Regular Expressions with Backreferences and Lookaheads Capture NLOG

Yuya Uezato

July 11, 2024 @ ICALP

CyberAgent, Inc.

Introduction: Modern Regular Expressions

Classical Regular Expressions (REGEX)

Example. $$\begin{aligned} [\![aa\ a^*]\!] &= [\![aa(\epsilon + a + aa + aaa + \cdots)]\!] \\ &= \{a^n : n \geqslant 2\} \end{aligned}$$

Introduction: Modern Regular Expressions

Classical Regular Expressions (REGEX)

$$\text{Example. } [\![aa\ a^*]\!] = [\![aa(\epsilon + a + aa + aaa + \cdots)]\!] = \{a^n : n \geqslant 2\}$$

Modern REGEX = REGEX with Backreferences

$$(\ aa\,a^*\ )_x\quad \backslash x\quad \backslash x\ ^*$$

Introduction: Modern Regular Expressions

Classical Regular Expressions (REGEX)

Example. $$[\![aa\ a^*]\!] = [\![aa(\epsilon + a + aa + aaa + \cdots)]\!] = \{a^n : n \geqslant 2\}$$

Modern REGEX = REGEX with Backreferences

$$\boxed{(}\,aa\,a^*\boxed{)_x}\ \boxed{\backslash x}\ \boxed{\backslash x}^*$$

$\boxed{(}\ E\ \boxed{)_x}$: **Capturing** stores a string matched with E to a variable x

$\boxed{\backslash x}$: **Referring** reads the string stored in x

$(aa\,a^*)_x \setminus x \setminus x^*$ means $\{a^n : n$ is composite$\}$.

$(aa\,a^*)_x \setminus x \setminus x^*$ means $\{a^n : n$ is composite$\}$.

n is composite $\iff \exists i, j \geqslant 2.\ n = i \cdot j$

$(aa\,a^*)_x \setminus x \setminus x^*$ means $\{a^n : n$ is composite$\}$.

n is composite $\iff \exists i, j \geqslant 2.\ n = i \cdot j$

The expression behaves like the "Sieve of Eratosthenes":

$(aa\,a^*)_x \backslash x \backslash x^*$ means $\{a^n : n$ is composite$\}$.

n is composite $\iff \exists i, j \geqslant 2.\ n = i \cdot j$

The expression behaves like the “Sieve of Eratosthenes”:

$$\overbracket{aa}^{x} \backslash x \backslash x^*$$

$$(aa\,a^*)_x \ \ \backslash x \ \ \backslash x^* =$$

$(aa\,a^*)_x \setminus x \setminus x^*$ means $\{a^n : n$ is composite$\}$.

n is composite $\iff \exists i, j \geqslant 2.\ n = i \cdot j$

The expression behaves like the “Sieve of Eratosthenes”:

$$\overbrace{aa}^{x}\,(aa)\,(aa)^* \;(= \{a^{2\cdot j} : j \geqslant 2\})$$

$(aa\,a^*)_x \;\setminus x\; \setminus x^* =$

$(aa\,a^*)_x \setminus x \setminus x^*$ means $\{a^n : n$ is composite$\}$.

n is composite $\iff \exists i, j \geqslant 2.\ n = i \cdot j$

The expression behaves like the “Sieve of Eratosthenes”:

$$
\begin{aligned}
& \ \overset{x}{\overline{aa}}\,(aa)\,(aa)^*\ (= \{a^{2\cdot j} : j \geqslant 2\}) \\
(aa\,a^*)_x \setminus x \setminus x^* = & +\ \overset{x}{\overline{aaa}}\,(aaa)\,(aaa)^*\ (= \{a^{3\cdot j} : j \geqslant 2\})
\end{aligned}
$$

$(aa\,a^*)_x \setminus x \setminus x^*$ means $\{a^n : n$ is composite$\}$.

n is composite $\iff \exists i, j \geqslant 2.\ n = i \cdot j$

The expression behaves like the “Sieve of Eratosthenes”:

$$
\begin{aligned}
(aa\,a^*)_x \ \setminus x \ \setminus x^* = \ & \overbracket{aa}^{x}\,(aa)\,(aa)^* \ (= \{a^{2\cdot j} : j \geqslant 2\}) \\
& + \overbracket{aaa}^{x}\,(aaa)\,(aaa)^* \ (= \{a^{3\cdot j} : j \geqslant 2\}) \\
& + \overbracket{aaaa}^{x}\,(aaaa)\,(aaaa)^* \ (= \left\{a^{4\cdot j} : j \geqslant 2\right\}) \\
& + \cdots
\end{aligned}
$$

$(aa\,a^*)_x \setminus x \setminus x^*$ means $\{a^n : n$ is composite$\}$.

n is composite $\iff \exists i, j \geqslant 2.\ n = i \cdot j$

The expression behaves like the “Sieve of Eratosthenes”:

$$
\begin{aligned}
(aa\,a^*)_x \ \setminus x \ \setminus x^* = \ & \overbrace{aa}^{x}\ (aa)\ (aa)^*\ (= \{a^{2\cdot j} : j \geqslant 2\}) \\
& + \overbrace{aaa}^{x}\ (aaa)\ (aaa)^*\ (= \{a^{3\cdot j} : j \geqslant 2\}) \\
& + \overbrace{aaaa}^{x}\ (aaaa)\ (aaaa)^*\ (= \left\{a^{4\cdot j} : j \geqslant 2\right\}) \\
& + \cdots
\end{aligned}
$$

So, $[\![\, (aaa^*)_x \setminus x \setminus x^* \,]\!] = \{a^n : n$ is composite$\}$.

$(aa\,a^*)_x \setminus x \setminus x^*$ means $\{a^n : n \text{ is composite}\}$.

$$n \text{ is composite} \iff \exists i, j \geqslant 2.\ n = i \cdot j$$

The expression behaves like the “Sieve of Eratosthenes”:

$$
\begin{aligned}
(aa\,a^*)_x \ \setminus x \ \setminus x^* = \ & \overbrace{aa}^{x}\,(aa)\ (aa)^* \ (= \{a^{2\cdot j} : j \geqslant 2\}) \\
& + \overbrace{aaa}^{x}\,(aaa)\ (aaa)^* \ (= \{a^{3\cdot j} : j \geqslant 2\}) \\
& + \overbrace{aaaa}^{x}\,(aaaa)\ (aaaa)^* \ (= \left\{a^{4\cdot j} : j \geqslant 2\right\}) \\
& + \cdots
\end{aligned}
$$

So, $[\![\, (aaa^*)_x \setminus x \setminus x^* \,]\!] = \{a^n : n \text{ is composite}\}$.

Can we represent the prime numbers $\left\{a^2, a^3, a^5, a^7, \ldots\right\}$??

Introduction: Modern Regular Expressions

Classical Regular Expressions (REGEX)

$$\text{Example. } [\![aa\ a^*]\!] = [\![aa(\epsilon + a + aa + aaa + \cdots)]\!]$$
$$= \{a^n : n \geqslant 2\}$$

Modern REGEX = REGEX with Backreferences & Lookaheads

$$(\ aa\ a^*\ )_x \quad \backslash x \quad \backslash x\ ^*$$

$$?(\ aa\ a^*\ \$\ ) \quad !(\ \underbrace{(aa\ a^*)_x\ \backslash x\ \backslash x^*}_{\text{composite checker}}\ \$\ ) \quad a^*$$

Introduction: Modern Regular Expressions

Classical Regular Expressions (REGEX)

$$\text{Example. } [\![aa\ a^*]\!] = [\![aa(\epsilon + a + aa + aaa + \cdots)]\!] = \{a^n : n \geqslant 2\}$$

Modern REGEX = REGEX with Backreferences & Lookaheads

$$(\ aa\ a^*\ )_x \quad \backslash x \quad \backslash x\ ^*$$

$$?(\ aa\ a^*\ \$\ )\ \ !(\ \underbrace{(aa\ a^*)_x\ \backslash x\ \backslash x^*}_{\text{composite checker}}\ \ \$\ )\ \ a^*$$

Introduction: Modern Regular Expressions

Classical Regular Expressions (REGEX)

$$\text{Example. } [\![aa\ a^*]\!] = [\![aa(\epsilon + a + aa + aaa + \cdots)]\!]$$
$$= \{a^n : n \geqslant 2\}$$

Mode… Lookaheads

Positive Lookahead

$?(\ E\ )$: if the input matches with E, then we continue.

$$?(\ aa\ a^*\ \$\ )\ \ !(\ \underbrace{(aa\ a^*)_x\ \backslash x\ \backslash x^*}_{\text{composite checker}}\ \$\ )\ a^*$$

Introduction: Modern Regular Expressions

Classical Regular Expressions (REGEX)

Example. $[\![aa\ a^*]\!] = [\![aa(\epsilon + a + aa + aaa + \cdots)]\!]$

$\{a^n : n \geq 2\}$

Mode

Negative Lookahead

$!(\ E\ )$: if the input **never** matches with E, then we continue.

Posi

$?(\ E\ )$: if the input matches with E, then we continue.

