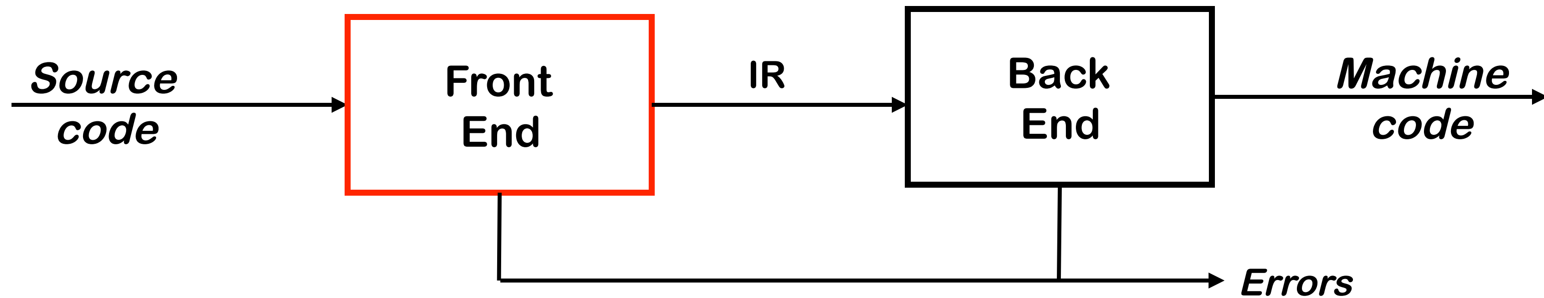
RUTGERS
THE STATE UNIVERSITY OF NEW JERSEY

CS415 Compilers

Lexical Analysis

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

The Front End

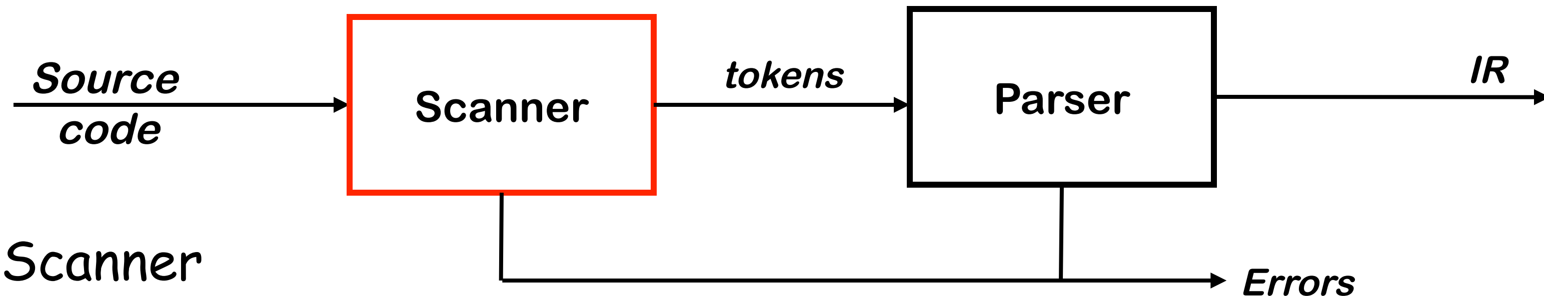


The purpose of the front end is to deal with the input language

- Perform a membership test: code ∈ source language?
- Is the program well-formed (semantically) ?
- Build an IR version of the code for the rest of the compiler

The front end is not monolithic

The Front End



Scanner

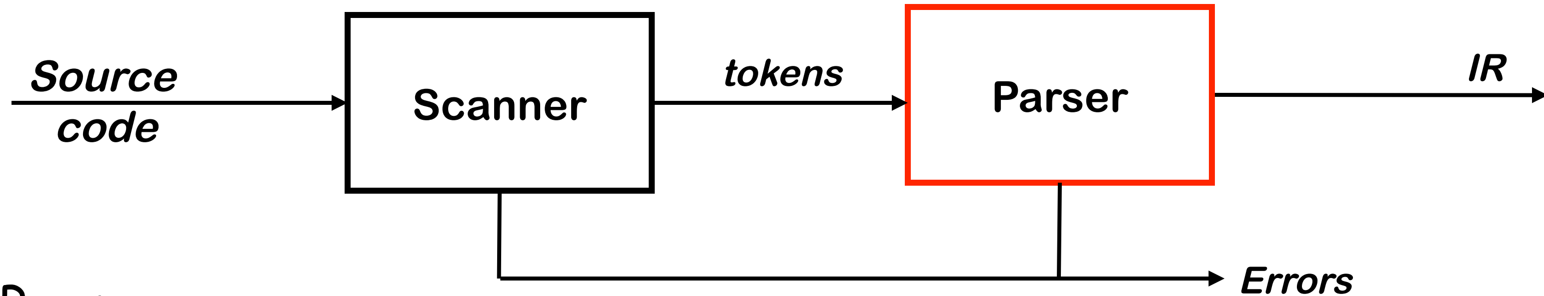
- Maps stream of characters into words
  - → Basic unit of syntax
  - → x = x + y ; *becomes*

    <id,x> <eq,=> <id,x> <pl,+> <id,y> <sc,; >
- Characters that form a word are its *lexeme*
- Its *part of speech* (or *syntactic category*) is called its *token type*
- Scanner discards white space & (often) comments

Speed is an issue in scanning

⇒ use a specialized recognizer

The Front End



Parser

- Checks stream of classified words (*parts of speech*) for grammatical correctness
- Determines if code is syntactically well-formed
- Guides checking at deeper levels than syntax
- Builds an IR representation of the code

We'll get to parsing in the next lectures

The Big Picture

- Language syntax is specified with *parts of speech*, not *words*
- Syntax checking matches *parts of speech* against a grammar

1.	*goal*	→	*expr*
2.	*expr*	→	*expr op term*
3.		|	*term*
4.	*term*	→	number
5.		|	id
6.	*op*	→	+
7.		|	–

