Slightly non-linear higher-order tree transducers

Lê Thành Dũng (Tito) Nguyễn (Aix-Marseille Univ.)
joint work with Gabriele Vanoni (IRIF, Paris)

STACS 2025 (Marseille $\xrightarrow{\text{online}}$ Jena)

- (Slightly non-)**linear**: as in *linear logic*
- **higher-order**: as in functional programming / *λ-calculus*

- (Slightly non-)**linear**: as in *linear logic*
- **higher-order**: as in functional programming / *λ-calculus*
- **tree transducers**: *automata* for tree-to-tree functions

- (Slightly non-)**linear**: as in *linear logic*
- **higher-order**: as in functional programming / *λ-calculus*
- **tree transducers**: *automata* for tree-to-tree functions

Comparing the expressive power of automata-like devices:

storing λ-terms vs. more conventional

- (Slightly non-)**linear**: as in *linear logic*
- **higher-order**: as in functional programming / *λ-calculus*
- **tree transducers**: *automata* for tree-to-tree functions

Comparing the expressive power of automata-like devices:

storing λ-terms vs. more conventional

First: conventional examples on strings

Two-way automata

Transitions: update finite state + move left/right depending on new state

Example: states $Q = \{q_1^{\rightarrow}, q_2^{\leftarrow}, q_3^{\leftarrow}\}$, initial state $q_1^{\rightarrow}$

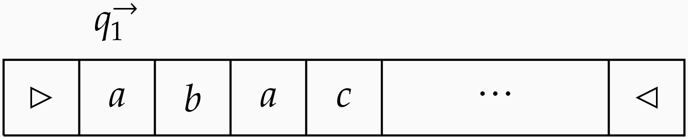
$$q_1^{\rightarrow}, (a|b) \mapsto q_1^{\rightarrow} \qquad q_1^{\rightarrow}, c \mapsto q_2^{\leftarrow} \qquad q_2^{\leftarrow}, (a|b|c) \mapsto q_3^{\leftarrow} \qquad q_3^{\leftarrow}, b \mapsto \text{accept}$$

Two-way automata

Transitions: update finite state + move left/right depending on new state

Example: states $Q = \{q_1^{\rightarrow}, q_2^{\leftarrow}, q_3^{\leftarrow}\}$, initial state $q_1^{\rightarrow}$

$$q_1^{\rightarrow}, (a|b) \mapsto q_1^{\rightarrow} \qquad q_1^{\rightarrow}, c \mapsto q_2^{\leftarrow} \qquad q_2^{\leftarrow}, (a|b|c) \mapsto q_3^{\leftarrow} \qquad q_3^{\leftarrow}, b \mapsto \text{accept}$$

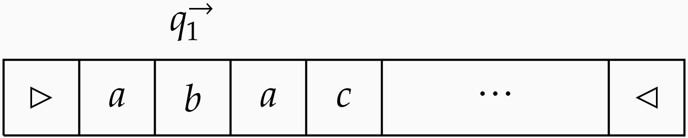


Two-way automata

Transitions: update finite state + move left/right depending on new state

Example: states $Q = \{q_1^{\rightarrow}, q_2^{\leftarrow}, q_3^{\leftarrow}\}$, initial state $q_1^{\rightarrow}$

$$\color{red}{q_1^{\rightarrow}, (a|b) \mapsto q_1^{\rightarrow}} \qquad q_1^{\rightarrow}, c \mapsto q_2^{\leftarrow} \qquad q_2^{\leftarrow}, (a|b|c) \mapsto q_3^{\leftarrow} \qquad q_3^{\leftarrow}, b \mapsto \text{accept}$$

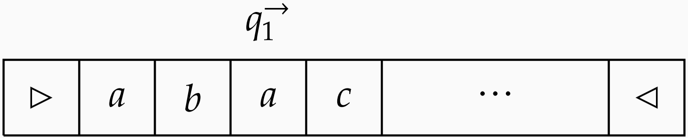


Two-way automata

Transitions: update finite state + move left/right depending on new state

Example: states $Q = \{q_1^{\rightarrow}, q_2^{\leftarrow}, q_3^{\leftarrow}\}$, initial state $q_1^{\rightarrow}$

$$\color{red}{q_1^{\rightarrow}, (a|b) \mapsto q_1^{\rightarrow}} \qquad q_1^{\rightarrow}, c \mapsto q_2^{\leftarrow} \qquad q_2^{\leftarrow}, (a|b|c) \mapsto q_3^{\leftarrow} \qquad q_3^{\leftarrow}, b \mapsto \text{accept}$$



Two-way automata

Transitions: update finite state + move left/right depending on new state

Example: states $Q = \{q_1^{\rightarrow}, q_2^{\leftarrow}, q_3^{\leftarrow}\}$, initial state $q_1^{\rightarrow}$

$$\color{red}{q_1^{\rightarrow}, (a|b) \mapsto q_1^{\rightarrow}} \qquad q_1^{\rightarrow}, c \mapsto q_2^{\leftarrow} \qquad q_2^{\leftarrow}, (a|b|c) \mapsto q_3^{\leftarrow} \qquad q_3^{\leftarrow}, b \mapsto \text{accept}$$

$q_1^{\rightarrow}$

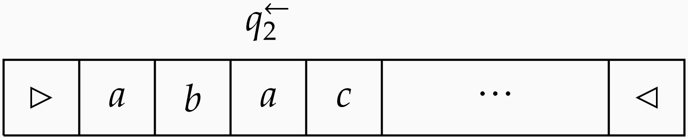
$\triangleright$	a	b	a	c	$\cdots$	$\triangleleft$

Two-way automata

Transitions: update finite state + move left/right depending on new state

Example: states $Q = \{q_1^{\rightarrow}, q_2^{\leftarrow}, q_3^{\leftarrow}\}$, initial state $q_1^{\rightarrow}$

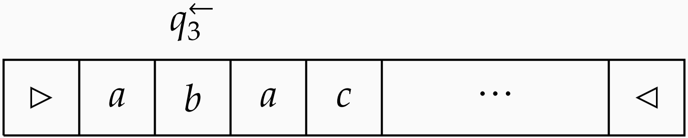
$$q_1^{\rightarrow}, (a|b) \mapsto q_1^{\rightarrow} \qquad q_1^{\rightarrow}, c \mapsto q_2^{\leftarrow} \qquad q_2^{\leftarrow}, (a|b|c) \mapsto q_3^{\leftarrow} \qquad q_3^{\leftarrow}, b \mapsto \text{accept}$$



Transitions: update finite state + move left/right depending on new state

Example: states $Q = \{q_1^{\rightarrow}, q_2^{\leftarrow}, q_3^{\leftarrow}\}$, initial state $q_1^{\rightarrow}$

$$q_1^{\rightarrow}, (a|b) \mapsto q_1^{\rightarrow} \qquad q_1^{\rightarrow}, c \mapsto q_2^{\leftarrow} \qquad q_2^{\leftarrow}, (a|b|c) \mapsto q_3^{\leftarrow} \qquad q_3^{\leftarrow}, b \mapsto \text{accept}$$

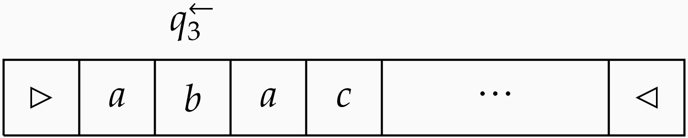


Two-way automata

Transitions: update finite state + move left/right depending on new state

Example: states $Q = \{q_1^{\rightarrow}, q_2^{\leftarrow}, q_3^{\leftarrow}\}$, initial state $q_1^{\rightarrow}$

$$q_1^{\rightarrow}, (a|b) \mapsto q_1^{\rightarrow} \qquad q_1^{\rightarrow}, c \mapsto q_2^{\leftarrow} \qquad q_2^{\leftarrow}, (a|b|c) \mapsto q_3^{\leftarrow} \qquad q_3^{\leftarrow}, b \mapsto \text{accept}$$



Theorem (Rabin & Scott / Shepherdson 1959)

Two-way automata ≡ one-way automata (≡ regular languages)

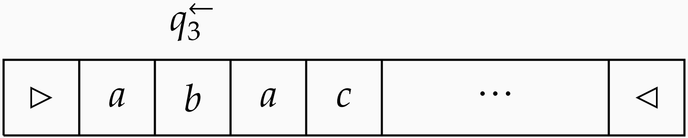
⟶ rightfully belong to "finite-state computation"

Two-way automata

Transitions: update finite state + move left/right depending on new state

Example: states $Q = \{q_1^{\rightarrow}, q_2^{\leftarrow}, q_3^{\leftarrow}\}$, initial state $q_1^{\rightarrow}$

$$q_1^{\rightarrow}, (a|b) \mapsto q_1^{\rightarrow} \qquad q_1^{\rightarrow}, c \mapsto q_2^{\leftarrow} \qquad q_2^{\leftarrow}, (a|b|c) \mapsto q_3^{\leftarrow} \qquad q_3^{\leftarrow}, b \mapsto \text{accept}$$



Theorem (Rabin & Scott / Shepherdson 1959)

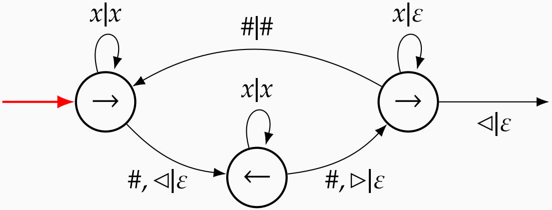
Two-way automata ≡ one-way automata (≡ regular languages)

⟶ rightfully belong to "finite-state computation"

⇒ so does their extension with string outputs

Two-way transducers

Example: $w_1\#\ldots\#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\#\ldots\#w_n \cdot \texttt{reverse}(w_n)$



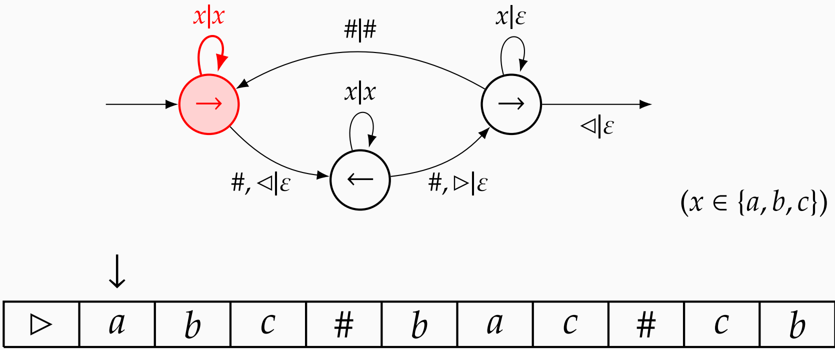
$(x \in \{a, b, c\})$

$\triangleright$	a	b	c	#	b	a	c	#	c	b	$\triangleleft$

Output:

Two-way transducers

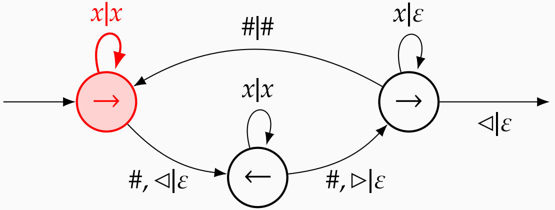
Example: $w_1\#\ldots\#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\#\ldots\#w_n \cdot \texttt{reverse}(w_n)$



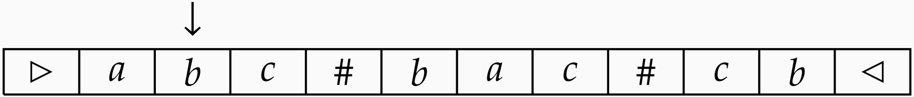
Output:

Two-way transducers

Example: $w_1\# \dots \# w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \dots \# w_n \cdot \texttt{reverse}(w_n)$



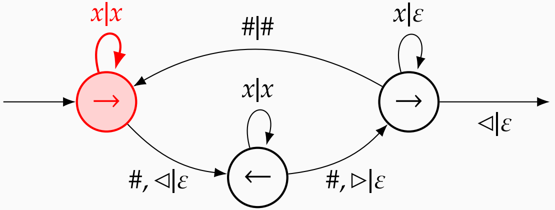
$(x \in \{a, b, c\})$



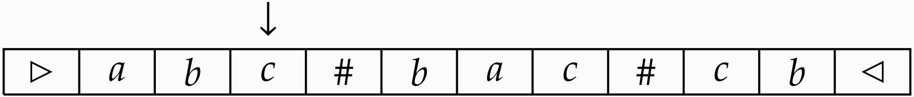
Output: a

Two-way transducers

Example: $w_1\#\ldots\#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\#\ldots\#w_n \cdot \texttt{reverse}(w_n)$



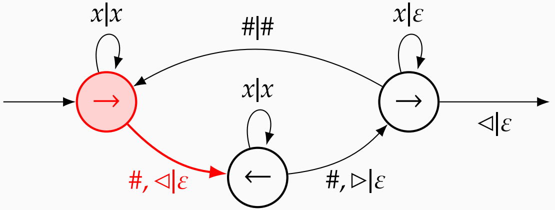
$(x \in \{a, b, c\})$



Output: *ab*

Two-way transducers

Example: $w_1\# \ldots \# w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \ldots \# w_n \cdot \texttt{reverse}(w_n)$



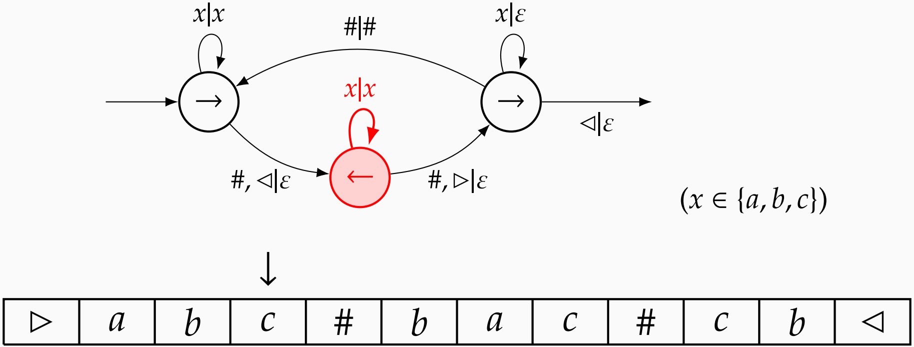
$(x \in \{a, b, c\})$

▷	a	b	c	#	b	a	c	#	c	b	◁

Output: abc

Two-way transducers

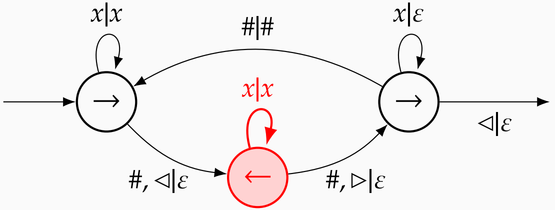
Example: $w_1\#\dots\#w_n \longmapsto w_1\cdot\texttt{reverse}(w_1)\#\dots\#w_n\cdot\texttt{reverse}(w_n)$



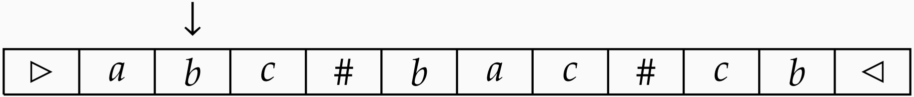
Output: *abc*

Two-way transducers

Example: $w_1\#\ldots\#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\#\ldots\#w_n \cdot \texttt{reverse}(w_n)$



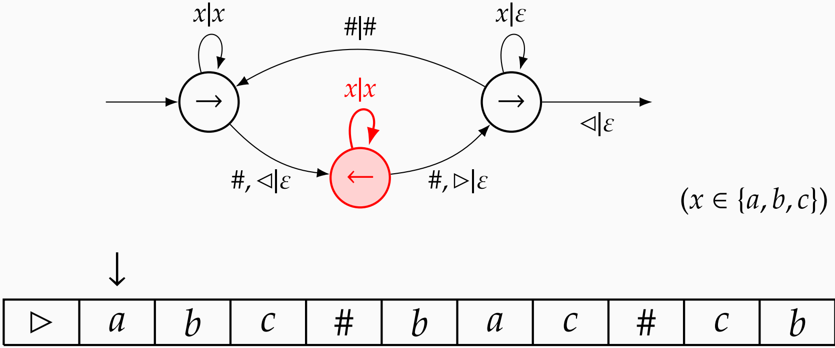
$(x \in \{a, b, c\})$



Output: *abcc*

Two-way transducers

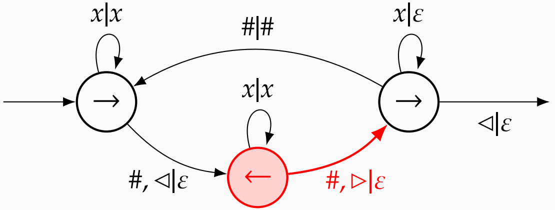
Example: $w_1\# \ldots \# w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \ldots \# w_n \cdot \texttt{reverse}(w_n)$



Output: $abccb$

Two-way transducers

Example: $w_1\#\ldots\#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\#\ldots\#w_n \cdot \texttt{reverse}(w_n)$



$(x \in \{a, b, c\})$

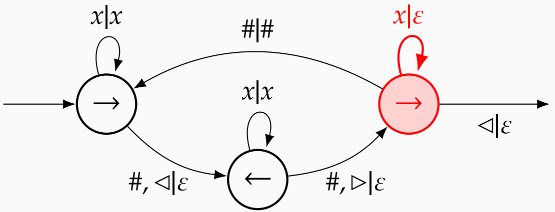
↓

▷	a	b	c	#	b	a	c	#	c	b	◁

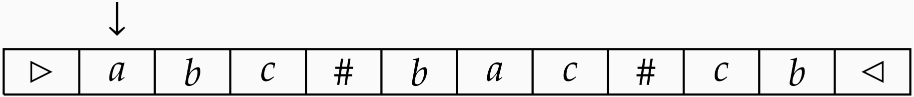
Output: $abccba$

Two-way transducers

Example: $w_1\# \dots \#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \dots \#w_n \cdot \texttt{reverse}(w_n)$



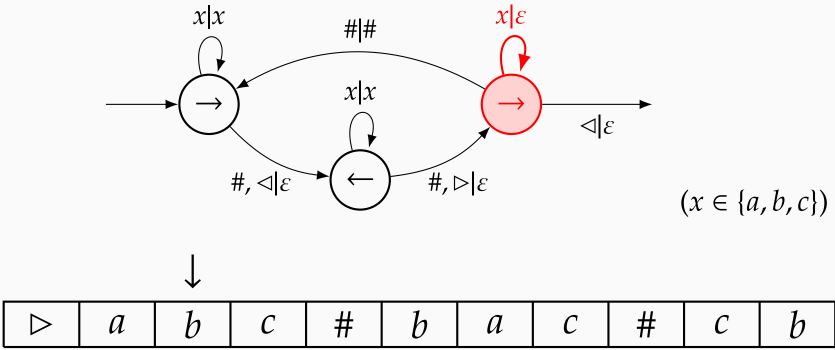
$(x \in \{a, b, c\})$



Output: *abccba*

Two-way transducers

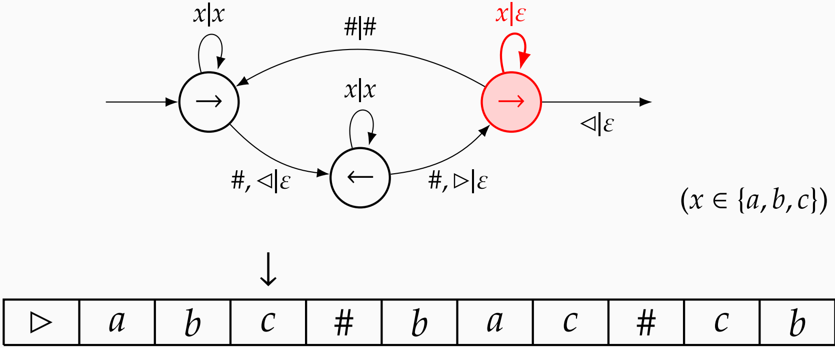
Example: $w_1\# \ldots \# w_n \longmapsto w_1 \cdot \mathtt{reverse}(w_1)\# \ldots \# w_n \cdot \mathtt{reverse}(w_n)$



Output: *abccba*

Two-way transducers

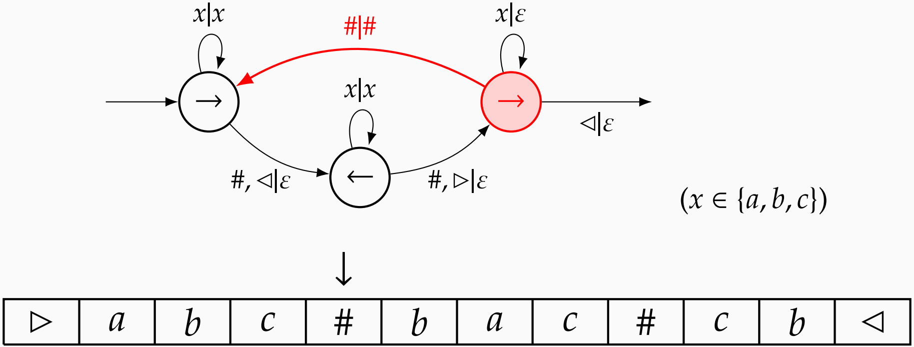
Example: $w_1\# \ldots \#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \ldots \#w_n \cdot \texttt{reverse}(w_n)$



Output: *abccba*

Two-way transducers

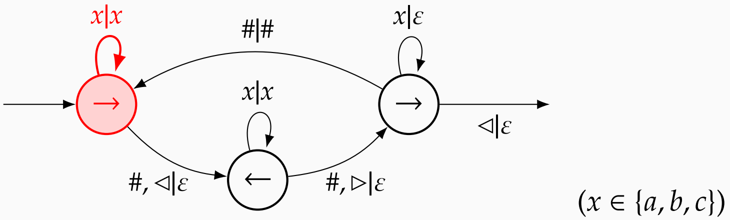
Example: $w_1\# \ldots \# w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \ldots \# w_n \cdot \texttt{reverse}(w_n)$

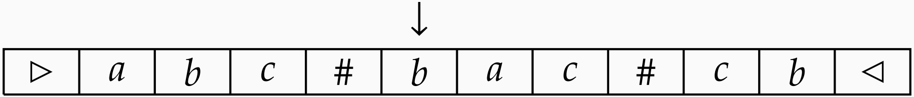


Output: *abccba*

Two-way transducers

Example: $w_1\# \dots \# w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \dots \# w_n \cdot \texttt{reverse}(w_n)$

