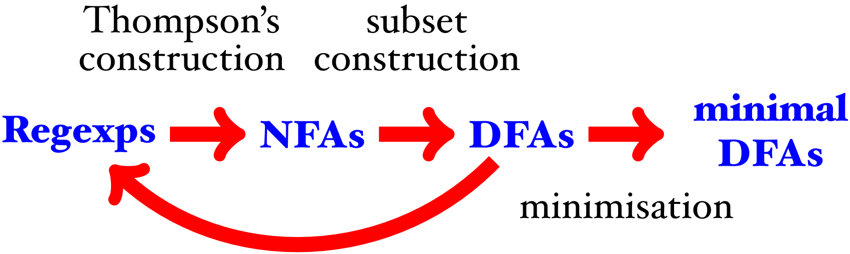
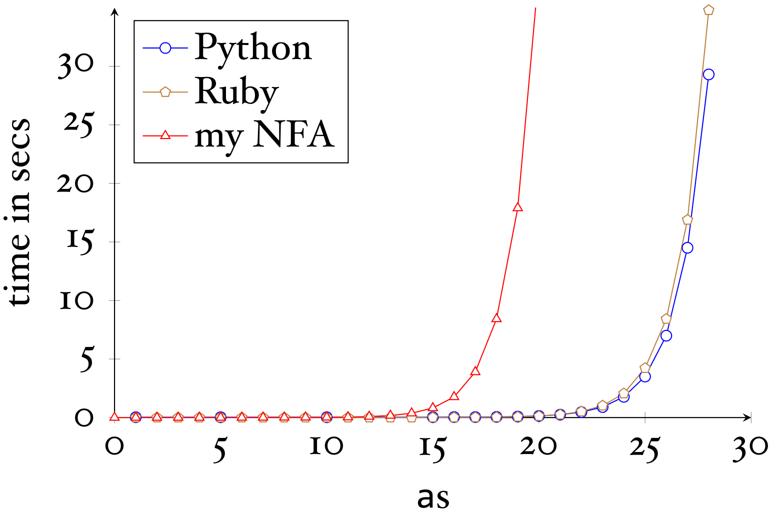
Automata and Formal Languages (4)

Email:	christian.urban at kcl.ac.uk
Office:	S1.27 (1st floor Strand Building)
Slides:	KEATS (also home work is there)

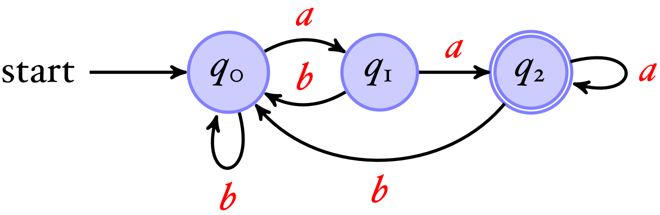
Regexps and Automata



$$(a?\{n\}) \cdot a\{n\}$$

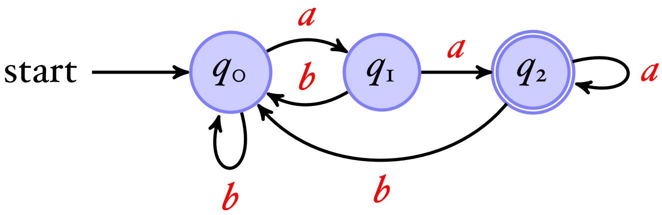


DFA to Rexp



$$q_0 = \epsilon + q_0\,b + q_1\,b + q_2\,b \qquad \text{(start state)}$$
$$q_1 = q_0\,a$$
$$q_2 = q_1\,a + q_2\,a$$

DFA to Rexp



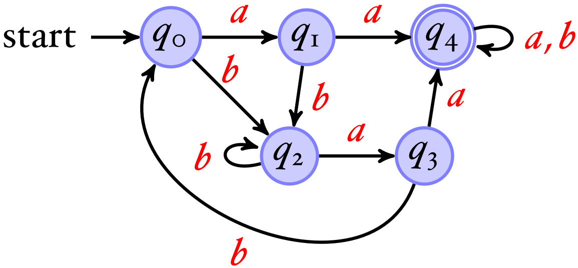
$$q_0 = \epsilon + q_0\,b + q_1\,b + q_2\,b \qquad \text{(start state)}$$
$$q_1 = q_0\,a$$
$$q_2 = q_1\,a + q_2\,a$$

