

# Compiling Techniques

## Lecture 4: Automatic Lexer Generation (EaC§2.4)

Christophe Dubach

2 October 2015

# Table of contents

- 1 Finite State Automata for Regular Expression
  - Finite State Automata
  - Non-determinism
- 2 From Regular Expression to Generated Lexer
  - Regular Expression to NFA
  - From NFA to DFA
- 3 Final Remarks

## Definition: finite state automata

A finite state automata is defined by:

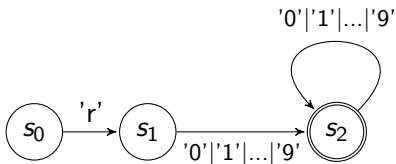
- $S$ , a finite set of states
- $\Sigma$ , an alphabet, or character set used by the recogniser
- $\delta(s, c)$ , a transition function (takes a state and a character and returns new state)
- $s_0$ , the initial or start state
- $S_F$ , a set of final states (a stream of characters is accepted iff the automata ends up in a final state)

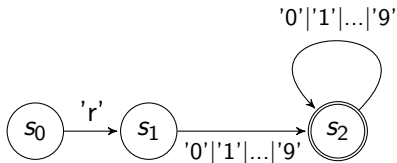
# Finite State Automata for Regular Expression

Example: register names

```
register ::= 'r' ('0'|'1'|...|'9') ('0'|'1'|...|'9')*
```

The RE (Regular Expression) corresponds to a recogniser (or finite state automata):





Finite State Automata (FSA) operation:

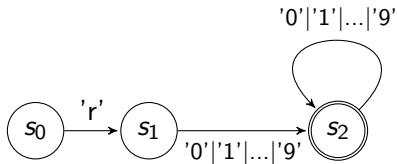
- Start in state  $s_0$  and take transitions on each input character
- The FSA accepts a word  $x$  iff  $x$  leaves it in a final state ( $s_2$ )

Examples:

- **r17** takes it through  $s_0, s_1, s_2$  and accepts
- **r** takes it through  $s_0, s_1$  and fails
- **a** starts in  $s_0$  and leads straight to failure

## Table encoding and skeleton code

To be useful a recogniser must be turned into code



### Table encoding RE

$\delta$	'r'	'0' '1' ... '9'	others
$s_0$	$s_1$	error	error
$s_1$	error	$s_2$	error
$s_2$	error	$s_2$	error

### Skeleton recogniser

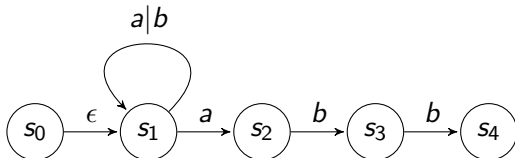
```

c = next character
state = s0
while (c ≠ EOF)
    state =  $\delta(\text{state}, c)$ 
    c = next character
if (state final)
    return success
else
    return error
  
```

## DFA

Each RE corresponds to a Deterministic Finite Automaton (DFA). However, it might be hard to construct directly.

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



This is a little different:

- $s_0$  has a transition on  $\epsilon$ , which can be followed without consuming an input character
- $s_1$  has two transitions on  $a$
- This is a **Non-deterministic Finite Automaton (NFA)**

## Non-deterministic vs deterministic finite automata

Deterministic finite state automata (DFA) :

- All edges leaving the same node have distinct labels
- There is no  $\epsilon$  transition

Non-deterministic finite state automata (NFA) :

- Can have multiple edges leaving from the same node and label with the same character
- Can have a special transition,  $\epsilon$
- This means we might have to backtrack



# Automatic Lexer Generation

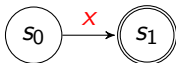
It is possible to systematically generate a lexer for any regular expression.

This can be done in three steps:

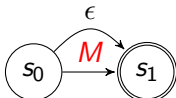
- 1 regular expression (RE)  $\rightarrow$  non-deterministic finite automata (NFA)
- 2 NFA  $\rightarrow$  deterministic finite automata (DFA)
- 3 DFA  $\rightarrow$  generated lexer

# 1st step: RE $\rightarrow$ NFA (Ken Thompson, CACM, 1968)

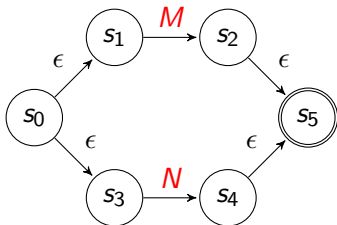
"x"



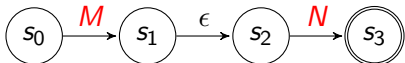
[M]



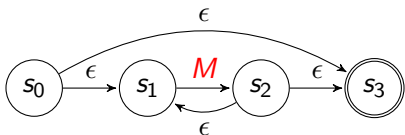
M|N



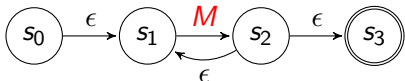
M N



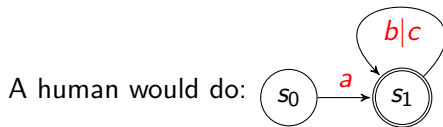
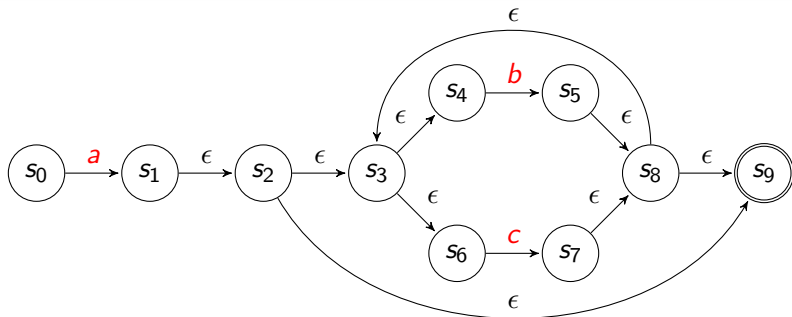
M\*



M+



# Example: $a(b|c)^*$



## Step 2: NFA $\rightarrow$ DFA

Executing a non-deterministic finite automata requires backtracking, which is inefficient. To overcome this, we need to construct a DFA using the NFA.

