MORE GENERAL MACHINES

CONFIGURATIONS AND COMPUTATION TRACES

More read-only algorithms

- Consider the language L over the Latin Alphabet consisting of strings that miss some letter.

  All English words are in L, but virtually no book is.

- L is a regular language: it is the intersection of the 26 languages $\{w \mid w \text{ uses } \sigma\}$ for $\sigma = \texttt{a}, \texttt{b}...$.

More read-only algorithms

- Consider the language L over the Latin Alphabet consisting of strings that miss some letter.

  All English words are in L, but virtually no book is.

- L is a regular language: it is the intersection of the 26 languages $\{w \mid w \text{ uses } \sigma\}$ for $\sigma = \texttt{a}, \texttt{b}...$.

- The smallest DFA that recognizes L has $\geqslant 2^{26} > 67,000,000$ states.

- The smallest NFA recognizing L has 27 states.

More read-only algorithms

- Consider the language L over the Latin Alphabet consisting of strings that miss some letter.

  All English words are in L, but virtually no book is.

- L is a regular language: it is the intersection of the 26 languages $\{w \mid w \text{ uses } \sigma\}$ for $\sigma = \mathtt{a}, \mathtt{b}...$.

- The smallest DFA that recognizes L has $\geqslant 2^{26} > 67,000,000$ states.

- The smallest NFA recognizing L has 27 states.

- ***Is there a deterministic algorithm recognizing L using a small number of states?***

A deterministic algorithm

- Algorithm: Scan for each digit separately, and repeat.

A deterministic algorithm

- Algorithm: Scan for each digit separately, and repeat.
- This cannot be done if we only read forward!
  The cursor would have to be scrolled back (or repositioned).
- So let's imagine a device that behaves just like an automaton,
  but can move the cursor both ways.

Extensions needed

- Each symbol read determines not only next state,
  but also next move: forward or backward.

Extensions needed

- Each symbol read determines not only next state,
  but also next move: forward or backward.
- Detecting ends of input requires end-markers:
  say $>$ (the ***gate***) on the left,
  and $\sqcup$ (the ***blank***) on the right.

Extensions needed

- Each symbol read determines not only next state,
  but also next move: forward or backward.
- Detecting ends of input requires end-markers:
  say $>$ (the ***gate***) on the left,
  and $\sqcup$ (the ***blank***) on the right.
- Termination signaled by the states, not the end of input.

Two-way automata

- A ***two-way automaton (2DFA)*** over an alphabet Σ:
  - Finite set of states Q
  - $s \in Q$, the *initial state*
  - $a \in S$, the *accepting state*
  - Transition partial-function: $\delta : Q \times \Gamma \rightharpoonup Q \times \mathbf{Act}$

  where $\Gamma = \Sigma \cup \{>, \sqcup\}$ and $\mathbf{Act} = \{+, -\}$.

Two-way automata

- A ***two-way automaton (2DFA)*** over an alphabet Σ:
  - Finite set of states Q
  - $s \in Q$, the *initial state*
  - $a \in S$, the *accepting state*
  - Transition partial-function: $\delta : Q \times \Gamma \rightharpoonup Q \times \mathbf{Act}$

  where $\Gamma = \Sigma \cup \{>, \sqcup\}$ and $\mathbf{Act} = \{+, -\}$.
- $\mathbf{Act}$ is the set of ***Actions***.

  Here they are $+$ for "step formward" and $-$ for "step back."
- Note: End-markers are added to the alphabet Σ.

Intended behavior of 2DFAs

- Write $q \xrightarrow{\sigma(\alpha)} p$ for $\delta(q, \sigma) = \langle p, \alpha \rangle$.

Intended behavior of 2DFAs

- Write $q \xrightarrow{\sigma(\alpha)} p$ for $\delta(q, \sigma) = \langle p, \alpha \rangle$.
- **The intent:**
  - A 2DFA operates on the input string extended with end-markers:
    Input 001201 appears as >001201⊔.

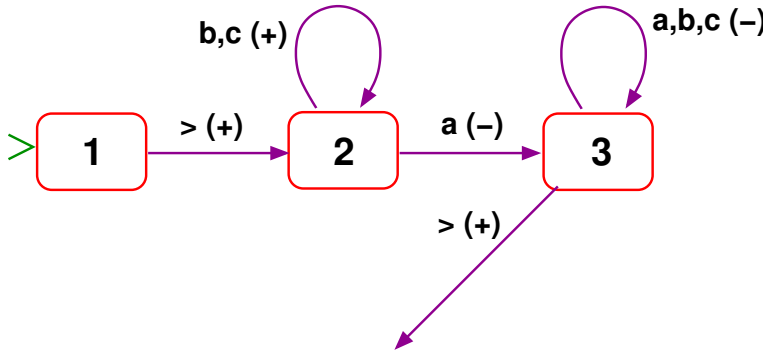
Intended behavior of 2DFAs

- Write $q \xrightarrow{\sigma(\alpha)} p$ for $\delta(q, \sigma) = \langle p, \alpha \rangle$.
- **The intent:**
  - A 2DFA operates on the input string extended with end-markers:
    Input `001201` appears as `>001201⊔`.
- A 2DFA scans one input symbol at a time.
  Visualize it as a ***cursor:***
  `>abc⊔` `>abc⊔` `>abc⊔`

A 2DFA for the "all-letters" language

- Here is a 2DFA over $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$
  that recognizs the strings using all three letter.

A 2DFA for the "all-letters" language



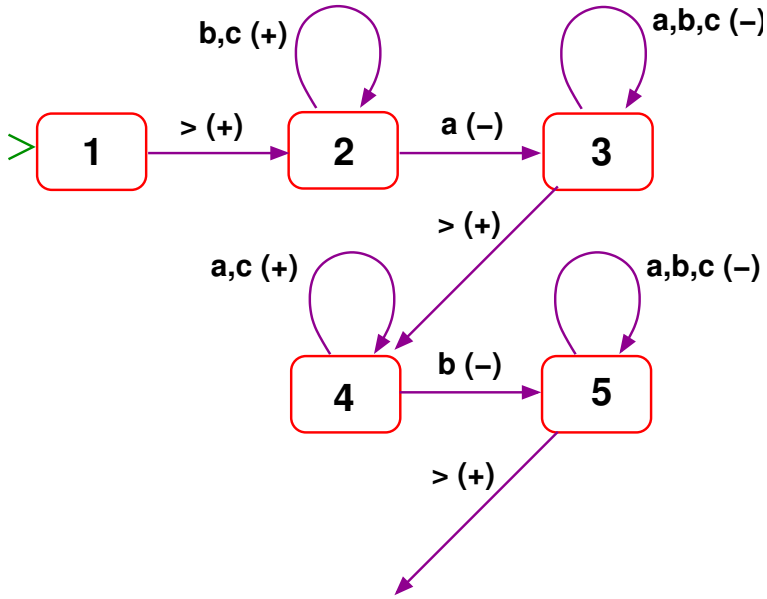
- Cycle through `b`'s and `c`'s until an `a` is found.

  If so, return to the gate;

  if not then then the blank end-maker is reached, for which there is no transition.

  The machine stops without accepting.

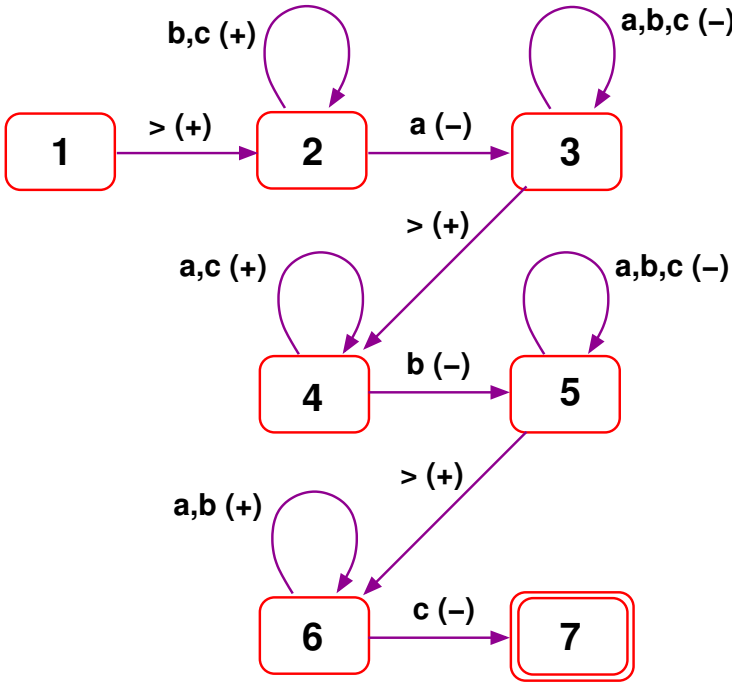
A 2DFA for the "all-letters" language



- Next cycle through `a`'s and `c`'s until a `b` is found.

  If so, return to the gate; if not then the final blank is reached, resulting as aboe in stopping without accepting.

A 2DFA for the "all-letters" language



- Cycle through `a`'s and `b`'s until a `c` is found.
  If so, accept. if not then stop at final blank without accepting.

Operational semantics of 2DFAs: configurations

- The 2DFA is our first device where execution steps
  consists in more than just a change of state.
- To describe a 2DFA's behavior we must account for the cursor position
  and therefore keep a record of the entire input for future use.

Operational semantics of 2DFAs: configurations

- The 2DFA is our first device where execution steps
  consists in more than just a change of state.
- To describe a 2DFA's behavior we must account for the cursor position
  and therefore keep a record of the entire input for future use.
- A ***cursored-string*** over Σ is a $\Sigma-$string with one symbol-position underlined.
- A ***configuration (cfg)*** is a pair $(q, \check{w})$ where
  - q is a state, and
  - $\check{w}$ is a cursored-string.
- Example: $(\mathbf{x}, >0101\underline{1}00\sqcup)$

Operational semantics of 2DFAs: configurations

- The 2DFA is our first device where execution steps consists in more than just a change of state.
- To describe a 2DFA's behavior we must account for the cursor position and therefore keep a record of the entire input for future use.
- A ***cursored-string*** over Σ is a $\Sigma-$string with one symbol-position underlined.
- A ***configuration (cfg)*** is a pair $(q, \check{w})$ where
  - q is a state, and
  - $\check{w}$ is a cursored-string.
- Example: $(\mathbf{x}, >0101\underline{1}00\sqcup)$
- The ***initial cfg for input*** w is the cfg $(s, \geq w\sqcup)$.

The YIELD relation between cfg's

- Given a 2DFA M its **Yield** relation $\Rightarrow_M$ is generated by
  - If $q \xrightarrow{\gamma(+)} p$ then $(q, u\underline{\gamma}\tau v) \Rightarrow (p, u\gamma\underline{\tau}v)$
  - If $q \overset{\gamma(-)}{\rightarrow} p$ then $(q, u\tau\underline{\gamma}v) \Rightarrow (p, u\underline{\tau}\gamma v)$

The YIELD relation between cfg's

- Given a 2DFA M its **Yield** relation $\Rightarrow_M$ is generated by
  - If $q \xrightarrow{\tau(+)} p$ then $(q, u\underline{\gamma}\tau v) \Rightarrow (p, u\gamma\underline{\tau}v)$
  - If $q \xrightarrow{\gamma(-)} p$ then $(q, u\tau\underline{\gamma}v) \Rightarrow (p, u\underline{\tau}\gamma v)$
- These clauses are the only ones in force.
  If a cfg ends with a cursored symbol, as in $(q, 011010\underline{)}$,
  then a transition $q \xrightarrow{0(+)} p$ does not apply.
- Similarly, a step-back transition has no effect when
  the cursor is at the first symbol.

Traces, accepted strings, recognized languages

- A cfg $c = (q, u\gamma v)$ is **terminal** if no transition applies.
  It is **accepting** if its state is an accepting state.

Traces, accepted strings, recognized languages

- A cfg $c = (q, u\gamma v)$ is **terminal** if no transition applies.
  It is **accepting** if its state is an accepting state.
- A **trace** of M for input w
  is a sequence $c_0 \Rightarrow c_1 \Rightarrow \cdots$,
  where c_0 is initial for w, and either
  - the sequence is infinite; or
  - the sequence is finite, and its last cfg is terminal.

Traces, accepted strings, recognized languages

- A cfg $c = (q, u\gamma v)$ is **terminal** if no transition applies.
  It is **accepting** if its state is an accepting state.
- A **trace** of M for input w
  is a sequence $c_0 \Rightarrow c_1 \Rightarrow \cdots$,
  where c_0 is initial for w, and either
  - the sequence is infinite; or
  - the sequence is finite, and its last cfg is terminal.
- The trace is **accepting** if
  it is finite and its last cfg is accepting.
- M **accepts** $w \in \Sigma^*$
  if its trace for input w is accepting.