Arden's Lemma:

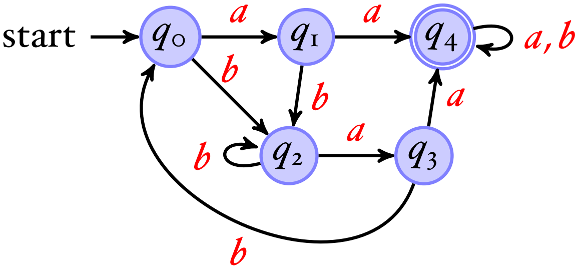
If $q = q\,r + s$ then $q = s\,r^*$

DFA Minimisation

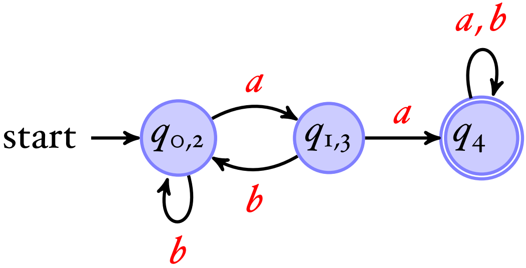
1. Take all pairs (q, p) with $q \neq p$
2. Mark all pairs that accepting and non-accepting states
3. For all unmarked pairs (q, p) and all characters c test whether
   $$(\delta(q, c), \delta(p, c))$$
   are marked. If yes, then also mark (q, p).
4. Repeat last step until no change.
5. All unmarked pairs can be merged.



	q_0	q_1	q_2	q_3
q_1				
q_2				
q_3				
q_4	$\star$	$\star$	$\star$	$\star$

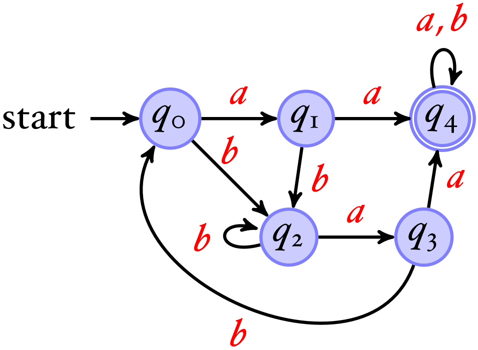


	q_0	q_1	q_2	q_3
q_1	★			
q_2		★		
q_3	★		★	
q_4	★	★	★	★

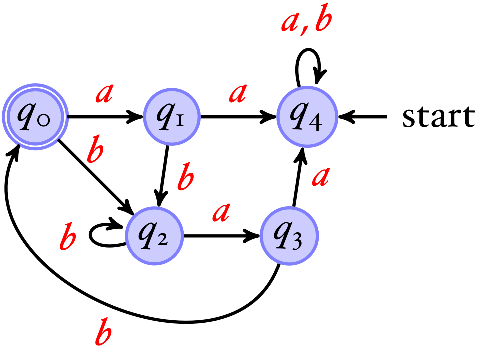


minimal automaton

Alternatives

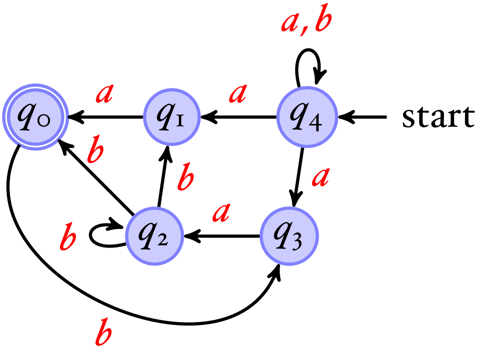


Alternatives



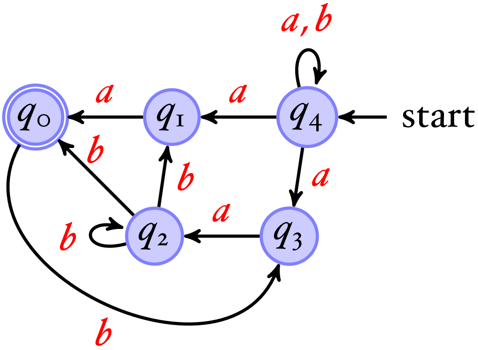
- exchange initial / accepting states

Alternatives



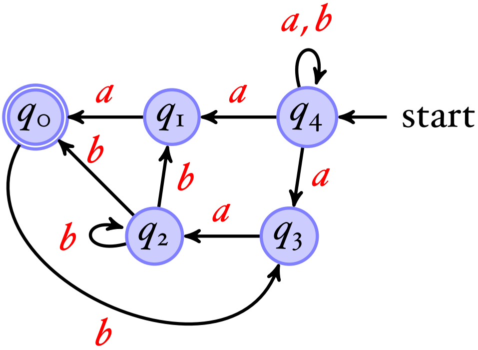
- exchange initial / accepting states
- reverse all edges