S = *goal*

T = { number, id, +, - }

N = { *goal*, *expr*, *term*, *op* }

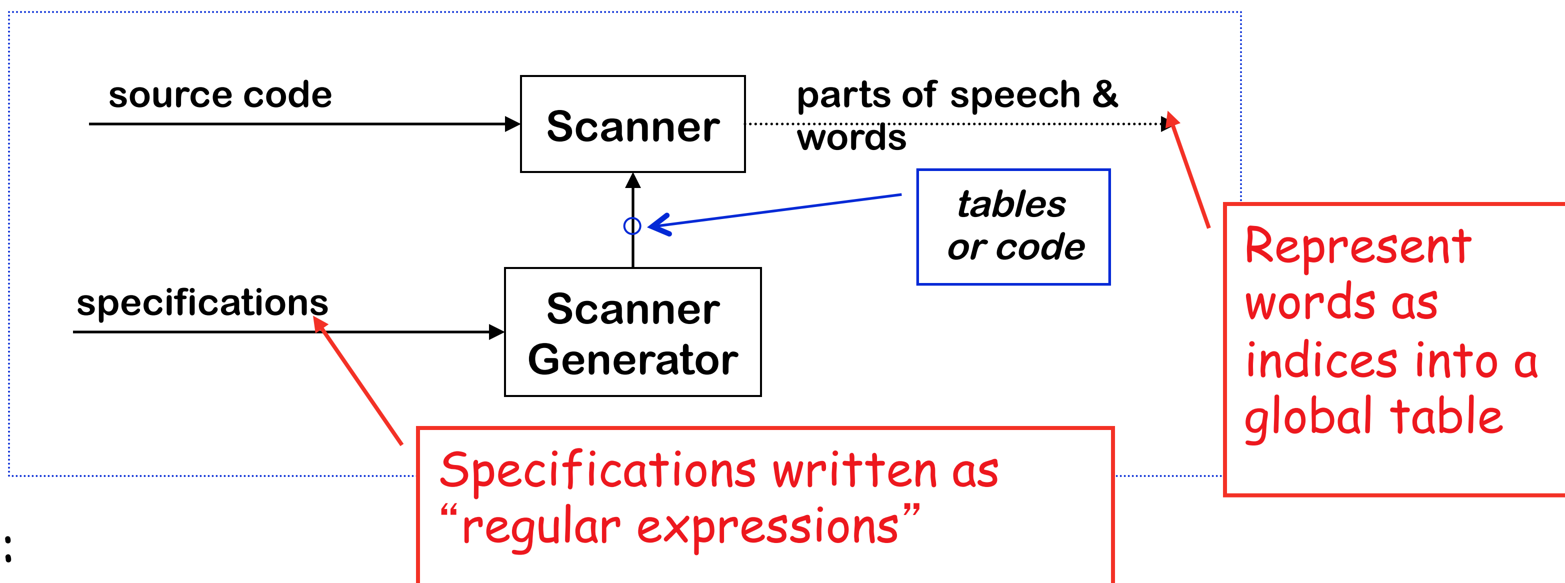
P = { 1, 2, 3, 4, 5, 6, 7}

RUTGERS

The Big Picture

Why study lexical analysis?

- We want to avoid writing scanners by hand



Goals:

→ To simplify specification & implementation of scanners

→ To understand the underlying techniques and technologies

Regular Expressions

Lexical patterns form a *regular language*

*** *any finite language is regular* ***

Regular expressions (REs) describe regular languages

Ever type "rm *.o a.out" ?

Regular Expression (over alphabet Σ)

- ε is a RE denoting the set {ε}
- If a is in Σ, then a is a RE denoting {a}
- If *x* and *y* are REs denoting *L(x)* and *L(y)* then
  - → $x \mid y$ is an RE denoting $L(x) \cup L(y)$
  - → xy is an RE denoting $L(x)L(y)$
  - → x^* is an RE denoting $L(x)^*$

Precedence is *closure*, then *concatenation*, then *alternation*

Set Operations

Operation	Definition
Union of L and M *Written L ∪ M*	$L \cup M = \{s \mid s \in L \text{ or } s \in M\}$
Concatenation of L and M *Written LM*	$LM = \{st \mid s \in L \text{ and } t \in M\}$
Kleene closure of L *Written L^**	$L^* = \bigcup_{0 \le i \le \infty} L^i$
Positive Closure of L *Written L^+*	$L^+ = \bigcup_{1 \le i \le \infty} L^i$

These definitions should be well known

RUTGERS

Examples of Regular Expressions

Identifiers:

Letter → (a|b|c| … |z|A|B|C| … |Z)

Digit → (0|1|2| … |9)

Identifier → *Letter* (*Letter* | *Digit*)*

Numbers:

Integer → (+|-|ε) (0| (1|2|3| … |9)(*Digit* *))

Decimal → *Integer* . *Digit* *

Real → (*Integer* | *Decimal*) E (+|-|ε) *Digit* *

Complex → (*Real* , *Real*)

Numbers can get much more complicated!

More Regular Expressions

- All strings of 1s and 0s ending in a 1

  (0 | 1)* 1

- All strings over lowercase letters where the vowels (a,e,i,o, & u) occur exactly once, in ascending order

  Cons → (b|c|d|f|g|h|j|k|l|m|n|p|q|r|s|t|v|w|x|y|z)
  *Cons** a *Cons** e *Cons** i *Cons** o *Cons** u *Cons**

- All strings of 1s and 0s that do not contain three 1s in a row:

More Regular Expressions

- All strings of 1s and 0s ending in a 1

  $(\underline{0} \mid \underline{1})^* \underline{1}$

- All strings over lowercase letters where the vowels (a,e,i,o, & u) occur exactly once, in ascending order

  Cons → (b|c|d|f|g|h|j|k|l|m|n|p|q|r|s|t|v|w|x|y|z)
  $Cons^* \ \underline{a} \ Cons^* \ \underline{e} \ Cons^* \ \underline{i} \ Cons^* \ \underline{o} \ Cons^* \ \underline{u} \ Cons^*$

- All strings of 1s and 0s that do not contain three 1s in a row:

  $(\underline{0}^* (\varepsilon \mid \underline{10} \mid \underline{110}) \underline{0}^*)^* (\varepsilon \mid \underline{1} \mid \underline{11})$

Regular Expressions (the point)

Regular expressions can be used to specify the words to be translated to parts of speech by a lexical analyzer

Using results from automata theory and theory of algorithms, we can automatically build recognizers from regular expressions

⇒ We study REs and associated theory to automate scanner construction !

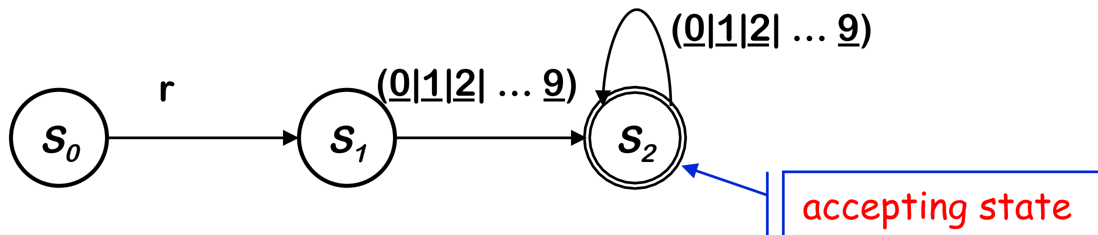
Example

Consider the problem of recognizing ILOC register names

Register → r (0|1|2| … | 9) (0|1|2| … | 9)*

- Allows registers of arbitrary number
- Requires at least one digit

RE corresponds to a recognizer (or DFA)



