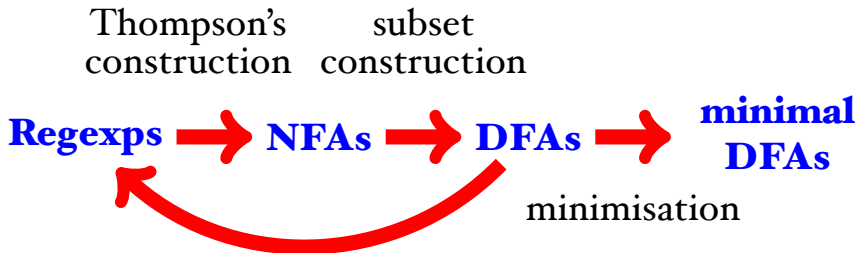


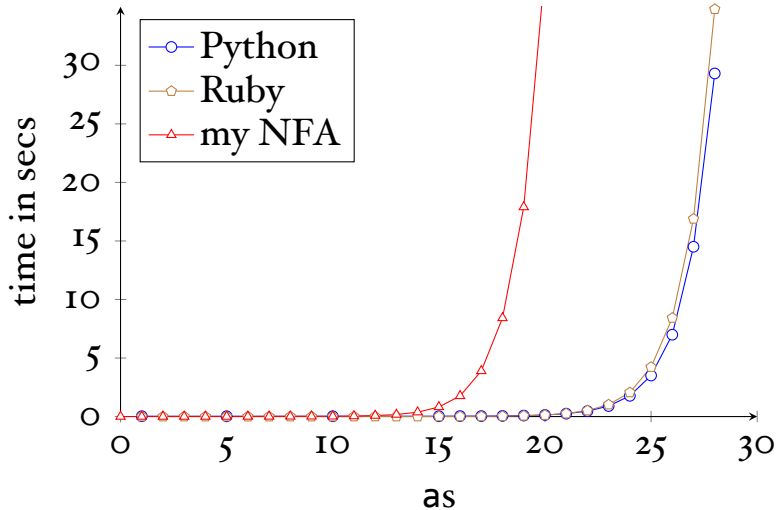
Automata and Formal Languages (4)

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

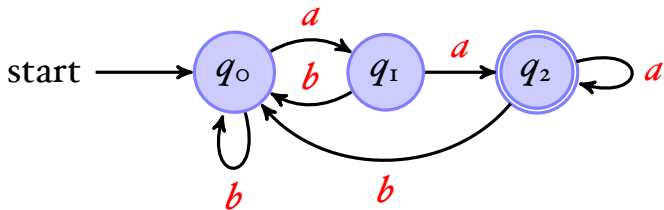
Regexps and Automata



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



DFA to Rexp

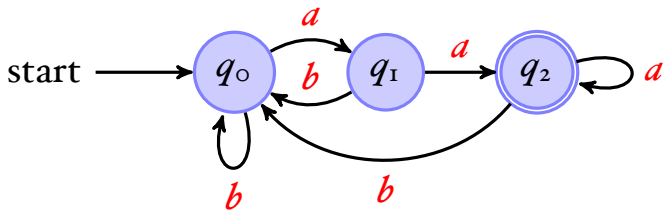


$$q_0 = \epsilon + q_0 b + q_1 b + q_2 b \quad (\text{start state})$$

$$q_1 = q_0 a$$

$$q_2 = q_1 a + q_2 a$$

DFA to Rexp



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

$$q_1 = q_0 a$$

$$q_2 = q_1 a + q_2 a$$

Arden's Lemma:

$$\text{If } q = qr + s \text{ then } q = sr^*$$

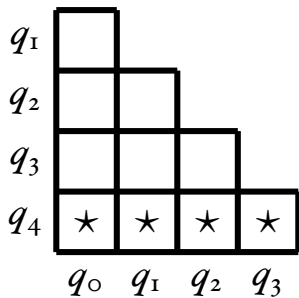
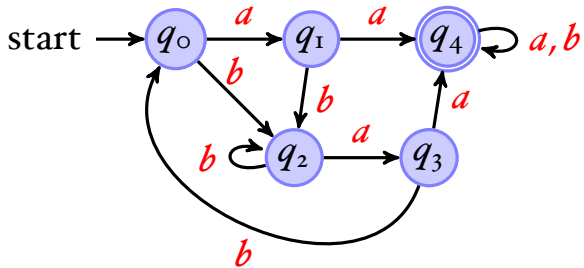
DFA Minimisation

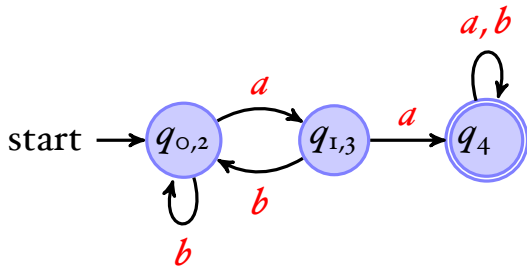
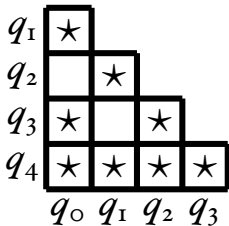
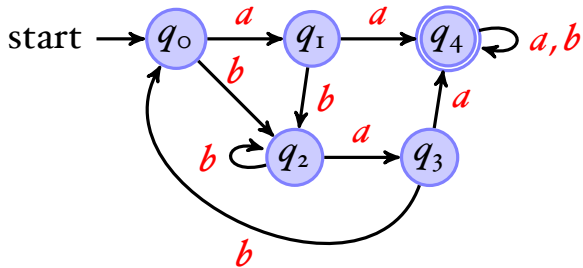
- 1 Take all pairs (q,p) with $q \neq p$
- 2 Mark all pairs that accepting and non-accepting states
- 3 For all unmarked pairs (q,p) and all characters c test whether

$$(\delta(q,c), \delta(p,c))$$

are marked. If yes, then also mark (q,p) .

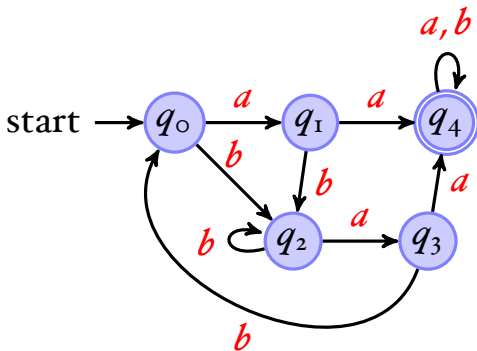
- 4 Repeat last step until no change.
- 5 All unmarked pairs can be merged.