$x\ ^*$

$$?(\ aa\ a^*\ \$\ )\ !(\ \underbrace{(aa\ a^*)_x\ \backslash x\ \backslash x^*}_{\text{composite checker}}\ \$\ )\ a^*$$

Introduction

Coding and experimenting in Python

```python
import re

prime = r'(?!((?P<X>(aaa*))(?P=X)(?P=X)*$))aa(a*)'

for i in range(1, 50):
    w = 'a' * i
    result = re.fullmatch(prime, w)
    print(i, result)
```

slide — -bash ▸ Emacs-arm64-11 — 110×28

```
CA-20023547:slide s22809$ python prime_demo.py
1 None
2 <re.Match object; span=(0, 2), match='aa'>
3 <re.Match object; span=(0, 3), match='aaa'>
4 None
5 <re.Match object; span=(0, 5), match='aaaaa'>
6 None
7 <re.Match object; span=(0, 7), match='aaaaaaa'>
8 None
9 None
10 None
11 <re.Match object; span=(0, 11), match='aaaaaaaaaaa'>
12 None
13 <re.Match object; span=(0, 13), match='aaaaaaaaaaaaa'>
14 None
15 None
16 None
17 <re.Match object; span=(0, 17), match='aaaaaaaaaaaaaaaaa'>
18 None
19 <re.Match object; span=(0, 19), match='aaaaaaaaaaaaaaaaaaa'>
20 None
21 None
```

Main Results on REWBLk

Question: *How expressive are modern REGEXs?*

Main Results on REWBLk

Question: *How expressive are modern REGEXs?*

Answer: We have two new results.

Main Results on REWBLk

Question: *How expressive are modern REGEXs?*

Answer: We have two new results.

In the paper, modern REGEXs are called as REWBLk.
(***R***egular ***E***xpressions ***W***ith ***B***ackreferences and ***L***oo***k***aheads).

Main Results on REWBLK

Question: *How expressive are modern REGEXs?*

Answer: We have two new results.

In the paper, modern REGEXs are called as REWBLK.
(***R***egular ***E***xpressions ***W***ith ***B***ackreferences and ***L***oo***k***aheads).

1. The language class of REWBLK $=$ **NL** (nondeterministic log-space).
   We have the following translation:

$$E : \text{REWBLK} \iff M : \text{nondeterministic log-space TM } s.t.\ [\![E]\!] = L(M)$$

Main Results on REWBLk

Question: *How expressive are modern REGEXs?*

Answer: We have two new results.

In the paper, modern REGEXs are called as REWBLk.
(***R***egular ***E***xpressions ***W***ith ***B***ackreferences and ***L***oo***k***aheads).

1. The language class of REWBLk $=$ **NL** (nondeterministic log-space).

   We have the following translation:

$$E : \text{REWBLk} \iff M : \text{nondeterministic log-space TM } s.t. \ [\![E]\!] = L(M)$$

2. The complexity of REWBLk-membership problem is **PSPACE**-complete.
   - **Input**: a REWBLk expression E & a string w
   - **Output**: True if E accepts w. False otherwise.

Main Results on REWBLk

Question: *How expressive are modern REGEXs?*

An

T

(*R*egula ... eads).

In this talk,
we only focus on 1 REWBLk $=$ **NL**

1 The language class of REWBLk $=$ **NL** (nondeterministic log-space).

We have the following translation:

$$E : \text{REWBLk} \iff M : \text{nondeterministic log-space TM } s.t.\ [\![E]\!] = L(M)$$

2 The complexity of REWBLk-membership problem is **PSPACE**-complete.

- **Input**: a REWBLk expression E & a string w
- **Output**: True if E accepts w. False otherwise.

Outline of this talk

1. Informal overview of REWBLk and results. ✅
2. Formal semantics of REWBLk
3. Idea of the proof of $\mathbf{NL} \subseteq$ REWBLk
4. Idea of the proof of $\mathbf{NL} \supseteq$ REWBLk

Syntax and Semantics of REWBLk

Syntax and Semantics of REWBLk

① *Syntax.* Expressions are inductively defined via the following grammar:

$$
\begin{array}{rcll}
E & ::= & \epsilon \mid \sigma \mid E + E \mid E\,E \mid E^* & \text{(REGEX part)} \\
& \mid & (E)_x \mid \backslash x & \text{(Backreferences part)} \\
& \mid & ?(E) \mid !(E) & \text{(Lookaheads part)}
\end{array}
$$

Syntax and Semantics of REWBLK

① *Syntax.* Expressions are inductively defined via the following grammar:

$$
\begin{array}{llll}
E & ::= & \epsilon \mid \sigma \mid E + E \mid E\,E \mid E^* & \text{(REGEX part)} \\
 & \mid & (E)_x \mid \backslash x & \text{(Backreferences part)} \\
 & \mid & ?(E) \mid !(E) & \text{(Lookaheads part)}
\end{array}
$$

Syntax and Semantics of REWBLK

① *Syntax.* Expressions are inductively defined via the following grammar:

$$
\begin{array}{rcll}
E & ::= & \epsilon \mid \sigma \mid E + E \mid E\,E \mid E^* & \text{(REGEX part)} \\
& \mid & (E)_x \mid \backslash x & \text{(Backreferences part)} \\
& \mid & ?(E) \mid !(E) & \text{(Lookaheads part)}
\end{array}
$$

Syntax and Semantics of REWBLk

① *Syntax.* Expressions are inductively defined via the following grammar:

$$E \quad ::= \quad \epsilon \mid \sigma \mid E + E \mid E\,E \mid E^* \qquad \text{(REGEX part)}$$

$$\mid \quad (E)_x \mid \backslash x \qquad \text{(Backreferences part)}$$

$$\mid \quad ?(E) \mid\ !(E) \qquad \text{(Lookaheads part)}$$

② *Configurations* $\langle \underbrace{p}_{\textit{index}}, \underbrace{\Lambda}_{\textit{variables assign.}} \rangle$. Pairs of an index and an assign.:

$$\begin{array}{lcccccccc} & & & & & \Lambda & & & \\ & & & & & \triangledown & & & \\ \text{input } w & : & a & b & \ldots & a & \ldots & a & \\ \text{index} & : & 0 & 1 & & p & & |w|-1 & |w| \end{array}$$

Syntax and Semantics of REWBLk

① ***Syntax.*** Expressions are inductively defined via the following grammar:

$$
\begin{array}{rcll}
E & ::= & \epsilon \mid \sigma \mid E + E \mid E\,E \mid E^* & \text{(REGEX part)} \\
 & \mid & (E)_x \mid \backslash x & \text{(Backreferences part)} \\
 & \mid & ?(E) \mid !(E) & \text{(Lookaheads part)}
\end{array}
$$

② ***Configurations*** $\langle \underbrace{p}_{\textit{index}}, \underbrace{\Lambda}_{\textit{variables assign.}} \rangle$. Pairs of an index and an assign.:

$$
\begin{array}{lccccccc}
 & & & & \Lambda & & & \\
 & & & & \triangledown & & & \\
\text{input } w & : & a & b & \dots\ a & \dots & a & \\
\text{index} & : & 0 & 1 & p & & |w|-1 & |w|
\end{array}
$$

For a variable x, $\Lambda(x)$ means the stored string in x.

Semantic Function & Language

③ ***Semantics function.*** Let w be an input string.

$$[\![E, \langle \underbrace{p}_{\text{index}}, \underbrace{\Lambda}_{\text{variables assign.}} \rangle]\!]_w = \{ \langle q_1, \Lambda_1 \rangle, \langle q_2, \Lambda_2 \rangle, \ldots \}.$$

Semantic Function & Language

③ ***Semantics function.*** Let w be an input string.

$$[\![E, \langle \underbrace{p}_{\text{index}}, \underbrace{\Lambda}_{\text{variables assign.}} \rangle]\!]_w = \{ \langle q_1, \Lambda_1 \rangle, \langle q_2, \Lambda_2 \rangle, \ldots \}.$$

It means that:

1. From the current configuration $\langle p, \Lambda \rangle$, we execute E;

$$\begin{array}{lcccccccccccccc} & & & & & & \Lambda & & & & & & & & \\ & & & & & & \triangledown & & & & & & & & \\ \text{input string } w & : & a & b & c & b \ldots & a & \ldots\ldots\ldots\ldots & b & \ldots & c & \ldots & b & a & a \\ \text{index} & : & 0 & 1 & 2 & 3 & p & & & & & & & & \end{array}$$

Semantic Function & Language

③ ***Semantics function.*** Let w be an input string.

$$[\![E, \langle \underset{\text{index}}{\underline{p}}, \underset{\text{variables assign.}}{\underline{\Lambda}} \rangle]\!]_w = \{ \langle q_1, \Lambda_1 \rangle, \langle q_2, \Lambda_2 \rangle, \ldots \}.$$

It means that:

1. From the current configuration $\langle p, \Lambda \rangle$, we execute E;
2. then obtain new configurations $\langle q_1, \Lambda_1 \rangle$

$$\begin{array}{lccccccccccccccc}
 & & & & & & & & \text{consumed by } E_{\Lambda_1} & & & & & & & \\
 & & & & & & & & & \triangledown & & & & & & \\
\text{input string } w & : & a & b & c & b & \ldots & a & \ldots\ldots\ldots & b & \ldots & c & \ldots & b & a & a \\
\text{index} & : & 0 & 1 & 2 & 3 & & p & & q_1 & & & & & &
\end{array}$$

Semantic Function & Language

③ ***Semantics function.*** Let w be an input string.

$$[\![E,\ \langle \underset{\text{index}}{\underline{p}},\ \underset{\text{variables assign.}}{\underline{\Lambda}} \rangle]\!]_w = \{\ \langle q_1, \Lambda_1\rangle,\ \langle q_2, \Lambda_2\rangle,\ \dots\ \}.$$

It means that:

1. From the current configuration $\langle p, \Lambda\rangle$, we execute E;
2. then obtain new configurations $\langle q_1, \Lambda_1\rangle$,$\langle q_2, \Lambda_2\rangle$, and so on.