On-site writing

A recognition algorithm for $\{a^n b^n\}$

- Since the language $\{a^n b^n \mid n \geqslant 0\}$ is not regular
  it is not recognized even by a 2-way automaton.

A recognition algorithm for $\{a^n b^n\}$

- Since the language $\{a^n b^n \mid n \geqslant 0\}$ is not regular
  it is not recognized even by a 2-way automaton.
- ***Can you think of a simple informal recognition algorithm?***

A recognition algorithm for $\{\mathtt{a}^n\mathtt{b}^n\}$

- Since the language $\{\mathtt{a}^n\mathtt{b}^n \mid n \geqslant 0\}$ is not regular
  it is not recognized even by a 2-way automaton.
- How about repeating this:
  cross off initial $\mathtt{a}$ (say by replacing it with $>$),
  then traverse the input and cross off final $\mathtt{b}$.
- Stop and accept if and when neither $\mathtt{a}$ nor $\mathtt{b}$
  are present for a new cycle.

Accepting a^3b^3

$\geq$**aaabbb**␣

Accepting a^3b^3

$\geq$**aaabbb**␣

Accepting a^3b^3

>aaabbb␣

Accepting a^3b^3

>≥aabbb␣

Accepting a^3b^3

>>aabbb␣

Accepting a^3b^3

>>a<u>a</u>bbb␣

Accepting a^3b^3

>>aabbb␣

Accepting a^3b^3

>>aabbb␣

Accepting a^3b^3

>>aabbb␣

Accepting a^3b^3

>>aabbb␣

Accepting a^3b^3

>>aabbb␣

Accepting a^3b^3

>>aabb␣␣

Accepting a^3b^3

>>aabb␣␣

Accepting a^3b^3

>>aabb␣␣

Accepting a^3b^3

>>a<u>a</u>bb␣␣

Accepting a^3b^3

>>aabb␣␣

Accepting a^3b^3

>≥aabb␣␣

Accepting a^3b^3

>>≥abb␣␣

Accepting a^3b^3

>>>abb␣␣

Accepting a^3b^3

>>>abb␣␣

Accepting a^3b^3

>>>abb␣␣

Accepting a^3b^3

>>>abb␣␣

Accepting a^3b^3

>>>abb␣␣

Accepting a^3b^3

>>>ab␣␣␣

Accepting a^3b^3

>>>ab␣␣␣

Accepting a^3b^3

>>>abuuu

Accepting a^3b^3

>>≥ab␣␣␣

Accepting a^3b^3

>>>abuuu

Accepting a^3b^3

>>>><b␣␣␣

Accepting a^3b^3

>>>>bUUU

Accepting a^3b^3

>>>>b␣␣␣

Accepting a^3b^3

>>>>bUUU

Accepting a^3b^3

>>>>␣␣␣␣

Accepting a^3b^3

>>>>⊔⊔⊔⊔

Implementing string overwrite

- A generalization of 2DFA: the ***on-site acceptor***, commonly known as ***LBA***.
- The new operation: overwrite a symbol by another.

  I.e. use the input for read/write memory. The components:

Implementing string overwrite

- A generalization of 2DFA: the ***on-site acceptor***, commonly known as ***LBA***.
- The new operation: overwrite a symbol by another.
  I.e. use the input for read/write memory. The components:
  - Basic alphabet Σ,
    additional symbols, including $>, \sqcup$ in extended alphabet Γ.
  - A finite set Q of ***states***.

    Two distinguished states: $s, a \in Q$, the ***start*** and ***accept*** states.
  - A transition partial-function:

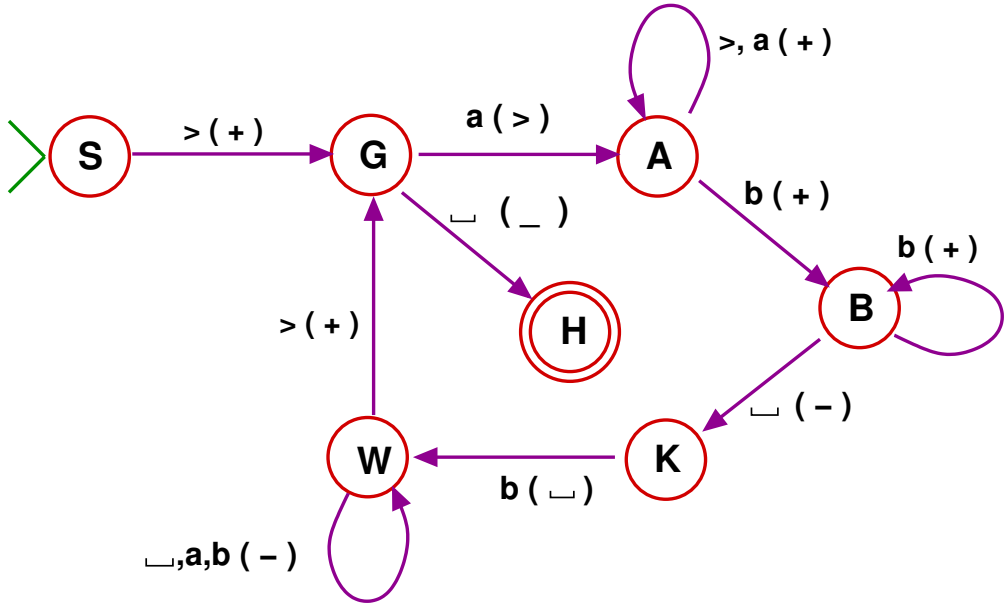
    $$\delta : \; Q \times \Gamma \rightharpoonup Q \times \mathbf{Act} \quad \text{where} \quad \mathbf{Act} = \{+, -\} \cup \Gamma.$$

Implementing string overwrite

- A generalization of 2DFA: the ***on-site acceptor***, commonly known as ***LBA***.
- The new operation: overwrite a symbol by another.
  I.e. use the input for read/write memory. The components:
  - Basic alphabet Σ,
    additional symbols, including $>, \sqcup$ in extended alphabet Γ.
  - A finite set Q of ***states***.

  Two distinguished states: $s, a \in Q$, the ***start*** and ***accept*** states.
  - A transition partial-function:
    $$\delta : \; Q \times \Gamma \rightharpoonup Q \times \mathbf{Act} \quad \text{where} \quad \mathbf{Act} = \{+, -\} \cup \Gamma.$$
- Action "γ" is the overwriting with $\gamma \in \Gamma$.
- We write (again) $q \xrightarrow{\sigma(\alpha)} p$ for
  $\delta(q, \sigma) = \langle p, \alpha \rangle$

An LBA for the crossing-off algorithm



LBA operation: Configurations

- The building stone is the configuration (cfg), just like 2DFA.
  Reminder:
- A ***cursored-string*** over Σ is a string over Σ
  with one symbol-position underlined.

LBA operation: Configurations

- The building stone is the configuration (cfg), just like 2DFA.
  Reminder:
- A ***cursored-string*** over Σ is a string over Σ
  with one symbol-position underlined.
- A ***configuration (cfg)*** is a pair $(q, \check{w})$ where
  - q is a state, and
  - $\check{w}$ is a cursored-string.

  e.g. $(\text{A}, >0101\underline{1}00\sqcup)$

LBA operation: Configurations

- The building stone is the configuration (cfg), just like 2DFA.
  Reminder:
- A ***cursored-string*** over Σ is a string over Σ
  with one symbol-position underlined.
- The ***initial cfg*** for w: $(s, \geq w \sqcup)$

LBA operation: Yield

- The ***Yield*** relation $\Rightarrow$ between configurations
  extends the Yield for 2DFAs:
  - If $q \xrightarrow{\gamma(+)} p$ then $(q, u\underline{\gamma}\tau v) \Rightarrow (p, u\gamma\underline{\tau}v)$
  - If $q \xrightarrow{\gamma(-)} p$ then $(q, u\tau\underline{\gamma}v) \Rightarrow (p, u\underline{\tau}\gamma v)$
  - ***NEW*** If $q \xrightarrow{\gamma(\tau)} p$ then $(q, u\underline{\gamma}v) \Rightarrow (p, u\underline{\tau}v)$
- ***What if τ and γ are the same?***

LBA operation: Traces and acceptance

- A cfg $c = (q, u\gamma v)$ is **terminal** if no rule applies.
- A cfg c is **accepting**
  if it is terminal and its state is the accepting state.

LBA operation: Traces and acceptance

- A cfg $c = (q, u\gamma v)$ is ***terminal*** if no rule applies.
- A cfg c is ***accepting***
  if it is terminal and its state is the accepting state.
- A ***terminating computation-trace of*** M ***for input*** w:

  $c_0 \Rightarrow c_1 \Rightarrow \cdots \Rightarrow c_n$

  where c_0 is initial for w and c_n is terminal.

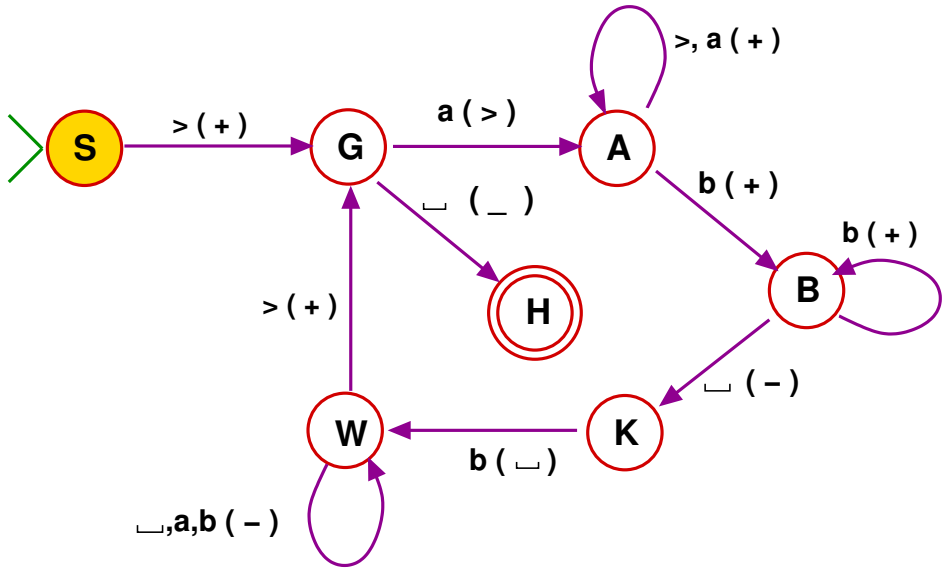
  The trace is ***accepting*** if c_n is accepting.

