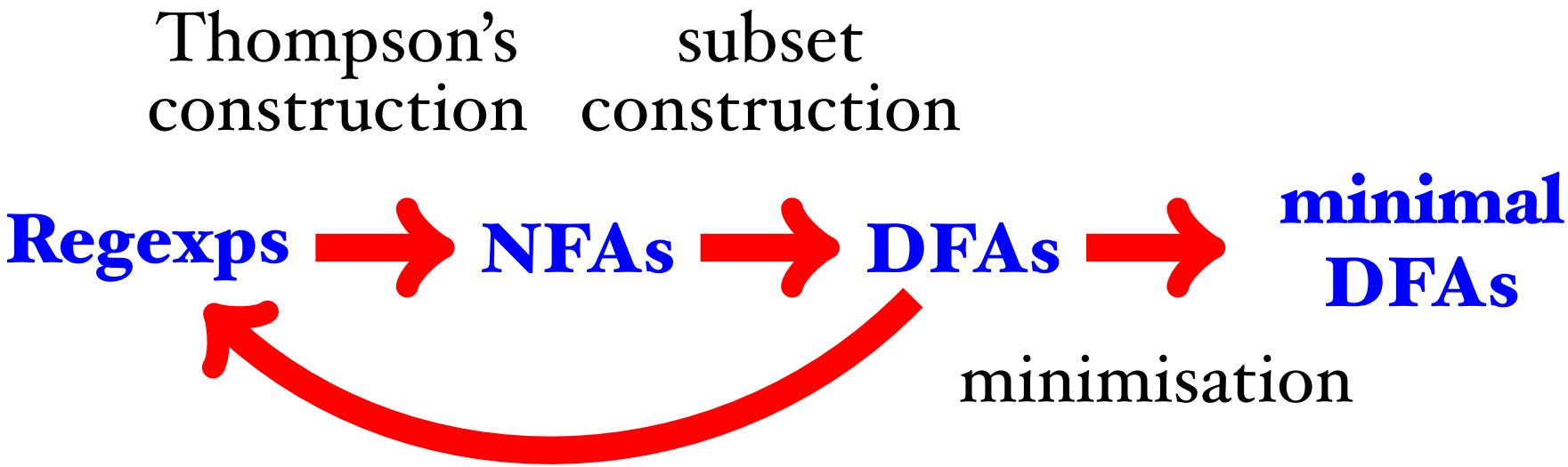
Automata and Formal Languages (4)

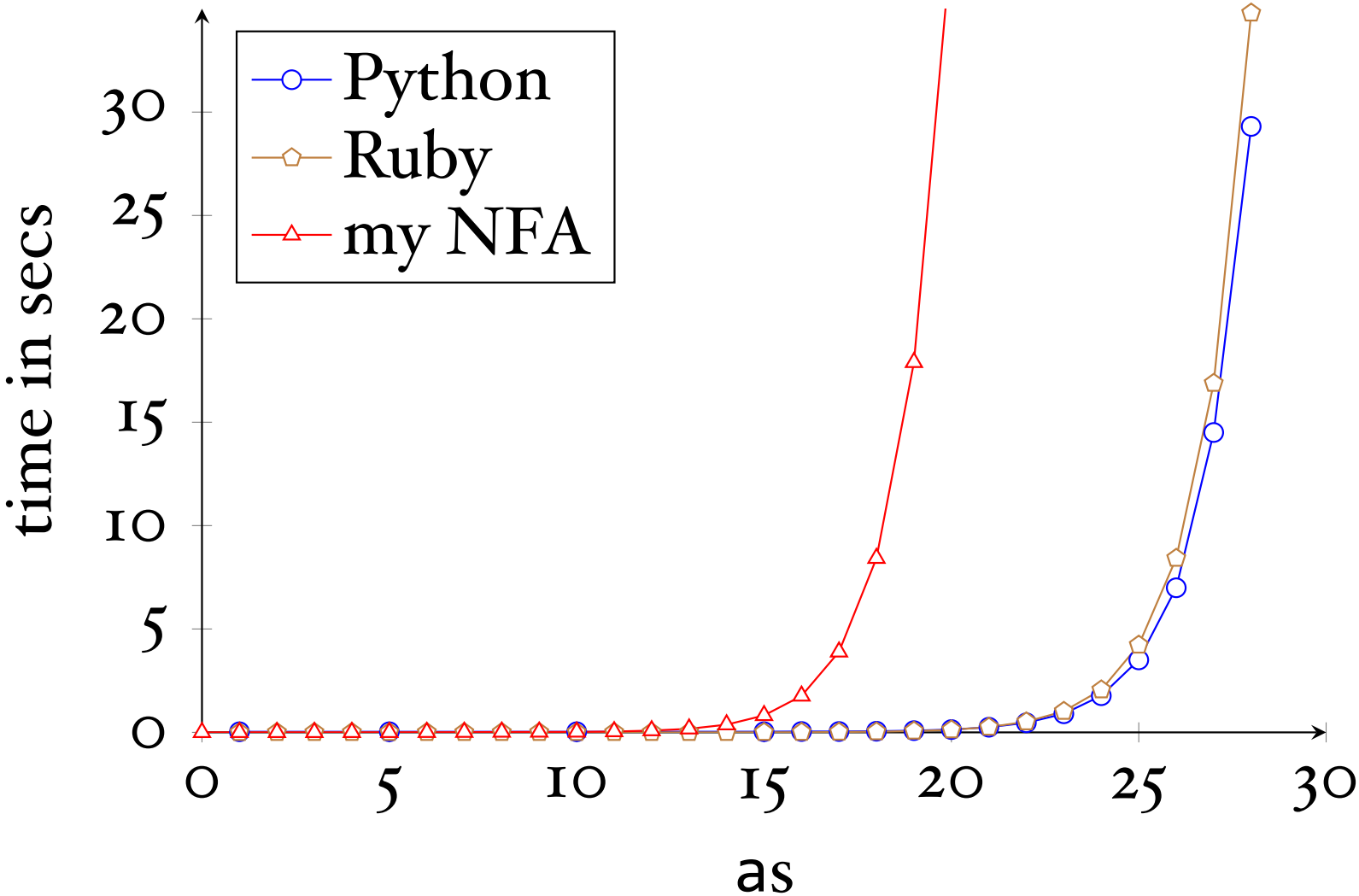
Email: christian.urban at kcl.ac.uk
Office: S1.27 (1st floor Strand Building)

Slides: KEATS (also home work is there)

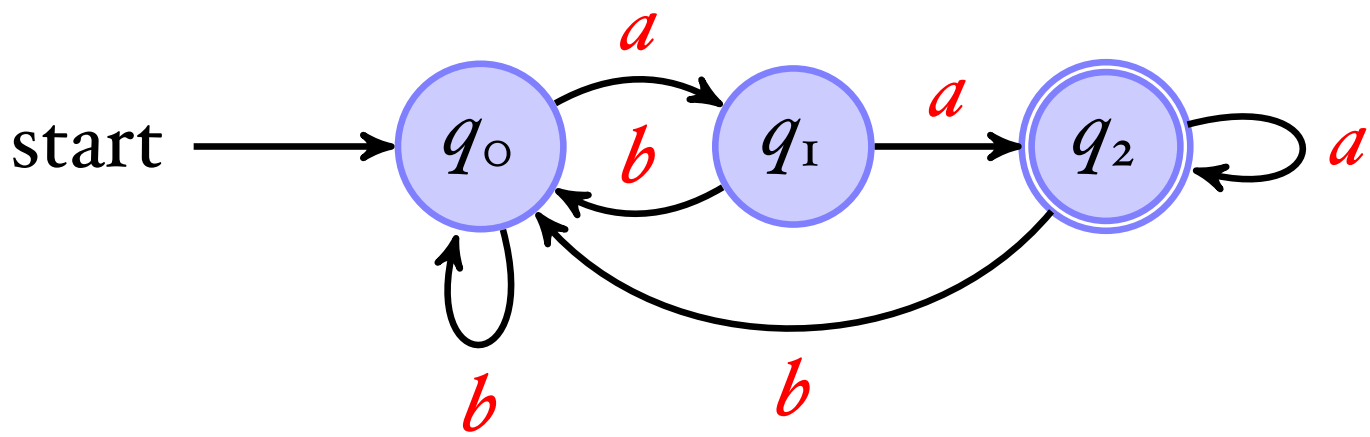
Regexps and Automata



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

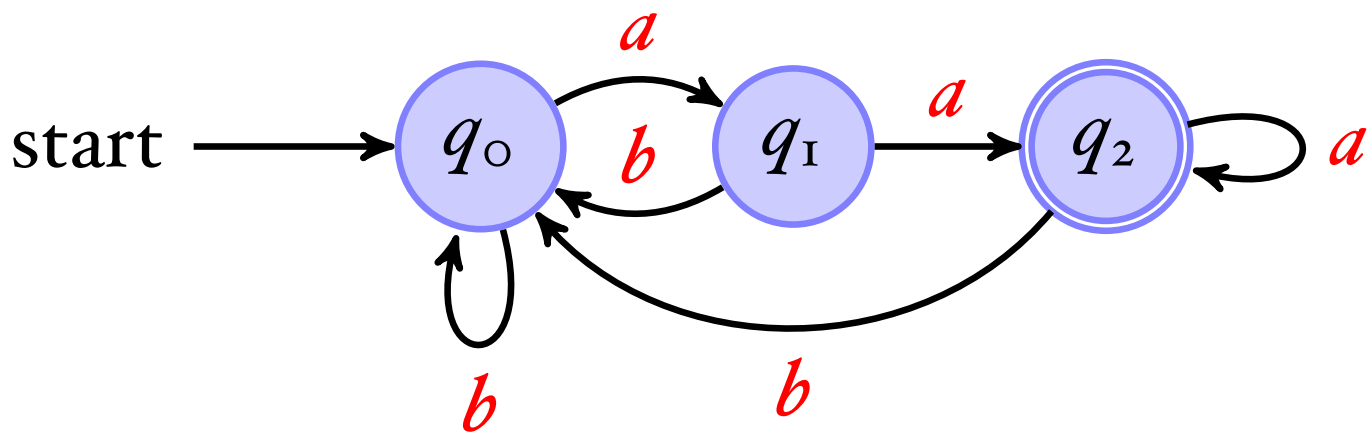


DFA to Rexp



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

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:

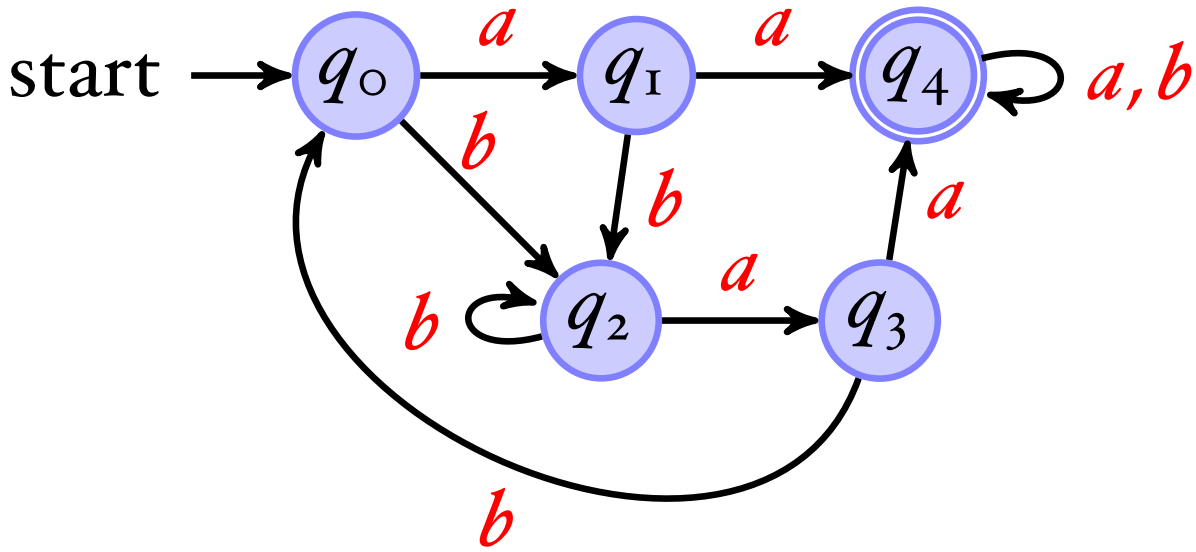
$$\text{If } q = qr + s \text{ then } q = sr^*$$

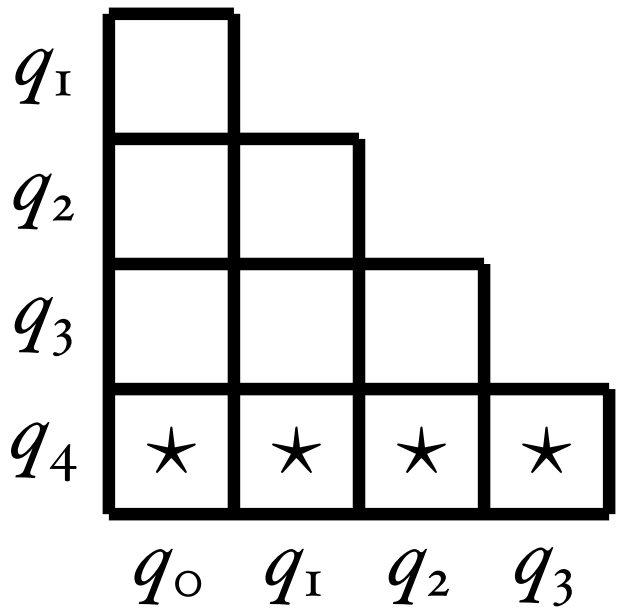
DFA Minimisation

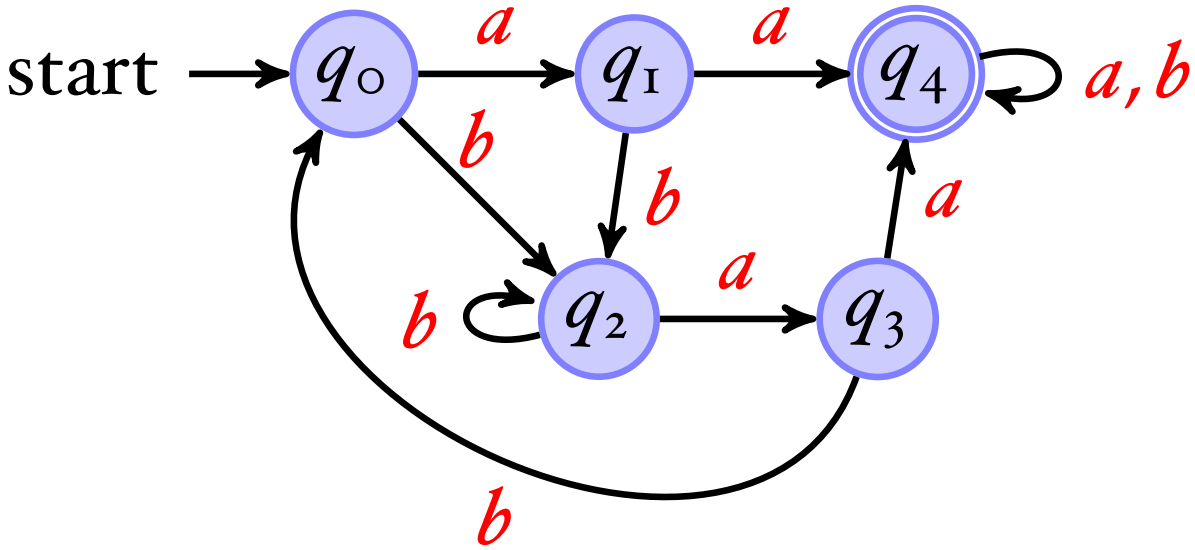
1. Take all pairs (q, p) with $q \neq p$
2. Mark all pairs that accepting and non-accepting states
3. For all unmarked pairs (q, p) and all characters c test whether

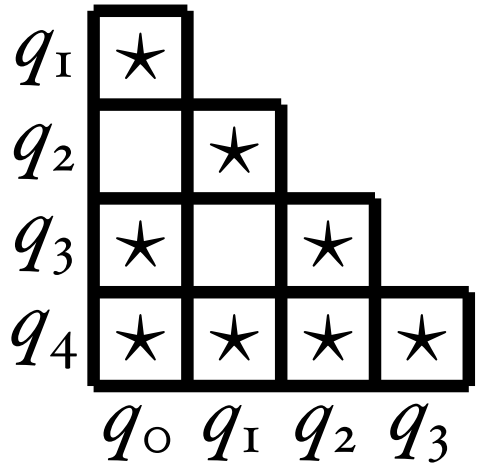
$$(\delta(q, c), \delta(p, c))$$

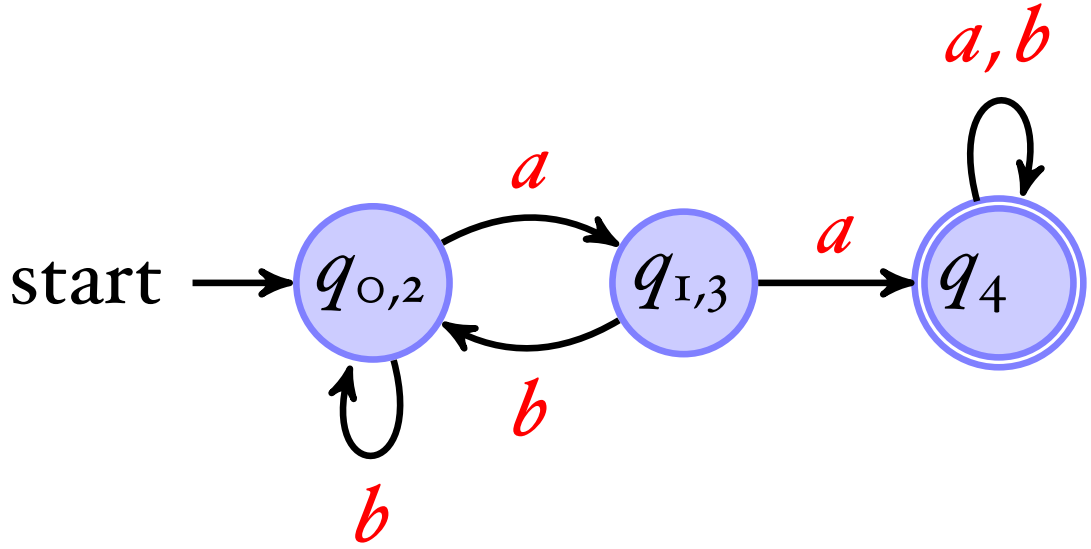
   are marked. If yes, then also mark (q, p).
4. Repeat last step until no change.
5. All unmarked pairs can be merged.


