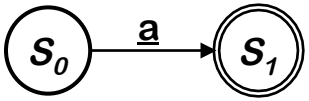
NFA for ab

Ken Thompson, CACM, 1968

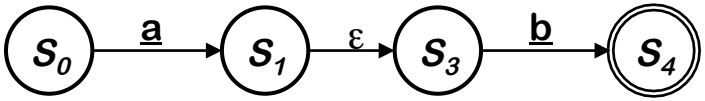
RE →NFA using Thompson's Construction

Key idea

- NFA pattern for each symbol and each operator
- Each NFA has a single start and accept state
- Join them with ε moves in precedence order



NFA for a



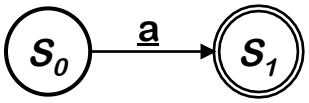
NFA for ab

Ken Thompson, CACM, 1968

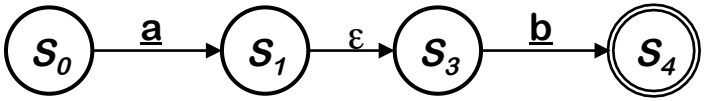
RE →NFA using Thompson's Construction

Key idea

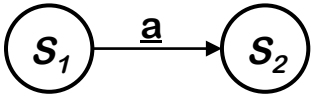
- NFA pattern for each symbol and each operator
- Each NFA has a single start and accept state
- Join them with ε moves in precedence order

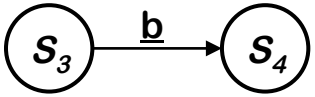


NFA for a



NFA for ab





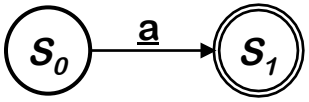
NFA for a | b

Ken Thompson, CACM, 1968

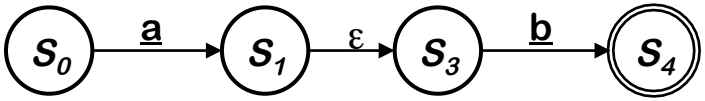
RE →NFA using Thompson's Construction

Key idea

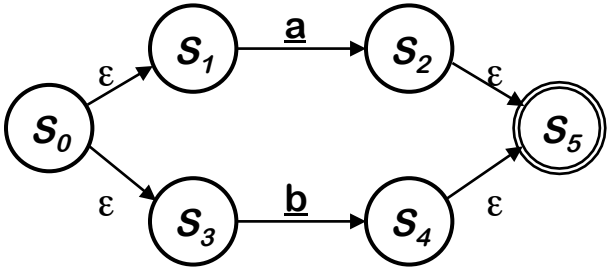
- NFA pattern for each symbol and each operator
- Each NFA has a single start and accept state
- Join them with ε moves in precedence order



NFA for a



NFA for ab



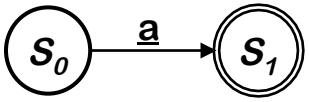
NFA for a | b

Ken Thompson, CACM, 1968

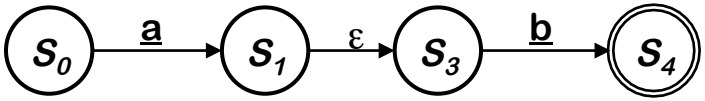
RE →NFA using Thompson's Construction

Key idea

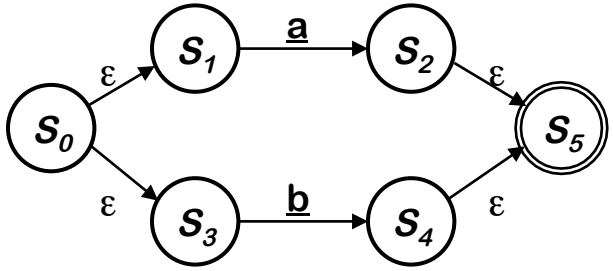
- NFA pattern for each symbol and each operator
- Each NFA has a single start and accept state
- Join them with ε moves in precedence order