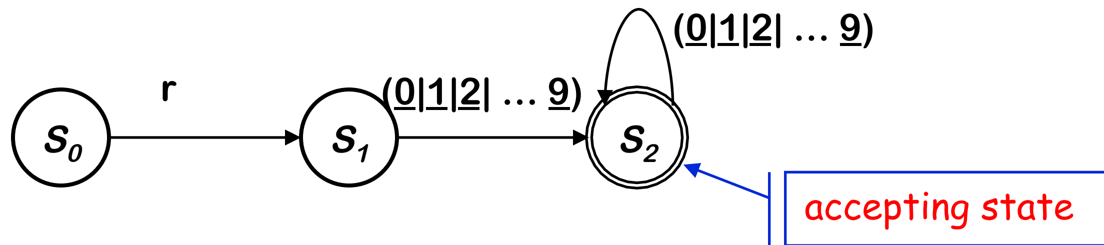
Recognizer for *Register*

Transitions on other inputs go to an error state, s_e

Example (continued)

DFA operation

- Start in state S_0 & take transitions on each input character
- DFA accepts a word x *iff* x leaves it in a final state (S_2)



Recognizer for *Register*

So,

- r17 takes it through s_0, s_1, s_2 and accepts
- r takes it through s_0, s_1 and fails
- a takes it straight to error state s_e (not shown here)

Example (continued)

To be useful, recognizer must turn into code

```
Char ← next character
State ← s0

while (Char ≠ EOF)
    State ← δ(State,Char)
    Char ← next character

if (State is a final state )
    then report success
    else  report failure
```

Skeleton recognizer

δ	r	0,1,2,3,4, 5,6,7,8,9	*All others*
s_0	s_1	s_e	s_e
s_1	s_e	s_2	s_e
s_2	s_e	s_2	s_e
s_e	s_e	s_e	s_e

Table encoding RE

Example (continued)

To be useful, recognizer must turn into code

```
Char ← next character
State ← s0

while (Char ≠ EOF)
    State ← δ(State,Char)
    perform specified action
    Char ← next character

if (State is a final state )
    then report success
    else  report failure
```

Skeleton recognizer

δ	r	0,1,2,3,4, 5,6,7,8,9	*All others*
s_0	s_1 *start*	s_e *error*	s_e *error*
s_1	s_e *error*	s_2 add	s_e *error*
s_2	s_e *error*	s_2 add	s_e *error*
s_e	s_e *error*	s_e *error*	s_e *error*

Table encoding RE

What if we need a tighter specification?

r *Digit Digit** allows arbitrary numbers

- Accepts r00000
- Accepts r99999
- What if we want to limit it to r0 through r31 ?

Write a tighter regular expression

- → *Register* → r ((0|1|2) (*Digit* | ε) | (4|5|6|7|8|9) | (3|30|31))
- → *Register* → r0|r1|r2| … |r31|r00|r01|r02| … |r09

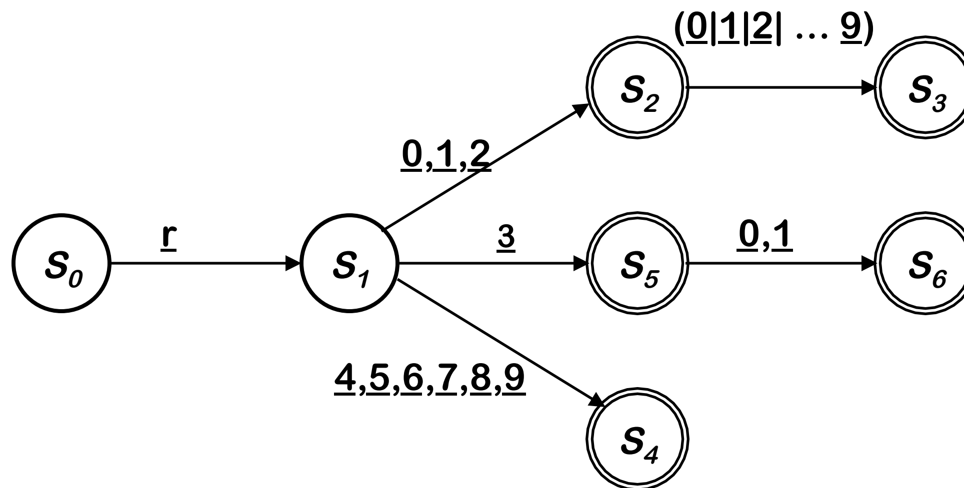
Produces a more complex DFA

- Has more states
- Same cost per transition
- Same basic implementation

Tighter register specification (continued)

The DFA for

Register → r ((0|1|2) (*Digit* | ε) | (4|5|6|7|8|9) | (3|30|31))



- Accepts a more constrained set of registers
- Same set of actions, more states

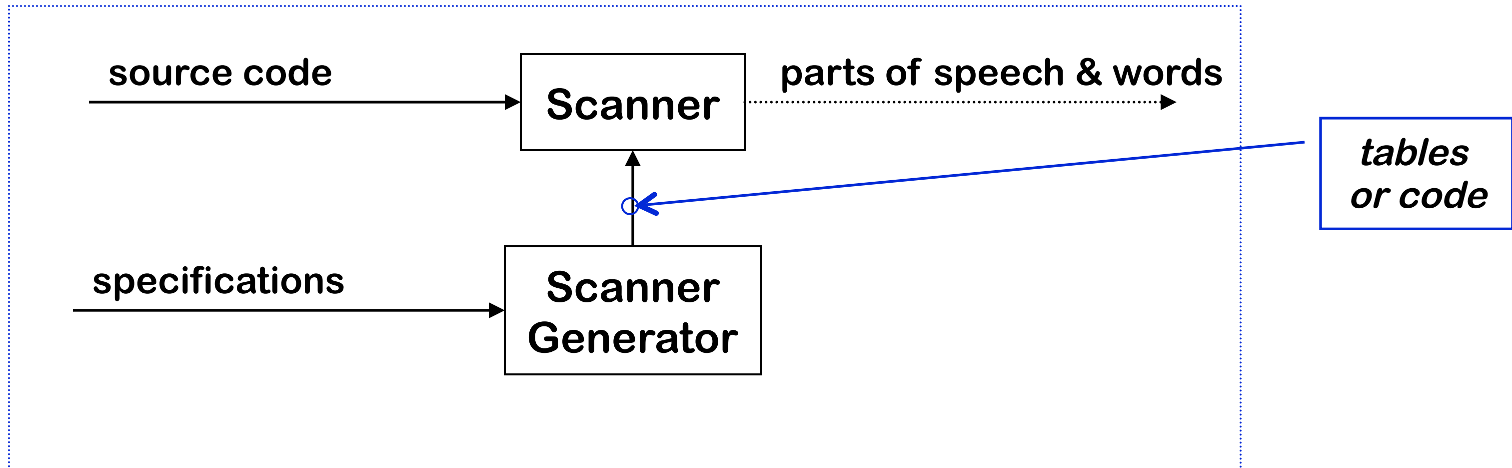
Tighter register specification (continued)

δ	r	0,1	2	3	4-9	All others
s_0	s_1	s_e	s_e	s_e	s_e	s_e
s_1	s_e	s_2	s_2	s_5	s_4	s_e
s_2	s_e	s_3	s_3	s_3	s_3	s_e
s_3	s_e	s_e	s_e	s_e	s_e	s_e
s_4	s_e	s_e	s_e	s_e	s_e	s_e
s_5	s_e	s_6	s_e	s_e	s_e	s_e
s_6	s_e	s_e	s_e	s_e	s_e	s_e
s_e	s_e	s_e	s_e	s_e	s_e	s_e