start
q_0
q_1
q_4
q_2
q_3
a
a
a, b
b
b
a
b
a
b


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

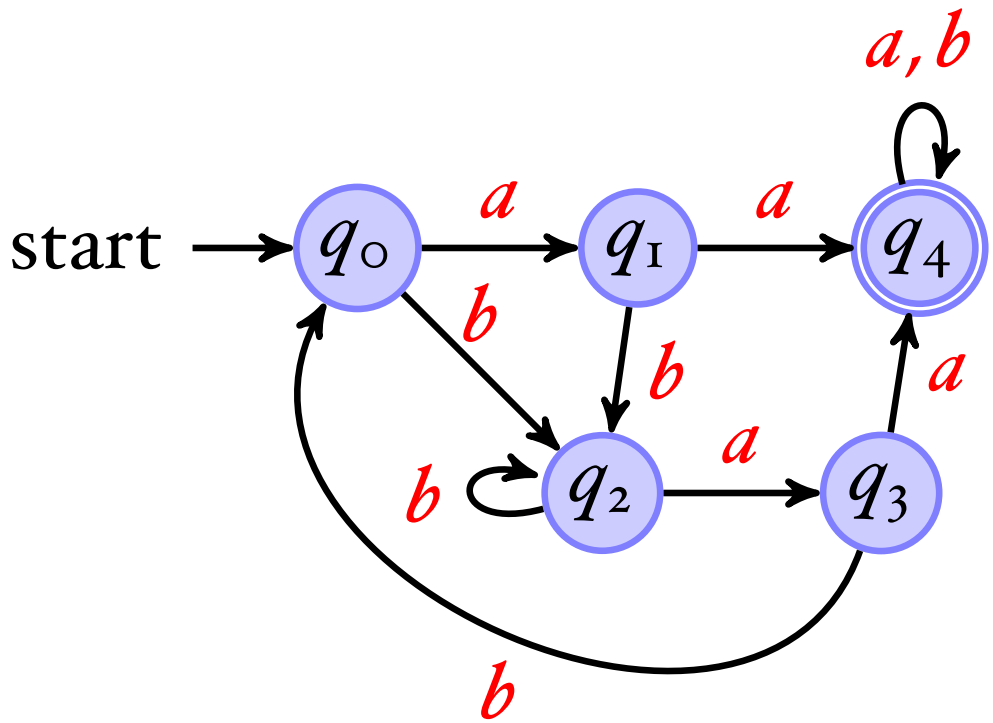




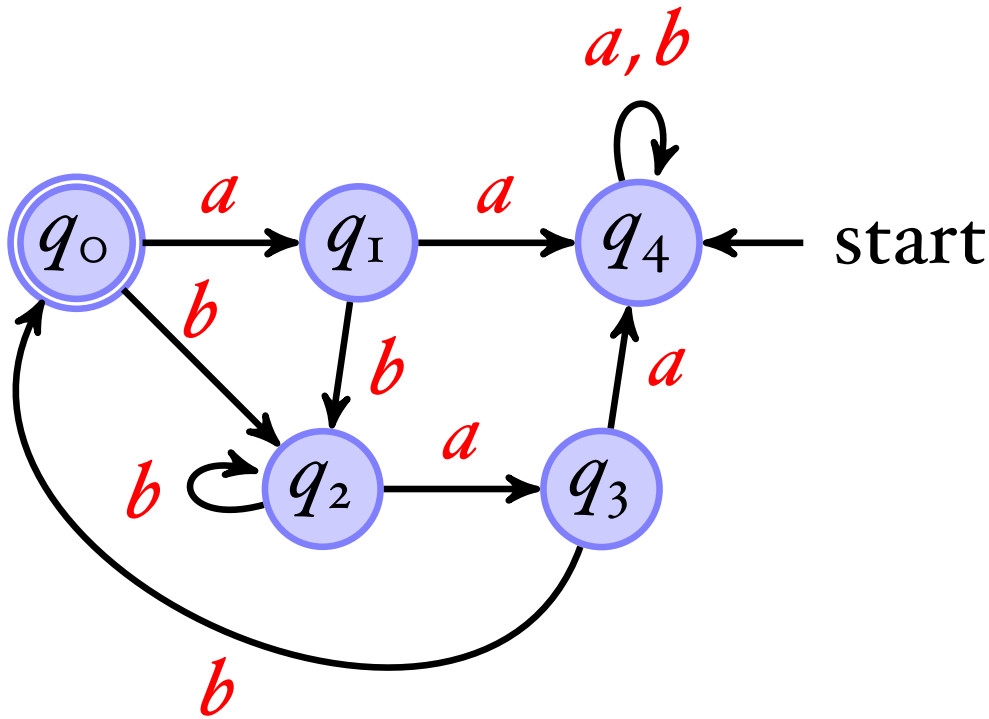


minimal automaton

Alternatives

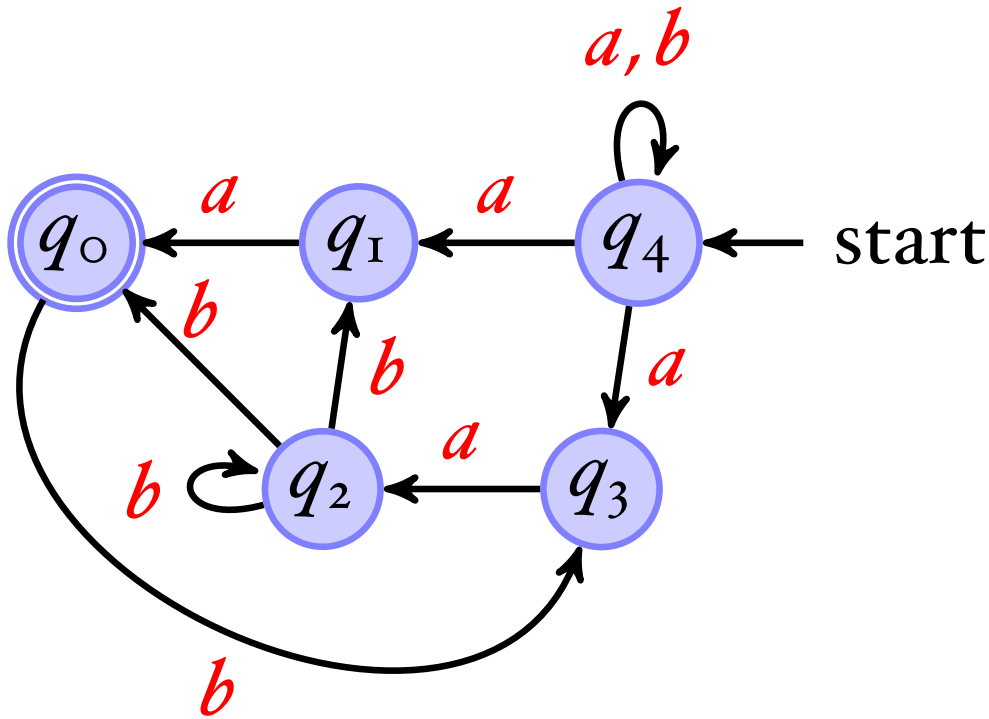


Alternatives



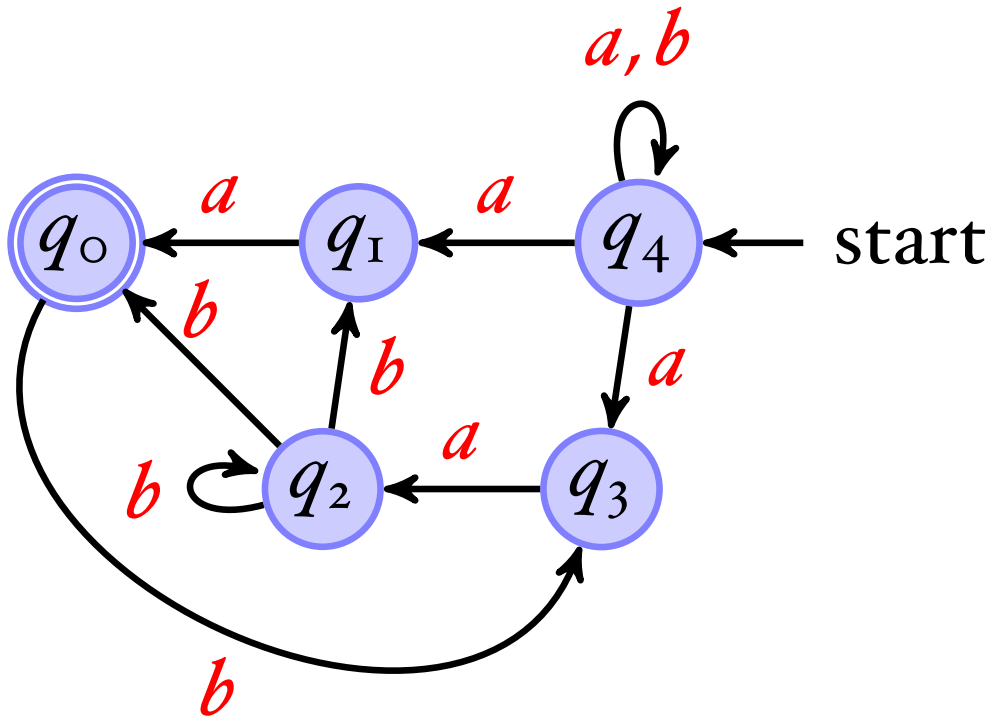
- exchange initial / accepting states

Alternatives



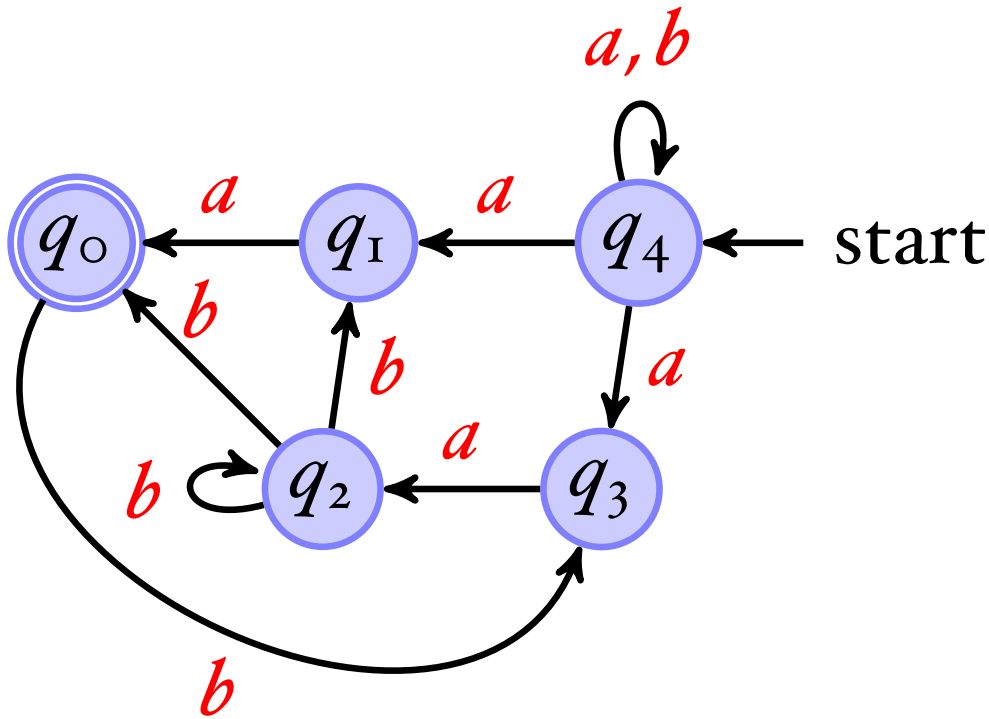
- exchange initial / accepting states
- reverse all edges