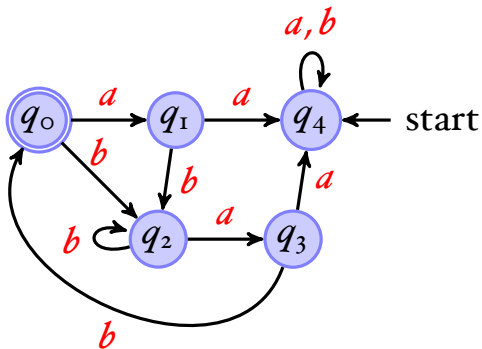


minimal automaton

Alternatives

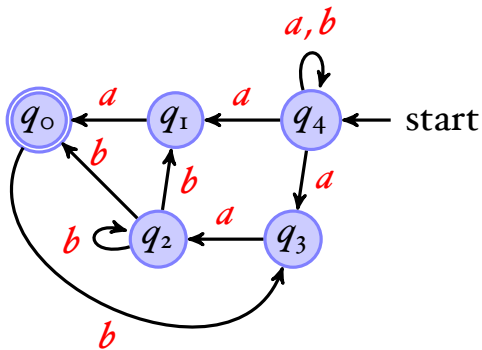


Alternatives



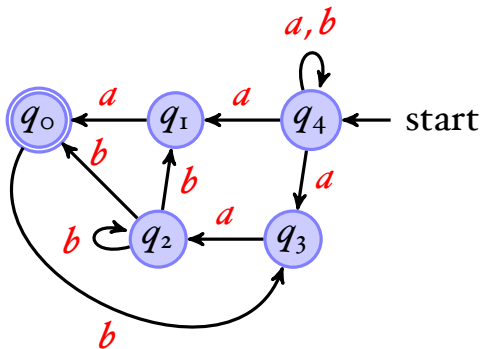
- exchange initial / accepting states

Alternatives



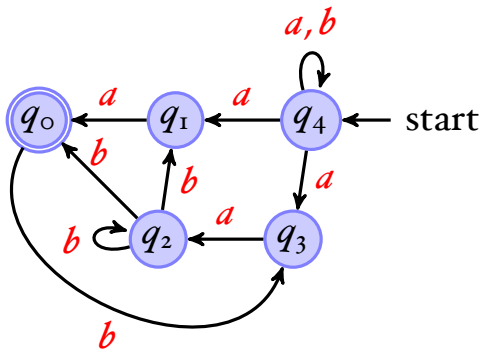
- exchange initial / accepting states
- reverse all edges

Alternatives



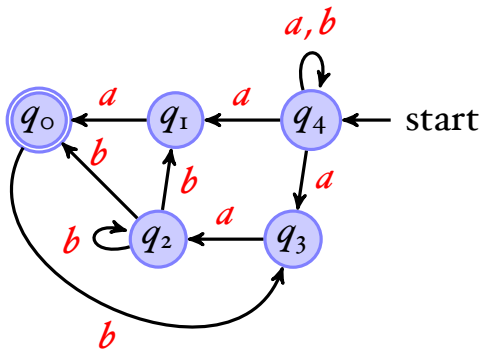
- exchange initial / accepting states
- reverse all edges
- subset construction \Rightarrow DFA

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:

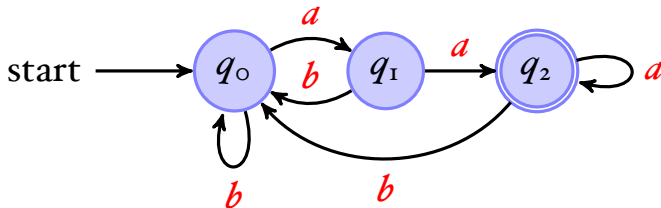
A language is **regular** iff there exists a regular expression that recognises all its strings.

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

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

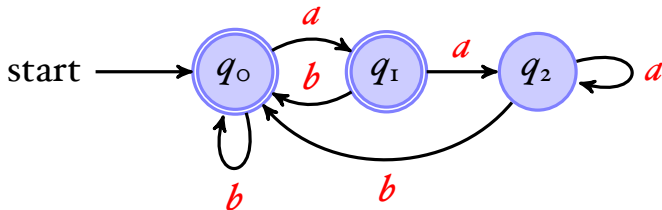
Negation

Regular languages are closed under negation:



Negation

Regular languages are closed under negation:



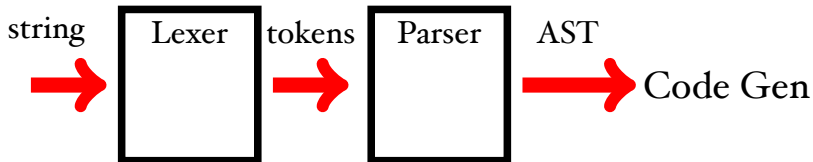
But requires that the automaton is **completed!**

```
1 write "Fib";
2 read n;
3 minus1 := 0;
4 minus2 := 1;
5 while n > 0 do {
6     temp := minus2;
7     minus2 := minus1 + minus2;
8     minus1 := temp;
9     n := n - 1
10 };
11 write "Result";
12 write minus2
```



```
1 write "Input a number ";
2 read n;
3 while n > 1 do {
4     if n % 2 == 0
5     then n := n/2
6     else n := 3*n+1;
7 };
8 write "Yes";
```