LBA operation: Traces and acceptance

- A cfg $c = (q, u\gamma v)$ is **terminal** if no rule applies.
- A cfg c is **accepting**
  if it is terminal and its state is the accepting state.
- M **accepts** $w \in \Sigma^*$ if there is an accepting trace for input w.
- The language **recognized** by M is
  $\mathcal{L}(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$

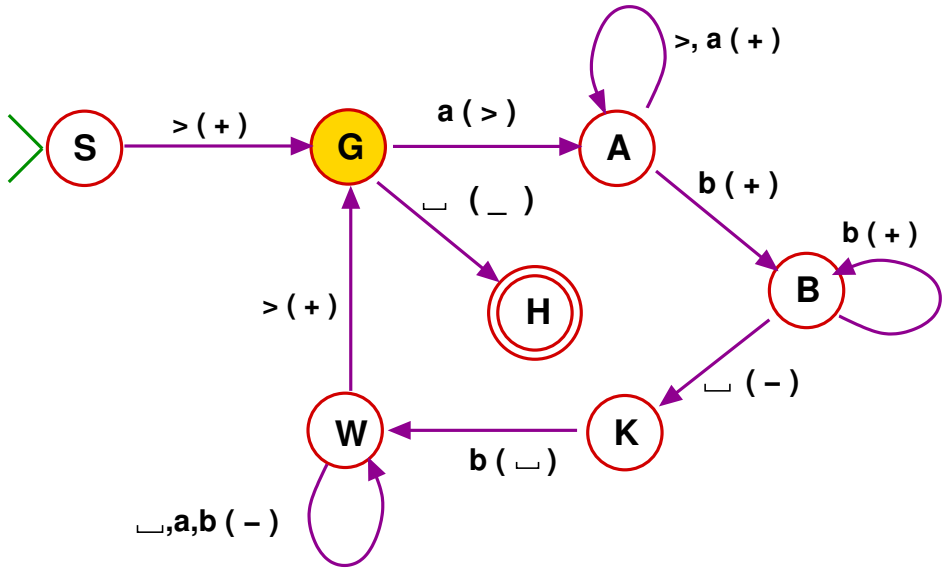
Example: Accepting trace for aabb

$(S, \geq \texttt{aabb}\sqcup)$



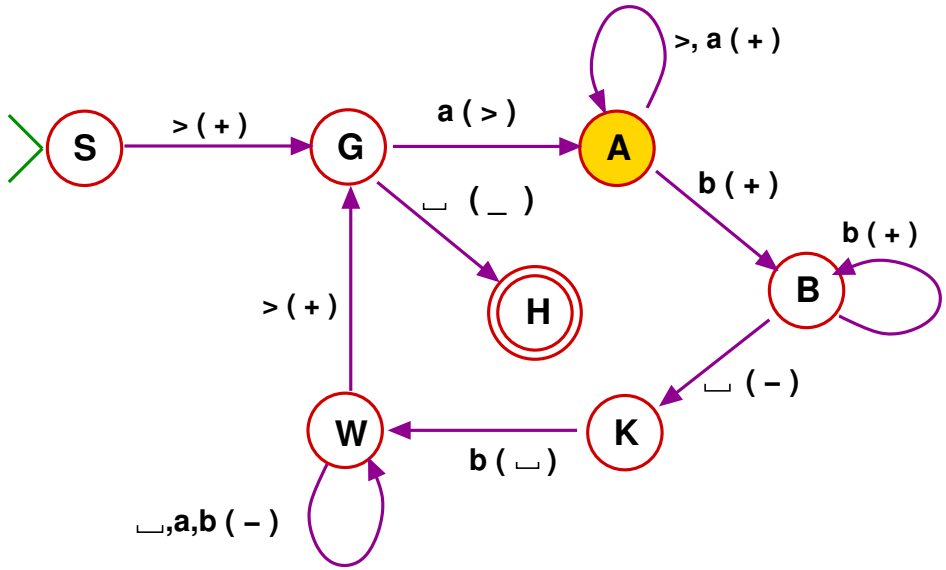
Example: Accepting trace for aabb

$(G, >\underline{a}abb\sqcup)$



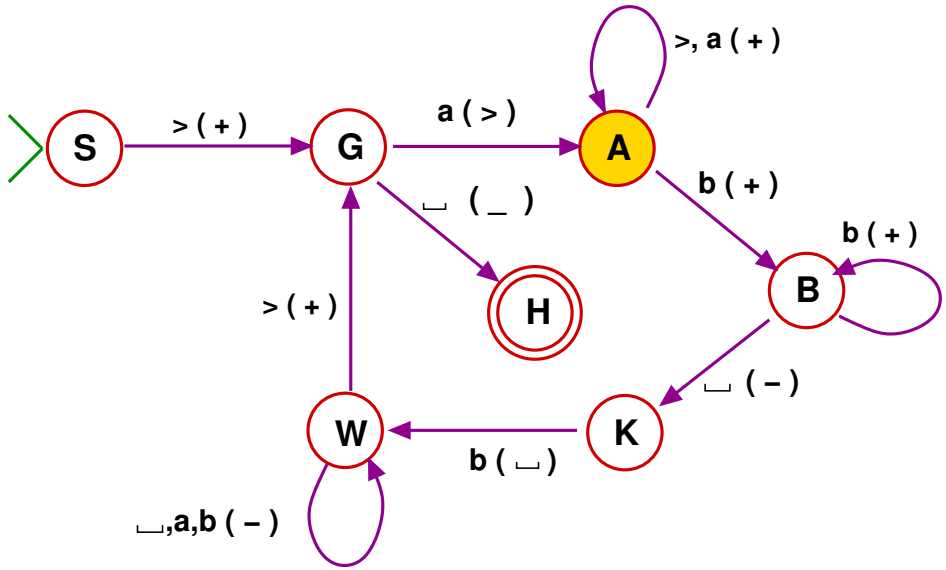
Example: Accepting trace for aabb

$(A, >\geq \texttt{abb}\sqcup)$



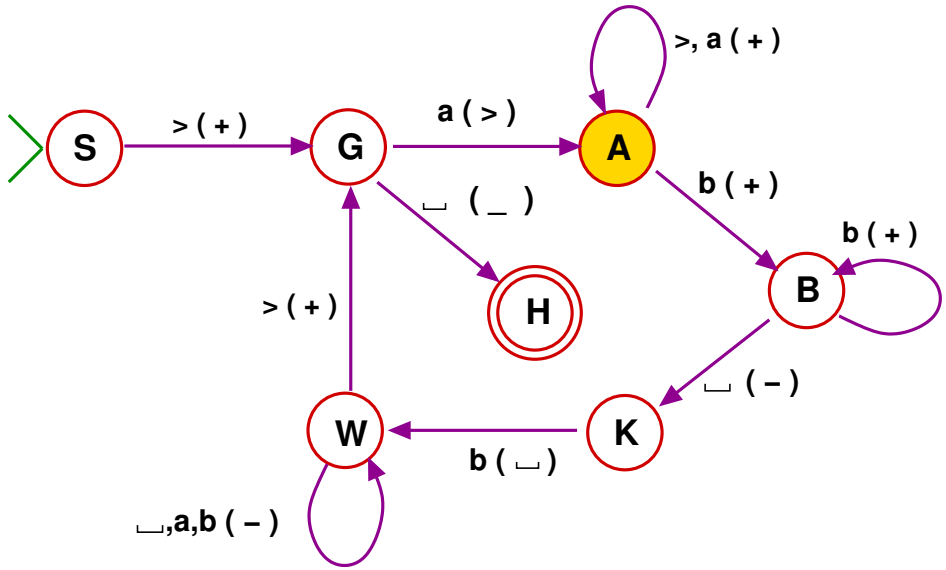
Example: Accepting trace for aabb

$(A, >>\underline{a}bb\sqcup)$



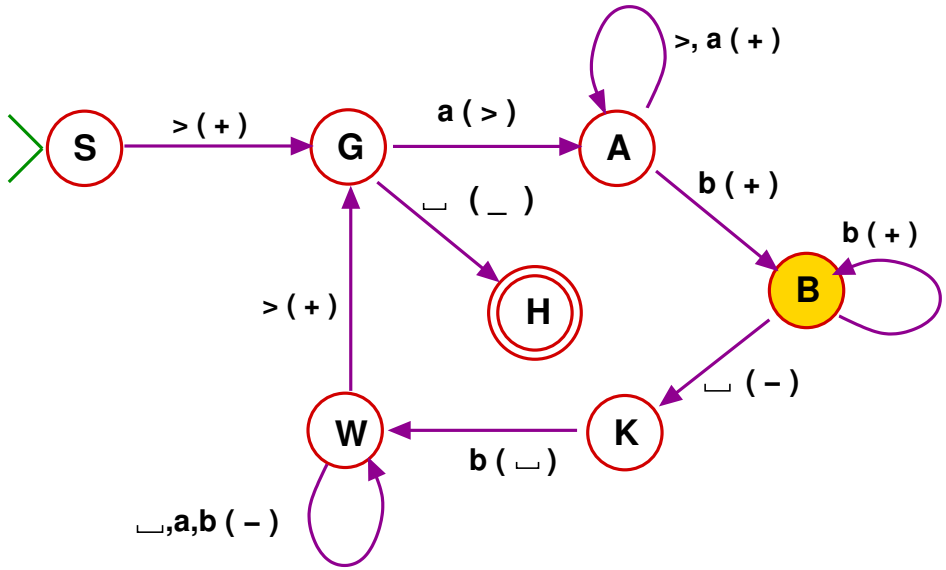
Example: Accepting trace for aabb

$(A, >>\underline{a}b\sqcup)$



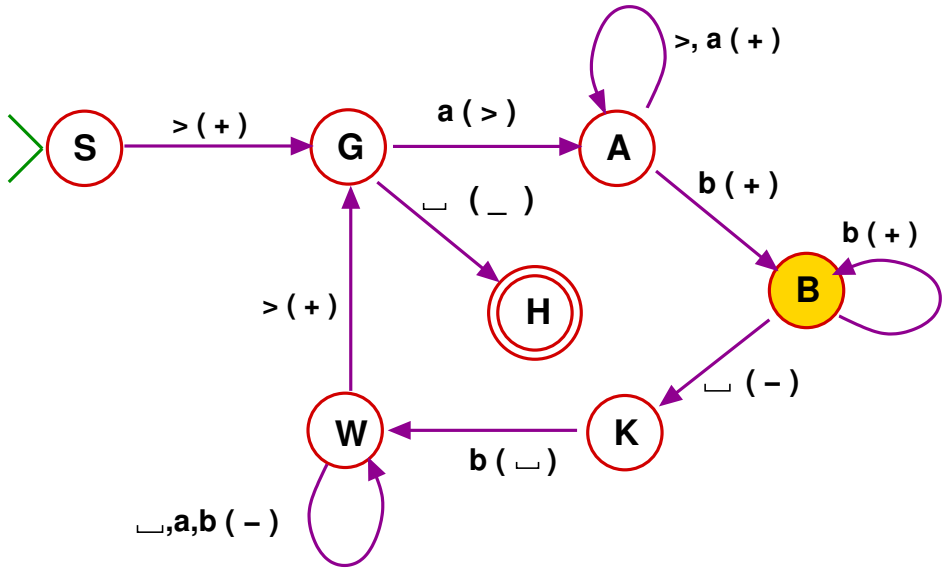
Example: Accepting trace for aabb

$(B, >>\texttt{abb}\sqcup)$



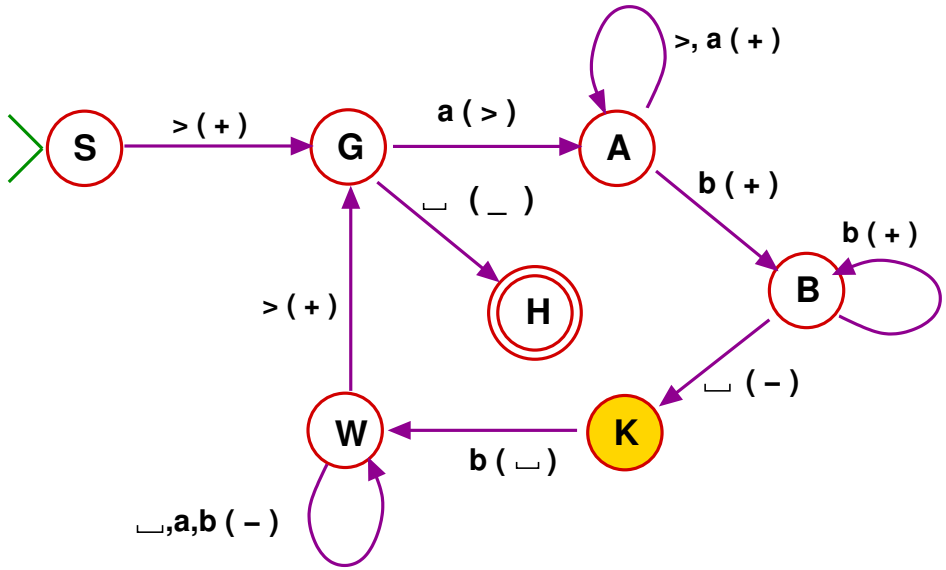
Example: Accepting trace for aabb

$(B, >>\texttt{abb}\sqcup)$

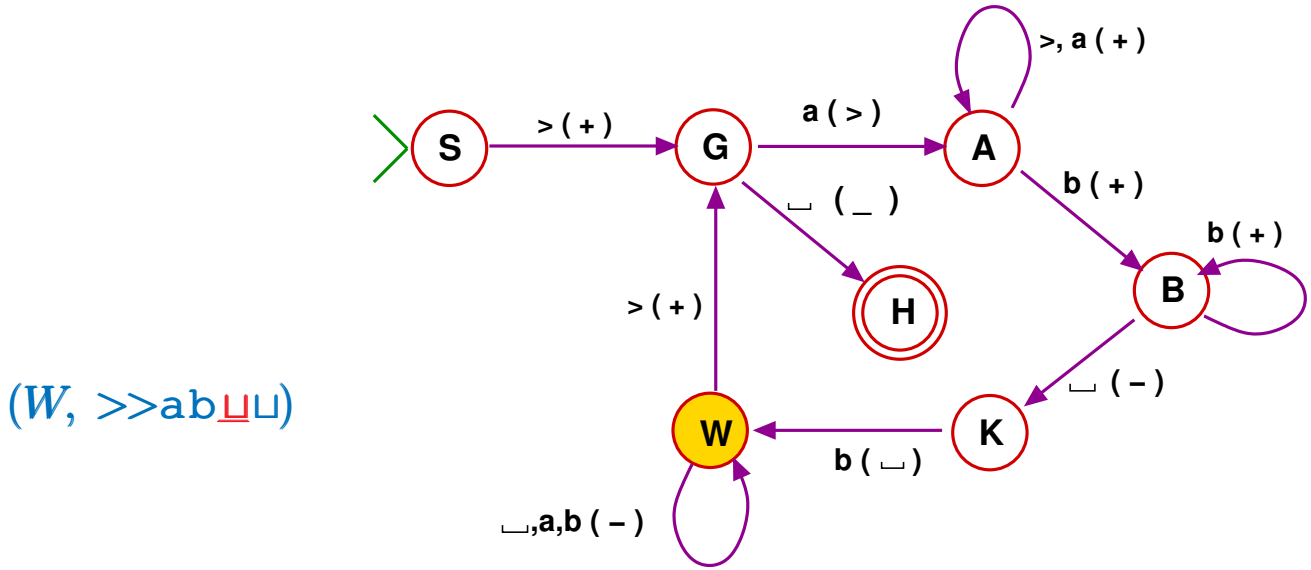


Example: Accepting trace for aabb

$(K, >>\mathtt{abb})$

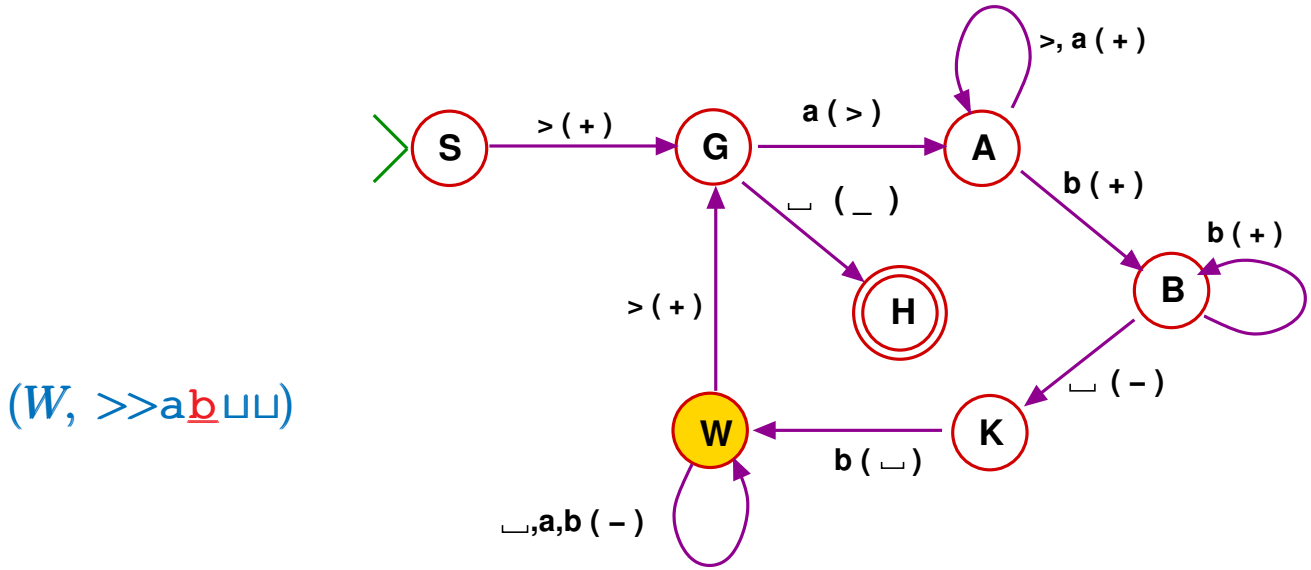


Example: Accepting trace for aabb



$(W, \texttt{>>ab␣␣})$

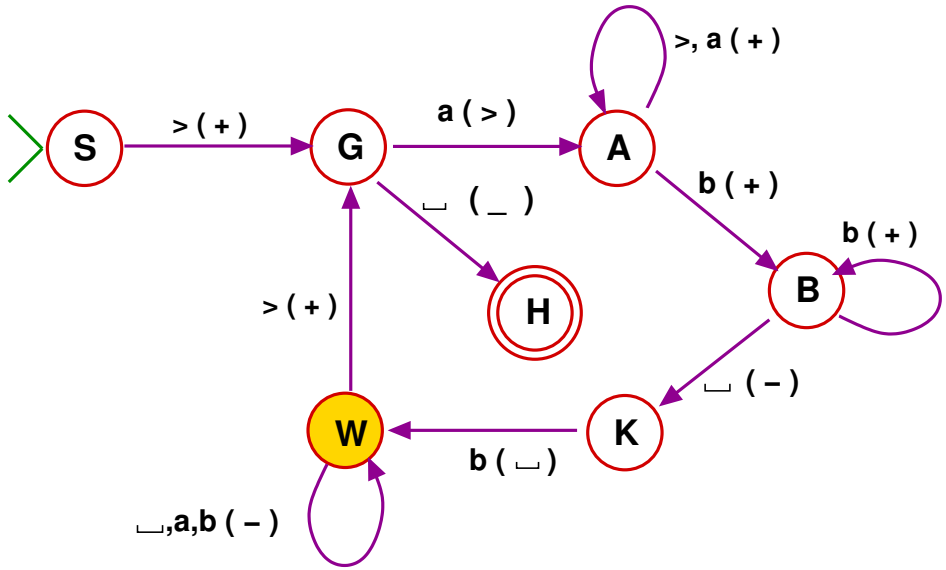
Example: Accepting trace for aabb



$(W,\ \texttt{>>a}\underline{\texttt{b}}\texttt{␣␣})$

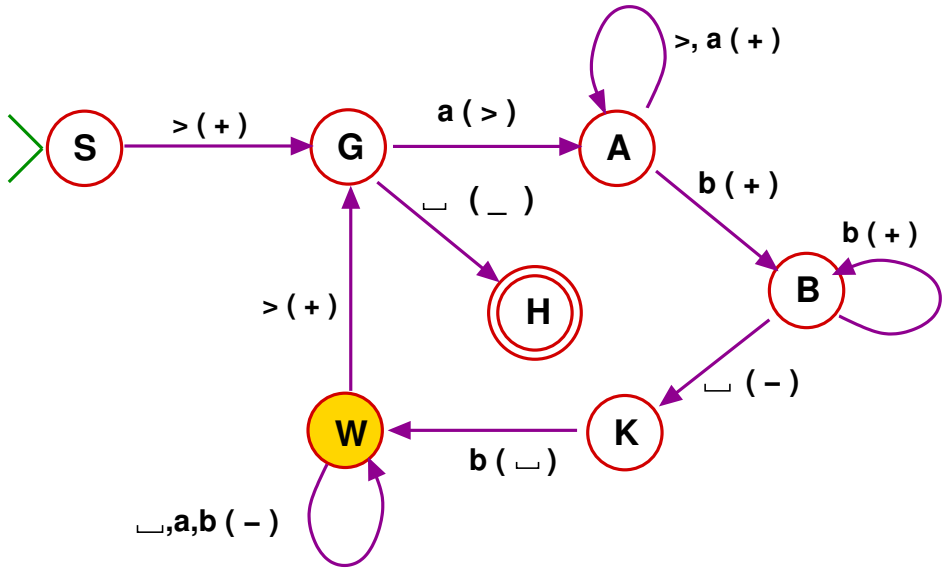
Example: Accepting trace for aabb

$(W, >>\underline{a}bUU)$



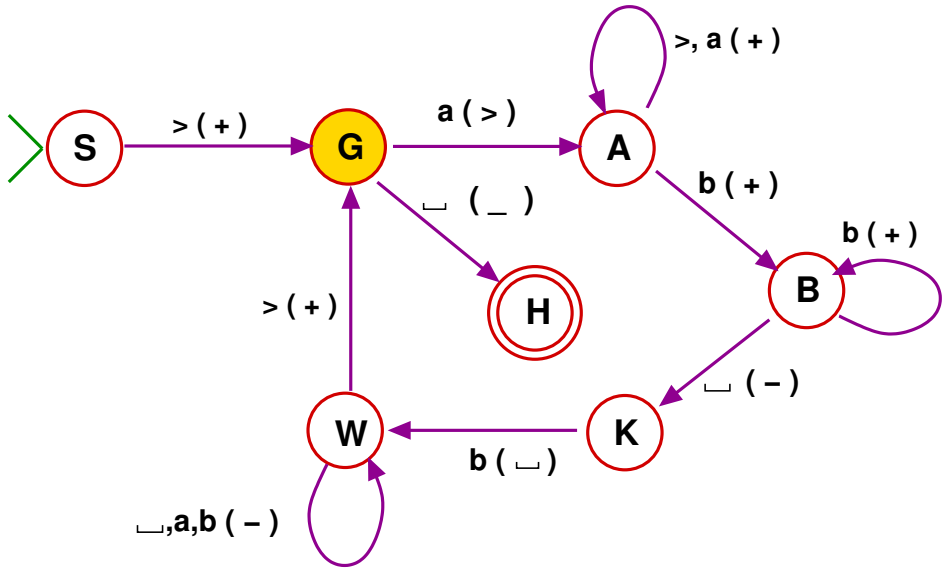
Example: Accepting trace for aabb

$(W, >\geq \texttt{ab}\sqcup\sqcup)$