Alternatives



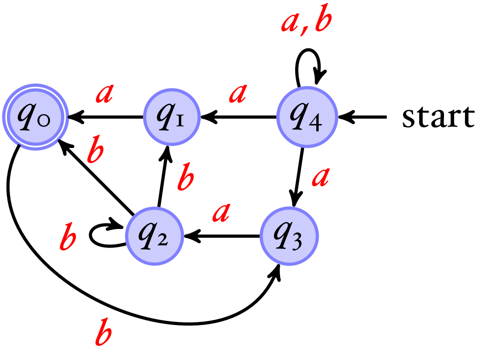
- exchange initial / accepting states
- reverse all edges
- subset construction ⇒ DFA

Alternatives



- exchange initial / accepting states
- reverse all edges
- subset construction $\Rightarrow$ DFA
- repeat once more

Alternatives



- exchange initial / accepting states
- reverse all edges
- subset construction $\Rightarrow$ DFA
- repeat once more $\Rightarrow$ minimal DFA

Regular Languages

Two equivalent definitions:

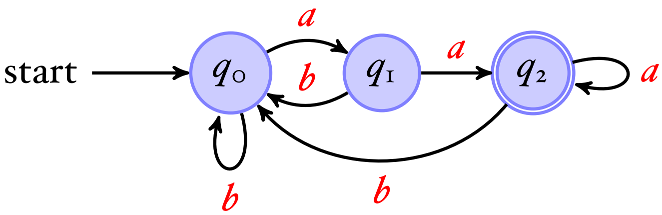
A language is regular iff there exists a regular expression that recognises all its strings.

A language is regular iff there exists an automaton that recognises all its strings.

for example $a^n b^n$ is not regular

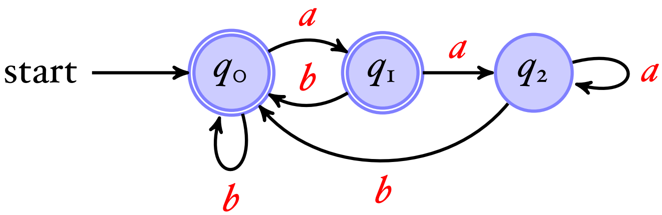
Negation

Regular languages are closed under negation:



Negation

Regular languages are closed under negation:



But requires that the automaton is completed!

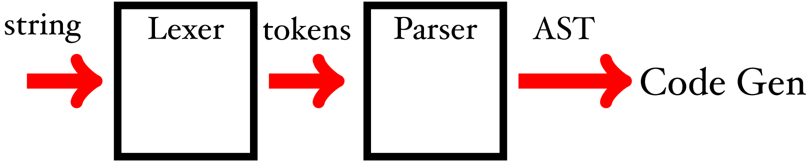
```
/* Fibonacci Program
   input: n */

write "Fib";
read n;
minus1 := 0;
minus2 := 1;
while n > 0 do {
       temp := minus2;
       minus2 := minus1 + minus2;
       minus1 := temp;
       n := n - 1
};
write "Result";
write minus2
```

??

```
write "Input a number ";
read n;
while n > 1 do {
  if n % 2 == 0
  then n := n/2
  else n := 3*n+1;
};
write "Yes";
```

A Compiler



"if true then then 42 else +"

KEYWORD:

if, then, else,

WHITESPACE:

" ", \n,

IDENT:

$LETTER \cdot (LETTER + DIGIT + \_)^*$

NUM:

$(NONZERODIGIT \cdot DIGIT^*) + 0$

OP:

+

COMMENT:

$/* \cdot (ALL^* \cdot \sim(*/) \cdot ALL^*) \cdot */$

”if true then then 42 else +”

```
KEYWORD(if),
WHITESPACE,
IDENT(true),
WHITESPACE,
KEYWORD(then),
WHITESPACE,
KEYWORD(then),
WHITESPACE,
NUM(42),
WHITESPACE,
KEYWORD(else),
WHITESPACE,
OP(+)
```