The main idea:

- We build a DFA which has one state for each set of states the NFA could end up in.
- A set of state is final in the DFA if it contains the final state from the NFA.
- Since the number of states in the NFA is finite ( $n$ ), the number of possible sets is also finite (maximum  $2^n$ ).

Assuming the state of the NFA are labelled  $s_i$  and the states of the DFA we are building are labelled  $q_i$ . We have two key functions:

- $\text{reachable}(s_i, \alpha)$  returns the set of states reachable from  $s_i$  by consuming  $\alpha$  from the input
- $\epsilon$ -closure( $s_i$ ) returns the set of states reachable from  $s_i$  by  $\epsilon$  (e.g., without consuming any input)

## The Subset Construction algorithm (Fixed point iteration)

```
 $q_0 = \epsilon\text{-closure}(s_0)$  ;  $Q = q_0$  ;  $WorkList = q_0$   
while ( $WorkList$  not empty)  
  remove a  $q$  from  $WorkList$   
  for each  $\alpha \in \Sigma$   
     $subset = \epsilon\text{-closure}(reachable(q, \alpha))$   
     $\delta[q, \alpha] = subset$   
    if ( $subset \notin Q$ ) then  
      add  $subset$  to  $Q$  and to  $WorkList$ 
```

## The algorithm (in English)

- Start from start state  $s_0$  of the NFA, compute its  $\epsilon$ -closure
- Build subset from all states reachable from  $q_0$  for an input  $\alpha$
- Add this subset to the transition table/function  $\delta$
- If the subset has not been seen before, add it to the worklist
- Iterate until no new subset are created

## Informal proof of termination

- Q contains no duplicates (test before adding)
- similarly we will never add twice the same subset to the worklist
- bounded number of states; maximum  $2^n$  subsets, where  $n$  is number of state in NFA

⇒ **the loop halts**

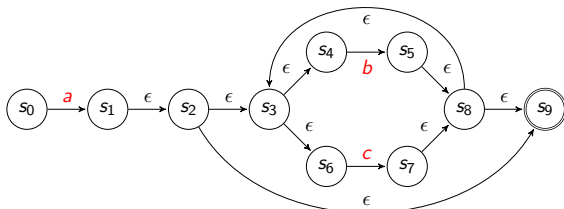
## End result

- S contains all the reachable NFA states
- It tries each symbol in each  $s_i$
- It builds every possible NFA configuration

⇒ **Q and  $\delta$  form the DFA**

# NFA $\rightarrow$ DFA

$a(b|c)^*$



	NFA states	$\epsilon$ -closure(reachable( $q, \alpha$ ))		
		a	b	c
$q_0$	$S_0$	$q_1$	none	none
$q_1$	$S_1, S_2, S_3,$ $S_4, S_6, S_9$	none	$q_2$	$q_3$
$q_2$	$S_5, S_8, S_9,$ $S_3, S_4, S_6$	none	$q_2$	$q_3$
$q_3$	$S_7, S_8, S_9,$ $S_3, S_4, S_6$	none	$q_2$	$q_3$



# Resulting DFA for $a(b|c)^*$

Graph

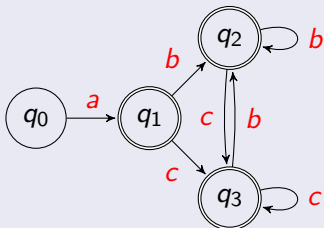


Table encoding

	a	b	c
$q_0$	$q_1$	error	error
$q_1$	error	$q_2$	$q_3$
$q_2$	error	$q_2$	$q_3$
$q_3$	error	$q_2$	$q_3$

- Smaller than the NFA
- All transitions are deterministic (no need to backtrack!)
- Could be even smaller  
 (see EaC§2.4.4 Hopcroft's Algorithm for minimal DFA)
- Can generate the lexer using skeleton recogniser seen earlier

## What can be so hard?

Poor language design can complicate lexing

- Reserved words (keywords) are important: **PL/I**  
if then then = else; else else = then
- Insignificant blanks (whitespaces): **Fortran & Algol68**  
do 10 i = 1,25  $\cong$  do 10 i = 1,25 (loop)  
do 10 i = 1.25  $\cong$  do10i = 1.25 (assignment)
- String constants with special characters: **C,C++,Java**  
newline, tab, quote, comment delimiters, ...

# Building Scanners

The important point:

- All this technology lets us automate scanner construction
- Implementer writes down regular expressions
- Scanner generator builds NFA, DFA and then writes out code
- This reliable process produces fast and robust scanners

For most modern language features, this works:

- As a language designer you should think twice before introducing a feature that defeats a DFA-based scanner
- The ones we have seen (e.g., insignificant blanks, non-reserved keywords) have not proven particularly useful or long lasting

## Next lecture

Parsing:

- Context-Free Grammars
- Dealing with ambiguity
- Recursive descent parser