consumed by E

$$\begin{array}{lcllllllll}
 & & & & & & \overset{\Lambda_1}{\triangledown} & & \overset{\Lambda_2}{\triangledown} & \\
\text{input string } w & : & a\ b\ c\ b & \dots & a & \dots\dots\dots\dots & b & \dots & c & \dots\ b\ a\ a \\
\text{index} & : & 0\ 1\ 2\ 3 & & p & & q_1 & & q_2 &
\end{array}$$

Semantic Function & Language

③ *Semantics function.* Let w be an input string.

$$[\![E, \langle \underbrace{p}_{\text{index}}, \underbrace{\Lambda}_{\text{variables assign.}} \rangle]\!]_w = \{ \langle q_1, \Lambda_1 \rangle, \langle q_2, \Lambda_2 \rangle, \dots \}.$$

④ *Acceptance.* E **accepts** w if w can be perfectly consumed by E.

$$\exists \langle p, \Lambda \rangle \in [\![E, \langle 0, \Lambda_\epsilon \rangle]\!]_w.\ p = |w| \qquad \forall x.\, \Lambda_\epsilon(x) = \epsilon$$

Semantic Function & Language

③ ***Semantics function.*** Let w be an input string.

$$[\![E,\ \langle \underbrace{p}_{\text{index}},\ \underbrace{\Lambda}_{\text{variables assign.}} \rangle]\!]_w = \{\ \langle q_1, \Lambda_1 \rangle,\ \langle q_2, \Lambda_2 \rangle,\ \ldots\ \}.$$

④ ***Acceptance.*** E **accepts** w if w can be perfectly consumed by E.

$$\exists \langle p, \Lambda \rangle \in [\![E,\ \langle 0, \Lambda_\epsilon \rangle]\!]_w.\ p = |w| \qquad \forall x.\, \Lambda_\epsilon(x) = \epsilon$$

		Λ_ϵ $\triangledown$					
input string w	:	a	b	c	…………	a	
index	:	0	1	2		$\lvert w\rvert - 1$	$\lvert w\rvert$

Semantic Function & Language

③ ***Semantics function.*** Let w be an input string.

$$[\![E,\ \langle \underbrace{p}_{\text{index}},\ \underbrace{\Lambda}_{\text{variables assign.}} \rangle]\!]_w = \{\ \langle q_1, \Lambda_1\rangle,\ \langle q_2, \Lambda_2\rangle,\ \ldots\ \}.$$

④ ***Acceptance.*** E **accepts** w if w can be perfectly consumed by E.

$$\exists\langle p, \Lambda\rangle \in [\![E,\ \langle 0, \Lambda_\epsilon\rangle]\!]_w.\ p = |w| \qquad \forall x.\, \Lambda_\epsilon(x) = \epsilon$$

consumed by E

Λ

$\triangledown$

input string w	:	a	b	c	$\ldots\ldots\ldots\ldots$	a	
index	:	0	1	2		$\lvert w\rvert - 1$	$\lvert w\rvert$

Semantic Function & Language

③ ***Semantics function.*** Let w be an input string.

$$[\![E,\ \langle \underbrace{p}_{\text{index}},\ \underbrace{\Lambda}_{\text{variables assign.}} \rangle]\!]_w = \{\ \langle q_1, \Lambda_1 \rangle,\ \langle q_2, \Lambda_2 \rangle,\ \dots\ \}.$$

④ ***Acceptance.*** E **accepts** w if w can be perfectly consumed by E.

$$\exists \langle p, \Lambda \rangle \in [\![E,\ \langle 0, \Lambda_\epsilon \rangle]\!]_w.\ p = |w| \qquad \forall x.\ \Lambda_\epsilon(x) = \epsilon$$

consumed by E

Λ

$\triangledown$

input string w	:	a	b	c	$\dots\dots\dots\dots$	a		
index	:	0	1	2		$\lvert w\rvert - 1$	$\lvert w\rvert$

Def. The language of E: $[\![E]\!] = \{w : E \text{ accepts } w\}$.

Semantics of REWBLk: REGEX part

Let w be an input string.

$$[\![\epsilon, \langle p, \Lambda \rangle]\!] = \{ \langle p, \Lambda \rangle \},$$

$$[\![\sigma, \langle p, \Lambda \rangle]\!] = \begin{cases} \{ \langle p+1, \Lambda \rangle \} & \text{if } w[p] = \sigma \text{ and } p+1 \leqslant |w| \\ \emptyset & \text{otherwise} \end{cases}$$

Examples

Semantics of REWBLk: REGEX part

Let w be an input string.

$$
\begin{aligned}
[\![\epsilon,\ \langle p, \Lambda\rangle]\!] &= \{\, \langle p, \Lambda\rangle \,\}, \\
[\![\sigma,\ \langle p, \Lambda\rangle]\!] &= \begin{cases} \{\, \langle p+1, \Lambda\rangle \,\} & \text{if } w[p] = \sigma \text{ and } p+1 \leqslant |w| \\ \emptyset & \text{otherwise} \end{cases}
\end{aligned}
$$

Examples

Semantics of REWBLk: REGEX part

Let w be an input string.

$$
\begin{aligned}
[\![\epsilon, \langle p, \Lambda \rangle]\!] &= \{\, \langle p, \Lambda \rangle \,\}, \\
[\![\sigma, \langle p, \Lambda \rangle]\!] &= \begin{cases} \{\, \langle p+1, \Lambda \rangle \,\} & \text{if } w[p] = \sigma \text{ and } p+1 \leqslant |w| \\ \emptyset & \text{otherwise} \end{cases}
\end{aligned}
$$

Examples

$$
[\![a,\ a \quad b \quad b \quad \overset{\Lambda}{\overset{\triangledown}{a}}\ ]\!] = a \quad b \quad b \quad a \quad \overset{\Lambda}{\triangledown}
$$

Semantics of REWBLk: REGEX part

Let w be an input string.

$$\llbracket \epsilon, \langle p, \Lambda \rangle \rrbracket = \{ \langle p, \Lambda \rangle \},$$

$$\llbracket \sigma, \langle p, \Lambda \rangle \rrbracket = \begin{cases} \{ \langle p+1, \Lambda \rangle \} & \text{if } w[p] = \sigma \text{ and } p+1 \leqslant |w| \\ \emptyset & \text{otherwise} \end{cases}$$

Examples

$$\llbracket b, \; a \quad b \quad b \quad \overset{\Lambda}{\overset{\triangledown}{a}} \; \rrbracket = \emptyset$$

Semantics of REWBLκ: REGEX part

$$[\![E_1 + E_2,\ \langle p, \Lambda\rangle]\!] \ = \ [\![E_1,\ \langle p, \Lambda\rangle]\!] \cup [\![E_2,\ \langle p, \Lambda\rangle]\!]$$

Semantics of REWBLK: REGEX part

$$[\![E_1 + E_2,\ \langle p, \Lambda\rangle]\!] \ = \ [\![E_1,\ \langle p, \Lambda\rangle]\!] \cup [\![E_2,\ \langle p, \Lambda\rangle]\!]$$

$$[\![E_1\ E_2,\ \ \langle p, \Lambda\rangle]\!] \ = \ \bigcup_{\langle q, \Lambda'\rangle \in [\![E_1, \langle p, \Lambda\rangle]\!]} [\![E_2,\ \langle q, \Lambda'\rangle]\!]$$

Semantics of REWBLк: REGEX part

$$
\begin{aligned}
[\![E_1 + E_2,\ \langle p, \Lambda\rangle]\!] &= [\![E_1,\ \langle p, \Lambda\rangle]\!] \cup [\![E_2,\ \langle p, \Lambda\rangle]\!] \\
[\![E_1\ E_2,\ \langle p, \Lambda\rangle]\!] &= \bigcup_{\langle q, \Lambda'\rangle \in [\![E_1, \langle p, \Lambda\rangle]\!]} [\![E_2,\ \langle q, \Lambda'\rangle]\!] \\
[\![E^*,\ \langle p, \Lambda\rangle]\!] &= \bigcup_{i=0}^{\infty} [\![E^i,\ \langle p, \Lambda\rangle]\!] \\
&=
\end{aligned}
$$

Semantics of REWBLK: REGEX part

$$
\begin{aligned}
[\![E_1 + E_2,\ \langle p, \Lambda\rangle]\!] &= [\![E_1,\ \langle p, \Lambda\rangle]\!] \cup [\![E_2,\ \langle p, \Lambda\rangle]\!] \\
[\![E_1\ E_2,\ \langle p, \Lambda\rangle]\!] &= \bigcup_{\langle q, \Lambda'\rangle \in [\![E_1, \langle p, \Lambda\rangle]\!]} [\![E_2,\ \langle q, \Lambda'\rangle]\!] \\
[\![E^*,\ \langle p, \Lambda\rangle]\!] &= \bigcup_{i=0}^{\infty} [\![E^i,\ \langle p, \Lambda\rangle]\!] \\
&= [\![\epsilon,\ \langle p, \Lambda\rangle]\!] \cup [\![E,\ \langle p, \Lambda\rangle]\!] \\
&\cup\ [\![E\ E,\ \langle p, \Lambda\rangle]\!] \cup [\![E\ E\ E,\ \langle p, \Lambda\rangle]\!] \cup \cdots
\end{aligned}
$$

Syntax and Semantics of Backreferences

$(E)_x$: We save a string consumed by applying E to the variable x.

$$[\![(E)_x,\ \langle p, \Lambda\rangle]\!] = \Big\{\langle\, p,\ \Lambda'[x := w[p..q)\,]\,\rangle\ :\ \langle q, \Lambda'\rangle \in [\![E,\ \langle p, \Lambda\rangle]\!]\Big\}$$

$\backslash x$: We use the stored string in x.

$$[\![\backslash x,\ \langle p, \Lambda\rangle]\!] = [\![\Lambda(x), \langle p, \Lambda\rangle]\!]$$

Example.

$$((a+b)^*)_x \,\#\, \backslash x$$

$$\begin{array}{l}\quad\ \Lambda_\epsilon \\ \quad\ \ \triangledown \\ \text{input}: \ a\ \ b\ \ a\ \ a\ \ \#\ \ a\ \ b\ \ a\ \ a\end{array}$$

Syntax and Semantics of Backreferences

$(E)_x$: We save a string consumed by applying E to the variable x.

$$[\![(E)_x,\ \langle p, \Lambda \rangle]\!] = \left\{ \langle\, p,\ \Lambda'[x := w[p..q)\,]\,\rangle \ :\ \langle q, \Lambda' \rangle \in [\![E,\ \langle p, \Lambda \rangle]\!] \right\}$$

$\backslash x$: We use the stored string in x.

$$[\![\backslash x,\ \langle p, \Lambda \rangle]\!] = [\![\Lambda(x), \langle p, \Lambda \rangle]\!]$$

Example.

$$((a+b)^*)_x \,\#\, \backslash x$$

$$x \mapsto abaa$$

$$\text{input}: \quad \overbrace{a \quad b \quad a \quad a}^{\text{consumed}} \quad \overset{\triangledown}{\#} \quad a \quad b \quad a \quad a$$

Syntax and Semantics of Backreferences

$(E)_x$: We save a string consumed by applying E to the variable x.

$$[\![(E)_x,\ \langle p, \Lambda\rangle]\!] = \left\{\langle\, p,\ \Lambda'[x := w[p..q)\,]\,\rangle \ :\ \langle q, \Lambda'\rangle \in [\![E,\ \langle p, \Lambda\rangle]\!]\right\}$$

$\backslash x$: We use the stored string in x.

$$[\![\backslash x,\ \langle p, \Lambda\rangle]\!] = [\![\Lambda(x), \langle p, \Lambda\rangle]\!]$$

Example.

$$((a+b)^*)_x \ \# \ \backslash x$$

$$x \mapsto abaa$$

$$\triangledown$$

input : $a \quad b \quad a \quad a \quad \# \quad a \quad b \quad a \quad a$

Syntax and Semantics of Backreferences

$(E)_x$: We save a string consumed by applying E to the variable x.

$$[\![(E)_x,\ \langle p, \Lambda\rangle]\!] = \left\{ \langle\, p,\ \Lambda'[x := w[p..q)\,]\,\rangle\ :\ \langle q, \Lambda'\rangle \in [\![E,\ \langle p, \Lambda\rangle]\!] \right\}$$

$\backslash x$: We use the stored string in x.

$$[\![\backslash x,\ \langle p, \Lambda\rangle]\!] = [\![\Lambda(x), \langle p, \Lambda\rangle]\!]$$

Example.

$$((a+b)^*)_x \,\#\, \backslash x$$

$$x \mapsto abaa$$

$$\text{input}:\ \ a\ \ b\ \ a\ \ a\ \ \#\ \ \overbrace{a\ \ b\ \ a\ \ a}^{\text{consumed by } \backslash x}\ \ \triangledown$$

Syntax and Semantics of Backreferences

$(E)_x$: We save a string consumed by applying E to the variable x.

$$[\![(E)_x,\ \langle p, \Lambda \rangle]\!] = \left\{ \langle\, p,\ \Lambda'[x := w[p..q)\,]\,\rangle\ :\ \langle q, \Lambda' \rangle \in [\![E,\ \langle p, \Lambda \rangle]\!] \right\}$$

$\backslash x$: We use the stored string in x.

$$[\![\backslash x,\ \langle p, \Lambda \rangle]\!] = [\![\Lambda(x), \langle p, \Lambda \rangle]\!]$$

Example.

$$((a+b)^*)_x \,\#\, \backslash x$$

$$x \mapsto abaa$$

$$\text{input}: \quad a \quad b \quad a \quad a \quad \# \quad \overbrace{a \quad b \quad a \quad a}^{\text{consumed by } \backslash x} \quad \triangledown$$

So, $((a+b)^*)_x \,\#\, \backslash x$ accepts a non-CFL $\{w \,\#\, w : w \in (a+b)^*\}$.

Syntax and Semantics of Lookaheads

Negative Lookaheads. We continue if the computation of E fails:

$$[\![!(E),\ \langle p, \Lambda\rangle]\!] = \begin{cases} \langle p,\ \Lambda\rangle & \text{if } [\![E,\ \langle p, \Lambda\rangle]\!] = \emptyset, \\ \emptyset & \text{otherwise} \end{cases}$$

Syntax and Semantics of Lookaheads

Negative Lookaheads. We continue if the computation of E fails:

$$[\![!(E),\ \langle p, \Lambda\rangle]\!] = \begin{cases} \langle p,\ \Lambda\rangle & \text{if } [\![E,\ \langle p, \Lambda\rangle]\!] = \emptyset, \\ \emptyset & \text{otherwise} \end{cases}$$

Example: Implementing End-of-String checker \$

In REGEX libraries, the special symbol \$ checks if we are in the EOS pos.

Syntax and Semantics of Lookaheads

Negative Lookaheads. We continue if the computation of E fails:

$$[\![!(E),\ \langle p, \Lambda\rangle]\!] = \begin{cases} \langle p,\ \Lambda\rangle & \text{if } [\![E,\ \langle p, \Lambda\rangle]\!] = \emptyset, \\ \emptyset & \text{otherwise} \end{cases}$$

Example: Implementing End-of-String checker \$

In REGEX libraries, the special symbol \$ checks if we are in the EOS pos.

\$ is syntactic sugar for $!(\Sigma)$:

Syntax and Semantics of Lookaheads

Negative Lookaheads. We continue if the computation of E fails:

$$[\![!(E),\ \langle p, \Lambda\rangle]\!] = \begin{cases} \langle p,\ \Lambda\rangle & \text{if } [\![E,\ \langle p, \Lambda\rangle]\!] = \emptyset, \\ \emptyset & \text{otherwise} \end{cases}$$

Example: Implementing End-of-String checker \$

In REGEX libraries, the special symbol \$ checks if we are in the EOS pos.

\$ is syntactic sugar for $!(\Sigma)$:

$$[\![\Sigma,\ \overset{\triangledown}{a}\ \ b\ \ a\ \ \ ]\!] = a\ \ \overset{\triangledown}{b}\ \ a$$

Syntax and Semantics of Lookaheads

Negative Lookaheads. We continue if the computation of E fails:

$$[\![!(E),\ \langle p, \Lambda\rangle]\!] = \begin{cases} \langle p,\ \Lambda\rangle & \text{if } [\![E,\ \langle p, \Lambda\rangle]\!] = \emptyset, \\ \emptyset & \text{otherwise} \end{cases}$$

Example: Implementing End-of-String checker \$

In REGEX libraries, the special symbol \$ checks if we are in the EOS pos.

\$ is syntactic sugar for $!(\Sigma)$:

$$[\![\Sigma,\ \overset{\triangledown}{a} \quad b \quad a \qquad]\!] = a \quad \overset{\triangledown}{b} \quad a \implies [\![!(\Sigma),\ \overset{\triangledown}{a} \quad b \quad a \qquad]\!] = \emptyset$$

Syntax and Semantics of Lookaheads

Negative Lookaheads. We continue if the computation of E fails:

$$\llbracket !(E),\ \langle p, \Lambda\rangle \rrbracket = \begin{cases} \langle p,\ \Lambda\rangle & \text{if } \llbracket E,\ \langle p, \Lambda\rangle \rrbracket = \emptyset, \\ \emptyset & \text{otherwise} \end{cases}$$

Example: Implementing End-of-String checker \$

In REGEX libraries, the special symbol \$ checks if we are in the EOS pos.

\$ is syntactic sugar for $!(\Sigma)$:

$$\llbracket \Sigma,\ \overset{\triangledown}{a}\ \ b\ \ a\ \ \rrbracket = a\ \ \overset{\triangledown}{b}\ \ a \implies \llbracket !(\Sigma),\ \overset{\triangledown}{a}\ \ b\ \ a\ \ \rrbracket = \emptyset$$

$$\llbracket \Sigma,\ a\ \ b\ \ a\ \ \overset{\triangledown}{\ }\ \rrbracket = \emptyset$$

Syntax and Semantics of Lookaheads

Negative Lookaheads. We continue if the computation of E fails:

$$[\![!(E),\ \langle p, \Lambda\rangle]\!] = \begin{cases} \langle p,\ \Lambda\rangle & \text{if } [\![E,\ \langle p, \Lambda\rangle]\!] = \emptyset, \\ \emptyset & \text{otherwise} \end{cases}$$

Example: Implementing End-of-String checker \$

In REGEX libraries, the special symbol \$ checks if we are in the EOS pos.

\$ is syntactic sugar for $!(\Sigma)$:

$$[\![\Sigma,\ \overset{\triangledown}{a}\ \ b\ \ a\ \ ]\!] = a\ \ \overset{\triangledown}{b}\ \ a \implies [\![!(\Sigma),\ \overset{\triangledown}{a}\ \ b\ \ a\ \ ]\!] = \emptyset$$

$$[\![\Sigma,\ a\ \ b\ \ a\ \ \overset{\triangledown}{\ }]\!] = \emptyset \implies [\![!(\Sigma),\ a\ \ b\ \ a\ \ \overset{\triangledown}{\ }]\!] = a\ \ b\ \ a\ \ \overset{\triangledown}{\ }$$

Syntax and Semantics of Lookaheads

Negative Lookaheads. We continue if the computation of E fails:

$$[\![!(E),\ \langle p, \Lambda \rangle]\!] = \begin{cases} \langle p,\ \Lambda \rangle & \text{if } [\![E,\ \langle p, \Lambda \rangle]\!] = \emptyset, \\ \emptyset & \text{otherwise} \end{cases}$$

Example: Implementing End-of-String checker \$

In REGEX libraries, the special symbol \$ checks if we are in the EOS pos.

\$ is syntactic sugar for $!(\Sigma)$.

Proposition: Closed under complementation

Let E be an expression. $!(E\,\$)\Sigma^*$ accepts the complement of $[\![E]\!]$.

Syntax and Semantics of Lookaheads

Positive Lookaheads

$$[\![?(E),\ \langle p,\Lambda\rangle]\!] = \left\{\langle p,\Lambda'\rangle : \langle q,\Lambda'\rangle \in [\![E,\ \langle p,\Lambda\rangle]\!]_w\right\}.$$

Example

$$?\big((a^*)_x\big)$$

$$\begin{array}{lcl} & & \qquad\qquad\quad\ \ \Lambda \\ & & \qquad\qquad\quad\ \ \triangledown \\ \text{input string } w & : & a\ b\ c\ b\ \ldots\ a\ \ a\ a\ a\ \ b\ \ \ldots\ldots \\ \text{index} & : & 0\ 1\ 2\ 3\ \ \ \ \ \ \ \ p \end{array}$$

Syntax and Semantics of Lookaheads

Positive Lookaheads

$$[\![?(E),\ \langle p, \Lambda\rangle]\!] = \Big\{\langle p, \Lambda'\rangle : \langle q, \Lambda'\rangle \in [\![E,\ \langle p, \Lambda\rangle]\!]_w\Big\}.$$

1. We apply E from the current position;

Example

$$?\big((a^*)_x\big)$$

$$x \mapsto aaaa$$

consumed

$$\begin{array}{lllllll} \text{input string } w & : & a\ b\ c\ b & \ldots & \overbrace{a\ \ a\ a\ a}^{} & \overset{\triangledown}{b} & \ldots\ldots \\ \text{index} & : & 0\ 1\ 2\ 3 & & p & q & \end{array}$$

Syntax and Semantics of Lookaheads

Positive Lookaheads

$$[\![?(E),\ \langle p,\Lambda\rangle]\!] = \left\{\langle p,\Lambda'\rangle : \langle q,\Lambda'\rangle \in [\![E,\ \langle p,\Lambda\rangle]\!]_w\right\}.$$

1. We apply E from the current position;
2. and go back to the invocation point with *inheriting* assignments.

Example

$$?\big((a^*)_x\big)$$

$$x \mapsto aaaa$$

input string w	:	$a\ b\ c\ b\ \ldots\ \overset{\triangledown}{a}\ \ a\ a\ a\ \ b\ \ \ldots\ldots$
index	:	$0\ 1\ 2\ 3\qquad\quad p$

Proving「NL $\subseteq$ REWBLᴋ」: Translating NL-machines to REWBLᴋ

Proving NL $\subseteq$ REWBLK

Our transformation consists of two steps:

M : nondeterministic log-space machine

$\downarrow$

$\mathcal{A}$: two-way multihead finite automaton

$\downarrow$

E : REWBLK

Proving NL $\subseteq$ REWBLK

Our transformation consists of two steps:

M : nondeterministic log-space machine

This translation comes from the following result:

*The language class of two-way multihead automata equals to **NL**.*

📘 *On Non-Determinancy in Simple Computing Devices*, J. Hartmanis, 1971.

$\mathcal{A}$: two-way multihead finite automaton

E : REWBLK

Proving NL $\subseteq$ REWBLK

Our transformation consists of two steps:

M : nondeterministic log-space machine

This translation comes from the following result:

*The language class of two-way multihead automata equals to **NL**.*

📘 *On Non-Determinancy in Simple Computing Devices*, J. Hartmanis, 1971.

$\mathcal{A}$: two-way multihead finite automaton

We only need to focus on this translation.

E : REWBLK

Proving NL $\subseteq$ REWBLk

Our transformation consists of two steps:

M : nondeterministic log-space machine

$\downarrow$

This translation comes from the following result:

The language class of two-way multihead automata equals to ***NL****.*

📘 *On Non-Determinancy in Simple Computing Devices*, J. Hartmanis, 1971.

$\mathcal{A}$: two-way multihead finite automaton

$\downarrow$

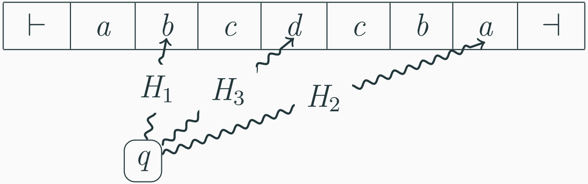
We only need to focus on this translation.

Simulating *heads* is easier than simulating log-space tapes.

E : REWBLk

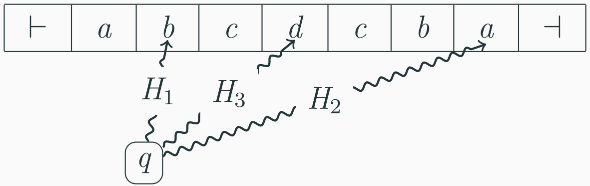
Quick and informal review of two-way multihead finite automata

The following is a configuration of a **3-head** two-way finite automaton:



Quick and informal review of two-way multihead finite automata

The following is a configuration of a **3-head** two-way finite automaton:

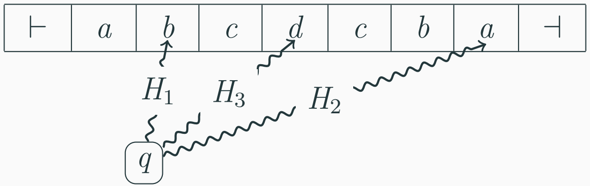


Generally, on every k-head two-way finite automaton,

- the input word is surrounded by end-markers $\vdash \cdots \dashv$.

Quick and informal review of two-way multihead finite automata

The following is a configuration of a **3-head** two-way finite automaton:

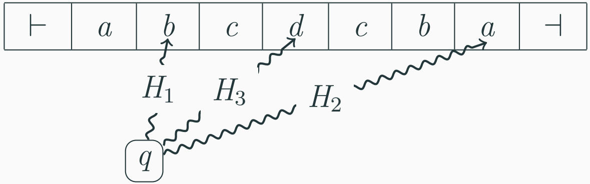


Generally, on every k-head two-way finite automaton,

- the input word is surrounded by end-markers $\vdash \cdots \dashv$.
- there are finitely many states;

Quick and informal review of two-way multihead finite automata

The following is a configuration of a **3-head** two-way finite automaton:

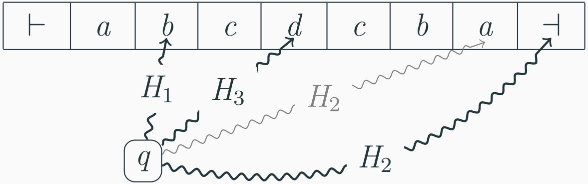


Generally, on every k-head two-way finite automaton,

- the input word is surrounded by end-markers $\vdash \cdots \dashv$.
- there are finitely many states;
- there are k-heads H_1, H_2, $\ldots$ H_k;

Quick and informal review of two-way multihead finite automata

The following is a configuration of a **3-head** two-way finite automaton:

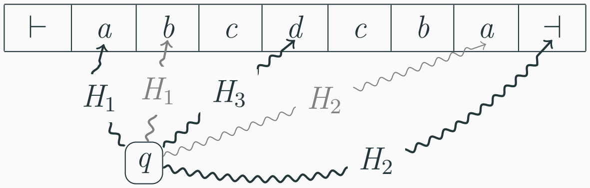


Generally, on every k-head two-way finite automaton,

- the input word is surrounded by end-markers $\vdash \cdots \dashv$.
- there are finitely many states;
- there are k-heads H_1, H_2, $\ldots$ H_k;
- as usual automata, each head can move right; and

Quick and informal review of two-way multihead finite automata

The following is a configuration of a **3-head** two-way finite automaton:



Generally, on every k-head two-way finite automaton,

- the input word is surrounded by end-markers $\vdash \cdots \dashv$.
- there are finitely many states;
- there are k-heads H_1, H_2, … H_k;
- as usual automata, each head can move right; and
- each head can also move left.

Proving NL $\subseteq$ REWBLK

As example, we try to accept the following typical CFL with REWBLK:

$$L_{\text{reverse}} = \left\{ w\#\, w^R : w \in \Sigma^* \right\}$$

w^R is the reverse of w.

For example, $a\, b\, c\# c\, b\, a \in L_{\text{reverse}}$.

Proving NL $\subseteq$ REWBLK

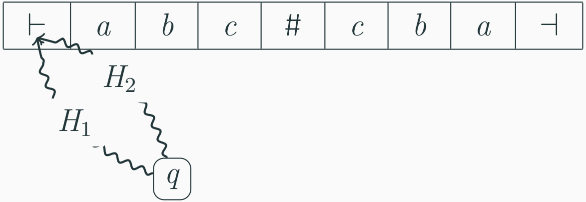
As example, we try to accept the following typical CFL with REWBLK:

$$L_{\text{reverse}} = \{\, w\#\,w^R : w \in \Sigma^* \,\}$$

w^R is the reverse of w.

For example, $a\,b\,c\#\,c\,b\,a \in L_{\text{reverse}}$.

We use a 2-head two-way finite automaton as follows:



initial configuration

Proving NL $\subseteq$ REWBLK

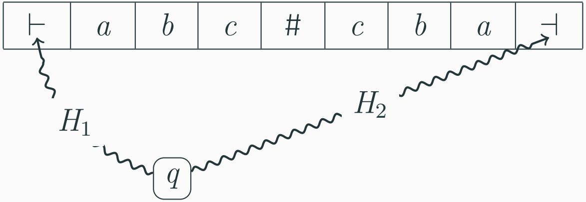
As example, we try to accept the following typical CFL with REWBLK:

$$L_{\text{reverse}} = \{ w \# w^R : w \in \Sigma^* \}$$

w^R is the reverse of w.

For example, $a\,b\,c\#\,c\,b\,a \in L_{\text{reverse}}$.

We use a 2-head two-way finite automaton as follows:



$\boxed{\text{move } H_2 \text{ to } \dashv \text{ while checking that } \# \text{ appears only once.}}$

Proving NL $\subseteq$ REWBLK

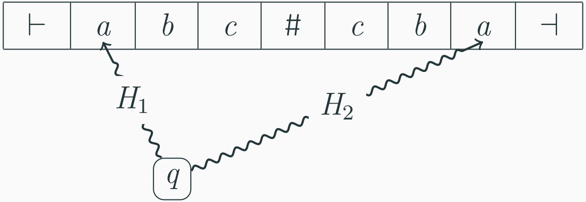
As example, we try to accept the following typical CFL with REWBLK:

$$L_{\text{reverse}} = \{ w \# w^R : w \in \Sigma^* \}$$

w^R is the reverse of w.

For example, $a\,b\,c \# c\,b\,a \in L_{\text{reverse}}$.

We use a 2-head two-way finite automaton as follows:



move heads toward the center while they read the same symbol.

Proving NL $\subseteq$ REWBLK

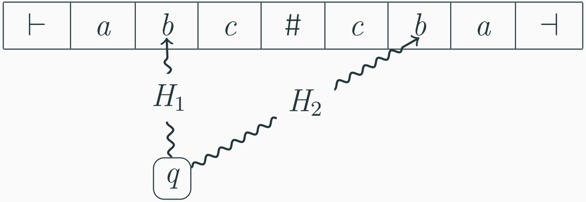
As example, we try to accept the following typical CFL with REWBLK:

$$L_{\text{reverse}} = \{ w \# w^R : w \in \Sigma^* \}$$

w^R is the reverse of w.

For example, $a\,b\,c\,\#\,c\,b\,a \in L_{\text{reverse}}$.

We use a 2-head two-way finite automaton as follows:



move heads toward the center while they read the same symbol.

Proving NL $\subseteq$ REWBLK

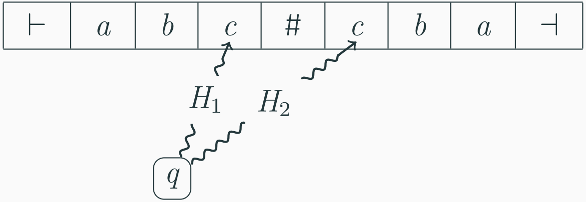
As example, we try to accept the following typical CFL with REWBLK:

$$L_{\text{reverse}} = \left\{ w \,\#\, w^R : w \in \Sigma^* \right\}$$

w^R is the reverse of w.

For example, $a\,b\,c\,\#\,c\,b\,a \in L_{\text{reverse}}$.

We use a 2-head two-way finite automaton as follows:



move heads toward the center while they read the same symbol.

Proving NL $\subseteq$ REWBLk

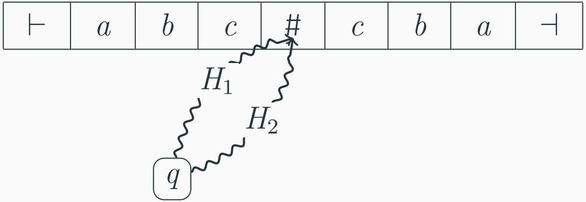
As example, we try to accept the following typical CFL with REWBLk:

$$L_{\text{reverse}} = \{ w\# w^R : w \in \Sigma^* \}$$

w^R is the reverse of w.

For example, $a\,b\,c\#\,c\,b\,a \in L_{\text{reverse}}$.

We use a 2-head two-way finite automaton as follows:



if the heads reach # at the same time, we accept the input.

Proving NL $\subseteq$ REWBLK

$$L_{\text{reverse}} = \left\{ w \# w^R : w \in \Sigma^* \right\}$$

w^R is the reverse of w.

To simulate two heads H_1 and H_2, we use two variables V_1 and V_2.

Proving NL $\subseteq$ REWBLK

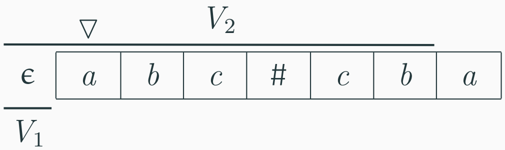
$$L_{\text{reverse}} = \left\{ w\#\, w^R : w \in \Sigma^* \right\}$$

w^R is the reverse of w.

To simulate two heads H_1 and H_2, we use two variables V_1 and V_2.

① We initialize V_1 and V_2 by:

$$?\big((\epsilon)_{V_1}\big) \qquad ?\big((\Sigma^*)_{V_2}\ \Sigma\ \$\big)$$



Proving NL $\subseteq$ REWBLK

$$L_{\text{reverse}} = \left\{ w \# w^R : w \in \Sigma^* \right\}$$

w^R is the reverse of w.

To simulate two heads H_1 and H_2, we use two variables V_1 and V_2.

② Check V_1 and V_2 scan the same symbol by

$$\bigcup_{\sigma \in \Sigma} ?(\backslash V_1\ \color{red}{\sigma}\color{black})\ ?(\backslash V_2\ \color{blue}{\sigma}\color{black})$$

$\triangledown$ V_2

ϵ	$\color{red}{a}$	b	c	$\#$	c	b	$\color{blue}{a}$

V_1

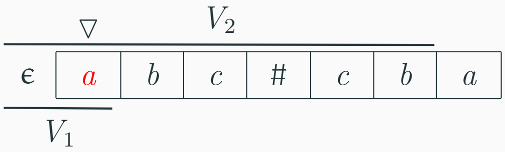
Proving NL $\subseteq$ REWBLk

$$L_{\text{reverse}} = \{ w\#w^R : w \in \Sigma^* \} \qquad w^R \text{ is the reverse of } w.$$

To simulate two heads H_1 and H_2, we use two variables V_1 and V_2.

③ Change/Move V_1 and V_2. First, we expand V_1. It is easy:

$$?\big((\backslash V_1\ \Sigma)_{V_1}\big)$$



Proving NL $\subseteq$ REWBLk

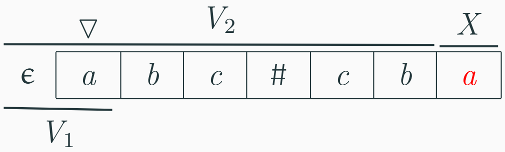
$$L_{\text{reverse}} = \left\{ w\#\, w^R : w \in \Sigma^* \right\}$$

w^R is the reverse of w.

To simulate two heads H_1 and H_2, we use two variables V_1 and V_2.

④ We shrink V_2 in two steps. First, find the complement string X:

$$?\left(\backslash V_2\ (\Sigma^*)_X\ \$\right)$$



Proving NL $\subseteq$ REWBLκ

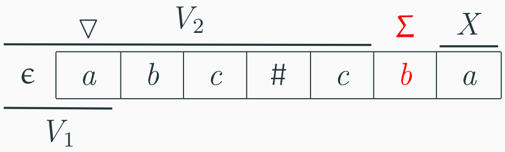
$$L_{\text{reverse}} = \left\{ w \# w^R : w \in \Sigma^* \right\}$$

w^R is the reverse of w.

To simulate two heads H_1 and H_2, we use two variables V_1 and V_2.

④ Using the complement string X, we shrink V_2:

$$?\Big((\Sigma^*)_{V_2} \ ({\color{red}\Sigma} \setminus X) \ \$ \Big)$$



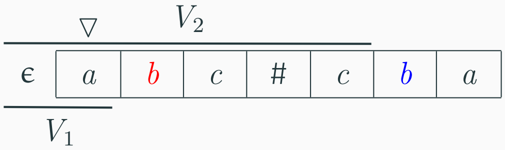
Proving NL $\subseteq$ REWBLK

$$L_{\text{reverse}} = \{ w\# w^R : w \in \Sigma^* \} \qquad w^R \text{ is the reverse of } w.$$

To simulate two heads H_1 and H_2, we use two variables V_1 and V_2.

⑤ We repeat the same. Check both the variables scan the same symbol:

$$\bigcup_{\sigma \in \Sigma} ?(\backslash V_1 \sigma)\ ?(\backslash V_2 \sigma)$$



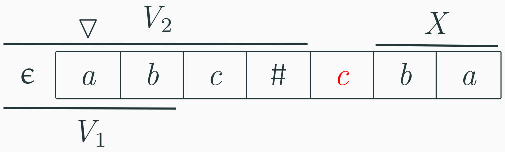
$$L_{\text{reverse}} = \{ w\#\, w^R : w \in \Sigma^* \}$$

w^R is the reverse of w.

To simulate two heads H_1 and H_2, we use two variables V_1 and V_2.

⑥ We again expand V_1 and shrink V_2 by

$$?((\backslash V_1\ \Sigma)_{V_1})\ \ ?(\backslash V_2\ (\Sigma^*)_X\ \$)\ \ ?((\Sigma^*)_{V_2}\ (\Sigma \setminus X)\ \$)$$



Proving NL $\subseteq$ REWBLk

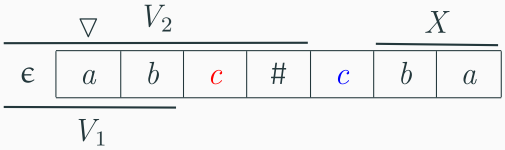
$$L_{\text{reverse}} = \{ w\# w^R : w \in \Sigma^* \}$$

w^R is the reverse of w.

To simulate two heads H_1 and H_2, we use two variables V_1 and V_2.

⑦ We again check both the variables scan the same symbol by

$$\bigcup_{\sigma \in \Sigma} ?(\backslash V_1\ \sigma)\ ?(\backslash V_2\ \sigma)$$



Proving NL ⊆ REWBLK

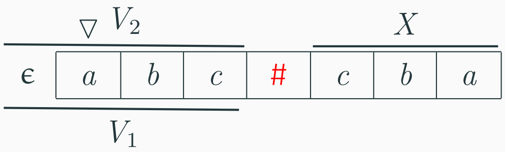
$$L_{\text{reverse}} = \left\{ w \# w^R : w \in \Sigma^* \right\}$$

w^R is the reverse of w.

To simulate two heads H_1 and H_2, we use two variables V_1 and V_2.

⑧ We again expand V_1 and shrink V_2.

$$?\big((\backslash V_1\ \Sigma)_{V_1}\big)\ \ ?\big(\backslash V_2\ (\Sigma^*)_X\ \$\big)\ \ ?\big((\Sigma^*)_{V_2}\ (\Sigma\ \backslash X)\ \$\big)$$



Proving NL $\subseteq$ REWBLK

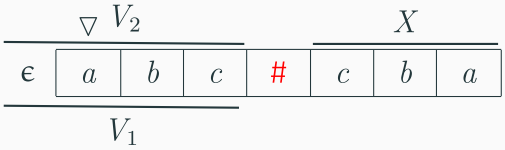
$$L_{\text{reverse}} = \{ w \# w^R : w \in \Sigma^* \}$$

w^R is the reverse of w.

To simulate two heads H_1 and H_2, we use two variables V_1 and V_2.

⑧ We again expand V_1 and shrink V_2.

$$?((\backslash V_1\ \Sigma)_{V_1})\ \ ?(\backslash V_2\ (\Sigma^*)_X\ \$)\ \ ?((\Sigma^*)_{V_2}\ (\Sigma\ \backslash X)\ \$)$$



On the basis of this simulation idea, we have「**Thm. NL** $\subseteq$ REWBLK.」

Proving REWBLk $\subseteq$ NL: Translating REWBLk to NL machines

Proving REWBLk $\subseteq$ NL

Basically, we generalize $\underbrace{\text{the classical regex-to-automaton translation}}_{\text{McNaughton–Yamada–Thompson construction}}$.

Proving REWBLK $\subseteq$ NL

Basically, we generalize $\underbrace{\text{the classical regex-to-automaton translation}}_{\text{McNaughton–Yamada–Thompson construction}}$.

For example, from the following expression,

$$E_{\mathsf{STconn}} = (\,V^*)_C\ ?(\#)\ \Big(?\big(\Sigma^*\,\#\,\backslash C \to (\Sigma^*)_C\,\#\big)\Big)^*\ \Sigma^*\ \#\ \backslash C$$

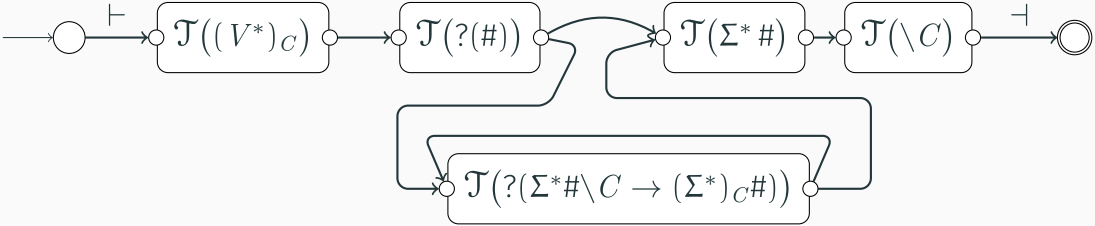
Proving REWBLK $\subseteq$ NL

Basically, we generalize $\underbrace{\text{the classical regex-to-automaton translation}}_{\text{McNaughton–Yamada–Thompson construction}}$.

For example, from the following expression,

$$E_{\mathsf{STconn}} = (V^*)_C\ ?(\#)\ \left(?(\Sigma^*\#\backslash C \to (\Sigma^*)_C\#)\right)^*\ \Sigma^*\ \#\ \backslash C$$

we obtain the following nlog-space machine $\mathcal{T}(E_{\mathsf{STconn}})$:
(note: inputs of machines are surrounded by $\vdash$ and $\dashv$)



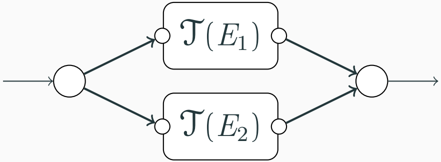
Translation rules for REGEX part

REWBLK : nlog-space Turing machines

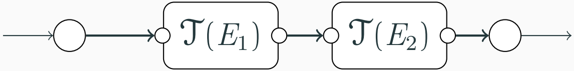
$\mathcal{T}(\epsilon) =$

$\mathcal{T}(\sigma) =$

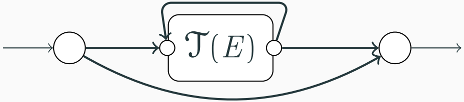
$\mathcal{T}(E_1 + E_2) =$



$\mathcal{T}(E_1\ E_2) \Rrightarrow$

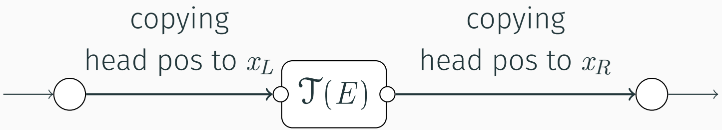


$(E)^* \Rrightarrow$



Translation rules for $(E)_x$ and $\backslash x$

$\mathcal{T}((E)_x) \quad =$

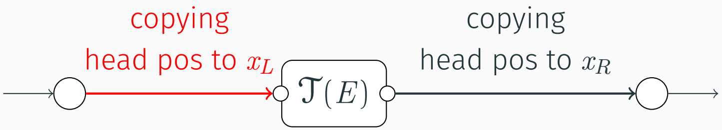


input tape	:	$\vdash \; a \; b \; c \; b \; \ldots \; \overset{\triangledown}{a} \; \ldots\ldots\ldots\ldots \; b \; \ldots \; c \; b \; a \; a \; \dashv$
pos	:	N
working tape x_L	:	
working tape x_R	:	

⚠️ We need to normalize and remove patterns like $(\cdots x \cdots)_x$, $(\cdots(\cdots)_x \cdots)_x$.

Translation rules for $(E)_x$ and $\backslash x$

$\mathcal{T}((E)_x) \quad =$

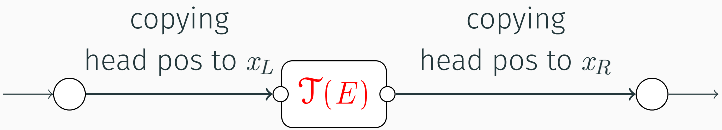


input tape	:	$\vdash \; a\; b\; c\; b\; \ldots\; \overset{\triangledown}{a}\; \ldots\ldots\ldots\ldots\; b\; \ldots\; c\; b\; a\; a \dashv$
pos	:	N
working tape x_L	:	binary encoding of N
working tape x_R	:	

⚠️ We need to normalize and remove patterns like $(\cdots x \cdots)_x$, $(\cdots (\cdots)_x \cdots)_x$.

Translation rules for $(E)_x$ and $\backslash x$

$\mathcal{T}((E)_x) \quad =$



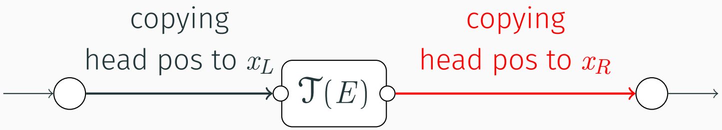
consumed by E

input tape	:	$\vdash \; a \; b \; c \; b \; \ldots \; \overbrace{a \; \ldots\ldots\ldots\ldots \; b}^{}$ $\; \ldots \; c \; b \; a \; a \; \dashv$
pos	:	N ... M
working tape x_L	:	binary encoding of N
working tape x_R	:	

⚠️ We need to normalize and remove patterns like $(\cdots x \cdots)_x$, $(\cdots(\cdots)_x \cdots)_x$.