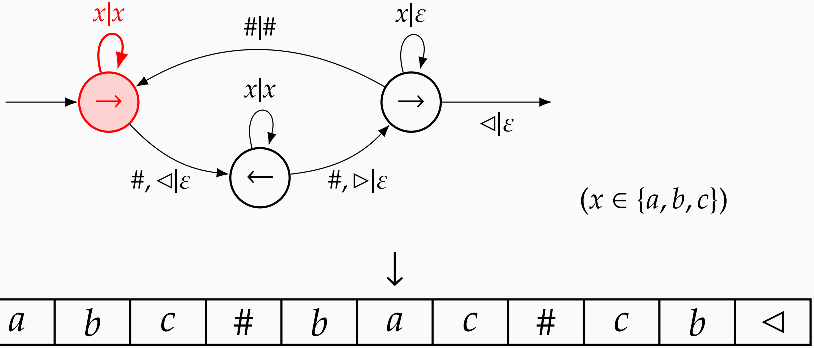




Output: $abccba\#$

Two-way transducers

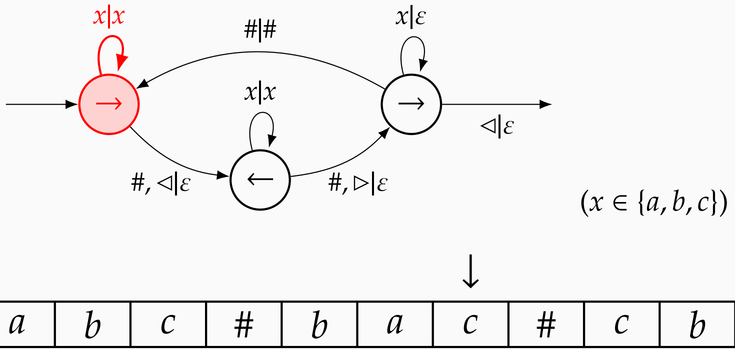
Example: $w_1\#\ldots\#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\#\ldots\#w_n \cdot \texttt{reverse}(w_n)$



Output: *abccba#b*

Two-way transducers

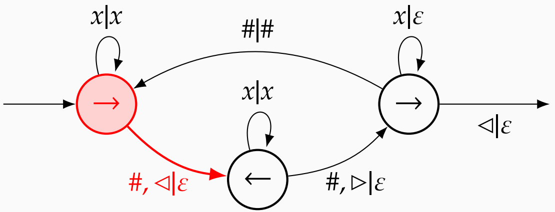
Example: $w_1\#\ldots\#w_n \longmapsto w_1 \cdot \mathtt{reverse}(w_1)\#\ldots\#w_n \cdot \mathtt{reverse}(w_n)$



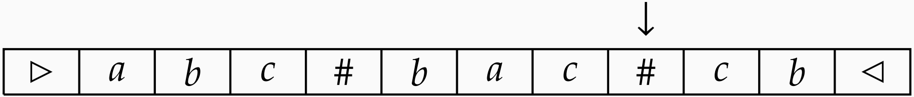
Output: *abccba#ba*

Two-way transducers

Example: $w_1\#\ldots\#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\#\ldots\#w_n \cdot \texttt{reverse}(w_n)$



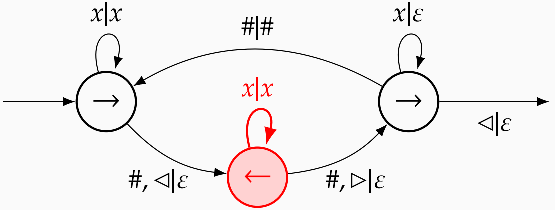
$(x \in \{a, b, c\})$



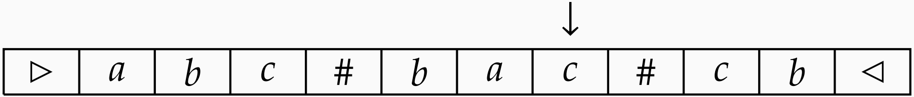
Output: *abccba#bac*

Two-way transducers

Example: $w_1\# \ldots \# w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \ldots \# w_n \cdot \texttt{reverse}(w_n)$



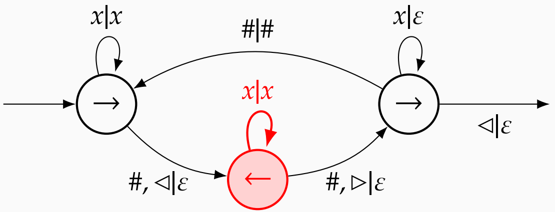
$(x \in \{a, b, c\})$



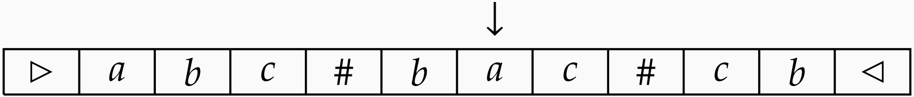
Output: *abccba#bac*

Two-way transducers

Example: $w_1\# \dots \# w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \dots \# w_n \cdot \texttt{reverse}(w_n)$



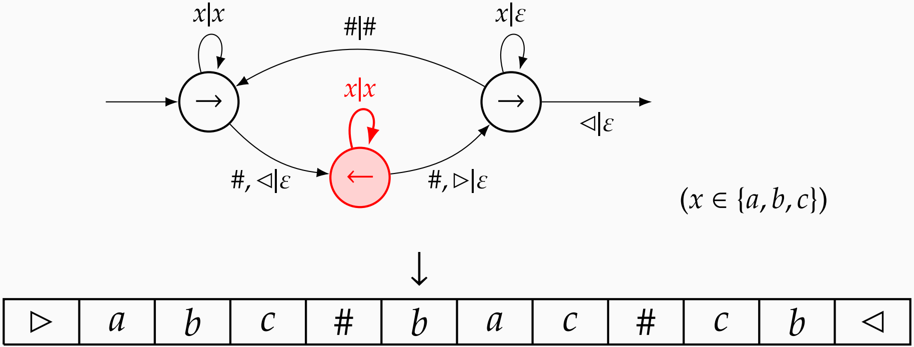
$(x \in \{a, b, c\})$



Output: *abccba#bacc*

Two-way transducers

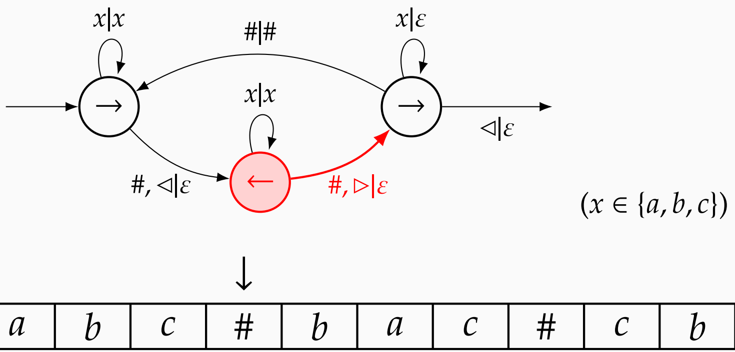
Example: $w_1\#\dots\#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\#\dots\#w_n \cdot \texttt{reverse}(w_n)$



Output: $abccba\#bacca$

Two-way transducers

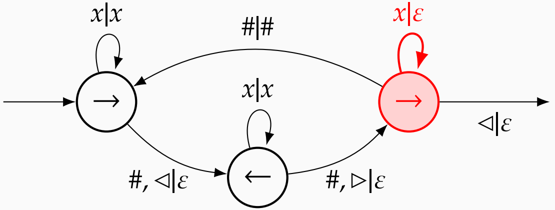
Example: $w_1\#\dots\#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\#\dots\#w_n \cdot \texttt{reverse}(w_n)$



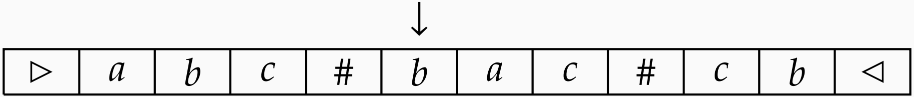
Output: *abccba#baccab*

Two-way transducers

Example: $w_1\# \dots \# w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \dots \# w_n \cdot \texttt{reverse}(w_n)$



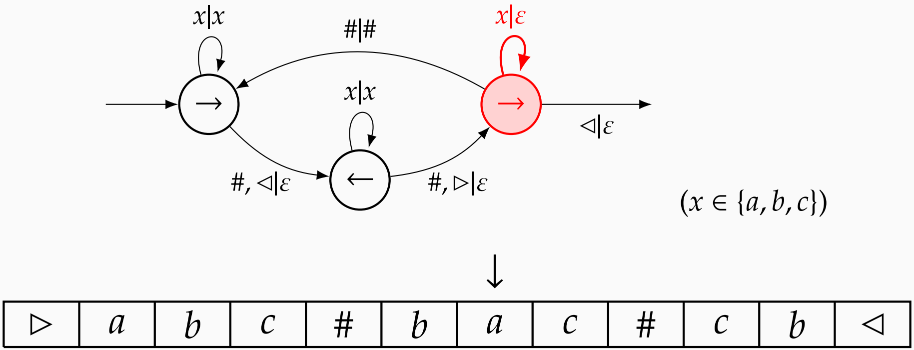
$(x \in \{a, b, c\})$



Output: *abccba#baccab*

Two-way transducers

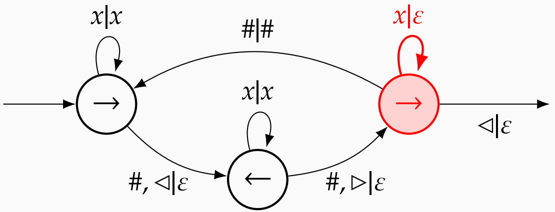
Example: $w_1\#\dots\#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\#\dots\#w_n \cdot \texttt{reverse}(w_n)$



Output: *abccba#baccab*

Two-way transducers

Example: $w_1\# \dots \# w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \dots \# w_n \cdot \texttt{reverse}(w_n)$



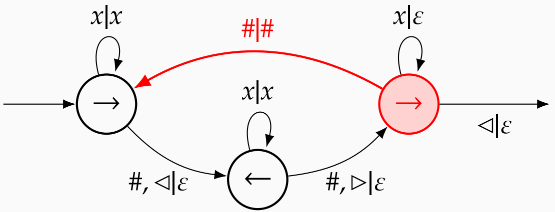
$(x \in \{a, b, c\})$

▷	a	b	c	#	b	a	c	#	c	b	◁

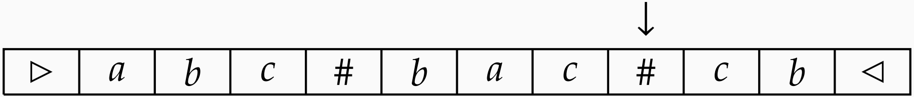
Output: *abccba#baccab*

Two-way transducers

Example: $w_1\# \dots \# w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \dots \# w_n \cdot \texttt{reverse}(w_n)$



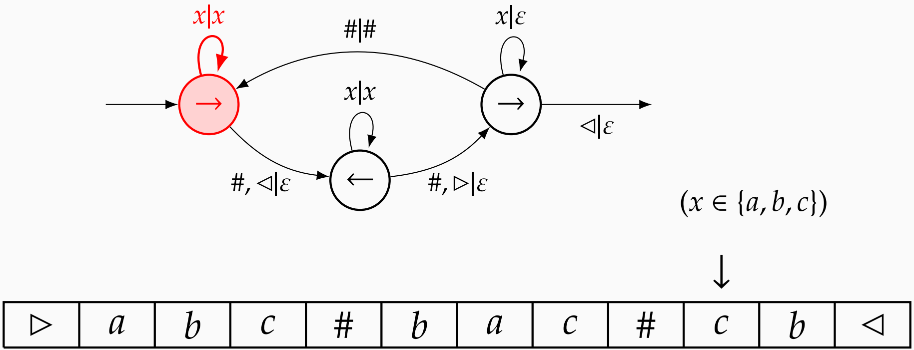
$(x \in \{a, b, c\})$



Output: *abccba#baccab*

Two-way transducers

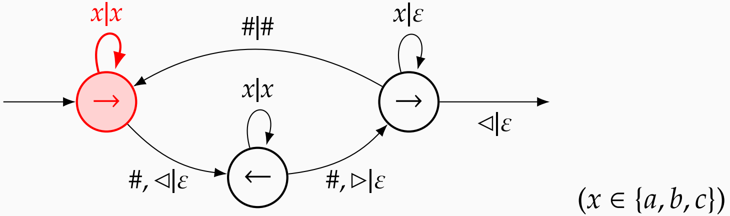
Example: $w_1\# \dots \#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \dots \#w_n \cdot \texttt{reverse}(w_n)$



Output: *abccba#baccab#*

Two-way transducers

Example: $w_1\# \dots \# w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \dots \# w_n \cdot \texttt{reverse}(w_n)$



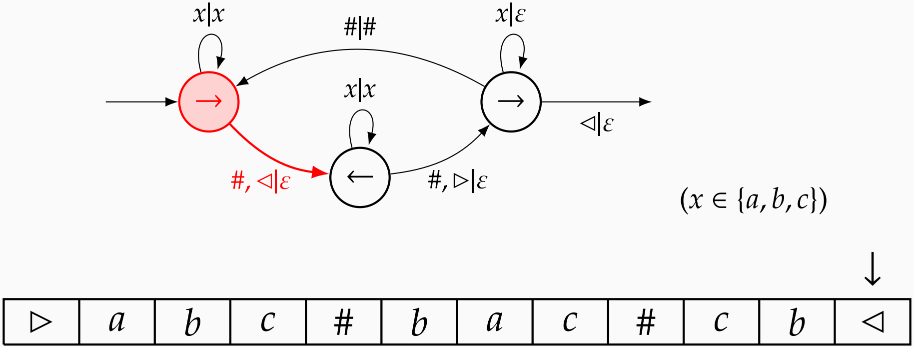
↓

▷	a	b	c	#	b	a	c	#	c	b	◁

Output: $abccba\#baccab\#c$

Two-way transducers

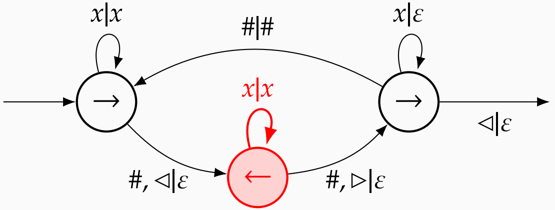
Example: $w_1\# \dots \# w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \dots \# w_n \cdot \texttt{reverse}(w_n)$



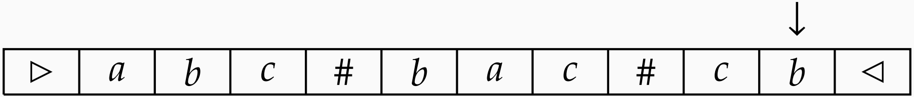
Output: $abccba\#baccab\#cb$

Two-way transducers

Example: $w_1\#\ldots\#w_n \longmapsto w_1 \cdot \mathtt{reverse}(w_1)\#\ldots\#w_n \cdot \mathtt{reverse}(w_n)$



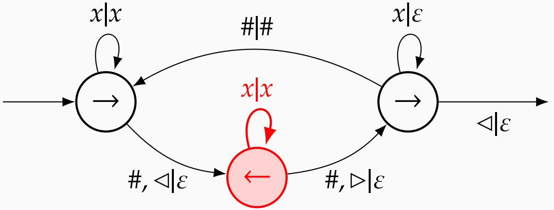
$(x \in \{a, b, c\})$



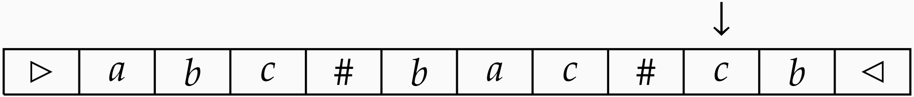
Output: *abccba#baccab#cb*

Two-way transducers

Example: $w_1\# \dots \# w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \dots \# w_n \cdot \texttt{reverse}(w_n)$



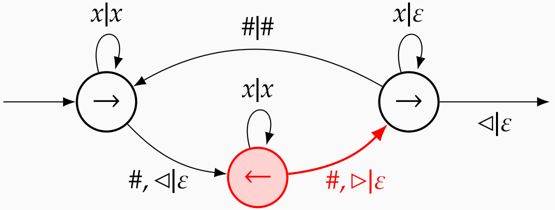
$(x \in \{a, b, c\})$



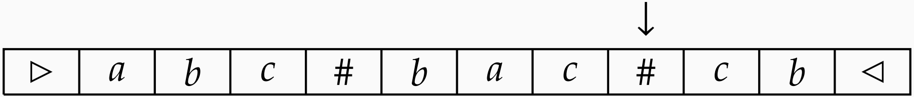
Output: *abccba#baccab#cbb*

Two-way transducers

Example: $w_1\# \ldots \# w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \ldots \# w_n \cdot \texttt{reverse}(w_n)$