Alternatives



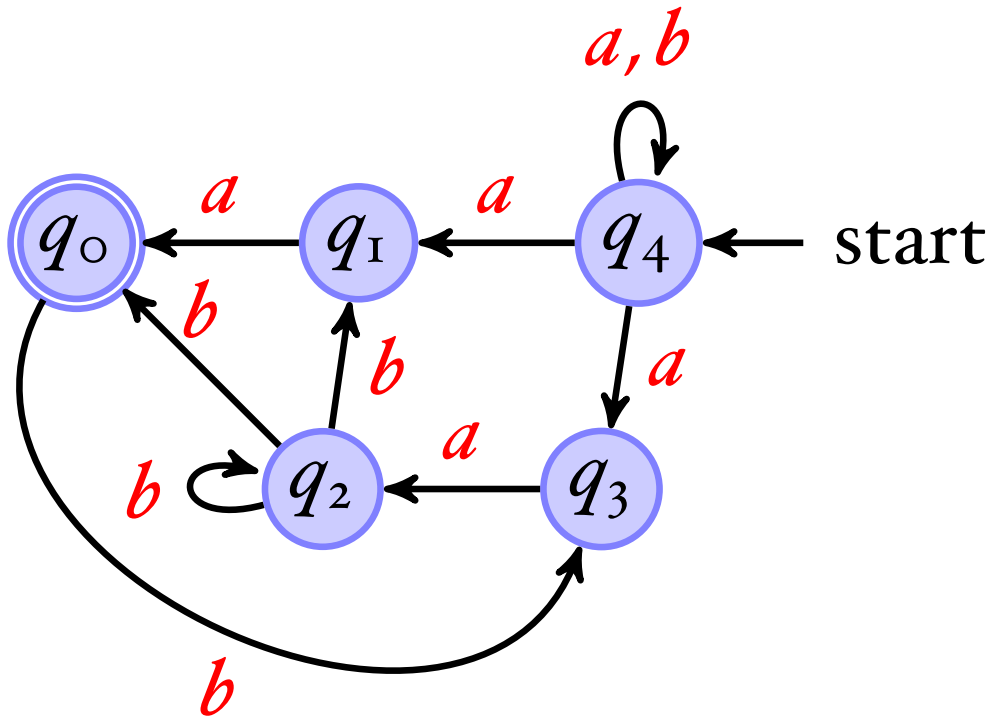
- exchange initial / accepting states
- reverse all edges
- subset construction $\Rightarrow$ DFA

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 ⇒ DFA
- repeat once more ⇒ minimal DFA

Regular Languages

Two equivalent definitions:

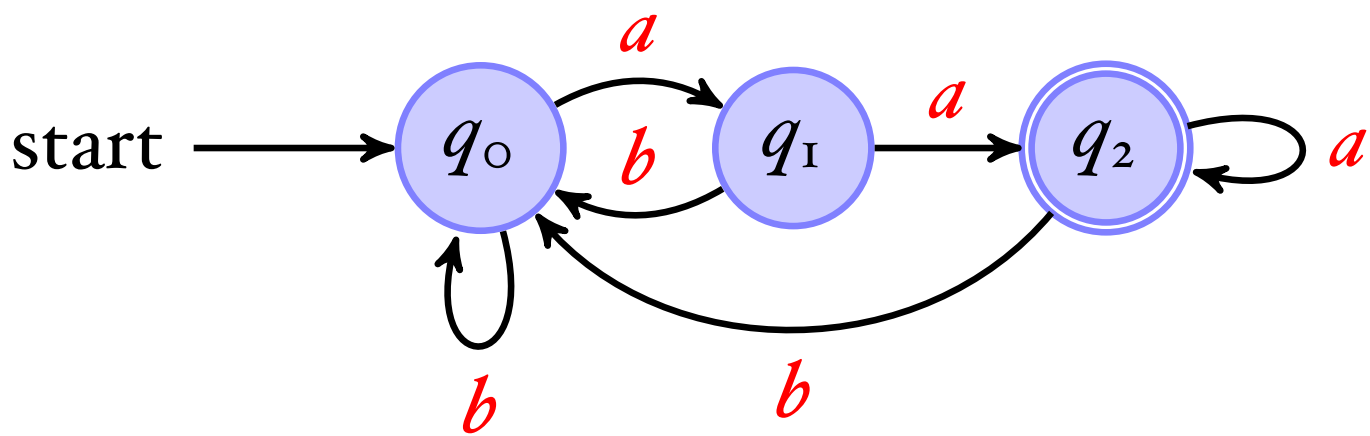
A language is regular iff there exists a regular expression that recognises all its strings.

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

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

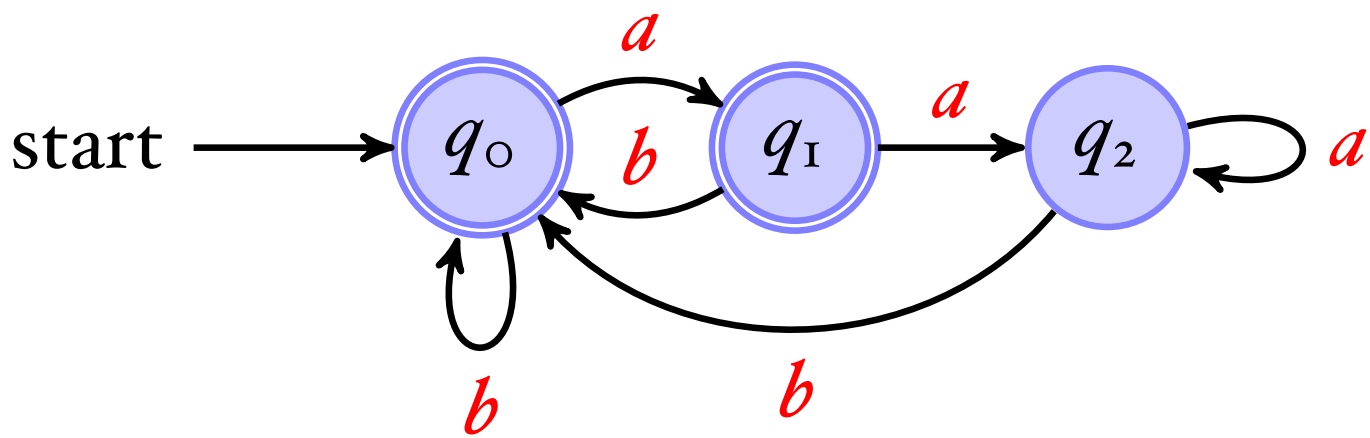
Negation

Regular languages are closed under negation:



Negation

Regular languages are closed under negation:



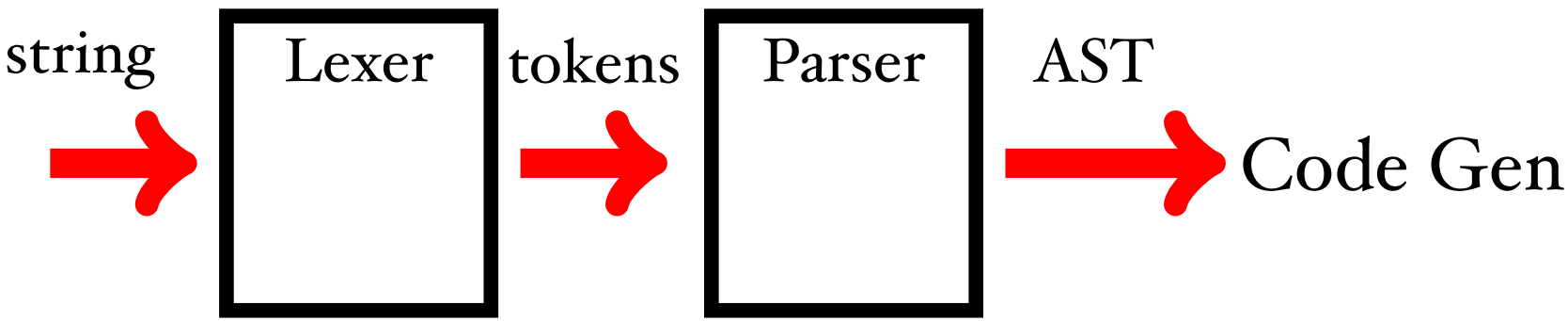
But requires that the automaton is completed!

```
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 +”

$$
\begin{array}{l}
\texttt{KEYWORD:} \\
\quad \texttt{if, then, else,} \\
\texttt{WHITESPACE:} \\
\quad \texttt{” ”, \textbackslash n,} \\
\texttt{IDENT:} \\
\quad \texttt{LETTER} \cdot (\texttt{LETTER} + \texttt{DIGIT} + \texttt{_})^* \\
\texttt{NUM:} \\
\quad (\texttt{NONZERODIGIT} \cdot \texttt{DIGIT}^*) + \texttt{0} \\
\texttt{OP:} \\
\quad \texttt{+} \\
\texttt{COMMENT:} \\
\quad \texttt{/*} \cdot \sim(\texttt{ALL}^* \cdot \texttt{(*/)} \cdot \texttt{ALL}^*) \cdot \texttt{*/}
\end{array}
$$

”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.

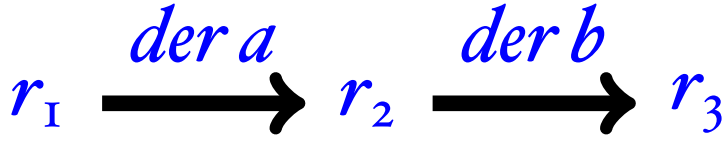
most posix matchers are buggy
http://www.haskell.org/haskellwiki/Regex_Posix

Sulzmann Matcher

We want to match the string abc using r_1:

Sulzmann Matcher

We want to match the string abc using r_1:

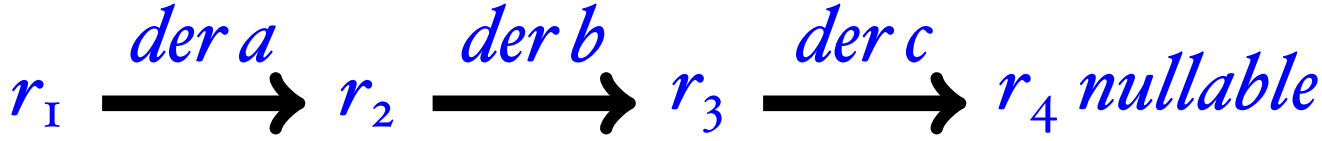


Sulzmann Matcher

We want to match the string abc using r_1:

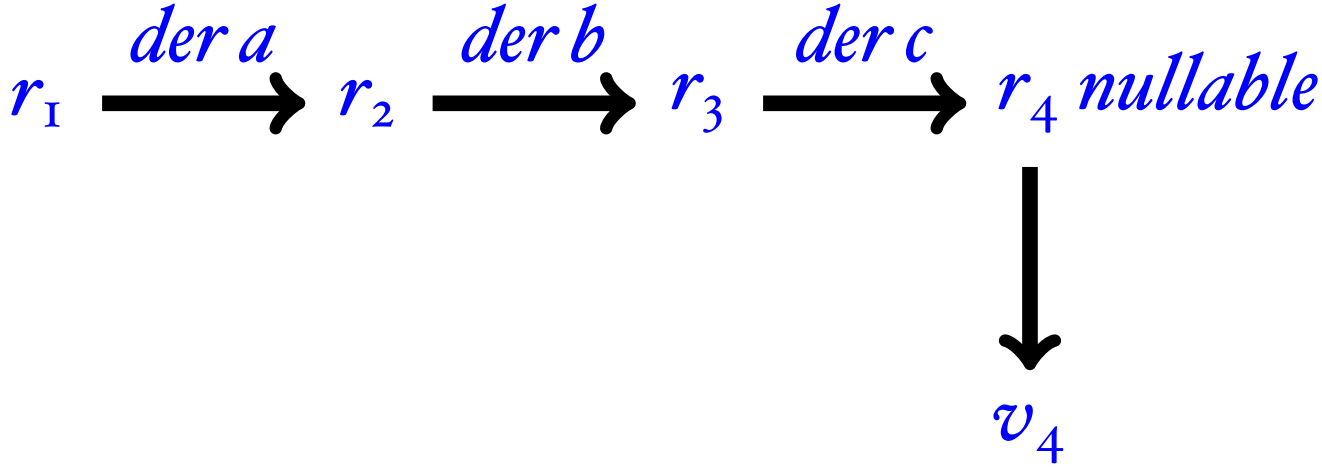
Sulzmann Matcher

We want to match the string abc using r_1:



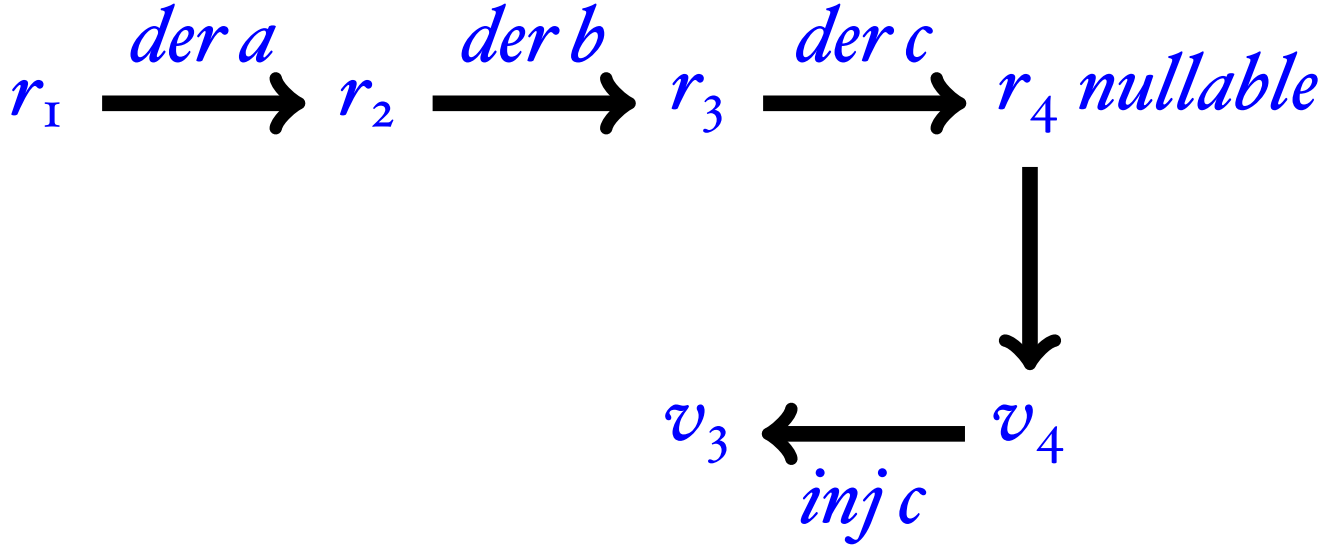
Sulzmann Matcher

We want to match the string abc using r_1:



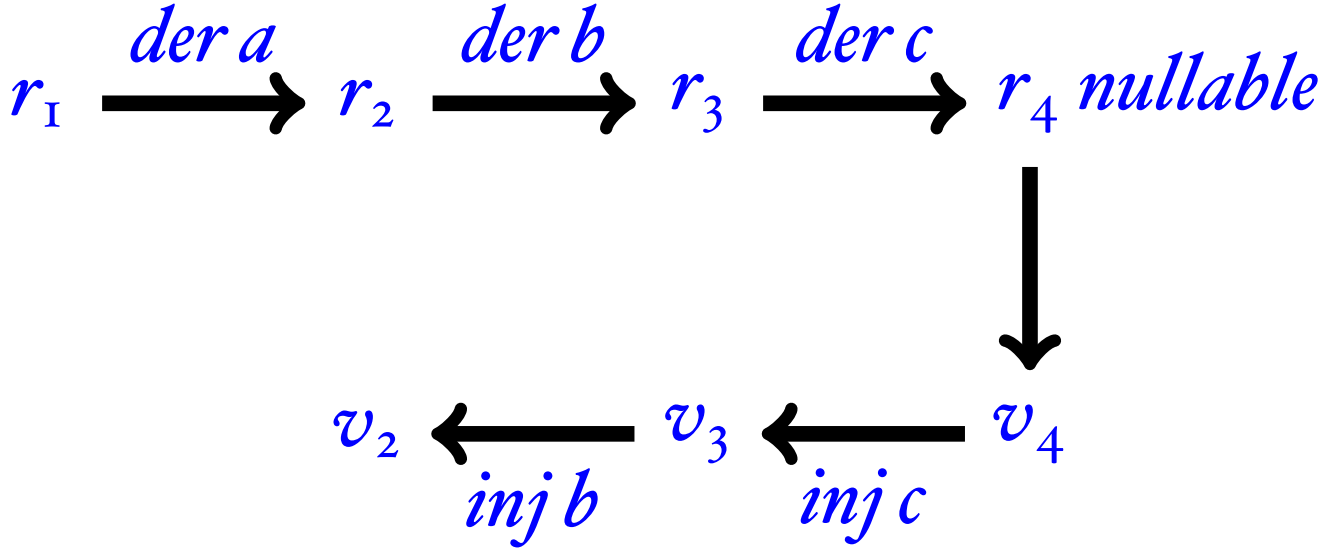
Sulzmann Matcher

We want to match the string abc using r_1:



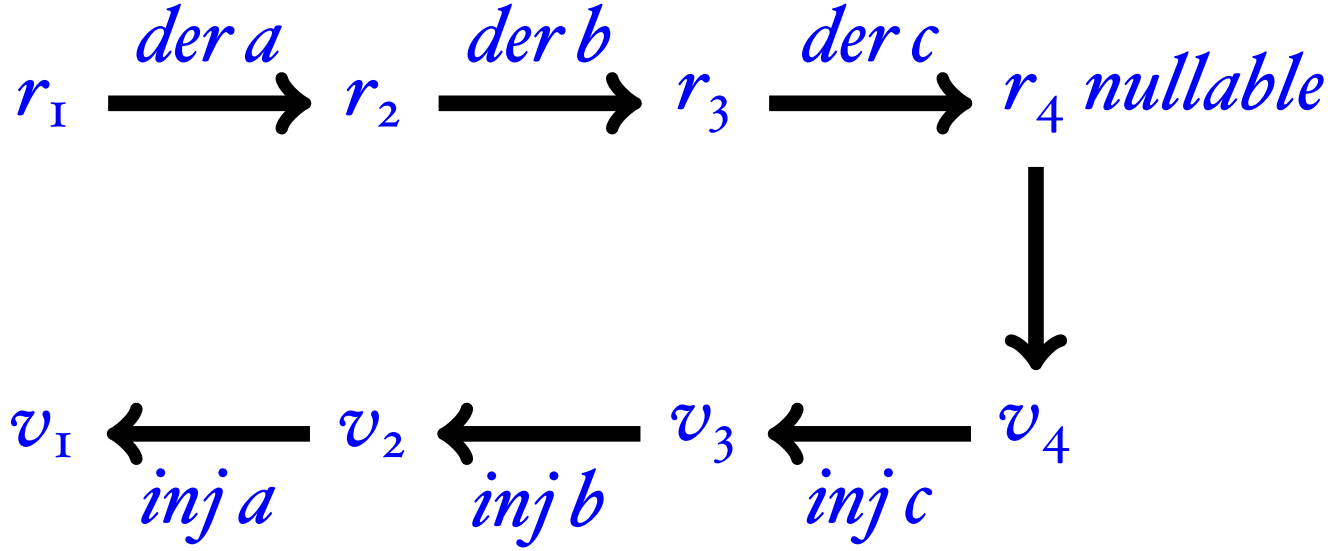
Sulzmann Matcher

We want to match the string abc using r_1:



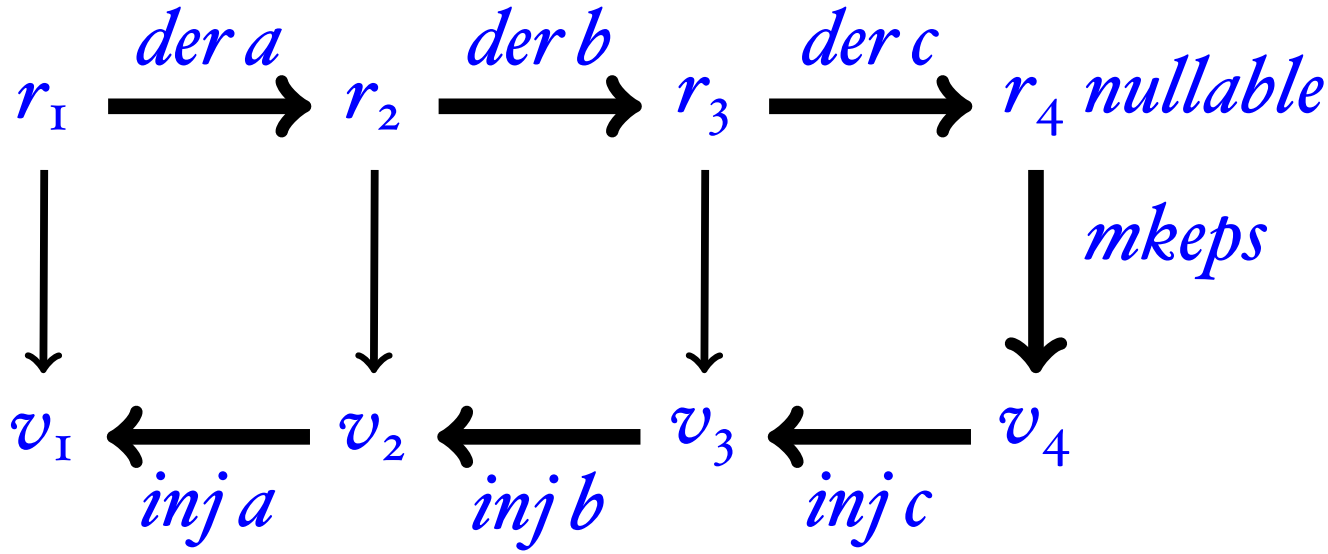
Sulzmann Matcher

We want to match the string abc using r_1:



Sulzmann Matcher

We want to match the string abc using r_1:



Regexes and Values

Regular expressions and their corresponding values:

$$
\begin{array}{rcl}
r & ::= & \varnothing \\
 & | & \epsilon \\
 & | & c \\
 & | & r_1 \cdot r_2 \\
 & | & r_1 + r_2 \\
 & | & \\
 & | & r^* \\
 & | &
\end{array}
\qquad
\begin{array}{rcl}
v & ::= & \\
 & & \mathit{Empty} \\
 & | & \mathit{Char}(c) \\
 & | & \mathit{Seq}(v_1, v_2) \\
 & | & \mathit{Left}(v) \\
 & | & \mathit{Right}(v) \\
 & | & [] \\
 & | & [v_1, \ldots v_n]
\end{array}
$$

Mkeps

Finding a (posix) value for recognising the empty string:

$$
\begin{array}{lcl}
mkeps\,\epsilon & \stackrel{\text{def}}{=} & Empty \\
mkeps\,r_1 + r_2 & \stackrel{\text{def}}{=} & \text{if } nullable(r_1) \\
 & & \text{then } Left(mkeps(r_1)) \\
 & & \text{else } Right(mkeps(r_2)) \\
mkeps\,r_1 \cdot r_2 & \stackrel{\text{def}}{=} & Seq(mkeps(r_1), mkeps(r_2)) \\
mkeps\,r^* & \stackrel{\text{def}}{=} & []
\end{array}
$$

Inject

Injecting ("Adding") a character to a value

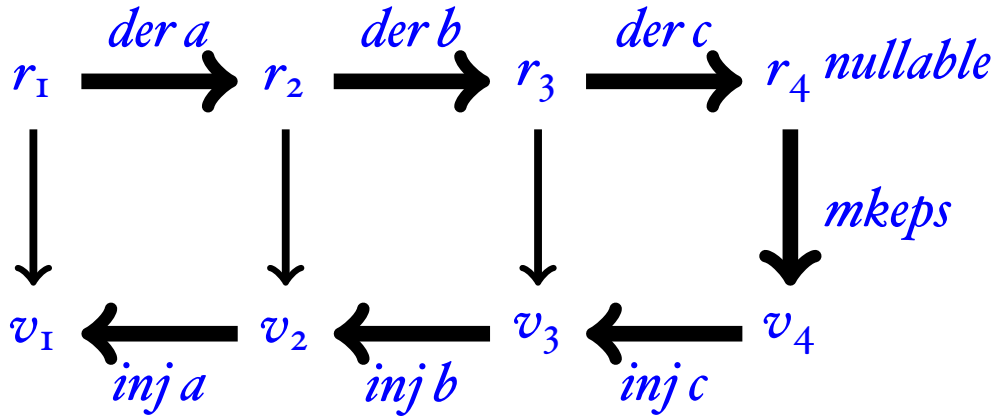
$$
\begin{array}{ll}
inj\,(c)\,c\,Empty & \stackrel{\text{def}}{=} Char\,c \\
inj\,(r_1 + r_2)\,c\,Left(v) & \stackrel{\text{def}}{=} Left(inj\,r_1\,c\,v) \\
inj\,(r_1 + r_2)\,c\,Right(v) & \stackrel{\text{def}}{=} Right(inj\,r_2\,c\,v) \\
inj\,(r_1 \cdot r_2)\,c\,Seq(v_1, v_2) & \stackrel{\text{def}}{=} Seq(inj\,r_1\,c\,v_1, v_2) \\
inj\,(r_1 \cdot r_2)\,c\,Left(Seq(v_1, v_2)) & \stackrel{\text{def}}{=} Seq(inj\,r_1\,c\,v_1, v_2) \\
inj\,(r_1 \cdot r_2)\,c\,Right(v) & \stackrel{\text{def}}{=} Seq(mkeps(r_1), inj\,r_2\,c\,v) \\
inj\,(r^*)\,c\,Sequ(v, vs) & \stackrel{\text{def}}{=} inj\,r\,c\,v :: vs
\end{array}
$$