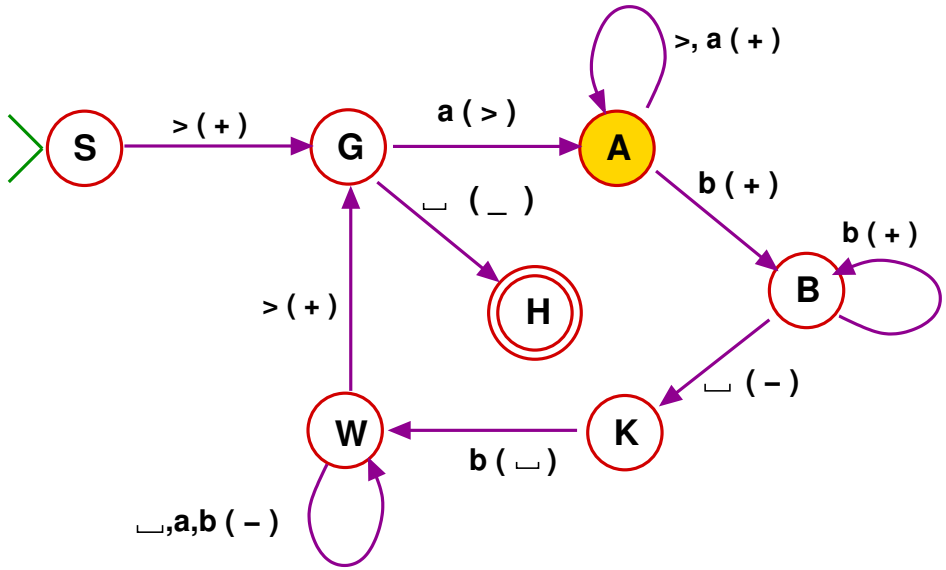
Example: Accepting trace for aabb

$(G,\ >>\underline{a}b\sqcup\sqcup)$



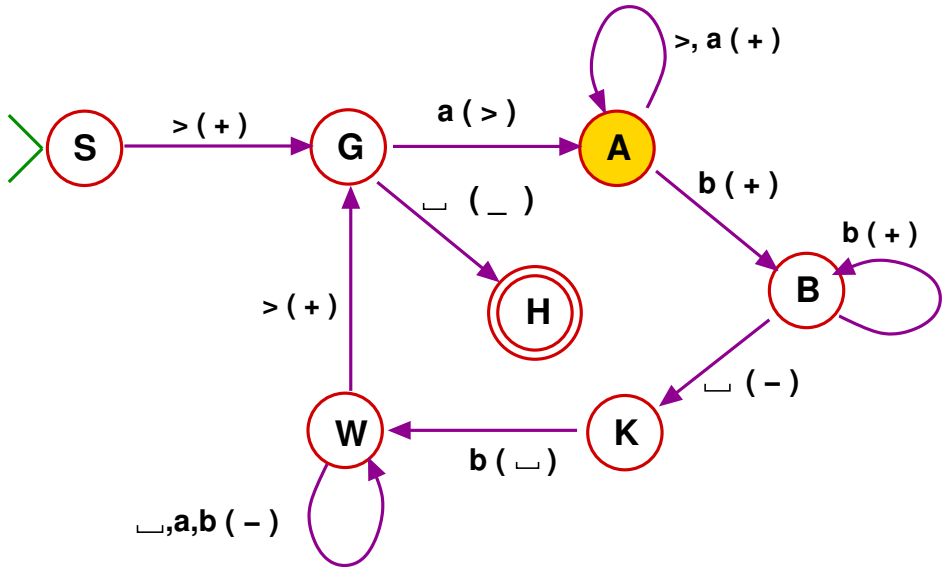
Example: Accepting trace for aabb

$(A, >>\geq b\sqcup\sqcup)$



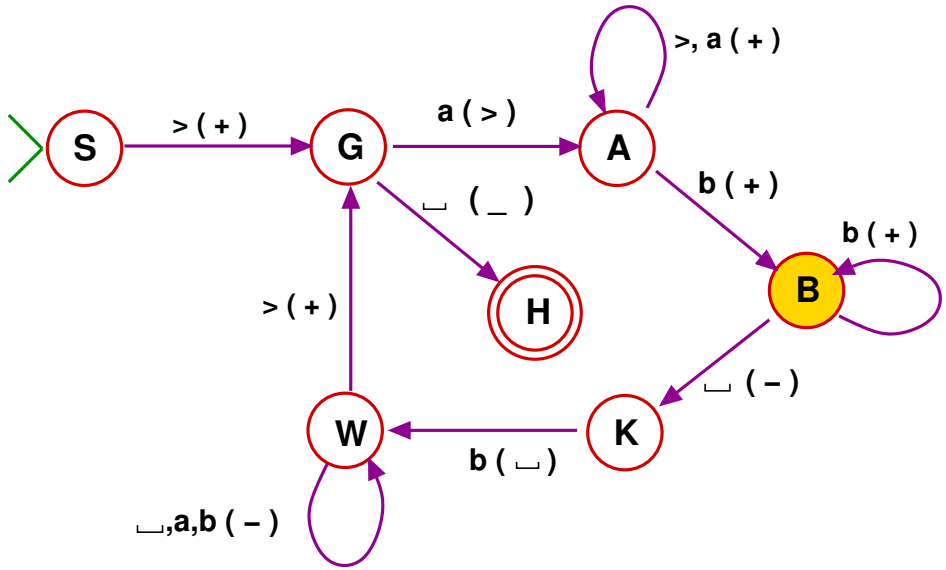
Example: Accepting trace for aabb

$(A,\ >>>\underline{b}\sqcup\sqcup)$



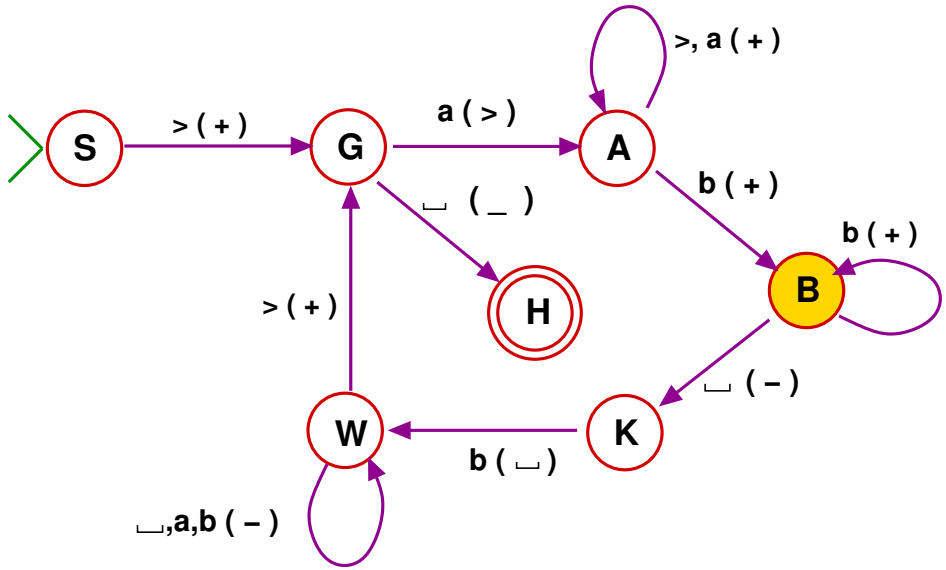
Example: Accepting trace for aabb

$(B,\ >>>\geq \sqcup\sqcup)$



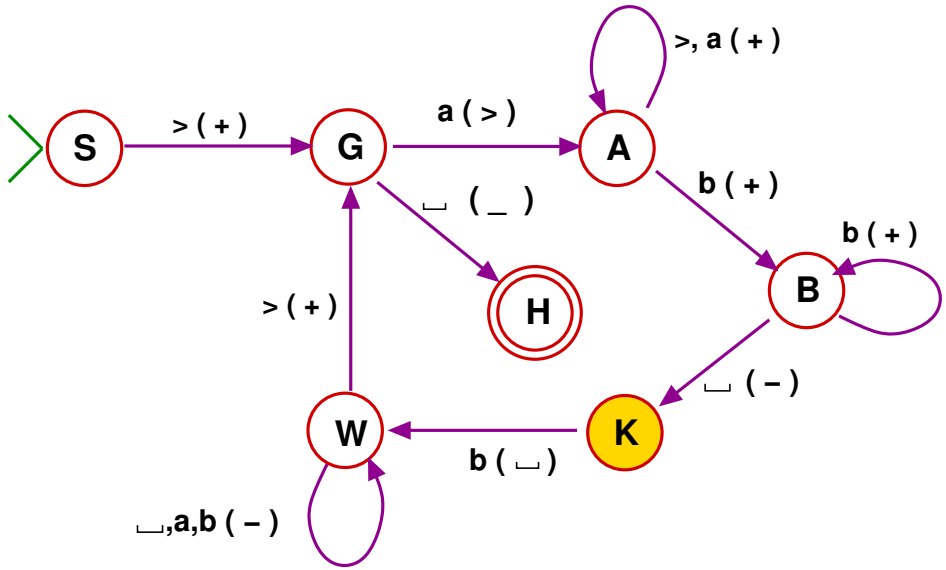
Example: Accepting trace for aabb

$(B, >>>\sqcup\sqcup)$

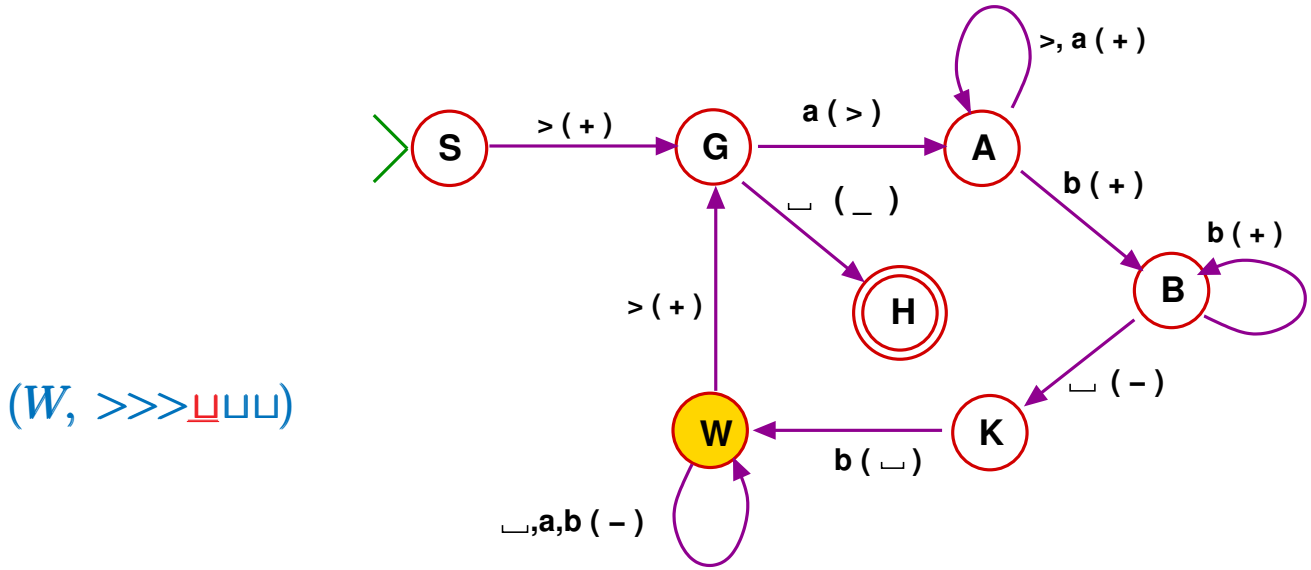


Example: Accepting trace for aabb

$(K,\ >>>\underline{b}\sqcup\sqcup)$



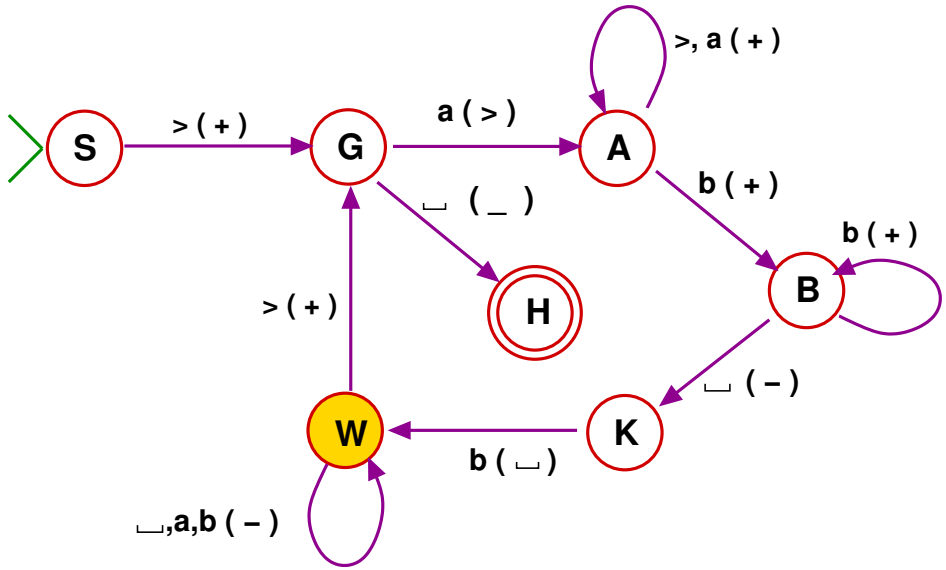
Example: Accepting trace for aabb



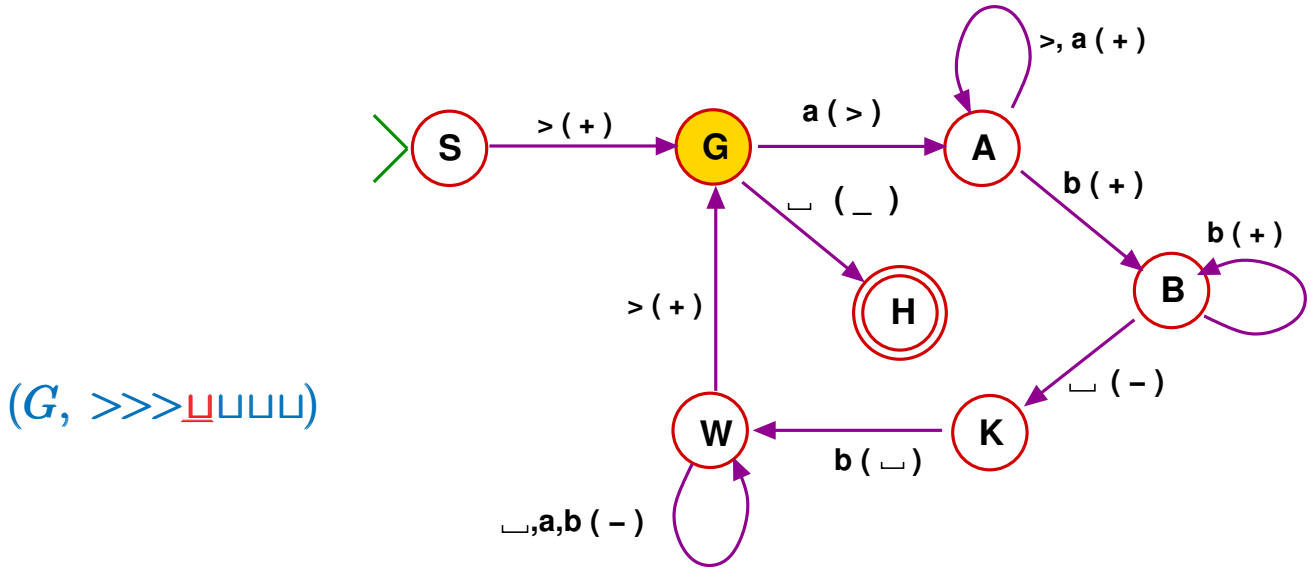
$(W,\ >>>\underline{\sqcup}\sqcup\sqcup)$

Example: Accepting trace for aabb

$(W,\ >>\geq \sqcup\sqcup\sqcup)$



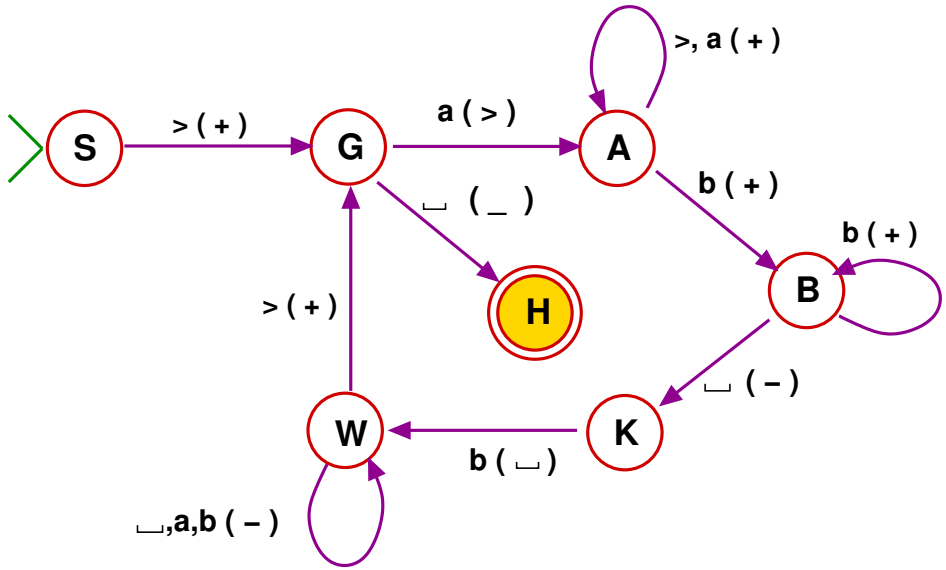
Example: Accepting trace for aabb



$(G,\ >>>\underline{\sqcup}\sqcup\sqcup\sqcup)$

Example: Accepting trace for aabb

$(H,\ >>>\sqcup\sqcup\sqcup\sqcup)$



MEMORY UNLEASHED

Knocking off the wall

- Devise an acceptor for those $w \in \{0,1\}^*$
  that are binary numerals for prime numbers.
- Intuitively clear that no algorithm can be on-site.

Knocking off the wall

- Devise an acceptor for those $w \in \{0,1\}^*$
  that are binary numerals for prime numbers.
- Intuitively clear that no algorithm can be on-site.
- An additional feature: claim new space.
- Same definition as on-site acceptors,
  but different semantic for step-forward:

Knocking off the wall

- Devise an acceptor for those $w \in \{0,1\}^*$
  that are binary numerals for prime numbers.
- Intuitively clear that no algorithm can be on-site.
- An additional feature: claim new space.
- Same definition as on-site acceptors,
  but different semantic for step-forward:
  - If $q \overset{\gamma(+)}{\rightarrow} p$ then $(q, u\underline{\gamma})$ implies $(p, u\gamma\underline{\sqcup})$

Knocking off the wall

- Devise an acceptor for those $w \in \{0,1\}^*$
  that are binary numerals for prime numbers.
- Intuitively clear that no algorithm can be on-site.
- An additional feature: claim new space.
- Same definition as on-site acceptors,
  but different semantic for step-forward:
  - If $q \overset{\gamma(+)}{\to} p$ then $(q, u\underline{\gamma})$ implies $(p, u\gamma\underline{\sqcup})$
- The machine appropriates new memory location
  and by overwrite can fill it with whatever it wants!

Knocking off the wall

- Devise an acceptor for those $w \in \{0,1\}^*$
  that are binary numerals for prime numbers.
- Intuitively clear that no algorithm can be on-site.
- An additional feature: claim new space.
- Same definition as on-site acceptors,
  but different semantic for step-forward:
  - If $q \overset{\gamma(+)}{\to} p$ then $(q, u\underline{\gamma})$ implies $(p, u\gamma\underline{\sqcup})$
- The machine appropriates new memory location
  and by overwrite can fill it with whatever it wants!
- This computation model is the **Turing acceptor**.

How Turing acceptors compute

- A cfg $c = (q, u\gamma v)$ is **terminal** if no transition applies.
- A terminal cfg c is **accepting** if its state is a.

How Turing acceptors compute

- A cfg $c = (q, u\gamma v)$ is **terminal** if no transition applies.
- A terminal cfg c is **accepting** if its state is a.
- A **computation-trace** of M for input w:

  $c_0 \Rightarrow c_1 \Rightarrow \cdots \Rightarrow c_n$

  where c_0 is initial for w and c_n is terminal.

How Turing acceptors compute

- A cfg $c = (q, u\gamma v)$ is **terminal** if no transition applies.
- A terminal cfg c is **accepting** if its state is a.
- A **computation-trace** of M for input w:

  $c_0 \Rightarrow c_1 \Rightarrow \cdots \Rightarrow c_n$

  where c_0 is initial for w and c_n is terminal.
- The trace is **accepting** if its terminal cfg is accepting.
- M **accepts** $w \in \Sigma^*$ if

  there is an accepting trace for input w.

How Turing acceptors compute

- A cfg $c = (q, u\gamma v)$ is **terminal** if no transition applies.
- A terminal cfg c is **accepting** if its state is a.
- A **computation-trace** of M for input w:

  $c_0 \Rightarrow c_1 \Rightarrow \cdots \Rightarrow c_n$

  where c_0 is initial for w and c_n is terminal.
- The trace is **accepting** if its terminal cfg is accepting.
- M **accepts** $w \in \Sigma^*$ if

  there is an accepting trace for input w.
- The language **recognized** by M is

  $\mathcal{L}(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$

Turing acceptors in the broader picture

- All our previous (deterministic) acceptors are special cases of Turing acceptors:

Turing acceptors in the broader picture

- All our previous (deterministic) acceptors are special cases of Turing acceptors:
  - **DFAs:** Only action is step-on.

Turing acceptors in the broader picture

- All our previous (deterministic) acceptors are special cases of Turing acceptors:
  - **DFAs:** Only action is step-on.
  - **2DFA:** Backward stepping permitted.

Turing acceptors in the broader picture

- All our previous (deterministic) acceptors are special cases of Turing acceptors:
  - **DFAs:** Only action is step-on.
  - **2DFA:** Backward stepping permitted.
  - **LBA:** Add overwriting.

Turing acceptors in the broader picture

- All our previous (deterministic) acceptors are special cases of Turing acceptors:
  - **DFAs:** Only action is step-on.
  - **2DFA:** Backward stepping permitted.
  - **LBA:** Add overwriting.
  - **Turing acceptors:** Dynamic computing space

    $q \xrightarrow{\sqcup(+)} p$ works at strings'-end.

Turing acceptors in the broader picture

- We can add to the Turing acceptors useful components.

Turing acceptors in the broader picture

- We can add to the Turing acceptors useful components.
  - **Nondeterministic Turing acceptors**
  - **Multi-string**

    Useful! Consider recognizing palindromes.
  - **Multi-cursors**
  - A plethora of programming constructs.

Turing acceptors in the broader picture

- We can add to the Turing acceptors useful components.
  - **Nondeterministic Turing acceptors**
  - **Multi-string**

    Useful! Consider recognizing palindromes.
  - **Multi-cursors**
  - A plethora of programming constructs.
- These are all hugely useful,

  improving efficiency, transparency, expressiveness, verification

Turing acceptors in the broader picture

- We can add to the Turing acceptors useful components.
  - **Nondeterministic Turing acceptors**
  - **Multi-string**

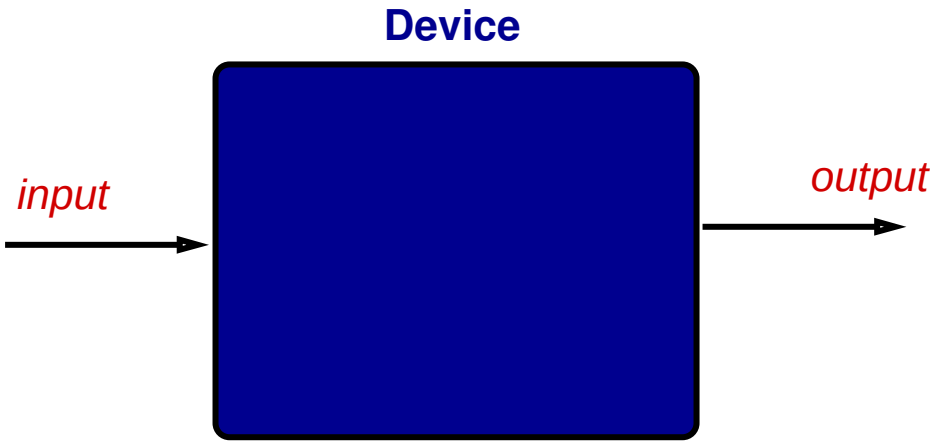
    Useful! Consider recognizing palindromes.
  - **Multi-cursors**
  - A plethora of programming constructs.
- These are all hugely useful,

  improving efficiency, transparency, expressiveness, verification
- But they do not yield new recognized languages!

  To be discussed later...

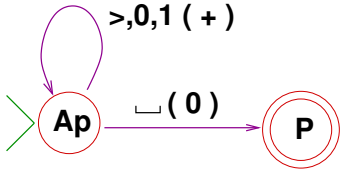
Transducers

a transducer



Turing transducers

- A Turing transducer over Σ
  computes a partial function $f : \Sigma^* \rightharpoonup \Sigma^*$.
- Example: append a 0 to the input:



Definition of Turing transducers

- A **Turing transducer** M over alphabet Σ
  is like a Turing acceptor over Σ except that: the accepting state becomes a ***print state*** (p).

Definition of Turing transducers

- A **Turing transducer** M over alphabet Σ
  is like a Turing acceptor over Σ except that: the accepting state becomes a ***print state*** (p).
- Stipulations:
  - The print state is terminal: no outgoing transitions.
  - Trailing blanks in a print configuration
    are not part of the output:

    If $(s, >w) \Rightarrow^* (p, >u\sqcup^k)$ then u is the **output** of M for input w.

Definition of Turing transducers

- A **Turing transducer** M over alphabet Σ
  is like a Turing acceptor over Σ except that: the accepting state becomes a ***print state*** (p).
- Stipulations:
  - The print state is terminal: no outgoing transitions.
  - Trailing blanks in a print configuration
    are not part of the output:
    If $(s, >w) \Rightarrow^* (p, >u\sqcup^k)$ then u is the **output** of M for input w.
- Trailing $\sqcup$s are unavoidable:
  Turing transducers cannot erase any part of the space they use.

Definition of Turing transducers

- A **Turing transducer** M over alphabet Σ
  is like a Turing acceptor over Σ except that: the accepting state becomes a ***print state*** (p).
- Stipulations:
  - The print state is terminal: no outgoing transitions.
  - Trailing blanks in a print configuration
    are not part of the output:
    If $(s, >w) \Rightarrow^* (p, >u\sqcup^k)$ then u is the **output** of M for input w.
- Trailing $\sqcup$s are unavoidable:
  Turing transducers cannot erase any part of the space they use.
- A transducer M **computes the partial function** $f : \Sigma^* \rightharpoonup \Sigma^*$ when

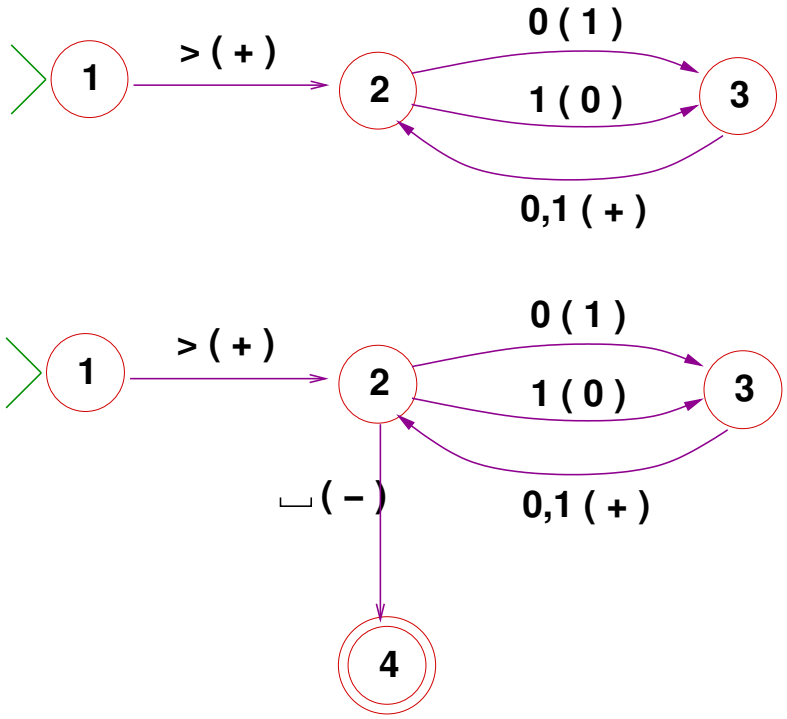
$$f(w) = u \quad \text{IFF} \quad (S, >w) \Rightarrow^* (P, >u\sqcup^k) \quad \text{for some } k \geqslant 0$$

- We say then that f is **Turing-computable**

EXAMPLES OF TURING MACHINES

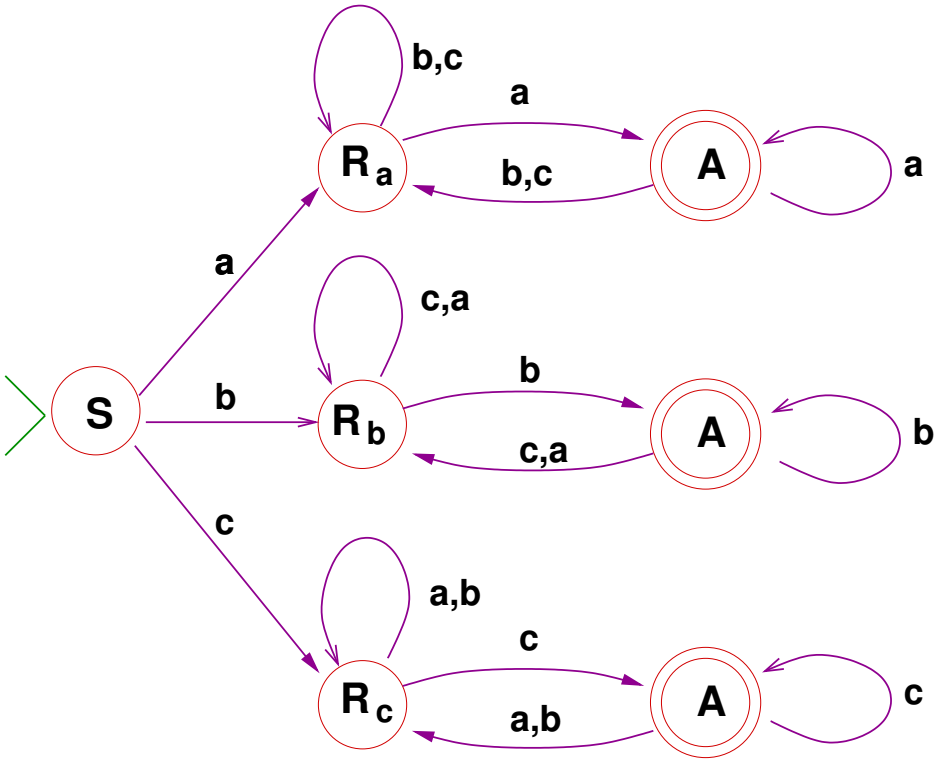
Example: The flip function

- $flip(0010) = 1101$
- Building the Turing transducer:

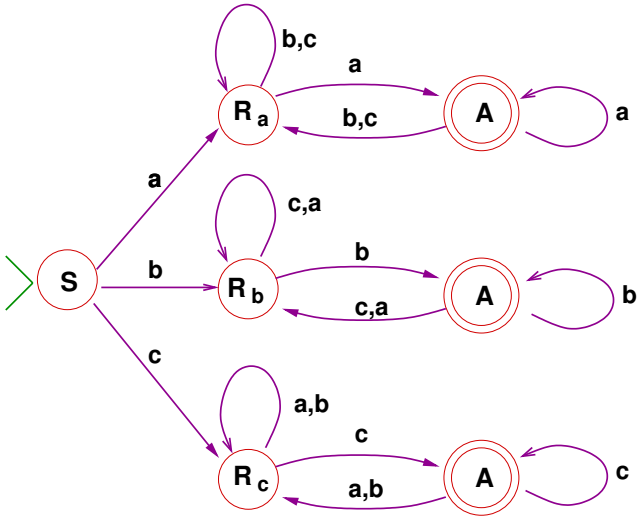


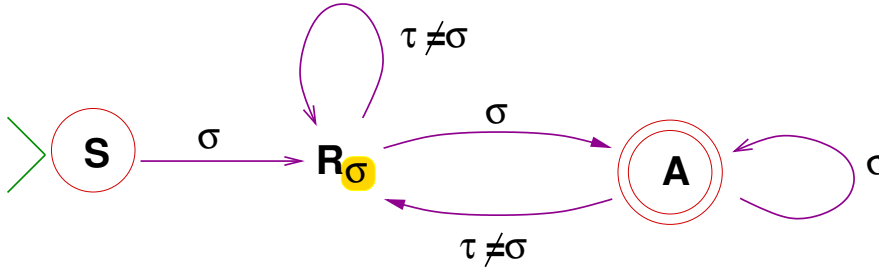
Modular presentation of transition diagrams

Consider a DFA recognizing *"first=last":*



Parameterizing states

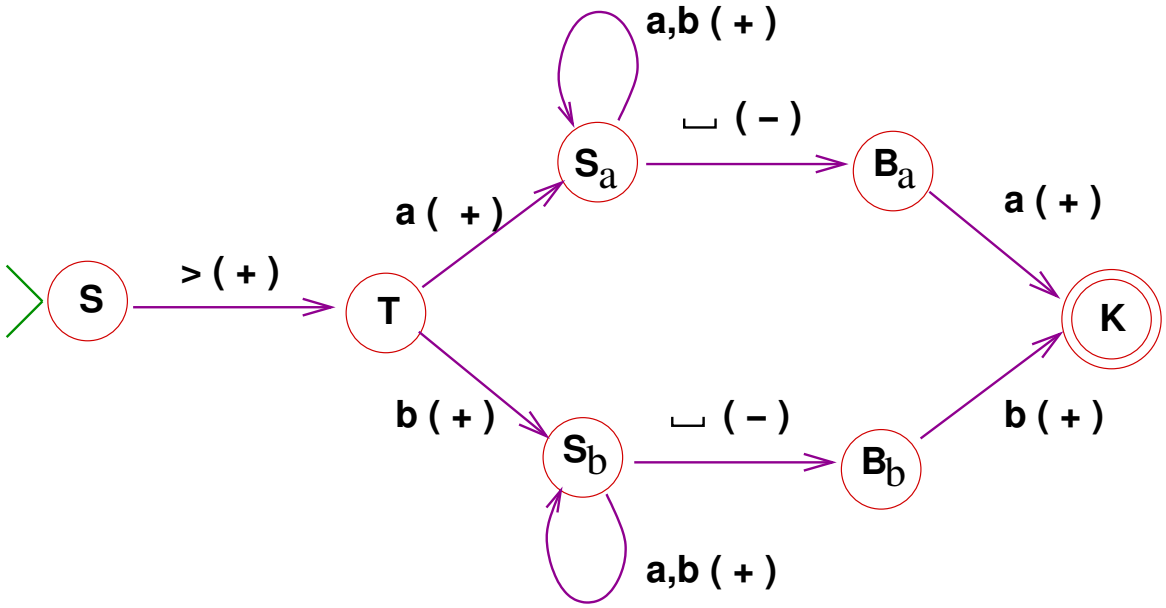


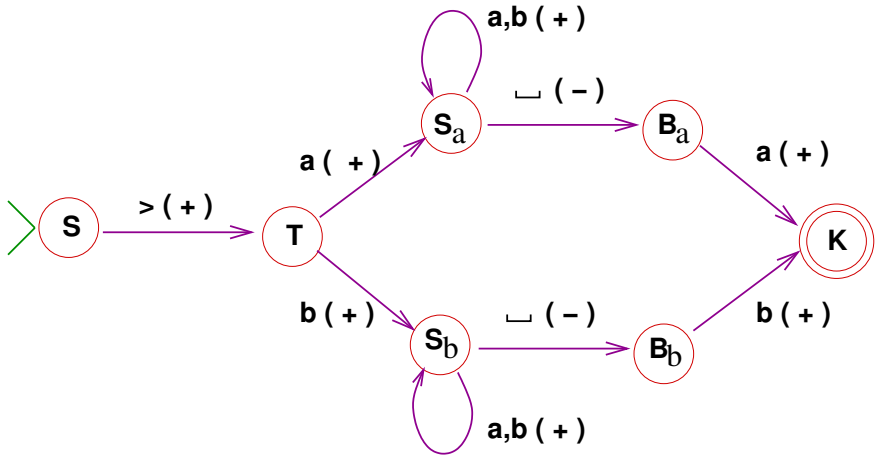


- The ***states*** are parameterized!

Same approach for a 2-way automaton

- Accept $w \in \{a,b\}^+$ iff first and last symbols are the same.
- A two-way automaton:





- Sometimes useful to forego the transition diagram and use a modular ***textual*** representation of δ.

$$
\begin{array}{lcll}
S & \overset{>(+)}{\rightarrow} & T & \\
T & \overset{\sigma(+)}{\rightarrow} & S_\sigma & \sigma = \mathtt{a},\mathtt{b} \\
S_\sigma & \overset{\tau(+)}{\rightarrow} & S_\sigma & \sigma,\tau = \mathtt{a},\mathtt{b} \\
S_\sigma & \overset{\text{␣}(-)}{\rightarrow} & B_\sigma & \sigma,\tau = \mathtt{a},\mathtt{b} \\
B_\sigma & \overset{\sigma(+)}{\rightarrow} & K & \sigma,\tau = \mathtt{a},\mathtt{b}
\end{array}
$$

Another example: Insert $\mathtt{1}$

- Map w to $\mathtt{1}w$.

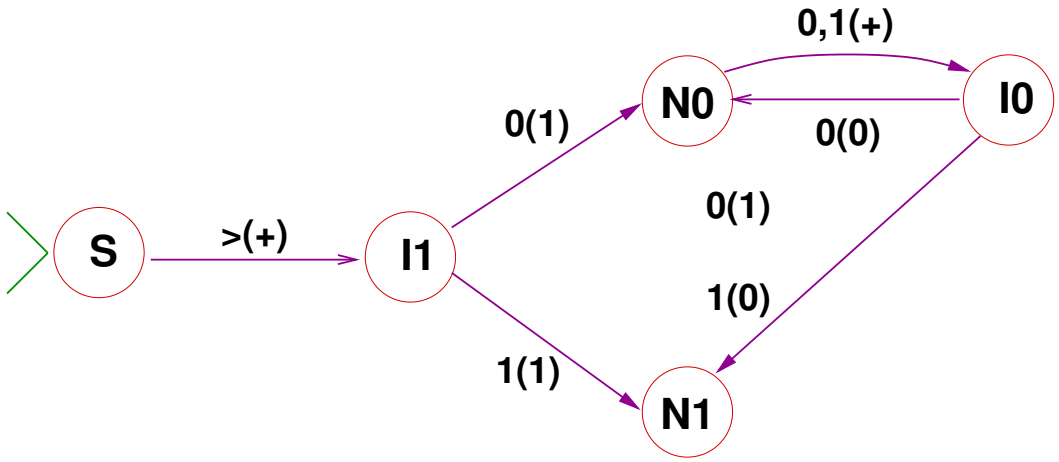
  Example: $f(1001) = 11001$

- Algorithm?