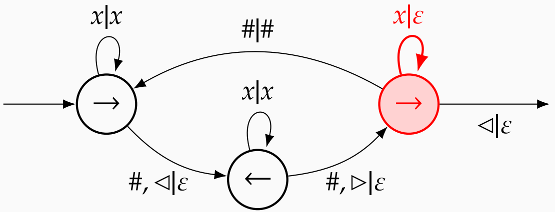
$(x \in \{a, b, c\})$



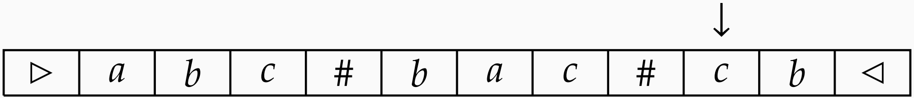
Output: *abccba#baccab#cbbc*

Two-way transducers

Example: $w_1\# \dots \# w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \dots \# w_n \cdot \texttt{reverse}(w_n)$



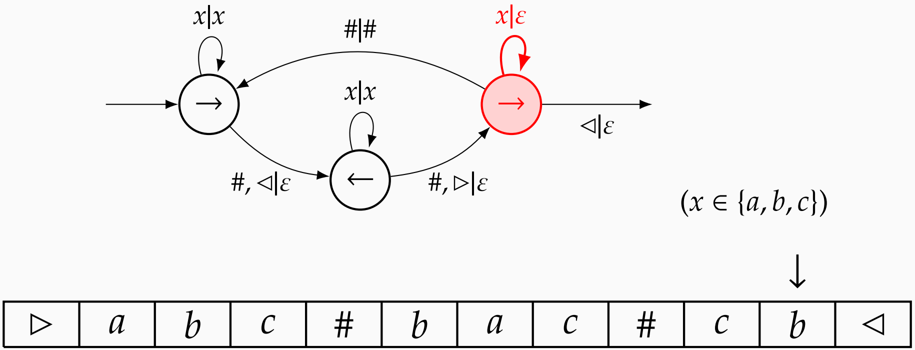
$(x \in \{a, b, c\})$



Output: *abccba#baccab#cbbc*

Two-way transducers

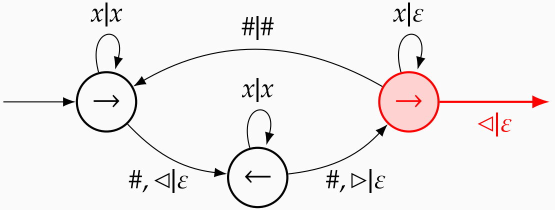
Example: $w_1\# \ldots \# w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \ldots \# w_n \cdot \texttt{reverse}(w_n)$



Output: *abccba#baccab#cbbc*

Two-way transducers

Example: $w_1\#\ldots\#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\#\ldots\#w_n \cdot \texttt{reverse}(w_n)$



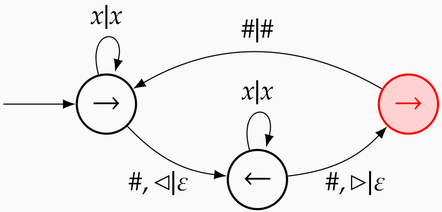
$(x \in \{a, b, c\})$

▷	a	b	c	#	b	a	c	#	c	b	◁

Output: *abccba#baccab#cbbc*

Two-way transducers are more powerful than one-way

Example: $w_1\#\ldots\#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\#\ldots\#w_n \cdot \texttt{reverse}(w_n)$



$(x \in \{a, b, c\})$

▷	a	b	c	#	b	a	c	#	c	b	◁

Output: *abccba#baccab#cbbc*

How to realize this with a one-way device?

$$\begin{aligned} \texttt{mapReverse}: \{a,b,c,\#\}^* &\to \{a,b,c,\#\}^* \\ w_1\# \dots \# w_n &\mapsto \texttt{reverse}(w_1)\# \dots \#\texttt{reverse}(w_n) \end{aligned}$$

a	c	a	b	#	b	c	#	c	a

$X = \varepsilon \qquad Y = \varepsilon$

How to realize this with a one-way device?

$$\texttt{mapReverse}: \begin{array}{ccc} \{a,b,c,\#\}^* & \rightarrow & \{a,b,c,\#\}^* \\ w_1\# \dots \#w_n & \mapsto & \texttt{reverse}(w_1)\# \dots \#\texttt{reverse}(w_n) \end{array}$$

$\downarrow$

a	c	a	b	$\#$	b	c	$\#$	c	a

$X = a \qquad Y = \varepsilon$

How to realize this with a one-way device?

$$\begin{aligned} \texttt{mapReverse} : \{a, b, c, \#\}^* &\to \{a, b, c, \#\}^* \\ w_1\# \dots \# w_n &\mapsto \texttt{reverse}(w_1)\# \dots \#\texttt{reverse}(w_n) \end{aligned}$$

	↓								
a	c	a	b	#	b	c	#	c	a

$X = ca \qquad Y = \varepsilon$

How to realize this with a one-way device?

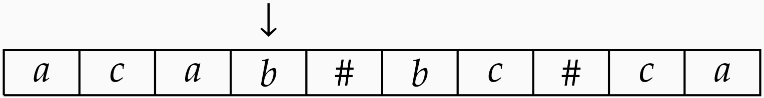
$$\begin{aligned}\texttt{mapReverse}:\ \{a,b,c,\#\}^* &\rightarrow \{a,b,c,\#\}^* \\ w_1\#\ldots\#w_n &\mapsto \texttt{reverse}(w_1)\#\ldots\#\texttt{reverse}(w_n)\end{aligned}$$

		↓							
a	c	a	b	#	b	c	#	c	a

$X = aca \qquad Y = \varepsilon$

How to realize this with a one-way device?

$$\texttt{mapReverse}: \{a,b,c,\#\}^* \to \{a,b,c,\#\}^*$$
$$w_1\#\ldots\#w_n \mapsto \texttt{reverse}(w_1)\#\ldots\#\texttt{reverse}(w_n)$$



$X = baca \qquad Y = \varepsilon$

How to realize this with a one-way device?

$$\begin{aligned} \texttt{mapReverse}: \quad \{a,b,c,\#\}^* &\to \{a,b,c,\#\}^* \\ w_1\# \dots \#w_n &\mapsto \texttt{reverse}(w_1)\# \dots \#\texttt{reverse}(w_n) \end{aligned}$$

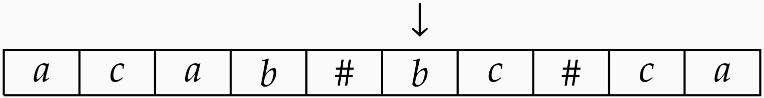
$\downarrow$

a	c	a	b	$\#$	b	c	$\#$	c	a

$X = \varepsilon \qquad Y = baca\#$

How to realize this with a one-way device?

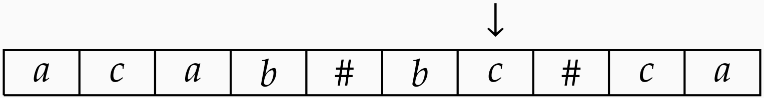
$$\texttt{mapReverse} : \{a,b,c,\#\}^* \to \{a,b,c,\#\}^*$$
$$w_1\# \ldots \#w_n \mapsto \texttt{reverse}(w_1)\# \ldots \#\texttt{reverse}(w_n)$$



$X = b \qquad Y = baca\#$

How to realize this with a one-way device?

$$\texttt{mapReverse} : \{a,b,c,\#\}^* \rightarrow \{a,b,c,\#\}^*$$
$$w_1\# \ldots \#w_n \mapsto \texttt{reverse}(w_1)\# \ldots \#\texttt{reverse}(w_n)$$



$X = cb \qquad Y = baca\#$

How to realize this with a one-way device?

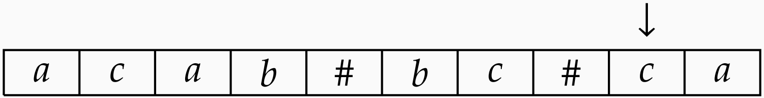
$$\texttt{mapReverse} : \{a, b, c, \#\}^* \to \{a, b, c, \#\}^*$$
$$w_1\# \ldots \# w_n \mapsto \texttt{reverse}(w_1)\# \ldots \#\texttt{reverse}(w_n)$$

a	c	a	b	$\#$	b	c	$\#$ ↓	c	a

$X = \varepsilon \qquad Y = baca\#cb\#$

How to realize this with a one-way device?

$$\texttt{mapReverse}: \{a,b,c,\#\}^* \to \{a,b,c,\#\}^*$$
$$w_1\# \ldots \#w_n \mapsto \texttt{reverse}(w_1)\# \ldots \#\texttt{reverse}(w_n)$$



$X = c \qquad Y = baca\#cb\#$

How to realize this with a one-way device?

$$\texttt{mapReverse}: \{a,b,c,\#\}^* \rightarrow \{a,b,c,\#\}^*$$

$$w_1\#\ldots\#w_n \mapsto \texttt{reverse}(w_1)\#\ldots\#\texttt{reverse}(w_n)$$

$\downarrow$

a	c	a	b	#	b	c	#	c	a

$X = ac \qquad Y = baca\#cb\#$

How to realize this with a one-way device?

$$\texttt{mapReverse}: \{a,b,c,\#\}^* \to \{a,b,c,\#\}^*$$
$$w_1\#\ldots\#w_n \mapsto \texttt{reverse}(w_1)\#\ldots\#\texttt{reverse}(w_n)$$

a	c	a	b	#	b	c	#	c	a

$X = ac$ $\quad Y = baca\#cb\#$ $\quad \texttt{mapReverse}(\ldots) = YX = baca\#cb\#ac$

How to realize this with a one-way device?

$$\texttt{mapReverse} : \{a, b, c, \#\}^* \to \{a, b, c, \#\}^*$$
$$w_1\# \dots \# w_n \mapsto \texttt{reverse}(w_1)\# \dots \#\texttt{reverse}(w_n)$$

a	c	a	b	#	b	c	#	c	a

$X = ac$ $\quad Y = baca\#cb\#$ $\quad \texttt{mapReverse}(\dots) = YX = baca\#cb\#ac$

Important point

X, Y concatenable, but **not inspectable** (“if $X[k] = a$ then…”)

$\longrightarrow$ control flow stays finite-state

How to realize this with a one-way device?

$$\begin{array}{rrcl} \texttt{mapReverse}: & \{a,b,c,\#\}^* & \to & \{a,b,c,\#\}^* \\ & w_1\#\ldots\#w_n & \mapsto & \texttt{reverse}(w_1)\#\ldots\#\texttt{reverse}(w_n) \end{array}$$

a	c	a	b	#	b	c	#	c	a

$X = ac \qquad Y = baca\#cb\# \qquad \texttt{mapReverse}(\ldots) = YX = baca\#cb\#ac$

Important point

X, Y concatenable, but **not inspectable** (“if $X[k] = a$ then…”)

$\longrightarrow$ control flow stays finite-state

What can you “reasonably” put in memory?

How to realize this with a one-way device?

$$\texttt{mapReverse} : \{a,b,c,\#\}^* \to \{a,b,c,\#\}^*$$
$$w_1\# \dots \# w_n \mapsto \texttt{reverse}(w_1)\# \dots \#\texttt{reverse}(w_n)$$

a	c	a	b	$\#$	b	c	$\#$	c	a

$X = ac$ $\quad Y = baca\#cb\#$ $\quad \texttt{mapReverse}(\dots) = YX = baca\#cb\#ac$

Important point

X, Y concatenable, but **not inspectable** (“if $X[k] = a$ then…”)

$\longrightarrow$ control flow stays finite-state

What can you “reasonably” put in memory? **LAMBDA: THE ULTIMATE**

Higher-order tree automata / transducers: simply typed λ-calculus

$$\text{Bottom-up tree aut.:} \quad a(b(c), c) \mapsto \texttt{accept?}(\delta_a(\delta_b(\delta_c), \delta_c)) \text{ with } \delta_a \colon Q^2 \to Q, \ldots$$

$$\text{Higher-order tree aut.:} \quad a(b(c), c) \mapsto \texttt{accept?}\ (t_a\ (t_b\ t_c)\ t_c) \text{ with } t_a : A^2 \Rightarrow A, \ldots$$

$$Q \text{ finite set} \qquad \text{vs.} \qquad A, B ::= o \mid A \times B \mid A \Rightarrow B$$

Higher-order tree automata / transducers: simply typed λ-calculus

Bottom-up tree aut.: $a(b(c),c) \mapsto \texttt{accept?}(\delta_a(\delta_b(\delta_c),\delta_c))$ with $\delta_a : Q^2 \to Q, \ldots$

Higher-order tree aut.: $a(b(c),c) \mapsto \texttt{accept?}\ (t_a\ (t_b\ t_c)\ t_c)$ with $t_a : A^2 \Rightarrow A, \ldots$

Q finite set vs. $A, B ::= o \mid A \times B \mid A \Rightarrow B$

Theorem (≃ Hillebrand & Kanellakis 1996 (Damm 1982?))

Higher-order tree automata recognize precisely regular tree languages.

Higher-order tree automata / transducers: simply typed λ-calculus

Bottom-up tree aut.: $a(b(c), c) \mapsto \texttt{accept?}(\delta_a(\delta_b(\delta_c), \delta_c))$ with $\delta_a : Q^2 \to Q, \ldots$

Higher-order tree aut.: $a(b(c), c) \mapsto \texttt{accept?}\ (t_a\ (t_b\ t_c)\ t_c)$ with $t_a : A^2 \Rightarrow A, \ldots$

Q finite set vs. $A, B ::= o \mid A \times B \mid A \Rightarrow B$

Theorem (≃ Hillebrand & Kanellakis 1996 (Damm 1982?))

Higher-order tree automata recognize precisely regular tree languages.

Higher-order *transducers*: make o a base type of non-inspectable trees

Higher-order tree automata / transducers: simply typed λ-calculus

Bottom-up tree aut.: $a(b(c), c) \mapsto \texttt{accept?}(\delta_a(\delta_b(\delta_c), \delta_c))$ with $\delta_a : Q^2 \to Q, \ldots$

Higher-order tree aut.: $a(b(c), c) \mapsto \texttt{accept?}\ (t_a\ (t_b\ t_c)\ t_c)$ with $t_a : A^2 \Rightarrow A, \ldots$

Q finite set vs. $A, B ::= o \mid A \times B \mid A \Rightarrow B$

Theorem (≃ Hillebrand & Kanellakis 1996 (Damm 1982?))

Higher-order tree automata recognize precisely regular tree languages.

Higher-order *transducers*: make o a base type of non-inspectable trees

memory type $A = o^k$ ▶ k-state *top-down* (sic!) *tree transducer*

Higher-order tree automata / transducers: simply typed λ-calculus

Bottom-up tree aut.: $a(b(c), c) \mapsto \texttt{accept?}(\delta_a(\delta_b(\delta_c), \delta_c))$ with $\delta_a : Q^2 \to Q, \ldots$

Higher-order tree aut.: $a(b(c), c) \mapsto \texttt{accept?}\ (t_a\ (t_b\ t_c)\ t_c)$ with $t_a : A^2 \Rightarrow A, \ldots$

Q finite set vs. $A, B ::= o \mid A \times B \mid A \Rightarrow B$

Theorem (≃ Hillebrand & Kanellakis 1996 (Damm 1982?))