A Compiler



”if true then then 42 else +”

KEYWORD:

if, then, else,

WHITESPACE:

” ”, \n,

IDENT:

LETTER · (LETTER + DIGIT + _)*

NUM:

(NONZERODIGIT · DIGIT*) + 0

OP:

+

COMMENT:

/* · ~ (ALL* · (* /) · ALL*) · */

”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 the 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 the 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

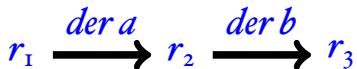
Sulzmann Matcher

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

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

Sulzmann Matcher

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



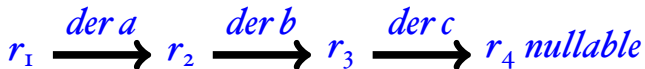
Sulzmann Matcher

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



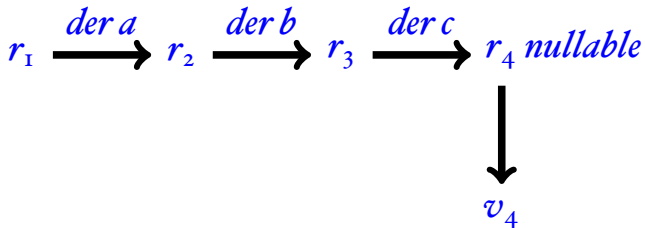
Sulzmann Matcher

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



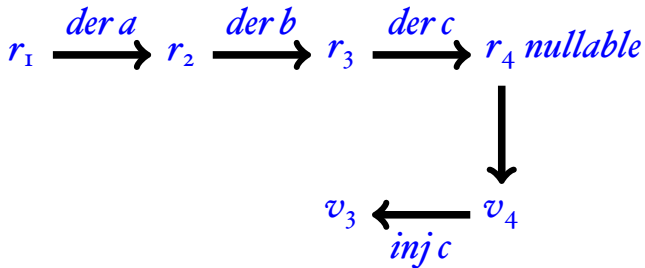
Sulzmann Matcher

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



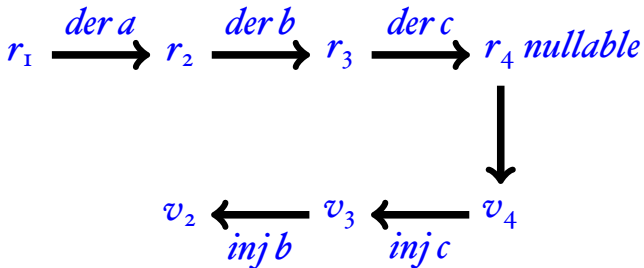
Sulzmann Matcher

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



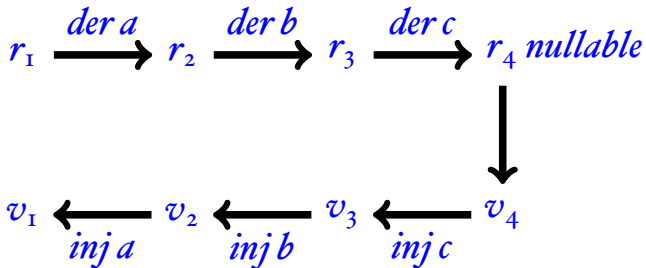
Sulzmann Matcher

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



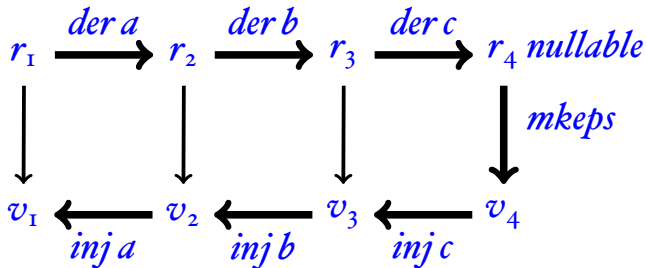
Sulzmann Matcher

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



Sulzmann Matcher

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



Regexes and Values

Regular expressions and their corresponding values:

$r ::= \emptyset$	$v ::=$
ϵ	<i>Empty</i>
c	<i>Char</i> (c)
$r_1 \cdot r_2$	<i>Seq</i> ($v_1 v_2$)
$r_1 + r_2$	<i>Left</i> (v)
r^*	<i>Right</i> (v)
	$[\]$
	$[v_1, \dots, v_n]$

Mkeps

Finding a (posix) value for recognising the empty string