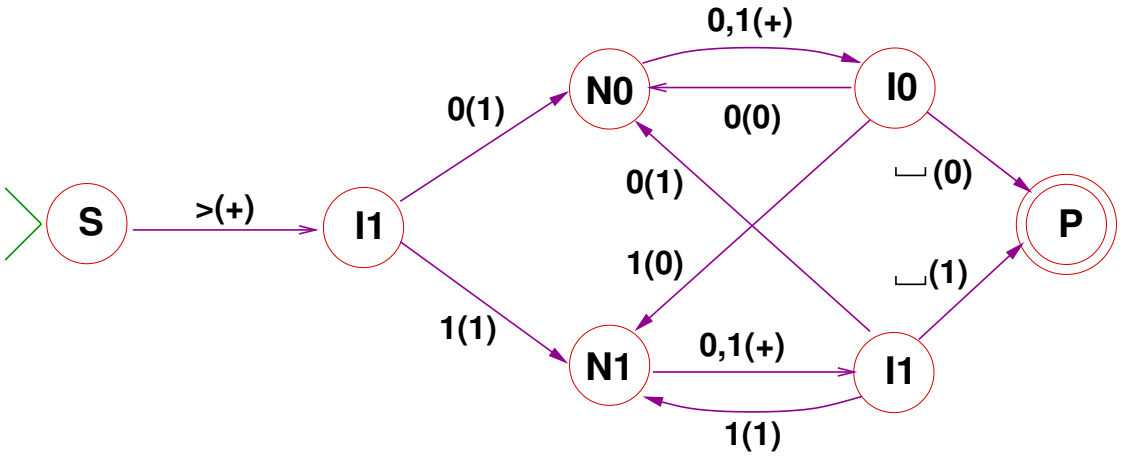
Another example: Insert 1

- For $w = \sigma_1 \cdots \sigma_{17}$:
  - 1 replaces σ_1
  - then σ_1 replaces σ_2
  - then σ_2 replaces σ_3 etc.

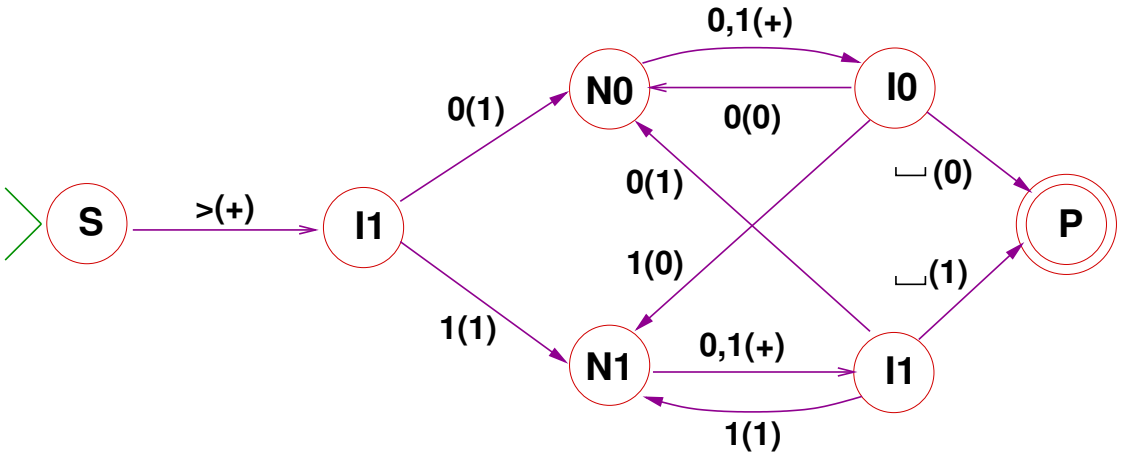
Another example: Insert 1



Another example: Insert 1



Another example: Insert 1



What if the alphabet is alphanumeric (36 symbols)?

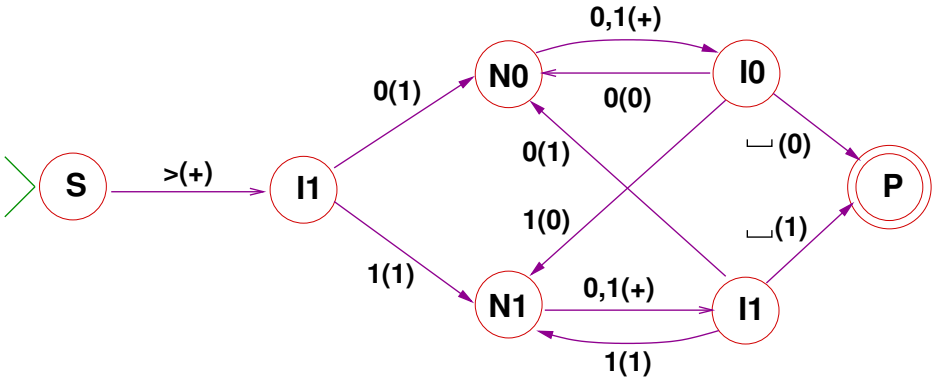
IT'S A MESS!

- We had labeled rules: $q \xrightarrow{\sigma} p$
- Here we need also coordinated labels for ***transitions***:

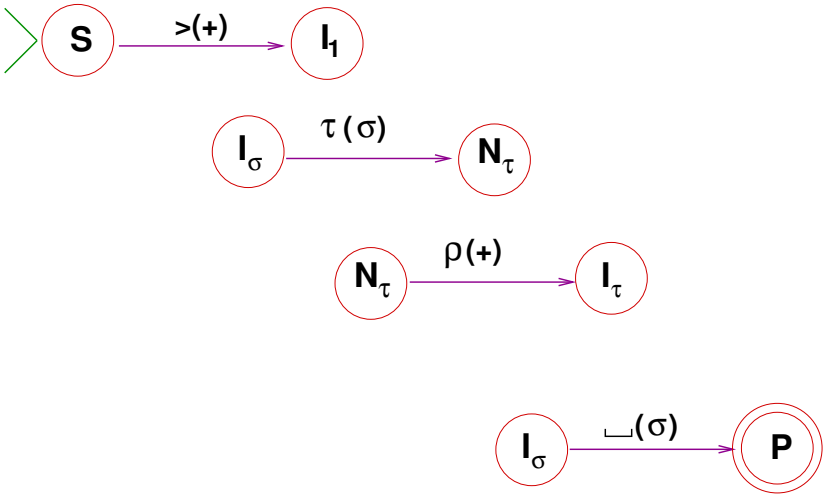
$$A_\sigma \xrightarrow{\sigma} B_\sigma$$

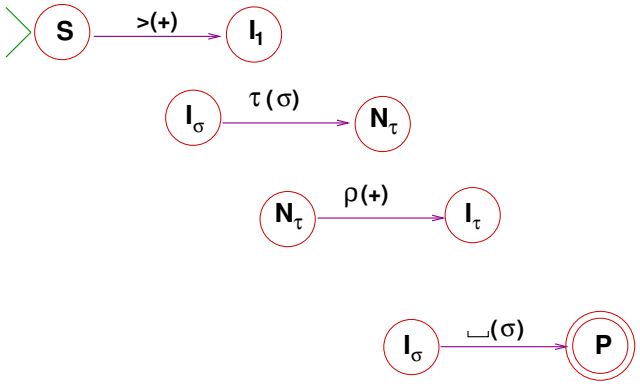
Modular presentation for inserting 1

Describe



by listing modular rules:





- Separate modules may be needed,
  to allow separate use of parameters.
- In textual format:

$$
\begin{array}{lcll}
S & \overset{>(+)}{\rightarrow} & I_1 & \\
I_\sigma & \overset{\tau(\sigma)}{\rightarrow} & N_\tau & \sigma, \tau \in \Sigma \\
N_\tau & \overset{\rho(+)}{\rightarrow} & I_\tau & \tau, \rho \in \Sigma \\
I_\sigma & \overset{\sqcup(\sigma)}{\rightarrow} & P & \sigma \in \Sigma
\end{array}
$$

Example: insert `1` *after second $*

- Input `0110$101$0$011`,
  output `0110$101$10$011`
- No second $ $\Longrightarrow$ *output = input*

$$
\begin{array}{lll}
S & \overset{>\,(+)}{\rightarrow} & D_1 \\
D_1 & \overset{\sigma\,(+)}{\rightarrow} & D_1 \quad (\sigma \neq \$) \\
D_1 & \overset{\$\,(+)}{\rightarrow} & D_2 \quad (\sigma \neq \$) \\
D_2 & \overset{\sigma\,(+)}{\rightarrow} & D_2 \quad (\sigma \neq \$) \\
D_2 & \overset{\$\,(+)}{\rightarrow} & I_1
\end{array}
$$

- Then proceed as for inserting 1 initially.

Example: Insert two symbols

- Insert `ab` at the head of the input.
- Algorithm: Cascade insertions of 2-symbol strings:

$$
\begin{array}{lcll}
S & \overset{>(+)}{\rightarrow} & I_{\text{abc}} & \\
I_{\sigma\tau\rho} & \overset{\tau(\sigma)}{\rightarrow} & N_\tau & \sigma, \tau \in \Sigma \\
N_\tau & \overset{\rho(+)}{\rightarrow} & I_\tau & \tau, \rho \in \Sigma \\
I_\sigma & \overset{\sqcup(\sigma)}{\rightarrow} & R & \sigma \in \Sigma \\
R & \overset{\tau(-)}{\rightarrow} & R & \tau \in \Sigma \\
R & \overset{>(>)}{\rightarrow} & P &
\end{array}
$$

Additional examples of Turing transducers:

Duplicate the first `b`

- Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$.
  $f : \Sigma^* \to \Sigma^*$ duplicates the first `b` if any.
- Example: $f(\mathtt{aabab}) = \mathtt{aabbab}$,
  $f(\mathtt{aa}) = \mathtt{aa}$.
- Construct a Turing transducer that computes f.
- Start state S, print state P.
- Transitions:

$$
\begin{array}{lll}
S & \overset{>\,(+)}{\longrightarrow} & A \\
A & \overset{a\,(+)}{\longrightarrow} & A \\
A & \overset{b\,(+)}{\longrightarrow} & I_b \\
A & \overset{\sqcup\,(-)}{\longrightarrow} & R \\
I_\sigma & \overset{\tau\,(\sigma)}{\longrightarrow} & N_\tau \quad \sigma,\tau = \mathtt{a},\mathtt{b} \\
N_\sigma & \overset{\tau\,(+)}{\longrightarrow} & I_\sigma \quad \sigma,\tau = \mathtt{a},\mathtt{b} \\
I_\sigma & \overset{\sqcup\,(\sigma)}{\longrightarrow} & R
\end{array}
$$

A computation trace

$$
\begin{array}{lcll}
S & \overset{>\,(+)}{\rightarrow} & A & \\
A & \overset{a\,(+)}{\rightarrow} & A & \\
A & \overset{b\,(+)}{\rightarrow} & I_b & \\
A & \overset{\sqcup\,(-)}{\rightarrow} & P & \\
I_\sigma & \overset{\tau\,(\sigma)}{\rightarrow} & N_\tau & \sigma, \tau = \mathtt{a}, \mathtt{b} \\
N_\sigma & \overset{\tau\,(+)}{\rightarrow} & I_\sigma & \sigma, \tau = \mathtt{a}, \mathtt{b} \\
I_\sigma & \overset{\sqcup\,(\sigma)}{\rightarrow} & P &
\end{array}
$$

Computation traces for input **baba**:

$(S, \underline{\geq}\texttt{baba})$

$\Rightarrow\ (A, >\underline{\texttt{b}}\texttt{aba})$

$\Rightarrow\ (I_b, >\texttt{b}\underline{\texttt{a}}\texttt{ba})$

$\Rightarrow\ (N_a, >\texttt{bb}\underline{\texttt{b}}\texttt{a})$

$\Rightarrow\ (I_a, >\texttt{bbb}\underline{\texttt{b}}\texttt{a})$

$\Rightarrow\ (N_b, >\texttt{bb}\underline{\texttt{a}}\texttt{a})$

$\Rightarrow\ (I_b, >\texttt{bba}\underline{\texttt{a}})$

$\Rightarrow\ (N_a, >\texttt{bba}\underline{\texttt{b}})$

$\Rightarrow\ (I_a, >\texttt{bbab}\underline{\sqcup})$

$\Rightarrow\ (P, >\texttt{babb}\underline{\texttt{a}})$

A failure trace

- Good practice:

  Check (some) exceptions, failures, borderline cases

- Check the transducer for input **aa**.

$$
\begin{array}{ll}
(S, \geq \mathtt{aa}) & \Rightarrow\ (R, > \mathtt{a}\underline{\mathtt{a}}) \\
\Rightarrow\ (A, > \underline{\mathtt{a}}\mathtt{a}) & \Rightarrow\ (R, > \underline{\mathtt{a}}\mathtt{a}) \\
\Rightarrow\ (A, > \mathtt{a}\underline{\mathtt{a}}) & \Rightarrow\ (R, \geq \mathtt{aa}) \\
\Rightarrow\ (A, > \mathtt{aa}\underline{\sqcup}) & \Rightarrow\ (P, \geq \mathtt{aa})
\end{array}
$$