RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

CS415 Compilers

Lexical Analysis
Part 4

Announcements

- Third homework has been posted. Due Monday, March 7

- First project (local instruction scheduler) NEW deadlines:
  code: March 9
  report: March 11

Project 1 clarifications

1. The order in which programs perform I/O has to be preserved. This requires a dependence notion on outputAI instructions.

2. Benchmark codes have storeAI and loadAI memory accesses only, with r0 as the base register. You must use the offset to determine whether a dependence exists or not.

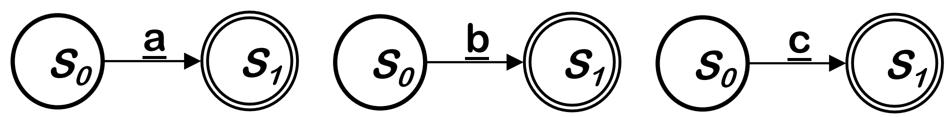
3. We will have some private tests that have store and load memory accesses. This is for extra credit.

4. Project reports should be around 6 pages long, with a max of 8 pages. We will not read your report beyond 8 pages. The report should include a short description of what you did, the outcome of your experiments, and how you interpret these outcomes. Use graphs/figures to show your results.

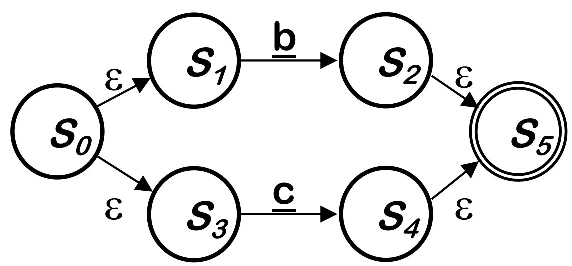
Example of Thompson's Construction

Let's try a (b | c)*

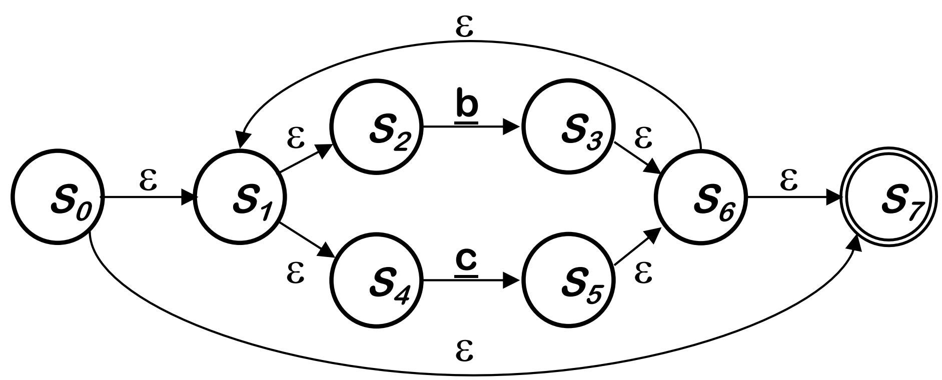
1. a, b, & c



2. b | c

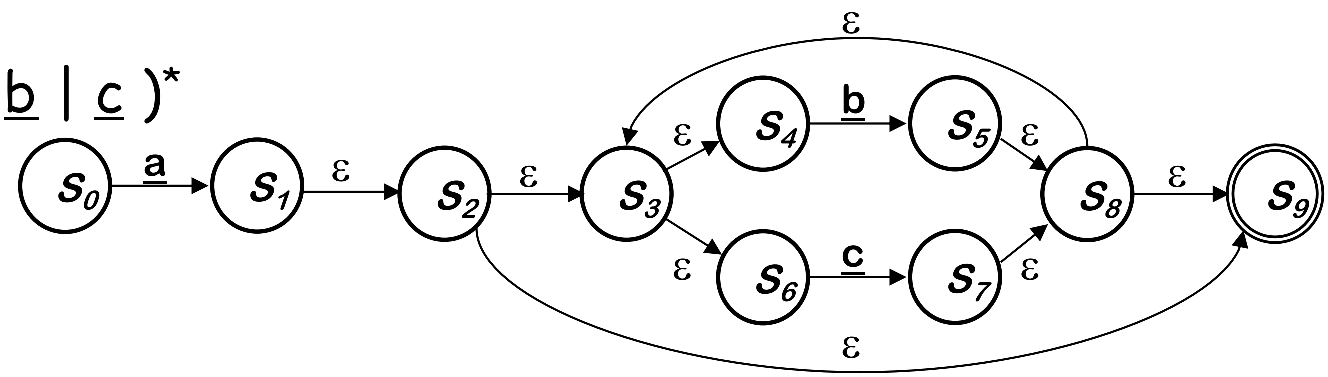


3. (b | c)*