”if true then then 42 else +”

```
KEYWORD(if),
IDENT(true),
KEYWORD(then),
KEYWORD(then),
NUM(42),
KEYWORD(else),
OP(+)
```

There is one small problem with the tokenizer.
How should we tokenize:

"x-3"

```
ID: …
OP:
  "+", "-"
NUM:
  (NONZERODIGIT · DIGIT*) + "0"
NUMBER:
  NUM + ("-" · NUM)
```

POSIX: Two Rules

- Longest match rule ("maximal munch rule"): The longest initial substring matched by any regular expression is taken as next token.
- Rule priority: For a particular longest initial substring, the first regular expression that can match determines the token.

POSIX: Two Rules

- Longest match rule ("maximal munch rule"): The longest initial substring matched by any regular expression is taken as next token.

- Rule priority: For a particular longest initial substring, the first regular expression that can match determines the token.

most posix matchers are buggy
http://www.haskell.org/haskellwiki/Regex_Posix

Sulzmann Matcher

We want to match the string abc using r_1:

$$r_1 \xrightarrow{der\ a} r_2$$

Sulzmann Matcher

We want to match the string abc using r_1:

$$r_1 \xrightarrow{der\, a} r_2 \xrightarrow{der\, b} r_3$$

Sulzmann Matcher

We want to match the string abc using r_1:

$$r_1 \xrightarrow{der\, a} r_2 \xrightarrow{der\, b} r_3 \xrightarrow{der\, c} r_4$$

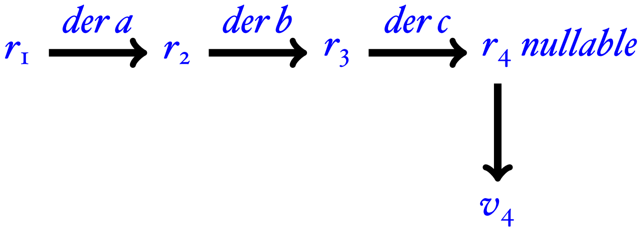
Sulzmann Matcher

We want to match the string *abc* using r_1:

$$r_1 \xrightarrow{der\,a} r_2 \xrightarrow{der\,b} r_3 \xrightarrow{der\,c} r_4\ nullable$$

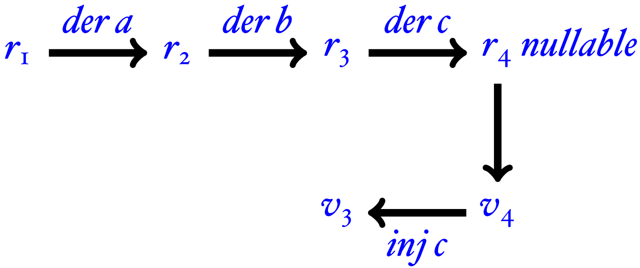
Sulzmann Matcher

We want to match the string *abc* using r_1:



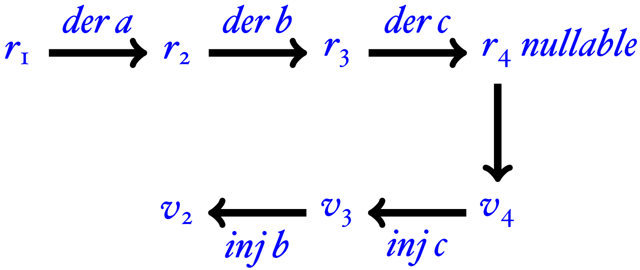
Sulzmann Matcher

We want to match the string *abc* using r_1:



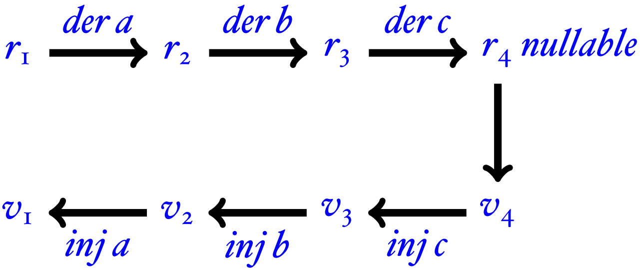
Sulzmann Matcher

We want to match the string *abc* using r_1:



Sulzmann Matcher

We want to match the string *abc* using r_1:



Sulzmann Matcher

We want to match the string *abc* using r_1: