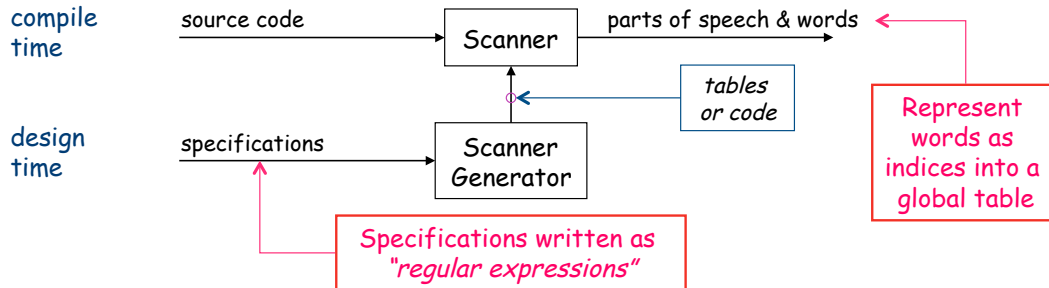


## Quick Review



Last 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

## Quick Review of Regular Expressions



- All strings of 1s and 0s ending in a 1  
 $(\underline{0} | \underline{1})^* \underline{1}$
- All strings over lowercase letters where the vowels (a,e,i,o, & u) occur exactly once, in ascending order  
Let *Cons* be  $(\underline{b} | \underline{c} | \underline{d} | \underline{f} | \underline{g} | \underline{h} | \underline{j} | \underline{k} | \underline{l} | \underline{m} | \underline{n} | \underline{p} | \underline{q} | \underline{r} | \underline{s} | \underline{t} | \underline{v} | \underline{w} | \underline{x} | \underline{y} | \underline{z})$   
 $Cons^* \underline{a} Cons^* \underline{e} Cons^* \underline{i} Cons^* \underline{o} Cons^* \underline{u} Cons^*$
- All strings of 1s and 0s that do not contain three 0s in a row:  
 $(\underline{1}^* (\underline{\epsilon} | \underline{01} | \underline{001}) \underline{1}^*)^* (\underline{\epsilon} | \underline{0} | \underline{00})$

## Example

(from Lab 1)

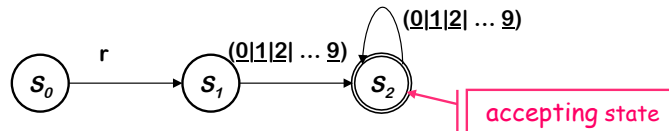


Consider the problem of recognizing ILOC register names

$Register \rightarrow r (0|1|2| \dots | 9) (0|1|2| \dots | 9)^*$

- 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,  $s_e$

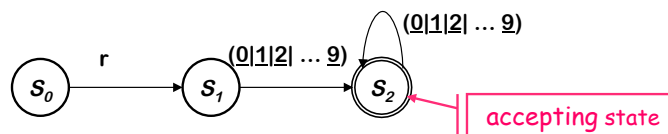
## Example

(continued)



DFA operation

- Start in state  $s_0$  & make 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  $s_e$

## Example

(continued)



To be useful, the recognizer must be converted 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*

$\delta$	r	0,1,2,3,4, 5,6,7,8,9	All others
s <sub>0</sub>	s <sub>1</sub>	s <sub>e</sub>	s <sub>e</sub>
s <sub>1</sub>	s <sub>e</sub>	s <sub>2</sub>	s <sub>e</sub>
s <sub>2</sub>	s <sub>e</sub>	s <sub>2</sub>	s <sub>e</sub>
s <sub>e</sub>	s <sub>e</sub>	s <sub>e</sub>	s <sub>e</sub>

*Table encoding the RE*

## Example

(continued)



We can add "actions" to each transition

```

Char ← next character
State ← s0
while (Char ≠ EOF)
  Next ← δ(State,Char)
  Act ← α(State,Char)
  perform action Act
  State ← Next
  Char ← next character
if (State is a final state)
  then report success
  else report failure
    
```

*Skeleton recognizer*

$\delta$	r	0,1,2,3,4, 5,6,7,8,9	All others
$\alpha$			
s <sub>0</sub>	s <sub>1</sub> <i>start</i>	s <sub>e</sub> <i>error</i>	s <sub>e</sub> <i>error</i>
s <sub>1</sub>	s <sub>e</sub> <i>error</i>	s <sub>2</sub> <i>add</i>	s <sub>e</sub> <i>error</i>
s <sub>2</sub>	s <sub>e</sub> <i>error</i>	s <sub>2</sub> <i>add</i>	s <sub>e</sub> <i>error</i>
s <sub>e</sub>	s <sub>e</sub> <i>error</i>	s <sub>e</sub> <i>error</i>	s <sub>e</sub> <i>error</i>

*Table encoding RE*



## What if we need a tighter specification?

$\underline{r}$  *Digit Digit\** allows arbitrary numbers

- Accepts  $\underline{r00000}$
- Accepts  $\underline{r99999}$
- What if we want to limit it to  $\underline{r0}$  through  $\underline{r31}$  ?

Write a tighter regular expression

- *Register*  $\rightarrow \underline{r} ( \underline{0|1|2} ) ( \textit{Digit} | \epsilon ) | ( \underline{4|5|6|7|8|9} ) | ( \underline{3|30|31} )$
- *Register*  $\rightarrow \underline{r0|r1|r2} | \dots | \underline{r31|r00|r01|r02} | \dots | \underline{r09}$

Produces a more complex DFA

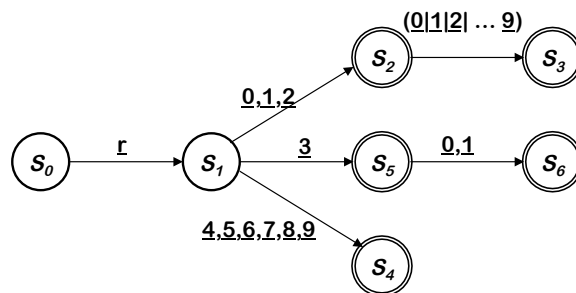
- DFA has more states
- DFA has **same cost** per transition (or per character)
- DFA has same basic implementation



## Tighter register specification (continued)

The DFA for

*Register*  $\rightarrow \underline{r} ( \underline{0|1|2} ) ( \textit{Digit} | \epsilon ) | ( \underline{4|5|6|7|8|9} ) | ( \underline{3|30|31} )$



- Accepts a more constrained set of register names
- Same set of actions, more states

## Tighter register specification (continued)



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

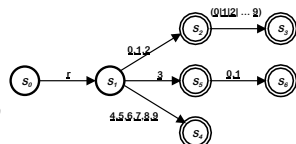
This table runs in the same skeleton recognizer

Table encoding RE for the tighter register specification

## Tighter register specification (continued)



State Action	r	0,1	2	3	4,5,6 7,8,9	other
0	1 <i>start</i>	e	e	e	e	e
1	e	2 <i>add</i>	2 <i>add</i>	5 <i>add</i>	4 <i>add</i>	e
2	e	3 <i>add</i>	3 <i>add</i>	3 <i>add</i>	3 <i>add</i>	e <i>exit</i>
3,4	e	e	e	e	e	e <i>exit</i>
6	e	6 <i>add</i>	e	e	e	e <i>exit</i>
6	e	e	e	e	e	x <i>exit</i>
e	e	e	e	e	e	e



## 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
    - Easy to build in an algorithmic way
    - Requires  $\epsilon$ -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 in the DFA
    - Hopcroft state minimization algorithm
  - Generate the scanner code
    - Additional specifications needed for the actions

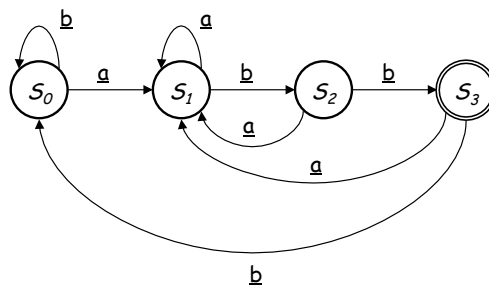
Introduce NFAs

Optional, but worthwhile

## Non-deterministic Finite Automata



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



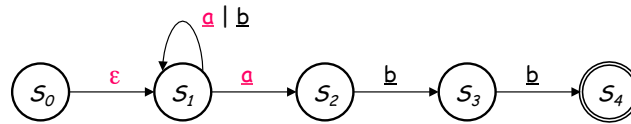
Each RE corresponds to a *deterministic finite automaton* (DFA)

- We know a DFA exists for each RE
- The DFA may be hard to build directly
- Automatic techniques will build it for us ...



## Non-deterministic Finite Automata

Here is a simpler RE for  $(a | b)^* abb$



This recognizer is more intuitive

- Structure seems to follow the RE's structure

This recognizer is not a DFA

- $S_0$  has a transition on  $\epsilon$
- $S_1$  has two transitions on  $a$

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



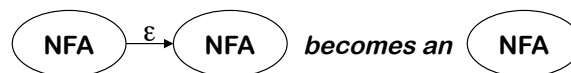
## Non-deterministic Finite Automata

An NFA accepts a string  $x$  iff  $\exists$  a path through the transition graph from  $s_0$  to a final state such that the edge labels spell  $x$ , ignoring  $\epsilon$ 's

- Transitions on  $\epsilon$  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  $\rightarrow$  DFA construction
- We can paste together NFAs with  $\epsilon$ -transitions



## Relationship between NFAs and DFAs

---



DFA is a special case of an NFA

- DFA has no  $\epsilon$  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 big 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!



# Where are we? Why are we doing this?

RE → NFA (Thompson's construction)

- Build an NFA for each term
- Combine them with  $\epsilon$ -moves

NFA → DFA (subset construction)

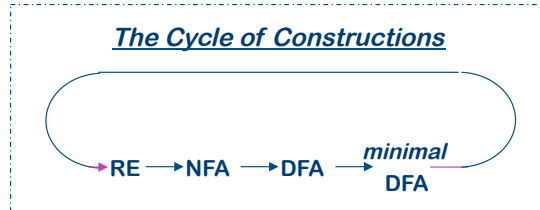
- Build the simulation

DFA → Minimal DFA

- Hopcroft's algorithm

DFA → RE

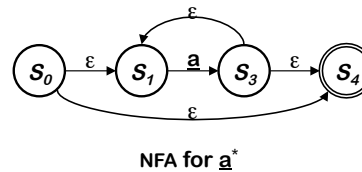
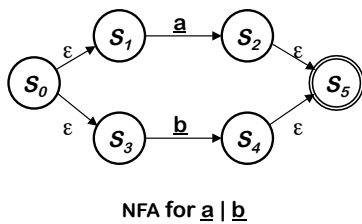
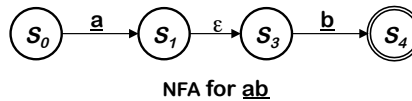
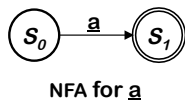
- All pairs, all paths problem
- Union together paths from  $s_0$  to a final state



# RE → NFA using Thompson's Construction

Key idea

- NFA pattern for each symbol & each operator
- Join them with  $\epsilon$  moves in precedence order

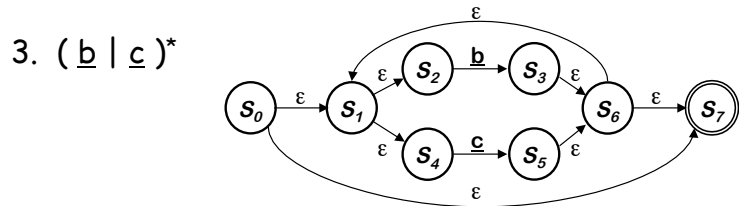
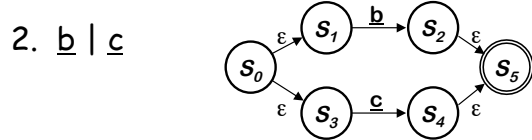


Ken Thompson, CACM, 1968

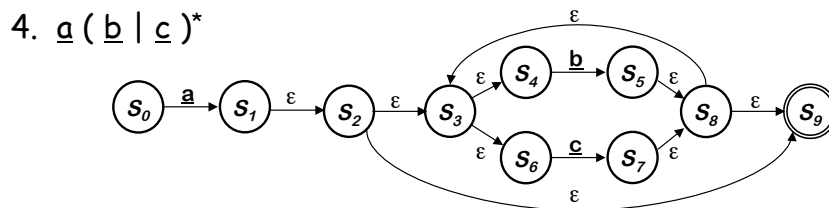


# Example of Thompson's Construction

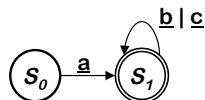
Let's try  $a(b|c)^*$



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



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



But, we can automate production of the more complex NFA version ...