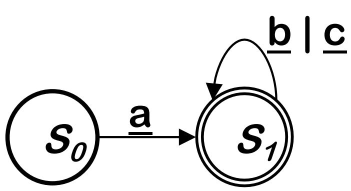


Example of Thompson's Construction (*con't*)

4. a (b | c)*



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



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

NFA $\rightarrow$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 set of states s_i by $\underline{a}$
- *ε-closure*(s_i) is set of states reachable from set of states s_i by ε

The algorithm (sketch):

- Start state derived from s_0 of the NFA
- Take its ε-closure S_0 = ε-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 $\rightarrow$DFA with Subset Construction

The algorithm:

$s_0 \leftarrow \varepsilon$-closure($q_0$)

add s_0 to S

while (S is still changing)
  for each $s_i \in S$
    for each $a \in \Sigma$
      $s_? \leftarrow \varepsilon$-closure(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)*

$\Rightarrow$ the loop halts

S contains all the reachable NFA states

It tries each symbol in each s_i.

It builds every possible NFA configuration.

$\Rightarrow$ *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

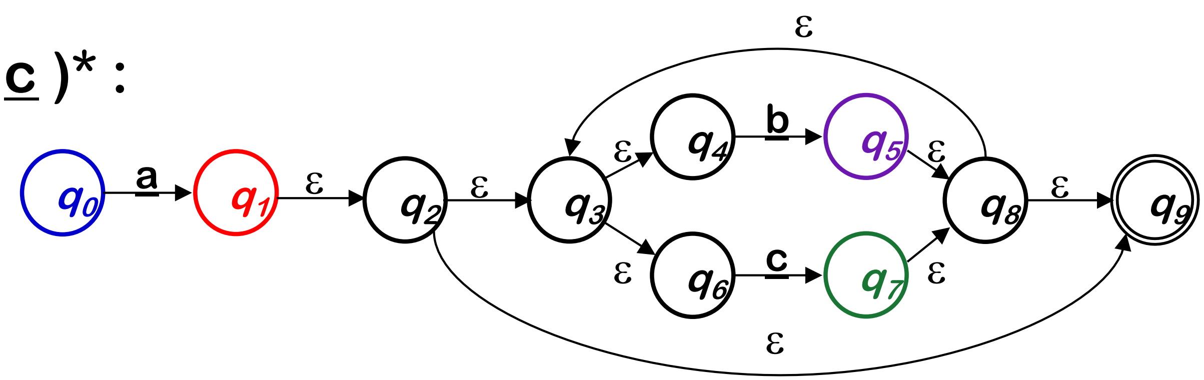
Other fixed-point computations

- 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

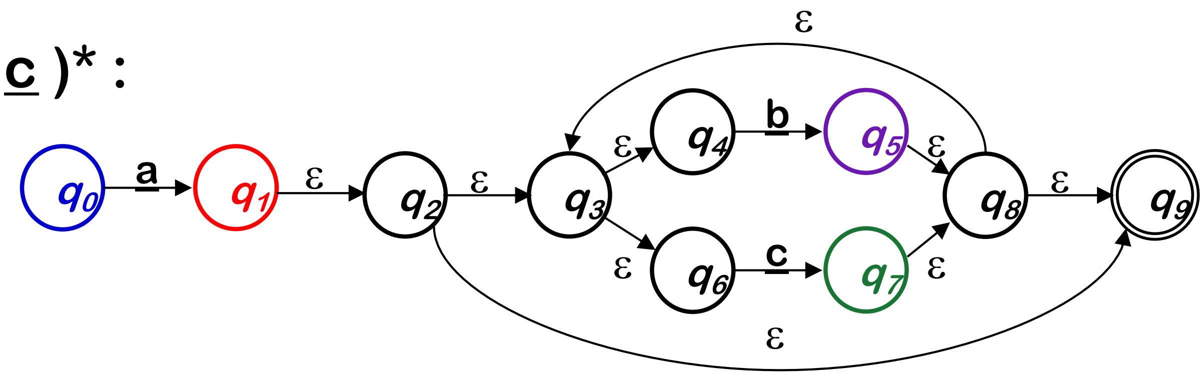
a (b | c)* :



Applying the subset construction:

NFA →DFA with Subset Construction

a (b | c)* :



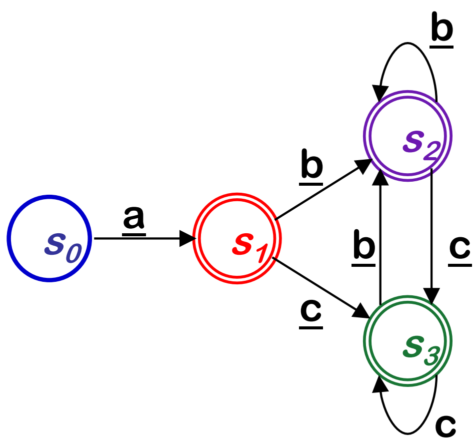
Applying the subset construction:

		ε-closure(move(s,*))		
	Sets of 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

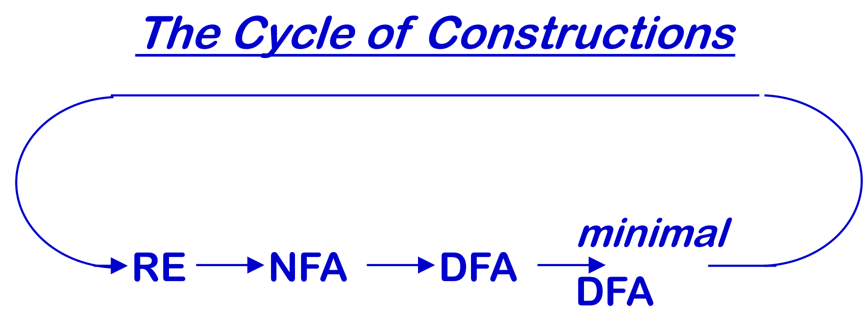
The DFA for a (b | c)*



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

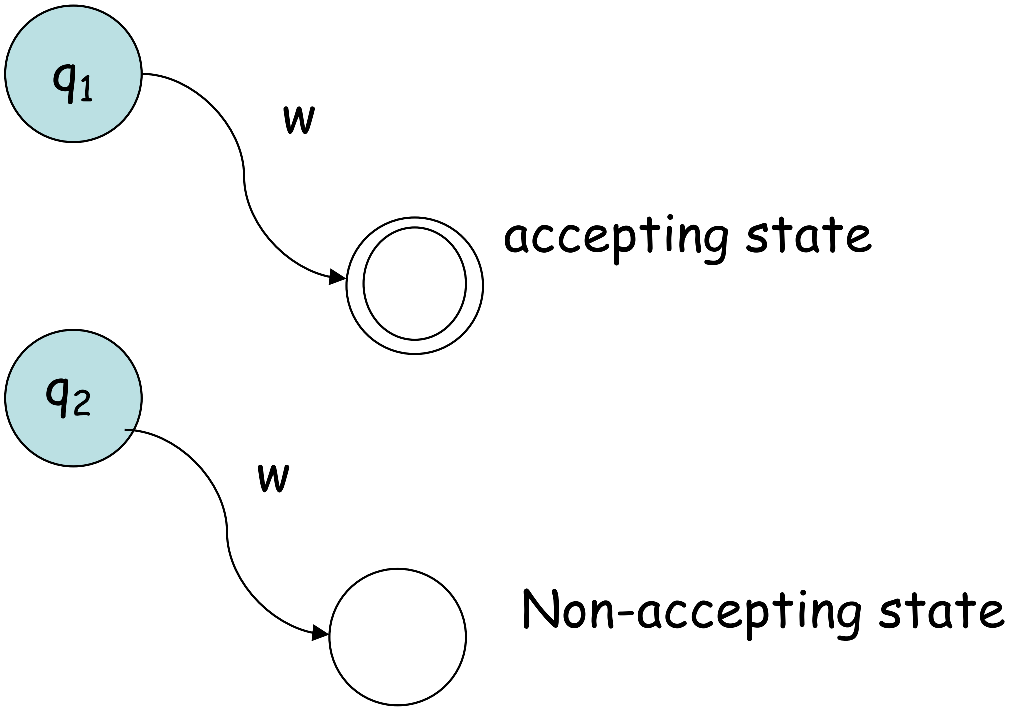
- Hopcroft's algorithm

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

How do we know whether two states encode the same information?



Intuition: Two states are equivalent if for all sequences of input symbols "w" they both lead to an accepting state, or both end up in a non-accepting state.

q_1 and q_2 are not equivalent.
"w" is a witness that they are not equivalent.

DFA Minimization

The Big Picture

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

RUTGERS

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 different sets $\Rightarrow$ two states must be in different sets, i.e., cannot be equivalent

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 different sets $\Rightarrow$ two states must be in different sets, i.e., cannot be equivalent

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: $\{F\}$ & $\{Q-F\}$ $(D = (Q,\Sigma,\delta,q_0,F))$

Splitting a set ("partitioning a set s 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, i.e., are considered equivalent, then s must be split
  - → q_a has transition on a, q_b does not ⇒ a splits s

DFA Minimization

The algorithm

$P \leftarrow \{ F, \{Q-F\}\}$
while (P is still changing)
  $T \leftarrow \{ \}$
  for each set $S \in P$
    $T \leftarrow T \cup split(S)$
  $P \leftarrow T$

split(S):
 for each $a \in \Sigma$
  if a splits S into
   $S_1, S_2, \ldots$ *then*
    return $\{S_1, S_2, \ldots\}$
  else return S

This is a fixed-point algorithm!

Why does this work?

- 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

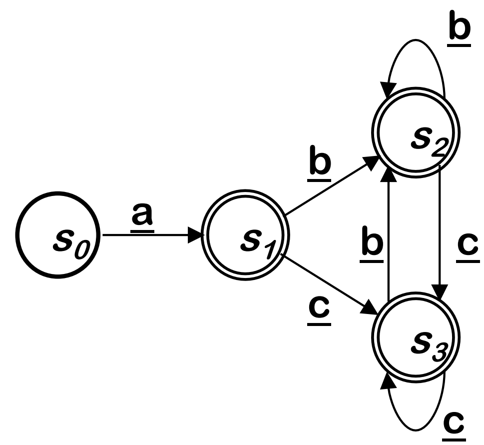
- Partitions are <u>never</u> combined

Back to our DFA Minimization example

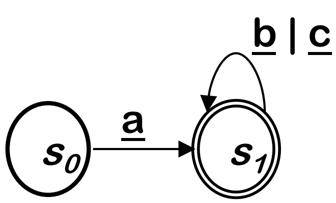
Then, apply the minimization algorithm

	Current Partition	Split on		
		a	b	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!

Another Example Register Specification

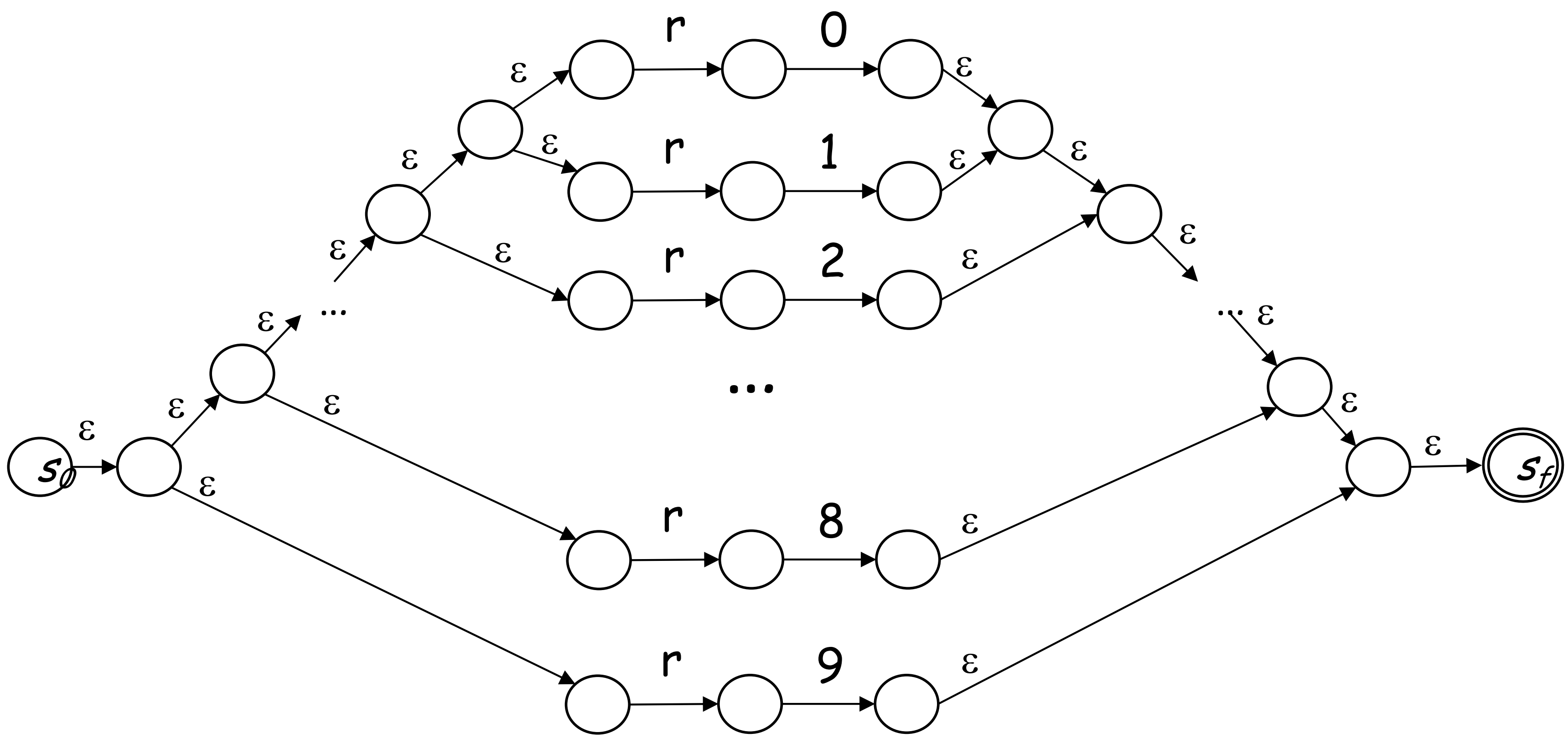
Start with a regular expression

$\underbrace{r0 \mid r1 \mid r2 \mid r3 \mid r4 \mid r5 \mid r6 \mid r7 \mid r8 \mid r9}$