$$\begin{aligned} \mathit{mkeps} \ \epsilon & \stackrel{\text{def}}{=} \ \mathit{Empty} \\ \mathit{mkeps} \ r_1 + r_2 & \stackrel{\text{def}}{=} \ \text{if } \mathit{nullable}(r_1) \\ & \quad \text{then } \mathit{Left}(\mathit{mkeps}(r_1)) \\ & \quad \text{else } \mathit{Right}(\mathit{mkeps}(r_2)) \\ \mathit{mkeps} \ r_1 \cdot r_2 & \stackrel{\text{def}}{=} \ \mathit{Seq}(\mathit{mkeps}(r_1), \mathit{mkeps}(r_2)) \\ \mathit{mkeps} \ r^* & \stackrel{\text{def}}{=} \ \square \end{aligned}$$

Inject

Injecting (“Adding”) a character to a value

$inj(c) c Empty$	$\stackrel{\text{def}}{=} Char c$
$inj(r_I + r_2) c Left(v)$	$\stackrel{\text{def}}{=} Left(inj r_I c v)$
$inj(r_I + r_2) c Right(v)$	$\stackrel{\text{def}}{=} Right(inj r_2 c v)$
$inj(r_I \cdot r_2) c Seq(v_I, v_2)$	$\stackrel{\text{def}}{=} Seq(inj r_I c v_I, v_2)$
$inj(r_I \cdot r_2) c Left(Seq(v_I, v_2))$	$\stackrel{\text{def}}{=} Seq(inj r_I c v_I, v_2)$
$inj(r_I \cdot r_2) c Right(v)$	$\stackrel{\text{def}}{=} Seq(mkeps(r_I), inj r_2 c v)$
$inj(r^*) c Sequ(v, vs)$	$\stackrel{\text{def}}{=} inj r c v :: vs$

inj: 1st arg \mapsto a rexp; 2nd arg \mapsto a character; 3rd arg \mapsto a value

Lexing

$lex\ r\ [] \stackrel{\text{def}}{=} \text{if } nullable(r) \text{ then } mkeps(r) \text{ else } error$
 $lex\ r\ c :: s \stackrel{\text{def}}{=} inj\ r\ c\ lex\ (der\ (c, r), s)$

lex: returns a value

Records

- new regex: $(x : r)$ new value: $Rec(x, v)$

Records

- new regex: $(x : r)$ new value: $Rec(x, v)$
- $nullable(x : r) \stackrel{\text{def}}{=} nullable(r)$
- $der\ c(x : r) \stackrel{\text{def}}{=} (x : der\ c\ r)$
- $mkeps(x : r) \stackrel{\text{def}}{=} Rec(x, mkeps(r))$
- $inj(x : r)\ c\ v \stackrel{\text{def}}{=} Rec(x, inj\ r\ c\ v)$