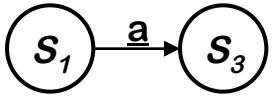
NFA for a



NFA for ab



NFA for a | b

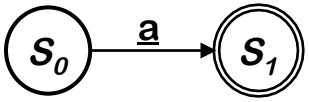


Ken Thompson, CACM, 1968

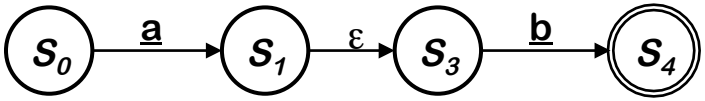
RE →NFA using Thompson's Construction

Key idea

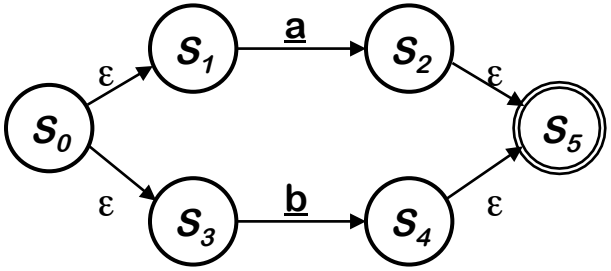
- NFA pattern for each symbol and each operator
- Each NFA has a single start and accept state
- Join them with ε moves in precedence order



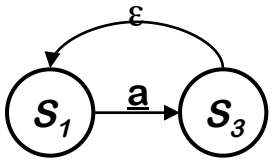
NFA for a



NFA for ab



NFA for a | b

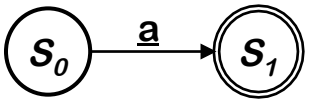


Ken Thompson, CACM, 1968

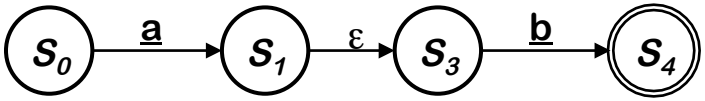
RE →NFA using Thompson's Construction

Key idea

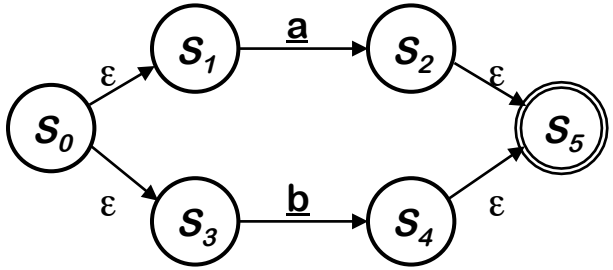
- NFA pattern for each symbol and each operator
- Each NFA has a single start and accept state
- Join them with ε moves in precedence order



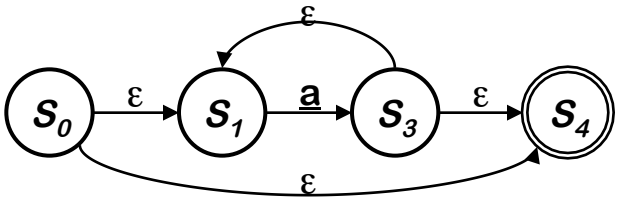
NFA for a



NFA for ab



NFA for a | b



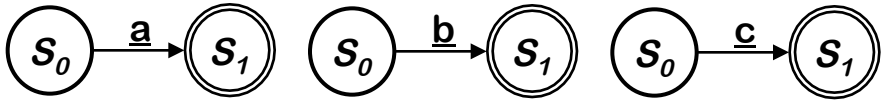
NFA for a^*

Ken Thompson, CACM, 1968

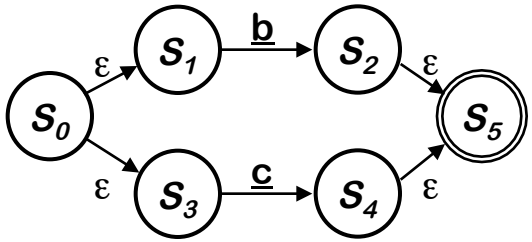
Example of Thompson's Construction

Let's try a (b | c)*

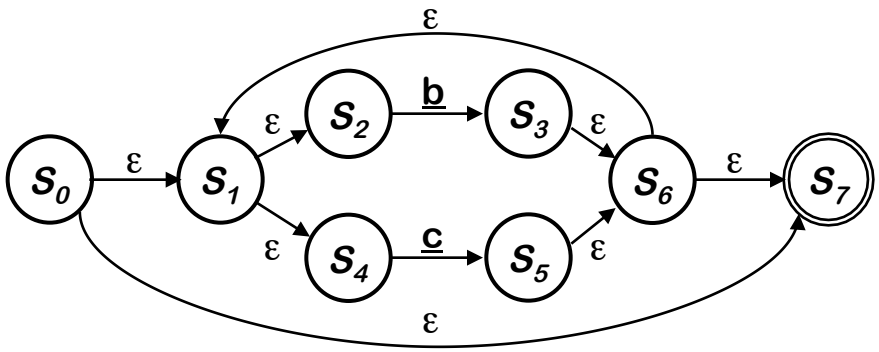
1. a, b, & c



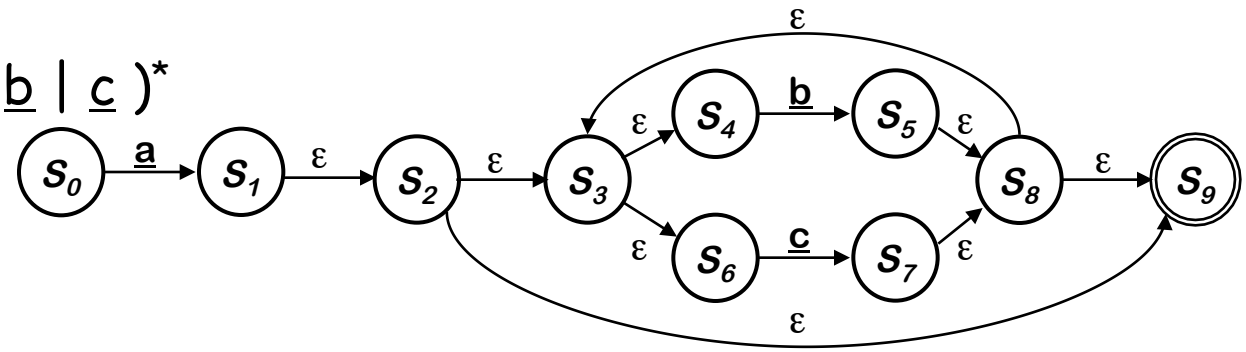
2. b | c



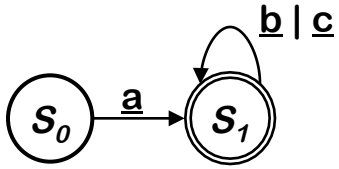
3. (b | c)*



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 , 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 (sketch):

- Start state derived from s_0 of the NFA
- Take its ε-closure $S_0 = \varepsilon\text{-closure}(s_0)$
- For each state S, compute move(S, a) for each $a \in \Sigma$, and take its ε-closure
- Iterate until no more states are added

Sounds more complex than it is…

NFA →DFA with Subset Construction

The algorithm:

$s_0 \leftarrow \varepsilon\text{-closure}(q_0)$

add s_0 to S

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

Let's think about why this works

The algorithm halts:

1. *S* contains no duplicates (test before adding)
2. $2^{|Q|}$ 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 symbol in each s_i.

It builds every possible NFA configuration.

⇒ *S and T form the DFA*

NFA →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

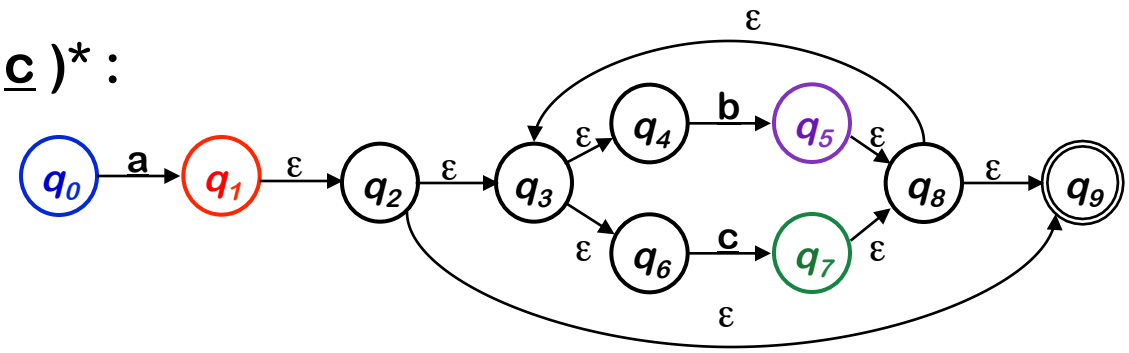
Other fixed-point computations

- Canonical construction of sets of LR(1) items
  - → Quite similar to the subset construction
- Classic data-flow analysis
  - → Solving sets of simultaneous set equations
- DFA minimization algorithm (coming up!)

We will see many more fixed-point computations

NFA →DFA with Subset Construction

a (b | c)* :



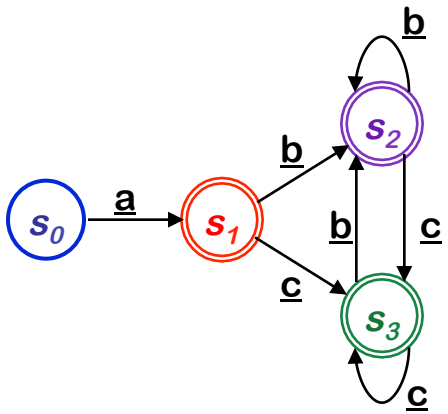
Applying the subset construction:

		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

Automating Scanner Construction

RE→NFA *(Thompson's construction)*

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

NFA →DFA *(subset construction)*

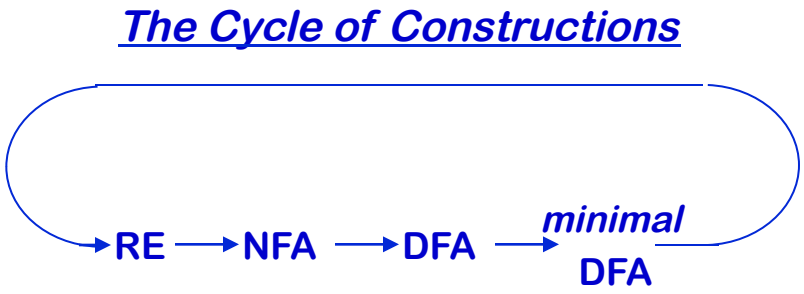
- Build the simulation

DFA →Minimal DFA

- 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



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:

- $\forall\ a \in \Sigma$, transitions on a lead to equivalent states (set) (DFA)

If a-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:

- $\forall\ a \in \Sigma$, transitions on a lead to equivalent states (DFA)

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

A partition P of S

- Each state $s \in S$ 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: $\{F\}$ & $\{Q-F\}$ $\quad (D = (Q, \Sigma, \delta, q_0, F))$

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

- 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
  - → q_a has transition on a, q_b does not ⇒ $\underline{a}$ splits s

DFA Minimization

The algorithm

```
P ← { F, {Q-F}}
while ( P is still changing)
    T ← { }
    for each set S ∈ P
          T ← T ∪ split(S)
    P ← T

split(S):
 for each a∈ Σ
    if a splits S into
         S1 , S2 , … then
            return {S1 , S2, …}
    else return S
```

This is a fixed-point algorithm!

Why does this work?

- F is the set of final states
- 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

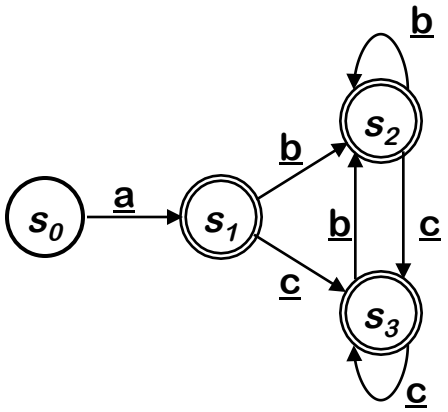
- Partitions are <u>never</u> combined

Back to our DFA Minimization example

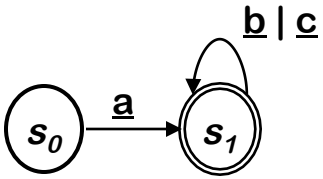
Then, apply the minimization algorithm

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

final states



To produce the minimal DFA



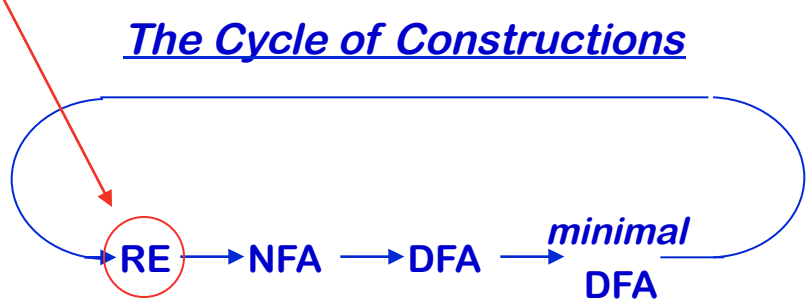
We observed that a human would design a simpler automaton than Thompson's construction & the subset construction did.