Translation rules for $(E)_x$ and $\backslash x$

$\mathcal{T}((E)_x) \quad =$



consumed by E

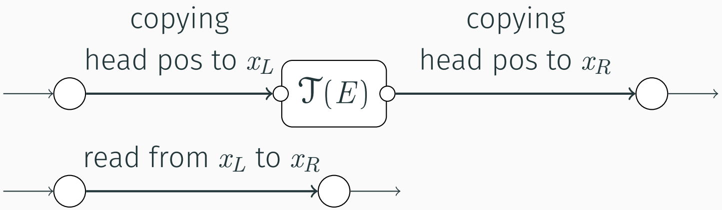
input tape	:	$\vdash \; a \; b \; c \; b \; \ldots \; \overbrace{a \; \ldots\ldots\ldots\ldots \; b}^{} \; \ldots \; c \; b \; a \; a \; \dashv$
pos	:	$N \qquad M$
working tape x_L	:	binary encoding of N
working tape x_R	:	binary encoding of M

⚠️ We need to normalize and remove patterns like $(\cdots x \cdots)_x$, $(\cdots(\cdots)_x \cdots)_x$.

Translation rules for $(E)_x$ and $\backslash x$

$\mathcal{T}((E)_x) \quad = $

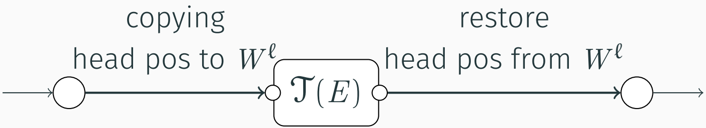
$\mathcal{T}(\backslash x) \quad = $



input tape	:	$\vdash \; a \; b \; c \; b \; \ldots \; a \; \overbrace{\ldots\ldots\ldots\ldots}^{\text{consumed by } E} \; \overset{\triangledown}{b} \; \ldots \; c \; b \; a \; a \; \dashv$
pos	:	$N \qquad M$
working tape x_L	:	binary encoding of N
working tape x_R	:	binary encoding of M

⚠️ We need to normalize and remove patterns like $(\cdots x \cdots)_x$, $(\cdots(\cdots)_x \cdots)_x$.

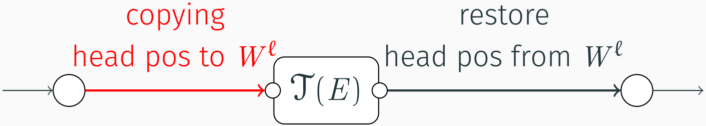
Translation rules for $?(E)$ and $!(E)$

$\mathcal{T}(?(E)^\ell) \quad =$ 

		$\triangledown$
input tape	:	$\vdash \; a \; b \; c \; b \; \ldots \; a \; \ldots\ldots\ldots\ldots \; b \; \ldots \; c \; b \; a \; a \; \dashv$
pos	:	N
working tape W^ℓ	:	

Translation rules for $?(E)$ and $!(E)$

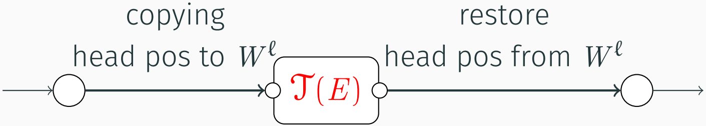
$\mathcal{T}(?(E)^{\ell}) \;=$



		$\triangledown$
input tape	:	$\vdash\ a\ b\ c\ b\ \ldots\ a\ \ldots\ldots\ldots\ldots\ b\ \ldots\ c\ b\ a\ a\ \dashv$
pos	:	N
working tape W^{ℓ}	:	binary encoding of N

Translation rules for $?(E)$ and $!(E)$

$$\mathcal{T}\left(?(E)^{\ell}\right) \quad =$$

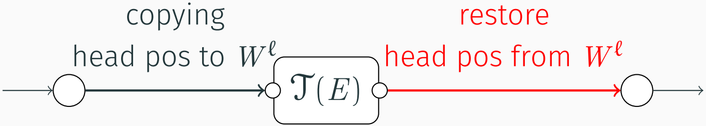


consumed by E

input tape	:	$\vdash \; a \; b \; c \; b \; \ldots \; a \; \ldots\ldots\ldots\ldots \; b \; \ldots \; c \; b \; a \; a \; \dashv$
pos	:	$N \qquad\qquad M$
working tape W^{ℓ}	:	binary encoding of N

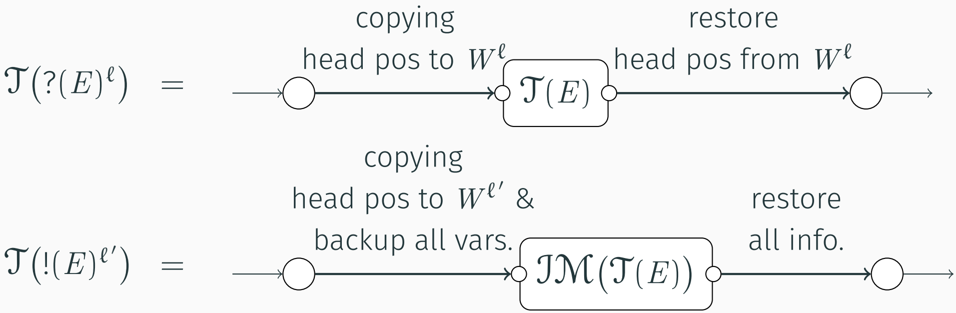
Translation rules for $?(E)$ and $!(E)$

$\mathcal{T}(?(E)^\ell) \quad =$



input tape	:	$\vdash \; a\; b\; c\; b \;\ldots\; \overset{\triangledown}{a} \;\ldots\ldots\ldots\ldots\; b \;\ldots\; c\; b\; a\; a \;\dashv$
pos	:	N (at a), M (at b)
working tape W^ℓ	:	binary encoding of N

Translation rules for $?(E)$ and $!(E)$



input tape	:	$\vdash\ a\ b\ c\ b\ \ldots\ \overset{\triangledown}{a}\ \ldots\ldots\ldots\ldots\ b\ \ldots\ c\ b\ a\ a\ \dashv$
pos	:	N (at a), M (at b)
working tape W^{ℓ}	:	binary encoding of N

Translation rules for $?(E)$ and $!(E)$

$$\mathcal{T}(?(E)^{\ell}) \quad = \quad \longrightarrow \bigcirc \xrightarrow[]{\substack{\text{copying} \\ \text{head pos to } W^{\ell}}} \boxed{\mathcal{T}(E)} \xrightarrow[]{\substack{\text{restore} \\ \text{head pos from } W^{\ell}}} \bigcirc \longrightarrow$$

$$\mathcal{T}(!(E)^{\ell'}) \quad = \quad \longrightarrow \bigcirc \xrightarrow[]{\substack{\text{copying} \\ \text{head pos to } W^{\ell'} \text{ \&} \\ \text{backup all vars.}}} \boxed{\mathcal{IM}(\mathcal{T}(E))} \xrightarrow[]{\substack{\text{restore} \\ \text{all info.}}} \bigcirc \longrightarrow$$

input tape	:	$\vdash \ a \ b \ c \ b \ \ldots \ \overset{\triangledown}{a} \ \ldots\ldots\ldots\ldots \ b \ \ldots \ c \ b \ a \ a \ \dashv$
pos	:	N (at a), M (at b)
working tape W^{ℓ}	:	binary encoding of N

$\mathcal{IM}(M)$ is the complement version of M computed by Immerman's construction. Please recall **NL** $=$ **co-NL** by Immerman–Szelepcsényi theorem.

Conclusion

	Language class	Comp. of Membership Problem
REGEX	$= \text{Dspace}(O(1))$	**NL**-complete (1)
REGEX + *Backreferences*	$\subseteq$ **NL** $[1]$ $\subseteq$ **INDEX** $[2]$	**NP**-complete (2)
REGEX + *Lookaheads*	$=$ REGEX $[3]$	**P**-complete (3)
REWBLk	$=$ **NL** NEW	**PSPACE**-complete NEW

[1] *Inside the class of regex languages*, M.L. Schmid, 2013

[2] *On the expressive power of regular expressions with backreferences*, T.Nogami & T. Terauchi, 2023

[3] *Alternation*, A. Chandra, D. Kozen, & L. Stockmeyer, 1981

(1) is equivalent to the st-connectivity (directed-graph connectivity) problem

(2) *Algorithms for finding patterns in strings*, A. Aho, 1990

(3) *A note on the space complexity of some decision problems for finite automata*, T. Jiang & B. Ravikumar, 1991

Bonus Slides

Normalizing expressions

Let's consider an expression:

$$(\backslash x\ (a\ \backslash x)_x\ \backslash x)_x,$$

which violates conditions $(\cdots \backslash x \cdots)_x$ and $(\cdots (\cdots)_x \cdots)_x$.

We can normalize it as follows:

$$\begin{aligned} & (\backslash x\ (a\ \backslash x)_x\ \backslash x)_x \\ \Rightarrow\quad & (\backslash x\ (a\ \backslash x)_x\ \backslash x)_y\ ?((\backslash y)_x) \\ \Rightarrow\quad & (\backslash x\ (a\ \backslash x)_z\ ?((\backslash z)_x)\ \backslash x)_y\ ?((\backslash y)_x) \end{aligned}$$

From outside to inside, we replace labels $(\cdots)_x$ with a frash one $(\cdots)_\alpha$ and then add $?((\backslash\alpha)_x)$ for copying the content:

$$(\cdots)_x \Rightarrow (\cdots)_\alpha ?((\backslash\alpha)_x)$$