Runs in the same skeleton recognizer

Table encoding RE for the tighter register specification

Constructing a Scanner - Quick Review



- The scanner is the first stage in the front end
- Specifications can be expressed using regular expressions
- Build tables and code from a DFA

Goal

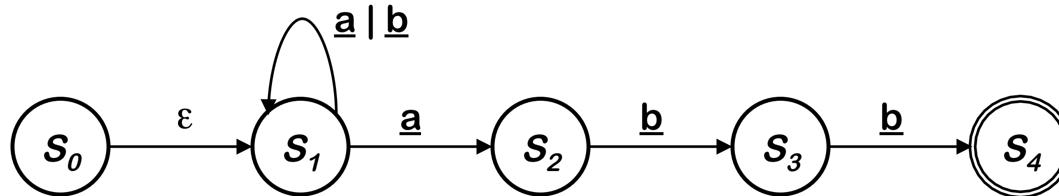
- We will show how to construct a finite state automaton to recognize any RE
- Overview:
  - → Direct construction of a nondeterministic finite automaton (NFA) to recognize a given RE
    - Requires ε-transitions to combine regular subexpressions
  - → Construct a deterministic finite automaton (DFA) 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

Non-deterministic Finite Automata

Each RE corresponds to a non *deterministic finite automaton* (NFA)

- May be hard to directly construct the right DFA

What about an RE such as (a | b)* abb ?



This is a little different

- S_0 has a transition on ε
- S_1 has two transitions on a

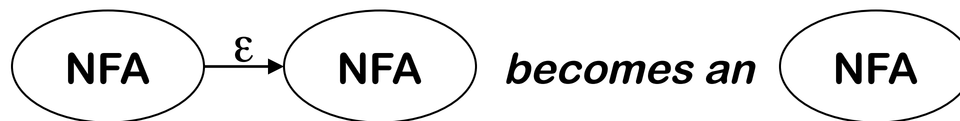
This is a *non-deterministic finite automaton* (NFA)

Non-deterministic Finite Automata

- An NFA accepts a string *x* iff ∃ a path though the transition graph from s_0 to a final state such that the edge labels spell *x*
- Transitions on ε consume no input
- To "run" the NFA, start in s_0 and *guess* the right transition at each step
  - Always guess correctly
  - If some sequence of correct guesses accepts x then accept

Why study NFAs?

- They are the key to automating the RE→DFA construction
- We can paste together NFAs with ε-transitions



Relationship between NFAs and DFAs

DFA is a special case of an NFA

- DFA has no ε transitions
- DFA's transition function is single-valued
- Same rules will work

DFA can be simulated with an NFA

→ *Obviously*

NFA can be simulated with a DFA *(less obvious)*

- Simulate sets of possible states
- Possible exponential blowup in the state space
- Still, one state per character in the input stream

Automating Scanner Construction

To convert a specification into code:

1. Write down the RE for the input language
2. Build a NFA
3. Build the DFA that simulates the NFA
4. Systematically shrink the DFA
5. Turn it into code

Scanner generators

- Lex and Flex work along these lines
- Algorithms are well-known and well-understood
- Key issue is interface to parser *(define all parts of speech)*
- You could build one in a weekend!

RUTGERS

Automating Scanner Construction

RE→ NFA *(Thompson's construction)*

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

NFA → DFA *(subset construction)*

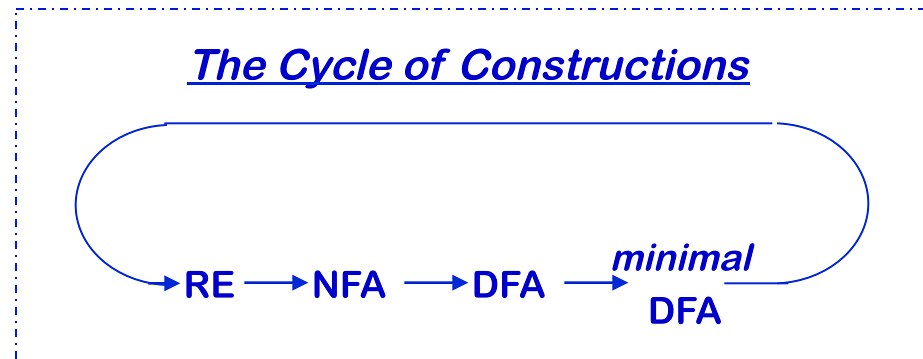
- Build the simulation

DFA → Minimal DFA

- Hopcroft's algorithm

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

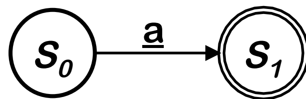
- All pairs, all paths problem
- Take the union of all paths from s_0 to an accepting state



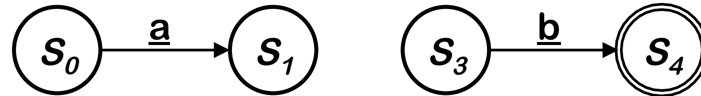
RE →NFA using Thompson's Construction

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



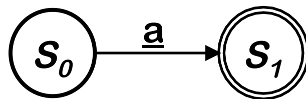
NFA for ab

Ken Thompson, CACM, 1968

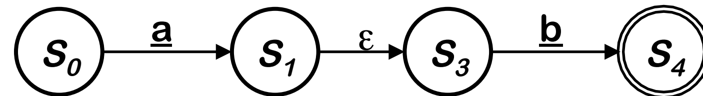
RE →NFA using Thompson's Construction

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



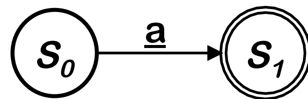
NFA for ab

Ken Thompson, CACM, 1968

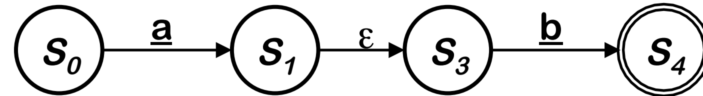
RE →NFA using Thompson's Construction

Key idea

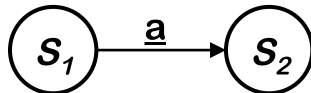
- NFA pattern for each symbol and each operator
- Each NFA has a single start and accept state
- Join them with ε moves in precedence order