Higher-order tree automata recognize precisely regular tree languages.

Higher-order *transducers*: make o a base type of non-inspectable trees

memory type $A = o^k$ ▶ k-state *top-down* (sic!) *tree transducer*

$A = (o^{\ell_1} \Rightarrow o) \times \cdots \times (o^{\ell_k} \Rightarrow o)$ ▶ k-state *macro tree transducer*, e.g. previous slide!

[Engelfriet & Vogler 1986], staple of "old-school" transducer theory

Higher-order tree automata / transducers: affine types

Problem (feature?): HO tree transducers can express *a lot* of functions
⟶ **Idea:** restrict expressivity using type system!

Higher-order tree automata / transducers: affine types

Problem (feature?): HO tree transducers can express *a lot* of functions
$\longrightarrow$ **Idea:** restrict expressivity using type system!

Theorem (claimed in my PhD; details: [Pradic & Price, MFPS'24])

Affine HO string transducers $\equiv$ *two-way transducers* ($\equiv$ *MSO transductions*)

i.e. replace $A \Rightarrow B$ by *affine* $A \multimap B$ which *can only use* A *once to produce* B
$\simeq$ "single use restrictions" in automata theory

Higher-order tree automata / transducers: affine types

Problem (feature?): HO tree transducers can express *a lot* of functions
$\longrightarrow$ **Idea:** restrict expressivity using type system!

Theorem (claimed in my PhD; details: [Pradic & Price, MFPS'24])

Affine HO string transducers $\equiv$ *two-way transducers* ($\equiv$ *MSO transductions*)

i.e. replace $A \Rightarrow B$ by *affine* $A \multimap B$ which *can only use* A *once to produce* B
$\simeq$ "single use restrictions" in automata theory

[Original thm. in "implicit automata" POV: express functions internally in λ-calculus]

Higher-order tree automata / transducers: affine types

Problem (feature?): HO tree transducers can express *a lot* of functions
$\longrightarrow$ **Idea:** restrict expressivity using type system!

Theorem (claimed in my PhD; details: [Pradic & Price, MFPS'24])

Affine HO string transducers ≡ two-way transducers (≡ MSO transductions)

i.e. replace $A \Rightarrow B$ by *affine* $A \multimap B$ which *can only use A once to produce B*
$\simeq$ "single use restrictions" in automata theory

[Original thm. in "implicit automata" POV: express functions internally in λ-calculus]

Conjecture (N. & Pradic, ICALP'21)

Affine HO tree automata $\subsetneq$ regular tree languages

Results of the paper (1)

Tree-walking: generalization of two-way automata

1 reading head moving around the tree in any direction

Theorem (N. & Vanoni, this paper)

Affine HO tree automata/transducers $\subseteq$ *reversible tree-walking aut./trans.*

Results of the paper (1)

Tree-walking: generalization of two-way automata

1 reading head moving around the tree in any direction

Theorem (N. & Vanoni, this paper)

Affine HO tree automata/transducers $\subseteq$ *reversible tree-walking aut./trans.*

Inexpressivity conjecture from last slide follows from:

Theorem (Bojańczyk & Colcombet 2005)

Tree-walking automata $\subsetneq$ *regular tree languages*

Results of the paper (1)

Tree-walking: generalization of two-way automata
1 reading head moving around the tree in any direction

Theorem (N. & Vanoni, this paper)

Affine HO tree automata/transducers	$\subseteq$	*reversible tree-walking aut./trans.*
Almost affine HO tree automata/transducers	$\subseteq$	*tree-walking aut./trans.*

Inexpressivity conjecture from last slide follows from:

Theorem (Bojańczyk & Colcombet 2005)

Tree-walking automata $\subsetneq$ *regular tree languages*

Almost affine [Kanazawa]: the base type o can be duplicated, but not the others
$\leftrightarrow$ "sharing" in the configuration graph of a tree-walking transducer

Results of the paper (2)

Lookaround = can inspect regular information at each node
= preprocessing by very simple transducers / MSO relabeling

Corollary (new proof of [Kanazawa 2008; Gallot, Lemay & Salvati 2020])

Affine HO tree transducers with lookaround	$\equiv$	*MSO transductions*
Almost affine HO tree trans. w/ lookaround	$\equiv$	*unfolding* $\circ$ *MSOT*

Results of the paper (2)

Lookaround = can inspect regular information at each node
= preprocessing by very simple transducers / MSO relabeling

Corollary (new proof of [Kanazawa 2008; Gallot, Lemay & Salvati 2020])

Affine HO tree transducers with lookaround	$\equiv$	*MSO transductions*
Almost affine HO tree trans. w/ lookaround	$\equiv$	*unfolding* $\circ$ *MSOT*

Other way to overcome inexpressivity [N. & Pradic]: add &/$\oplus$ types

$A \otimes B$ ("multiplicative") vs. $A \& B$ ("additive")

(better suited to "implicit automata" POV)

Results of the paper (3)

Exponential modality $!A$ makes A duplicable

$A, B ::= o \mid A \multimap B \mid !A \qquad (A \Rightarrow B = !A \multimap B)$

Affine = !-free

Almost affine = '!' only on o

Results of the paper (3)

Exponential modality $!A$ makes A duplicable

$A, B ::= o \mid A \multimap B \mid !A \qquad (A \Rightarrow B = !A \multimap B)$

Affine = !-free

Almost affine = '!' only on o

Almost !-depth 1: '!' only on almost affine types

Theorem (N. & Vanoni, this paper)

Almost !-depth 1 HO tree trans. w/ lookaround ≡ invisible pebble tree transducers

(tree-walking + unbounded stack of marked positions [Engelfriet et al. PODS'07])

Results of the paper (3)

Exponential modality $!A$ makes A duplicable

$A, B ::= o \mid A \multimap B \mid !A \qquad (A \Rightarrow B = !A \multimap B)$

Affine = !-free

Almost affine = '!' only on o

Almost !-depth 1: '!' only on almost affine types

Theorem (N. & Vanoni, this paper)

Almost !-depth 1 HO tree trans. w/ lookaround ≡ invisible pebble tree transducers (tree-walking + unbounded stack of marked positions [Engelfriet et al. PODS'07])

Main tool: *Interaction Abstract Machine* executing λ-terms (coauthor's expertise!), automaton-like variant of Girard's "Geometry of Interaction"

Results of the paper (3)

Exponential modality $!A$ makes A duplicable

$A, B ::= o \mid A \multimap B \mid !A \qquad (A \Rightarrow B = !A \multimap B)$

Affine = !-free

Almost affine = '!' only on o

Almost !-depth 1: '!' only on almost affine types

Theorem (N. & Vanoni, this paper)

Almost !-depth 1 HO tree trans. w/ lookaround ≡ invisible pebble tree transducers (tree-walking + unbounded stack of marked positions [Engelfriet et al. PODS'07])

Main tool: *Interaction Abstract Machine* executing λ-terms (coauthor's expertise!), automaton-like variant of Girard's "Geometry of Interaction"

Connections between GoI and two-way automata [Hines 2003]
tree-walking transducers [Katsumata 2008]

... but their category-theoretic version of GoI does not "scale" to almost !-depth 1

Results of the paper (3)

Thanks for your attention!

Exponential modality $!A$ makes A duplicable

$A, B ::= o \mid A \multimap B \mid !A \qquad (A \Rightarrow B = !A \multimap B)$

Affine = !-free

Almost affine = '!' only on o

Almost !-depth 1: '!' only on almost affine types

Theorem (N. & Vanoni, this paper)

Almost !-depth 1 HO tree trans. w/ lookaround ≡ invisible pebble tree transducers
(tree-walking + unbounded stack of marked positions [Engelfriet et al. PODS'07])

Main tool: *Interaction Abstract Machine* executing λ-terms (coauthor's expertise!),
automaton-like variant of Girard's "Geometry of Interaction"

Connections between GoI and two-way automata [Hines 2003]
tree-walking transducers [Katsumata 2008]
... but their category-theoretic version of GoI does not "scale" to almost !-depth 1