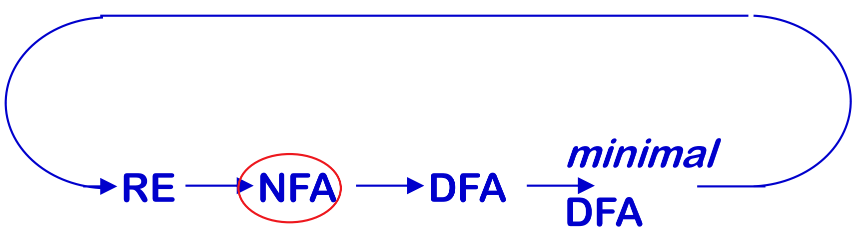
The Cycle of Constructions

Abbreviated Register Specification

Thompson's construction produces

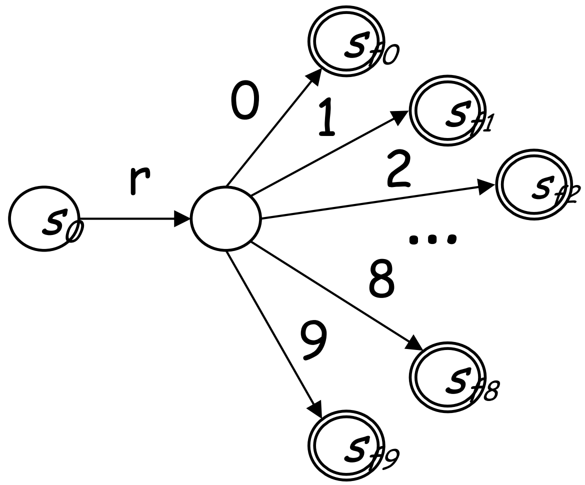


The Cycle of Constructions

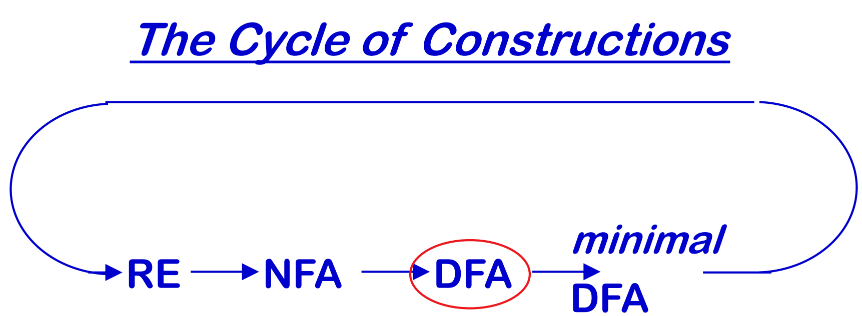


Abbreviated Register Specification

The subset construction builds

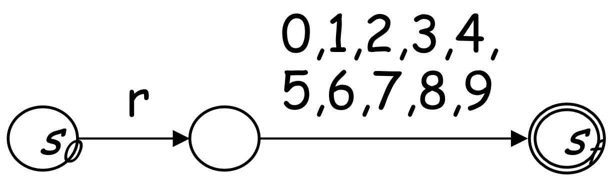


This is a DFA, but it has a lot of states …

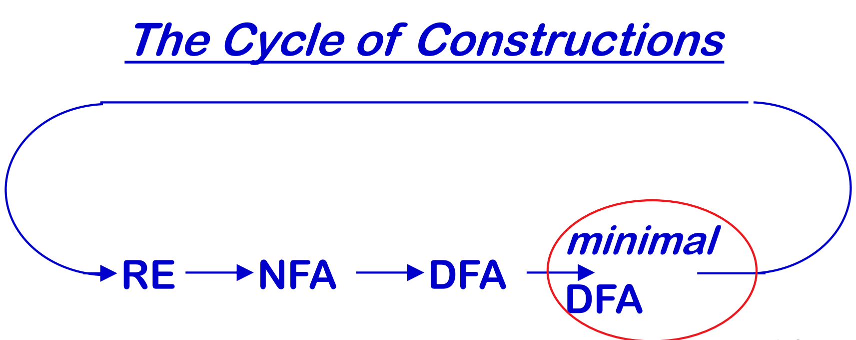


Abbreviated Register Specification

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

$$\begin{array}{lll} \textit{Term} & \rightarrow & [\text{a-zA-Z}]\ ([\text{a-zA-z}]\ |\ [\underline{0}\text{-}\underline{9}])^* \\ \textit{Op} & \rightarrow & \underline{+}\ |\ \underline{-}\ |\ \underline{*}\ |\ \underline{/} \\ \textit{Expr} & \rightarrow & (\ \textit{Term Op}\ )^*\ \textit{Term} \end{array}$$

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

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
- Strings of bit patterns that represent binary numbers which are divisible by 5

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, …

- Limited identifier "length" (Fortran 66 & PL/I)

Next class topic

Parsing
(Syntax Analysis)

EAC Chapters 3.1 - 3.2