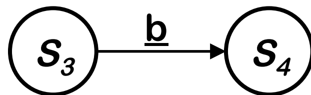


NFA for a



NFA for ab





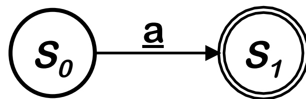
NFA for a | b

Ken Thompson, CACM, 1968

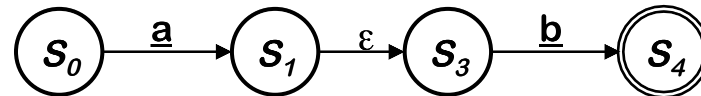
RE →NFA using Thompson's Construction

Key idea

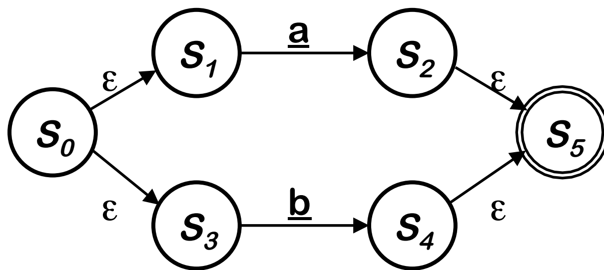
- NFA pattern for each symbol and each operator
- Each NFA has a single start and accept state
- Join them with ε moves in precedence order



NFA for a



NFA for ab



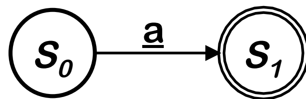
NFA for a | b

Ken Thompson, CACM, 1968

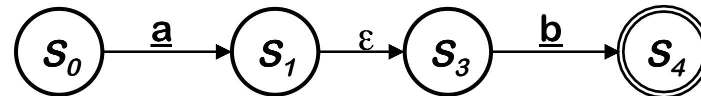
RE →NFA using Thompson's Construction

Key idea

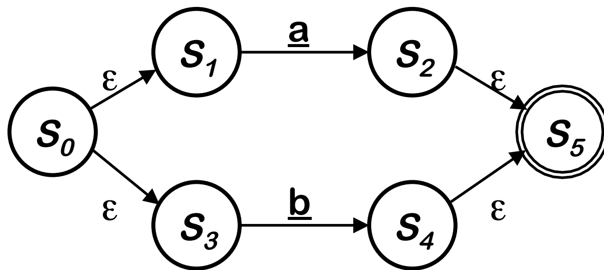
- NFA pattern for each symbol and each operator
- Each NFA has a single start and accept state
- Join them with ε moves in precedence order



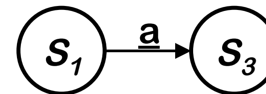
NFA for a



NFA for ab



NFA for a | b

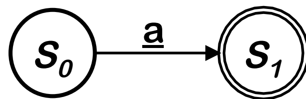


Ken Thompson, CACM, 1968

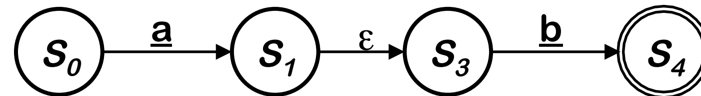
RE →NFA using Thompson's Construction

Key idea

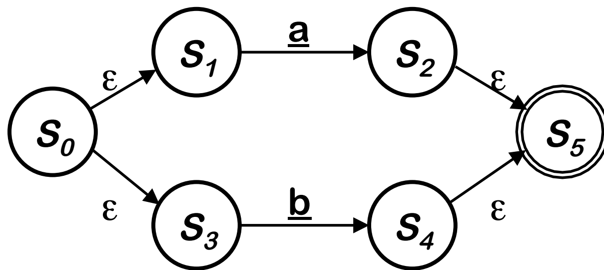
- NFA pattern for each symbol and each operator
- Each NFA has a single start and accept state
- Join them with ε moves in precedence order



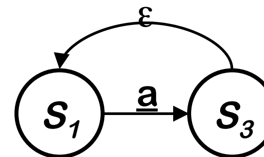
NFA for a



NFA for ab



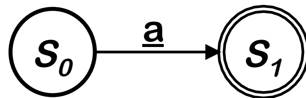
NFA for a | b



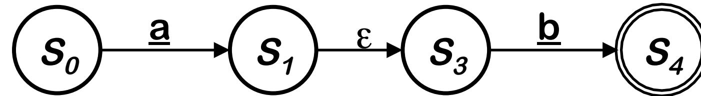
Ken Thompson, CACM, 1968

RE →NFA using Thompson's Construction

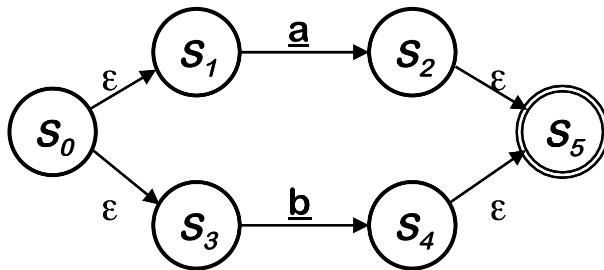
Key idea
- NFA pattern for each symbol and each operator
- Each NFA has a single start and accept state
- Join them with ε moves in precedence order



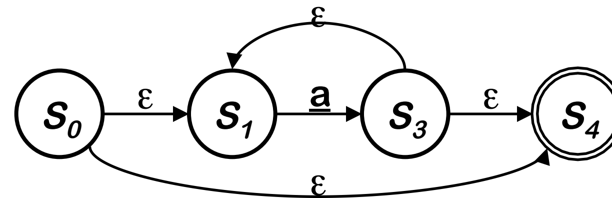
NFA for a



NFA for ab



NFA for a | b



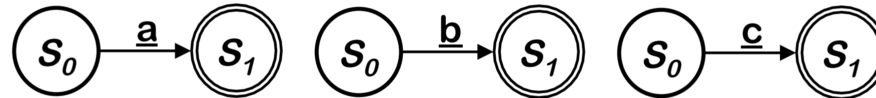
NFA for a^*

Ken Thompson, CACM, 1968

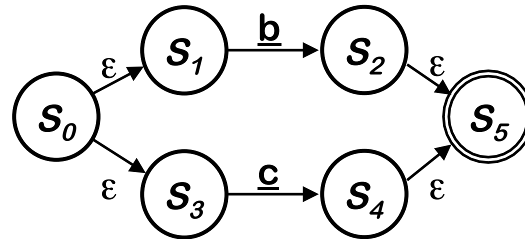
Example of Thompson's Construction

Let's try a (b | c)*

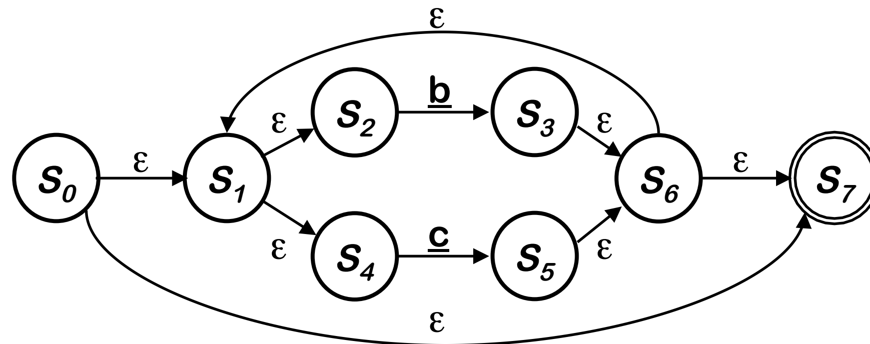
1. a, b, & c



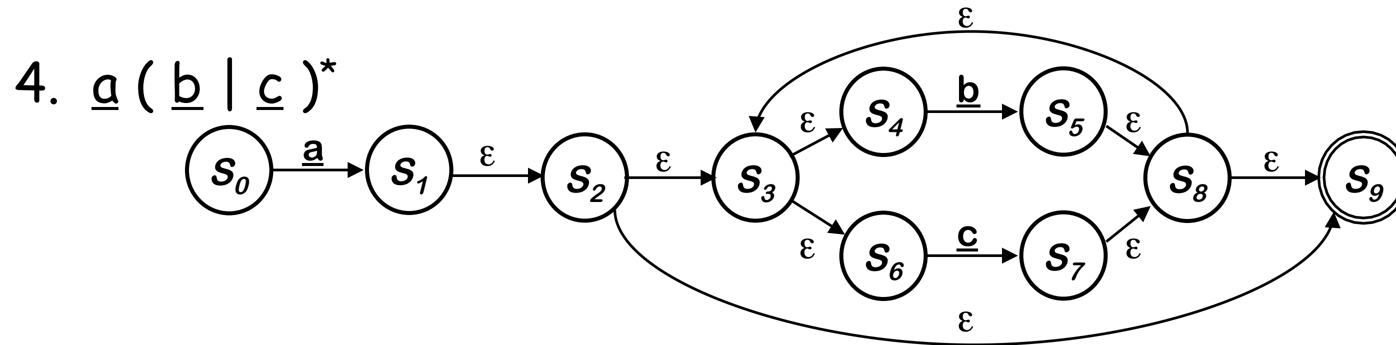
2. b | c



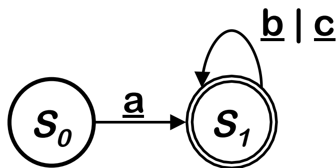
3. (b | c)*



Example of Thompson's Construction (con't



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 = ε-closure(s_0)
- Take the image of S_0, move(S_0, 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:

$s_0 \leftarrow \varepsilon$-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)*

⇒ 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 with Subset Construction

Example of a *fixed-point* computation

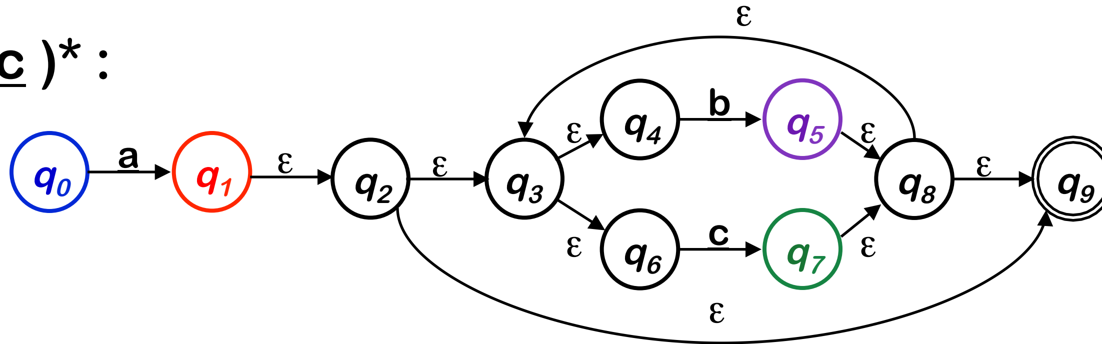
- Monotone construction of some finite set
- Halts when it stops adding to the set
- Proofs of halting & correctness are similar
- These computations arise in many contexts

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

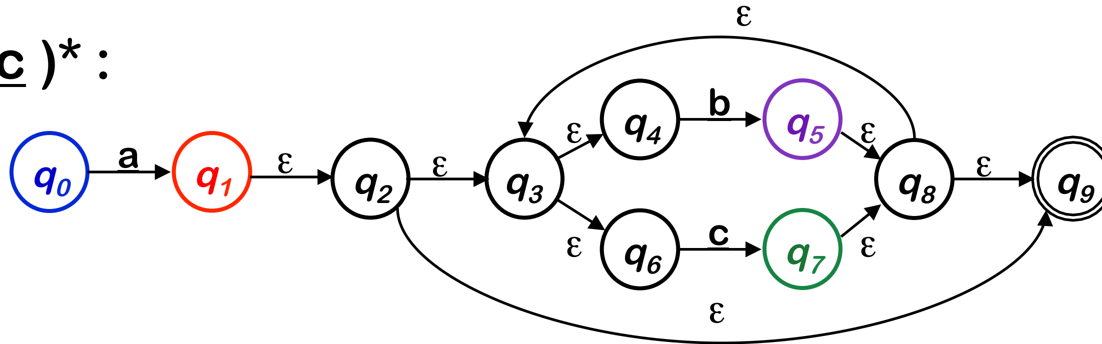
NFA →DFA with Subset Construction



Applying the subset construction:

		ε-closure(move(s,*))		
	NFA states	a	b	c
s_0	q_0			

NFA →DFA with Subset Construction

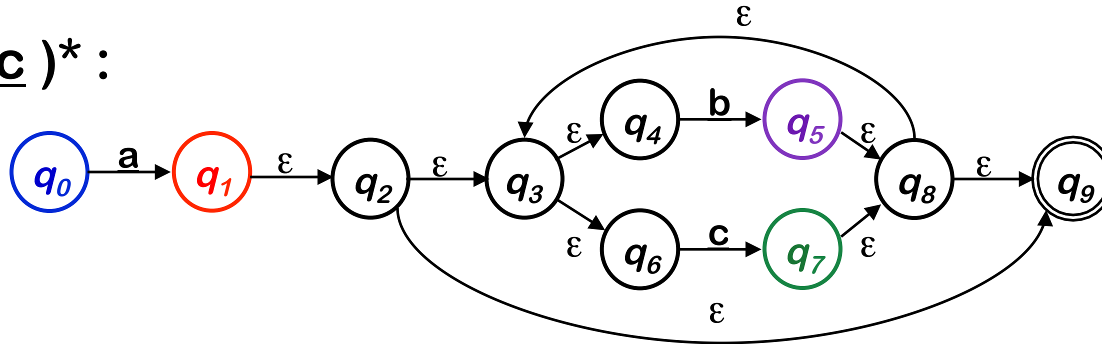


Applying the subset construction:

		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*

NFA →DFA with Subset Construction

a (b | c)* :