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

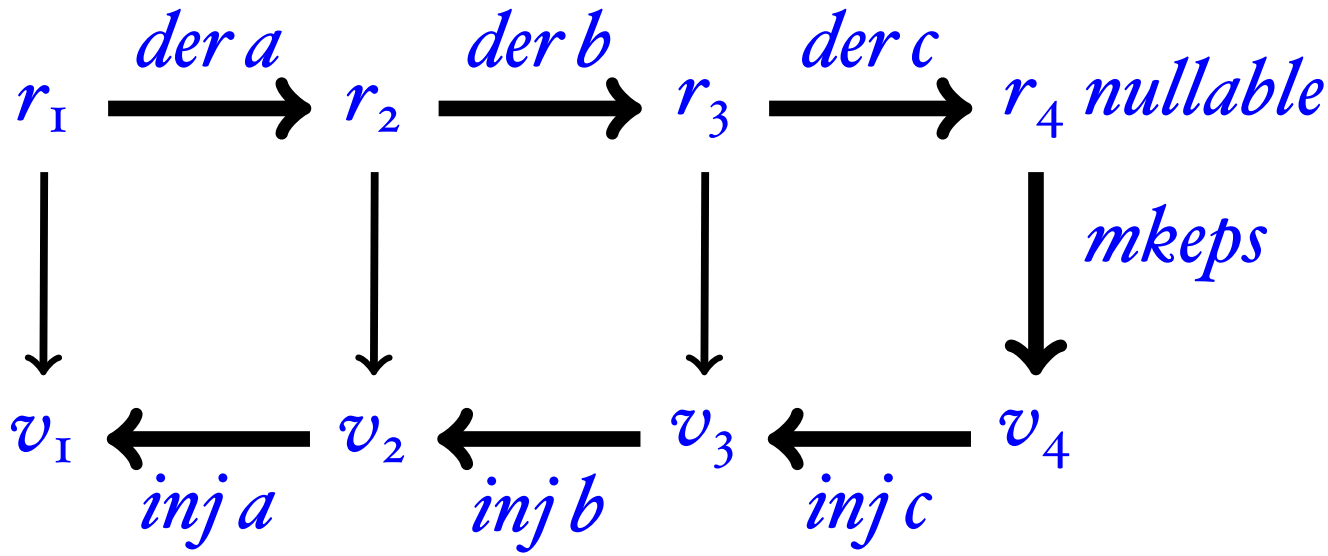
for extracting subpatterns $(z : ((x : ab) + (y : ba))$

While Tokens

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

Simplification

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



Rectification

rectification functions:

$$
\begin{array}{lcll}
r \cdot \varnothing & \mapsto & \varnothing & \\
\varnothing \cdot r & \mapsto & \varnothing & \\
r \cdot \epsilon & \mapsto & r & \lambda f_1 f_2\, v.\, Seq(f_1\, v, f_2\, Empty) \\
\epsilon \cdot r & \mapsto & r & \lambda f_1 f_2\, v.\, Seq(f_1\, Empty, f_2\, v) \\
r + \varnothing & \mapsto & r & \lambda f_1 f_2\, v.\, Left(f_1\, v) \\
\varnothing + r & \mapsto & r & \lambda f_1 f_2\, v.\, Right(f_2\, v) \\
r + r & \mapsto & r & \lambda f_1 f_2\, v.\, Left(f_1\, v)
\end{array}
$$

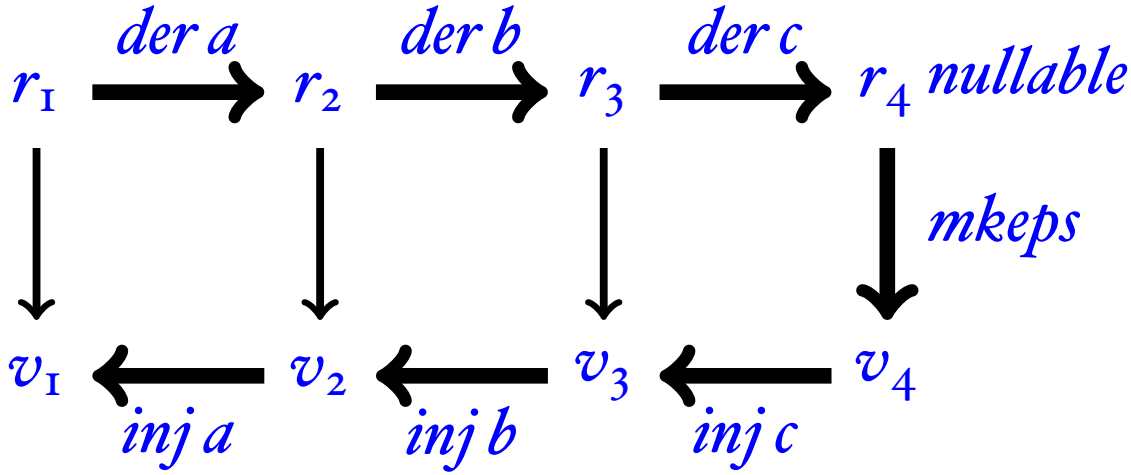
Rectification

rectification functions:

$$
\begin{array}{lcll}
r \cdot \varnothing & \mapsto & \varnothing & \\
\varnothing \cdot r & \mapsto & \varnothing & \\
r \cdot \epsilon & \mapsto & r & \lambda f_1 f_2\, v.\, Seq(f_1\, v, f_2\, Empty) \\
\epsilon \cdot r & \mapsto & r & \lambda f_1 f_2\, v.\, Seq(f_1\, Empty, f_2\, v) \\
r + \varnothing & \mapsto & r & \lambda f_1 f_2\, v.\, Left(f_1\, v) \\
\varnothing + r & \mapsto & r & \lambda f_1 f_2\, v.\, Right(f_2\, v) \\
r + r & \mapsto & r & \lambda f_1 f_2\, v.\, Left(f_1\, v)
\end{array}
$$

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

Lexing with Simplification

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

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