

Automata and Formal Languages (3)

Email: christian.urban at kcl.ac.uk

Office: S1.27 (1st floor Strand Building)

Slides: KEATS (also home work and course-work is there)

Regular Expressions

In programming languages they are often used to recognise:

- symbols, digits
- identifiers
- numbers (non-leading zeros)
- keywords
- comments

<http://www.regexper.com>

Last Week

Last week I showed you a regular expression matcher which works provably correct in all cases (we did not do the proving part though)

matches r s if and only if $s \in L(r)$

by Janusz Brzozowski (1964)

The Derivative of a Rexp

$$\mathit{der} c (\emptyset) \stackrel{\text{def}}{=} \emptyset$$

$$\mathit{der} c (\epsilon) \stackrel{\text{def}}{=} \emptyset$$

$$\mathit{der} c (d) \stackrel{\text{def}}{=} \text{if } c = d \text{ then } \epsilon \text{ else } \emptyset$$

$$\mathit{der} c (r_1 + r_2) \stackrel{\text{def}}{=} \mathit{der} c r_1 + \mathit{der} c r_2$$

$$\mathit{der} c (r_1 \cdot r_2) \stackrel{\text{def}}{=} \begin{array}{l} \text{if } \mathit{nullable}(r_1) \\ \text{then } (\mathit{der} c r_1) \cdot r_2 + \mathit{der} c r_2 \\ \text{else } (\mathit{der} c r_1) \cdot r_2 \end{array}$$

$$\mathit{der} c (r^*) \stackrel{\text{def}}{=} (\mathit{der} c r) \cdot (r^*)$$

$$\mathit{ders} [] r \stackrel{\text{def}}{=} r$$

$$\mathit{ders} (c :: s) r \stackrel{\text{def}}{=} \mathit{ders} s (\mathit{der} c r)$$

To see what is going on, define

$$\mathit{Der} c A \stackrel{\text{def}}{=} \{s \mid c :: s \in A\}$$

For $A = \{\mathit{foo}, \mathit{bar}, \mathit{frak}\}$ then

$$\mathit{Der} f A = \{\mathit{oo}, \mathit{rak}\}$$

$$\mathit{Der} b A = \{\mathit{ar}\}$$

$$\mathit{Der} a A = \emptyset$$

The Idea of the Algorithm

If we want to recognise the string abc with regular expression r then

- $Der a (L(r))$

The Idea of the Algorithm

If we want to recognise the string abc with regular expression r then

- 1 $Der a (L(r))$
- 2 $Der b (Der a (L(r)))$

The Idea of the Algorithm

If we want to recognise the string abc with regular expression r then

- 1 $Der a (L(r))$
- 2 $Der b (Der a (L(r)))$
- 3 $Der c (Der b (Der a (L(r))))$

The Idea of the Algorithm

If we want to recognise the string abc with regular expression r then

- 1 $Der a (L(r))$
- 2 $Der b (Der a (L(r)))$
- 3 $Der c (Der b (Der a (L(r))))$
- 4 finally we test whether the empty string is in this set

The Idea of the Algorithm

If we want to recognise the string abc with regular expression r then

- 1 $Der a (L(r))$
- 2 $Der b (Der a (L(r)))$
- 3 $Der c (Der b (Der a (L(r))))$
- 4 finally we test whether the empty string is in this set

The matching algorithm works similarly, just over regular expressions instead of sets.

Input: string *abc* and regular expression *r*

- 1 *der a r*
- 2 *der b (der a r)*
- 3 *der c (der b (der a r))*

Input: string *abc* and regular expression *r*

- 1 *der a r*
- 2 *der b (der a r)*
- 3 *der c (der b (der a r))*
- 4 finally check whether the last regular expression can match the empty string

We proved already

nullable(r) if and only if $\epsilon \in L(r)$

by induction on the regular expression.

We proved already

nullable(r) if and only if $\epsilon \in L(r)$

by induction on the regular expression.

Any Questions?

We need to prove

$$L(\text{der } c r) = \text{Der } c (L(r))$$

by induction on the regular expression.

Proofs about Rexps

- P holds for \emptyset , ϵ and c
- P holds for $r_1 + r_2$ under the assumption that P already holds for r_1 and r_2 .
- P holds for $r_1 \cdot r_2$ under the assumption that P already holds for r_1 and r_2 .
- P holds for r^* under the assumption that P already holds for r .

Proofs about Natural Numbers and Strings

- P holds for 0 and
- P holds for $n + 1$ under the assumption that P already holds for n

- P holds for $[]$ and
- P holds for $c::s$ under the assumption that P already holds for s

Languages

A **language** is a set of strings.

A **regular expression** specifies a language.

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

Languages

A **language** is a set of strings.

A **regular expression** specifies a language.

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

not all languages are regular, e.g. $a^n b^n$ is not

Regular Expressions

$r ::=$	\emptyset	null
	ϵ	empty string / "" / []
	c	character
	$r_1 \cdot r_2$	sequence
	$r_1 + r_2$	alternative / choice
	r^*	star (zero or more)

How about ranges $[a-z]$, r^+ and $\sim r$? Do they increase the set of languages we can recognise?

Negation of Regular Expr's

- $\sim r$ (everything that r cannot recognise)
- $L(\sim r) \stackrel{\text{def}}{=} UNIV - L(r)$
- $nullable(\sim r) \stackrel{\text{def}}{=} \text{not } (nullable(r))$
- $derc(\sim r) \stackrel{\text{def}}{=} \sim (derc r)$

Negation of Regular Expr's

- $\sim r$ (everything that r cannot recognise)
- $L(\sim r) \stackrel{\text{def}}{=} UNIV - L(r)$
- $nullable(\sim r) \stackrel{\text{def}}{=} \text{not } (nullable(r))$
- $derc(\sim r) \stackrel{\text{def}}{=} \sim (derc r)$

Used often for recognising comments:

$$/ \cdot * \cdot (\sim ([a-z]^* \cdot * \cdot / \cdot [a-z]^*)) \cdot * \cdot /$$

Negation

Assume you have an alphabet consisting of the letters *a*, *b* and *c* only. Find a (basic!) regular expression that matches all strings *except* *ab* and *ac*!

Automata

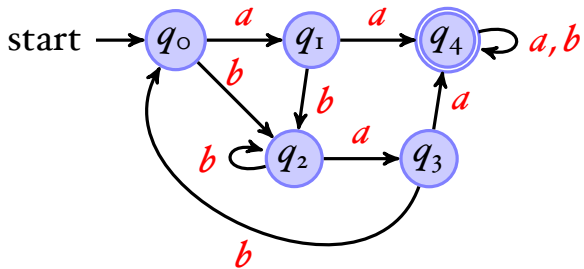
A **deterministic finite automaton** consists of:

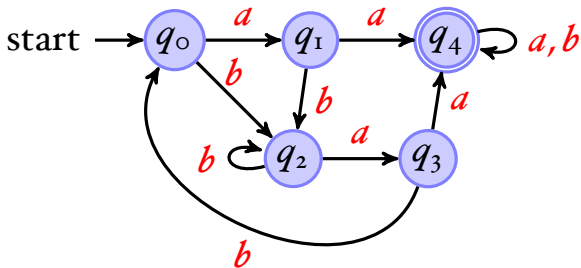
- a set of states
- one of these states is the start state
- some states are accepting states, and
- there is transition function

which takes a state as argument and a character and produces a new state

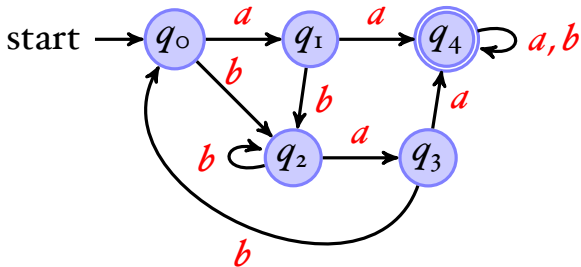
this function might not be everywhere defined

$$A(Q, q_0, F, \delta)$$





- the start state can be an accepting state
- it is possible that there is no accepting state
- all states might be accepting (but this does not necessarily mean all strings are accepted)



for this automaton δ is the function

$$\begin{array}{lll}
 (q_0, a) \rightarrow q_1 & (q_1, a) \rightarrow q_4 & (q_4, a) \rightarrow q_4 \\
 (q_0, b) \rightarrow q_2 & (q_1, b) \rightarrow q_2 & (q_4, b) \rightarrow q_4 \quad \dots
 \end{array}$$

Accepting a String

Given

$$A(\mathcal{Q}, q_0, F, \delta)$$

you can define

$$\begin{aligned}\hat{\delta}(q, []) &\stackrel{\text{def}}{=} q \\ \hat{\delta}(q, c :: s) &\stackrel{\text{def}}{=} \hat{\delta}(\delta(q, c), s)\end{aligned}$$

Accepting a String

Given

$$A(\mathcal{Q}, q_0, F, \delta)$$

you can define

$$\begin{aligned}\hat{\delta}(q, []) &\stackrel{\text{def}}{=} q \\ \hat{\delta}(q, c :: s) &\stackrel{\text{def}}{=} \hat{\delta}(\delta(q, c), s)\end{aligned}$$

Whether a string s is accepted by A ?

$$\hat{\delta}(q_0, s) \in F$$

Non-Deterministic Finite Automata

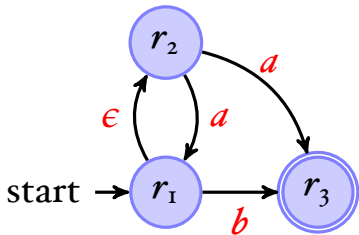
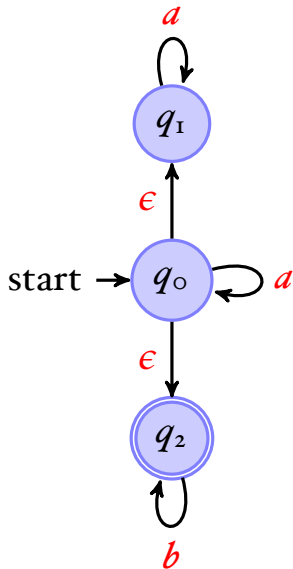
A non-deterministic finite automaton consists again of:

- a finite set of states
- one of these states is the start state
- some states are accepting states, and
- there is transition **relation**

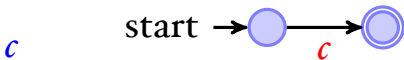
$$\begin{aligned}(q_1, a) &\rightarrow q_2 \\ (q_1, a) &\rightarrow q_3\end{aligned}$$

$$(q_1, \epsilon) \rightarrow q_2$$

Two NFA Examples

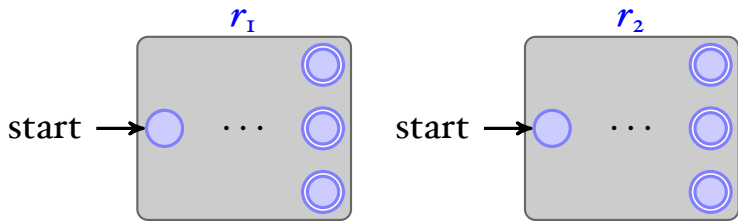


Rexp to NFA



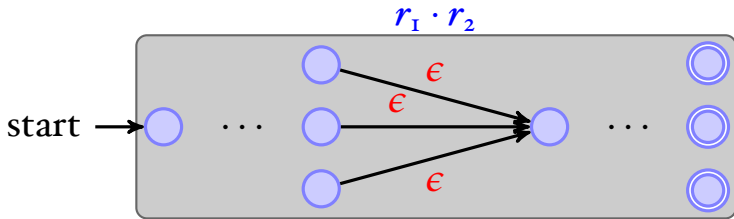
Case $r_1 \cdot r_2$

By recursion we are given two automata:



We need to (1) change the accepting nodes of the first automaton into non-accepting nodes, and (2) connect them via ϵ -transitions to the starting state of the second automaton.

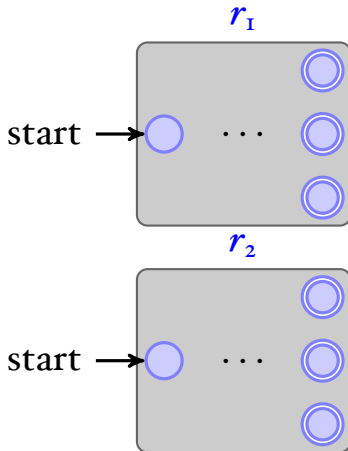
Case $r_1 \cdot r_2$



We need to (1) change the accepting nodes of the first automaton into non-accepting nodes, and (2) connect them via ϵ -transitions to the starting state of the second automaton.

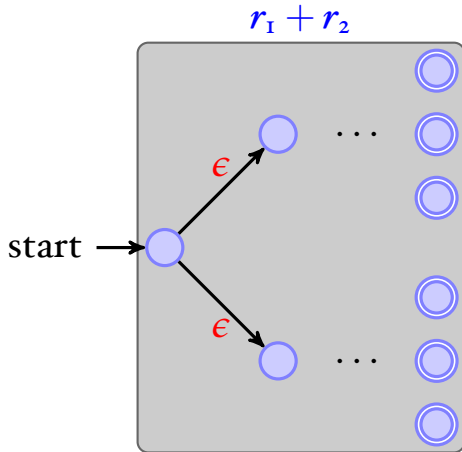
Case $r_1 + r_2$

By recursion we are given two automata:



We (1) need to introduce a new starting state and (2) connect it to the original two starting states.

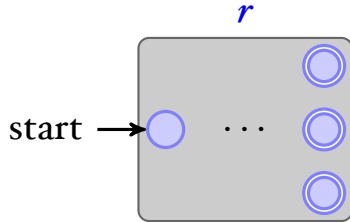
Case $r_1 + r_2$



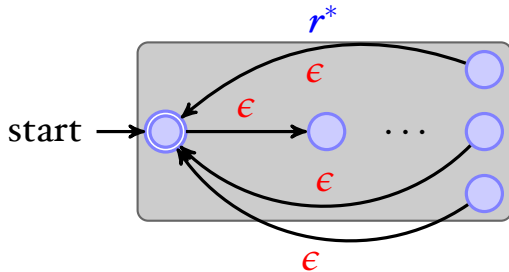
We (1) need to introduce a new starting state and (2) connect it to the original two starting states.

Case r^*

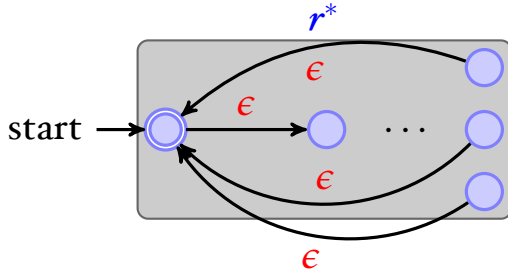
By recursion we are given an automaton for r :



Case r^*

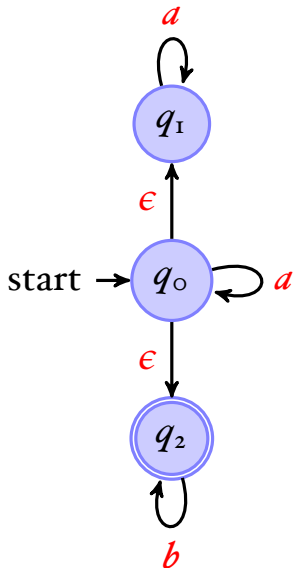


Case r^*



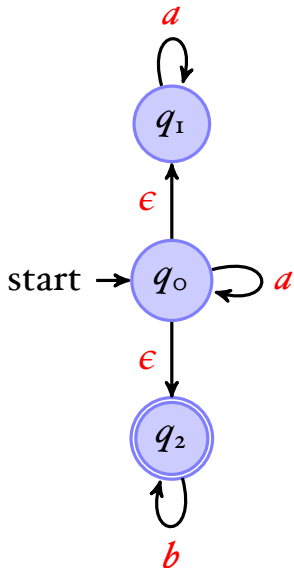
Why can't we just have an epsilon transition from the accepting states to the starting state?

Subset Construction



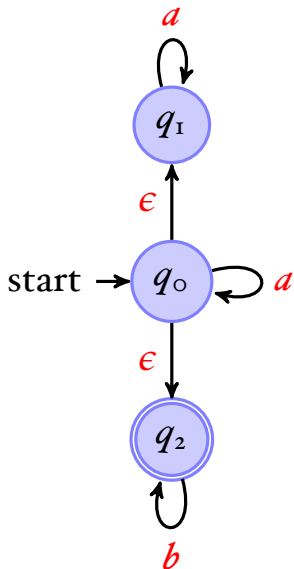
nodes	a	b
\emptyset		
$\{0\}$		
$\{1\}$		
$\{2\}$		
$\{0, 1\}$		
$\{0, 2\}$		
$\{1, 2\}$		
$\{0, 1, 2\}$		

Subset Construction



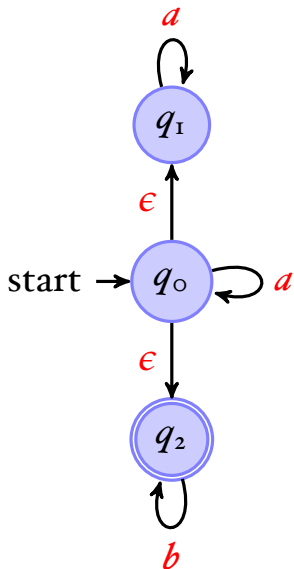
nodes	a	b
\emptyset	\emptyset	\emptyset
$\{0\}$		
$\{1\}$		
$\{2\}$		
$\{0, 1\}$		
$\{0, 2\}$		
$\{1, 2\}$		
$\{0, 1, 2\}$		

Subset Construction



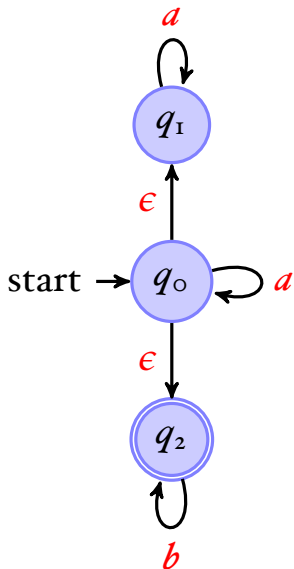
nodes	a	b
\emptyset	\emptyset	\emptyset
$\{0\}$	$\{0, 1, 2\}$	$\{2\}$
$\{1\}$	$\{1\}$	\emptyset
$\{2\}$	\emptyset	$\{2\}$
$\{0, 1\}$		
$\{0, 2\}$		
$\{1, 2\}$		
$\{0, 1, 2\}$		

Subset Construction



nodes	a	b
\emptyset	\emptyset	\emptyset
$\{0\}$	$\{0, 1, 2\}$	$\{2\}$
$\{1\}$	$\{1\}$	\emptyset
$\{2\}$	\emptyset	$\{2\}$
$\{0, 1\}$	$\{0, 1, 2\}$	$\{2\}$
$\{0, 2\}$	$\{0, 1, 2\}$	$\{2\}$
$\{1, 2\}$	$\{1\}$	$\{2\}$
$\{0, 1, 2\}$	$\{0, 1, 2\}$	$\{2\}$

Subset Construction



nodes	a	b
\emptyset	\emptyset	\emptyset
$\{0\}$	$\{0, 1, 2\}$	$\{2\}$
$\{1\}$	$\{1\}$	\emptyset
$\{2\}$ *	\emptyset	$\{2\}$
$\{0, 1\}$	$\{0, 1, 2\}$	$\{2\}$
$\{0, 2\}$ *	$\{0, 1, 2\}$	$\{2\}$
$\{1, 2\}$ *	$\{1\}$	$\{2\}$
s: $\{0, 1, 2\}$ *	$\{0, 1, 2\}$	$\{2\}$

Regexps and Automata

Thompson's construction subset construction

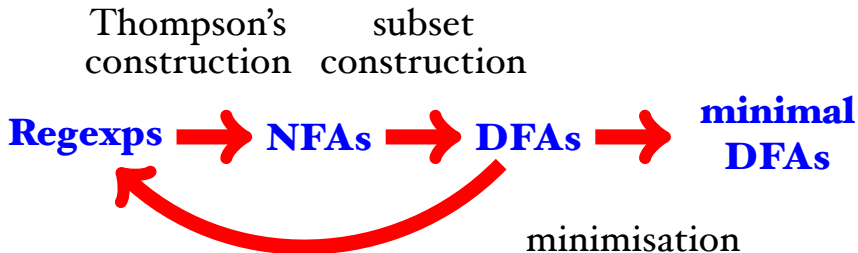
Regexps  **NFAs**  **DFAs**

Regexps and Automata

Thompson's construction subset construction



Regexps and Automata



Regular Languages

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

or **equivalently**

A language is **regular** iff there exists a deterministic finite automaton that recognises all its strings.

Regular Languages

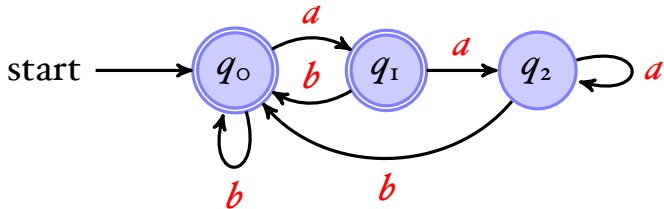
A language is **regular** iff there exists a regular expression that recognises all its strings.