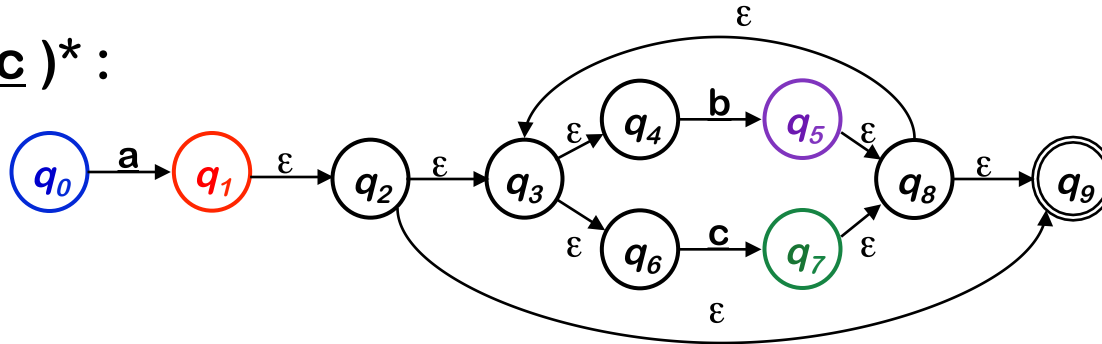


Applying the subset construction:

		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			

NFA →DFA with Subset Construction

a (b | c)* :

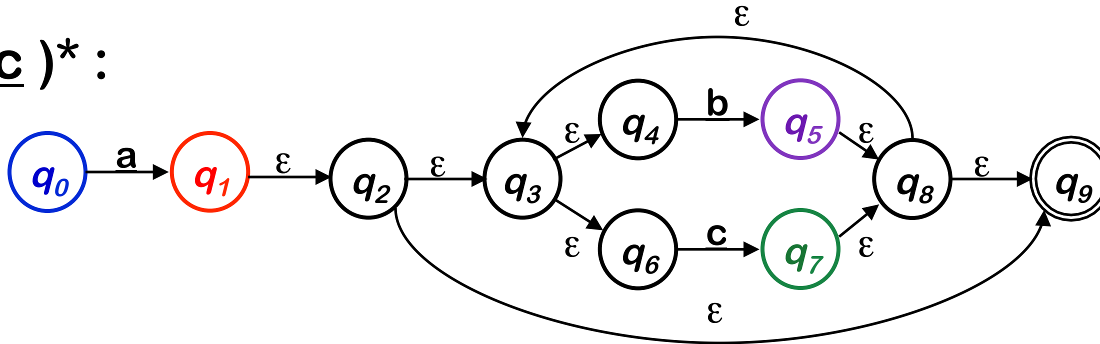


Applying the subset construction:

		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		

NFA →DFA with Subset Construction

a (b | c)* :

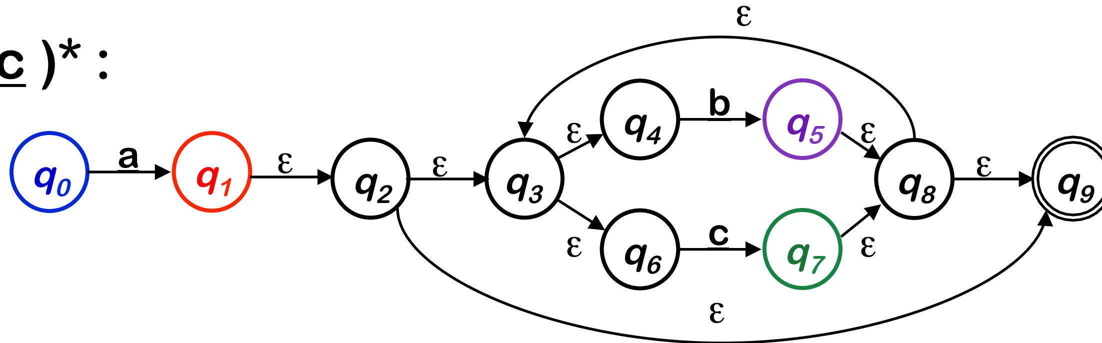


Applying the subset construction:

		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	$q_5, q_8, q_9, q_3, q_4, q_6$	

NFA →DFA with Subset Construction

a (**b** | **c**)* :

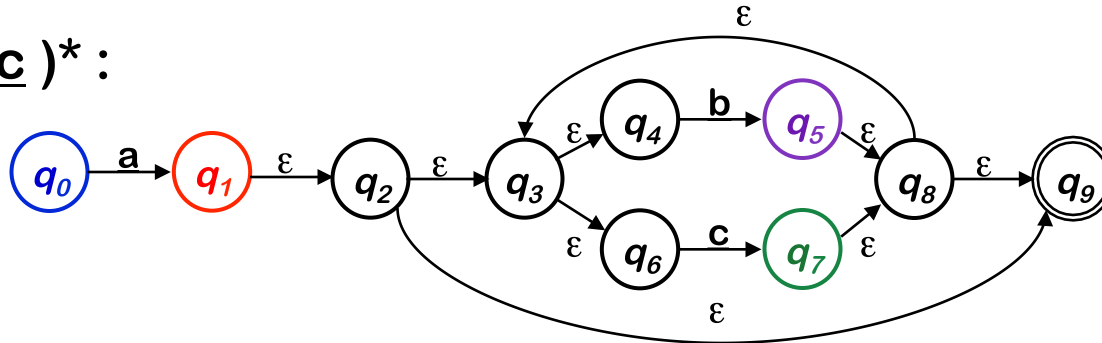


Applying the subset construction:

		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	$q_5, q_8, q_9, q_3, q_4, q_6$	$q_7, q_8, q_9, q_3, q_4, q_6$

NFA →DFA with Subset Construction

a (b | c)* :

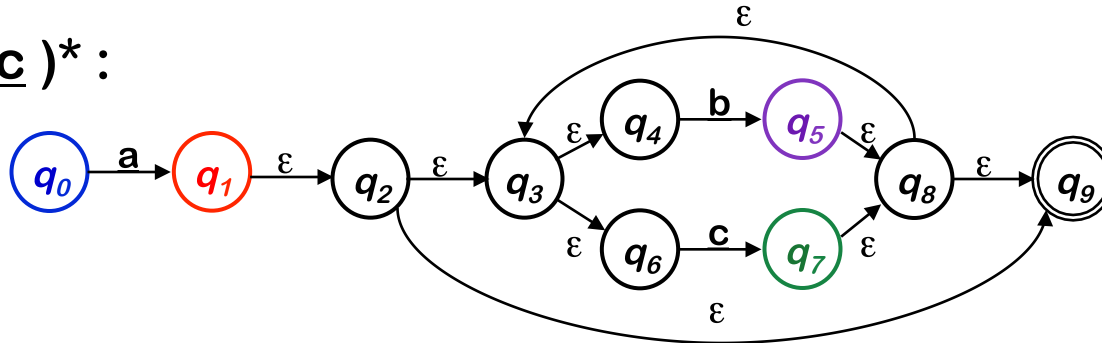


Applying the subset construction:

		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			
s_3	q_7, q_8, q_9, q_3, q_4, q_6			

NFA →DFA with Subset Construction

a (b | c)* :