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

Abbreviated Register Specification

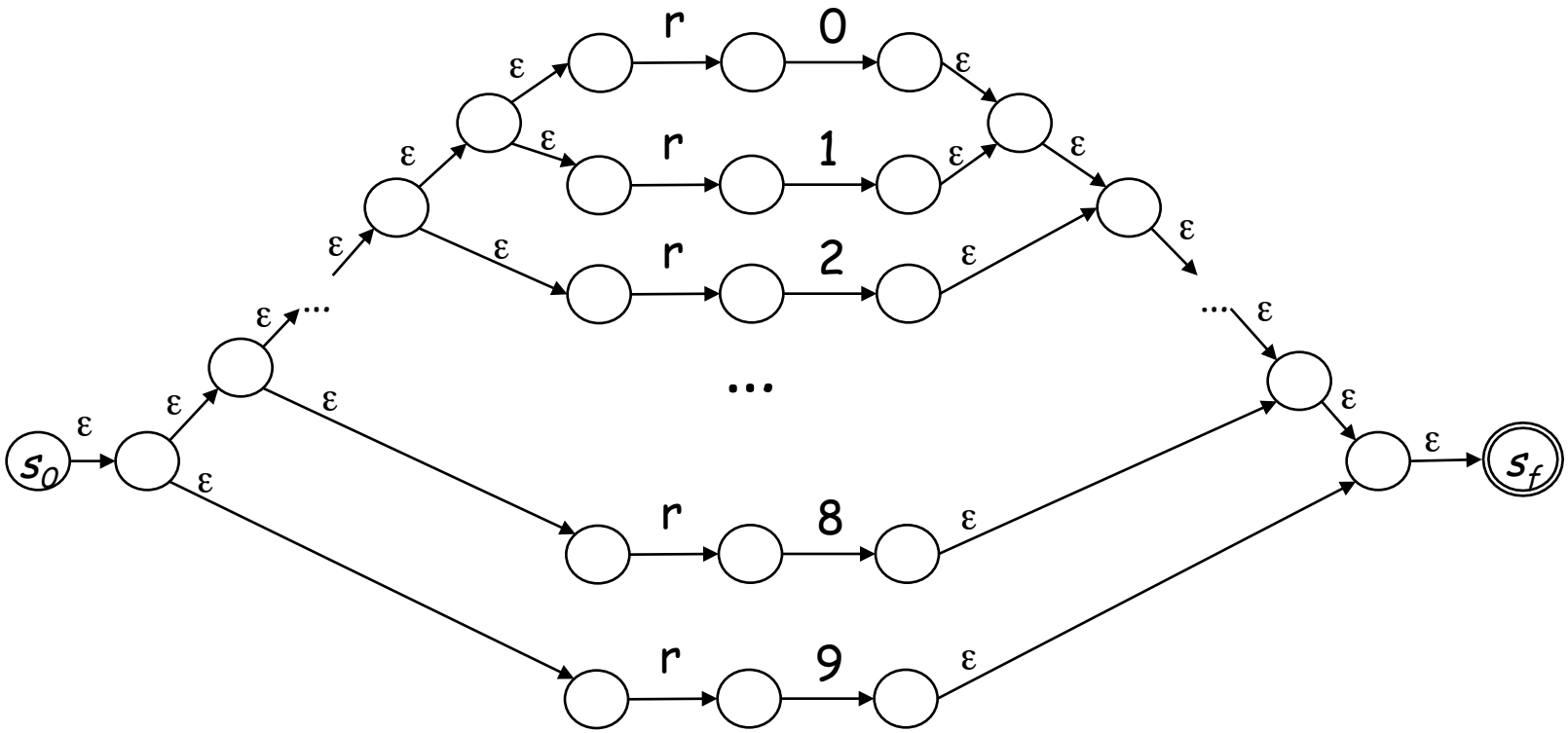
Start with a regular expression

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



Abbreviated Register Specification

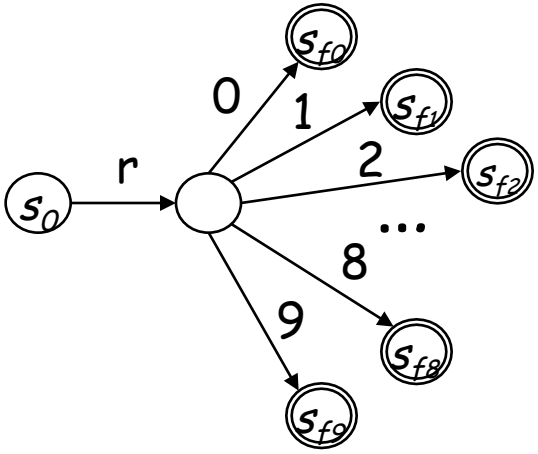
Thompson's construction produces



The Cycle of Constructions

Abbreviated Register Specification

The subset construction builds

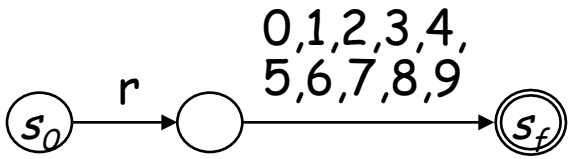


This is a DFA, but it has a lot of states …

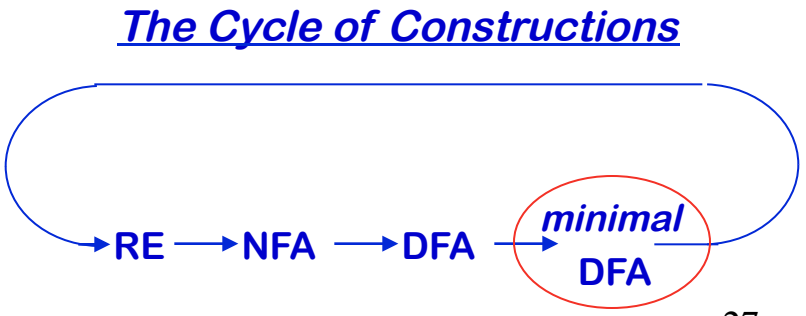
The Cycle of Constructions

Abbreviated Register Specification

The DFA minimization algorithm builds



This looks like what a skilled compiler writer would do!



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

- Strings with alternating 0's and 1's

  $(\varepsilon \mid 1) (01)^* (\varepsilon \mid 0)$
- Strings with an even number of 0's and 1's
- Strings of bit patterns that represent binary numbers which are divisible by 5

Announcement

1. Project 1 posted
2. Homework set 3 will be posted later today

Next Class: Syntax Analysis

Read EaC: 3.1 – 3.3