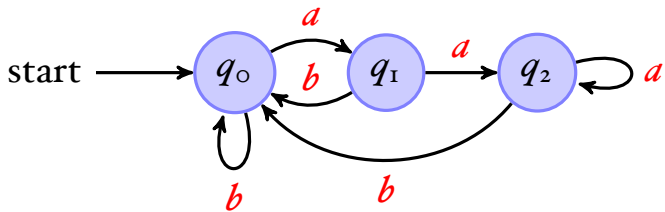
or **equivalently**

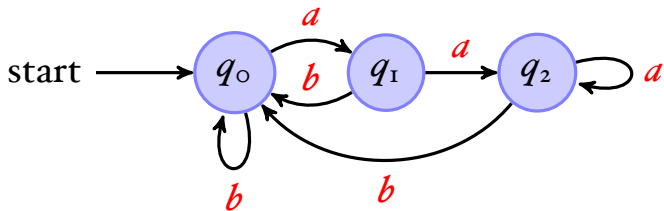
A language is **regular** iff there exists a deterministic finite automaton that recognises all its strings.

Why is every finite set of strings a regular language?

DFA to Rexp



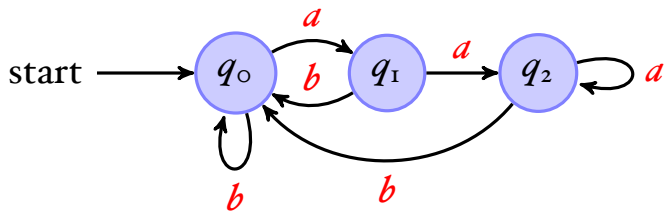


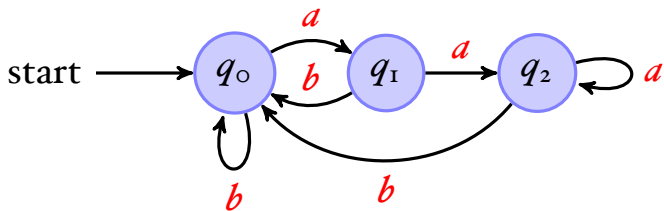


$$q_0 = 2q_0 + 3q_1 + 4q_2$$

$$q_1 = 2q_0 + 3q_1 + 1q_2$$

$$q_2 = 1q_0 + 5q_1 + 2q_2$$

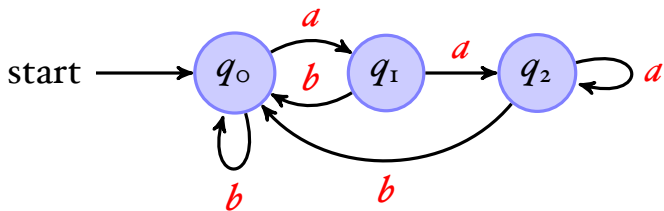




$$q_0 = \epsilon + q_0 b + q_1 b + q_2 b$$

$$q_1 = q_0 a$$

$$q_2 = q_1 a + q_2 a$$



$$q_0 = \epsilon + q_0 b + q_1 b + q_2 b$$

$$q_1 = q_0 a$$

$$q_2 = q_1 a + q_2 a$$

Arden's Lemma:

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

Given the function

$$\text{rev}(\emptyset) \stackrel{\text{def}}{=} \emptyset$$

$$\text{rev}(\epsilon) \stackrel{\text{def}}{=} \epsilon$$

$$\text{rev}(c) \stackrel{\text{def}}{=} c$$

$$\text{rev}(r_1 + r_2) \stackrel{\text{def}}{=} \text{rev}(r_1) + \text{rev}(r_2)$$

$$\text{rev}(r_1 \cdot r_2) \stackrel{\text{def}}{=} \text{rev}(r_2) \cdot \text{rev}(r_1)$$

$$\text{rev}(r^*) \stackrel{\text{def}}{=} \text{rev}(r)^*$$

and the set

$$\text{Rev } A \stackrel{\text{def}}{=} \{s^{-1} \mid s \in A\}$$

prove whether

$$L(\text{rev}(r)) = \text{Rev}(L(r))$$