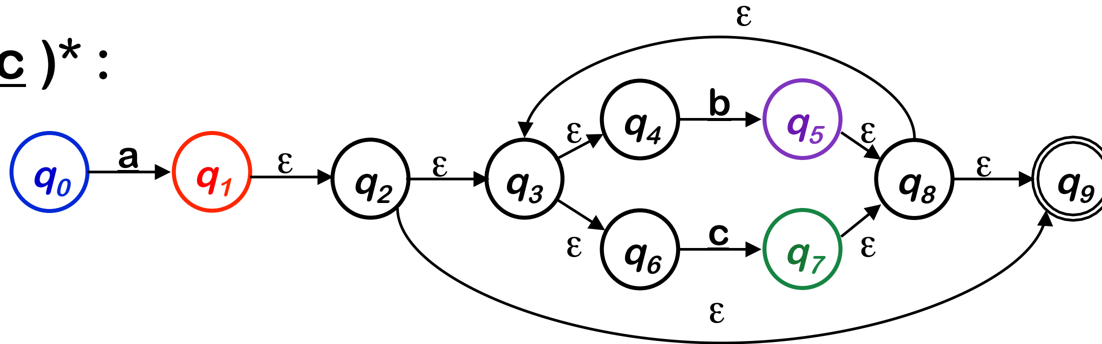


Applying the subset construction:

		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_3	$q_7, q_8, q_9, q_3, q_4, q_6$	none		

NFA →DFA with Subset Construction

a (b | c)* :

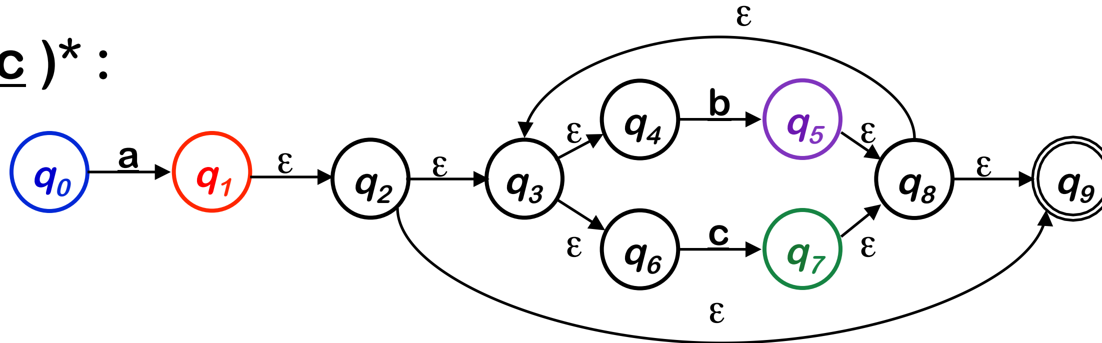


Applying the subset construction:

		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	$q_7, q_8, q_9, q_3, q_4, q_6$	*none*	s_2	

NFA →DFA with Subset Construction

a (b | c)* :

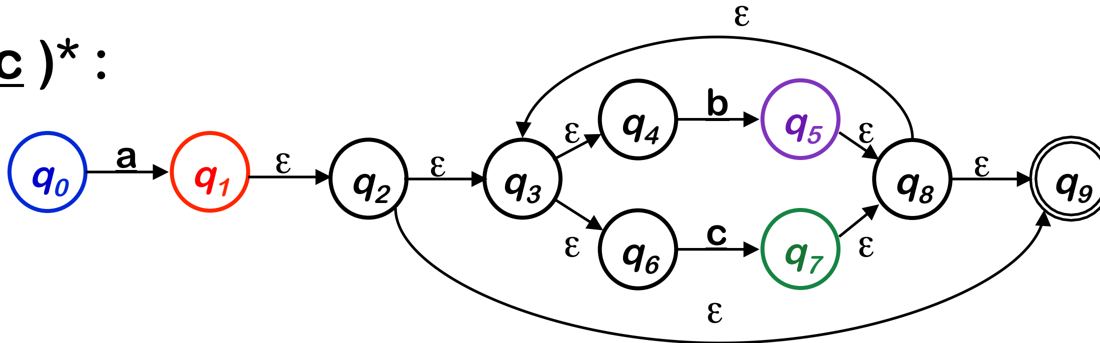


Applying the subset construction:

		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

NFA →DFA with Subset Construction

a (b | c)* :



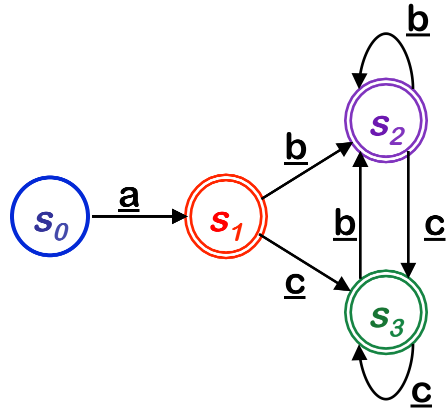
Applying the subset construction:

		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)*

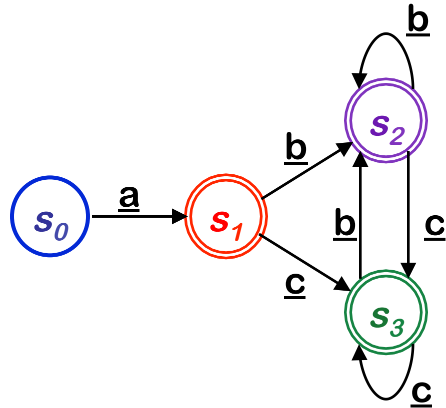


δ	a	b	c
s_0	s_1	-	-
s_1	-	s_2	s_3
s_2	-	s_2	s_3
s_3	-	s_2	s_3

- Ends up smaller than the NFA
- All transitions are deterministic

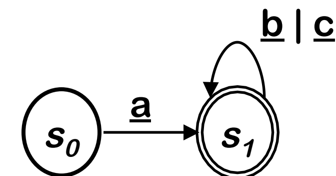
NFA →DFA with Subset Construction

The DFA for a (b | c)*

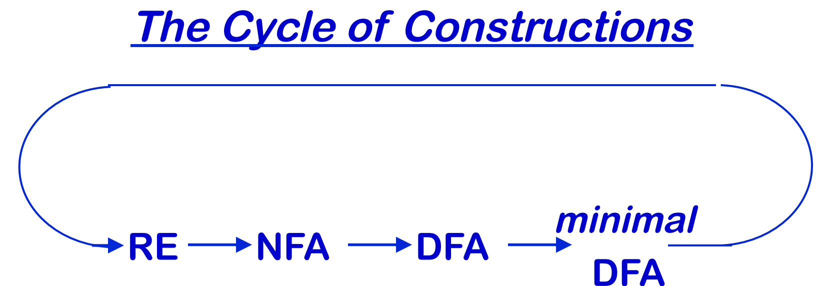


δ	a	b	c
s_0	s_1	-	-
s_1	-	s_2	s_3
s_2	-	s_2	s_3
s_3	-	s_2	s_3

- Ends up smaller than the NFA
- All transitions are deterministic
- Still not the smallest DFA



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

Announcements

- Instruction Scheduling Project
  - → Will be posted this Monday morning
  - → Not a group project

Next class

Syntax Analysis

Read EaC: 3.1 – 3.3