Records

- new regex: $(x : r)$ new value: $Rec(x, v)$
- $nullable(x : r) \stackrel{\text{def}}{=} nullable(r)$
- $der\ c(x : r) \stackrel{\text{def}}{=} (x : der\ c\ r)$
- $mkeps(x : r) \stackrel{\text{def}}{=} Rec(x, mkeps(r))$
- $inj(x : r)\ c\ v \stackrel{\text{def}}{=} Rec(x, inj\ r\ c\ v)$

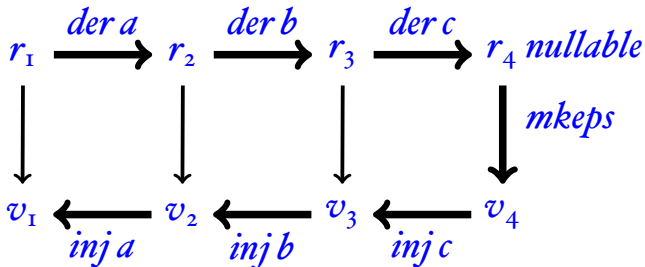
for extracting subpatterns $(z : ((x : ab) + (y : ba)))$

While Tokens

WHILE_REGS $\stackrel{\text{def}}{=}$ ((**"k"** : KEYWORD) +
(**"i"** : ID) +
(**"o"** : OP) +
(**"n"** : NUM) +
(**"s"** : SEMI) +
(**"p"** : (LPAREN + RPAREN)) +
(**"b"** : (BEGIN + END)) +
(**"w"** : WHITESPACE))*

Simplification

- If we simplify after the derivative, then we are building the value for the simplified regular expression, but *not* for the original regular expression.



Rectification

rectification
functions:

$$r \cdot \emptyset \mapsto \emptyset$$

$$\emptyset \cdot r \mapsto \emptyset$$

$$r \cdot \epsilon \mapsto r \quad \lambda f_1 f_2 v. \text{Seq}(f_1 v, f_2 \text{Empty})$$

$$\epsilon \cdot r \mapsto r \quad \lambda f_1 f_2 v. \text{Seq}(f_1 \text{Empty}, f_2 v)$$

$$r + \emptyset \mapsto r \quad \lambda f_1 f_2 v. \text{Left}(f_1 v)$$

$$\emptyset + r \mapsto r \quad \lambda f_1 f_2 v. \text{Right}(f_2 v)$$

$$r + r \mapsto r \quad \lambda f_1 f_2 v. \text{Left}(f_1 v)$$

Rectification

rectification
functions:

$$r \cdot \emptyset \mapsto \emptyset$$

$$\emptyset \cdot r \mapsto \emptyset$$

$$r \cdot \epsilon \mapsto r \quad \lambda f_1 f_2 v. \text{Seq}(f_1 v, f_2 \text{Empty})$$

$$\epsilon \cdot r \mapsto r \quad \lambda f_1 f_2 v. \text{Seq}(f_1 \text{Empty}, f_2 v)$$

$$r + \emptyset \mapsto r \quad \lambda f_1 f_2 v. \text{Left}(f_1 v)$$

$$\emptyset + r \mapsto r \quad \lambda f_1 f_2 v. \text{Right}(f_2 v)$$

$$r + r \mapsto r \quad \lambda f_1 f_2 v. \text{Left}(f_1 v)$$

old *simp* returns a rexp;

new *simp* returns a rexp and a rectification fun.

Lexing with Simplification

$\text{lex } r \ [] \stackrel{\text{def}}{=} \text{if } \text{nullable}(r) \text{ then } \text{mkeps}(r) \text{ else } \text{error}$

$\text{lex } r \ c \ :: \ s \stackrel{\text{def}}{=} \text{let } (r', \text{frect}) = \text{simp}(\text{der}(c, r))$
 $\text{inj } r \ c \ (\text{frect}(\text{lex}(r', s)))$

