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 $\sigma = \mathtt{a}, \mathtt{b}, \ldots$.

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 $\sigma = \mathbf{a}, \mathbf{b} \ldots$.

- 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 $\sigma = \mathbf{a}, \mathbf{b} \ldots$.
- 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: $\delta : Q \times \Gamma \rightharpoonup Q \times \mathrm{Act}$

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

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: $\delta : Q \times \Gamma \rightharpoonup Q \times \mathrm{Act}$

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

  Here they are $+$ for "step forward" 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**$\sqcup$.

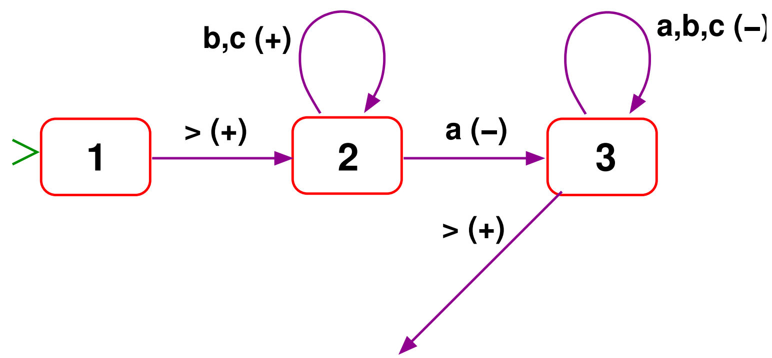
Intended behavior of 2DFAs

- 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**$\sqcup$.
- A 2DFA scans one input symbol at a time.
  Visualize it as a ***cursor:***
  $\underline{>}\mathbf{abc}\sqcup$ $\quad$ $>\mathbf{a}\underline{\mathbf{b}}\mathbf{c}\sqcup$ $\quad$ $>\mathbf{abc}\underline{\sqcup}$

A 2DFA for the "all-letters" language

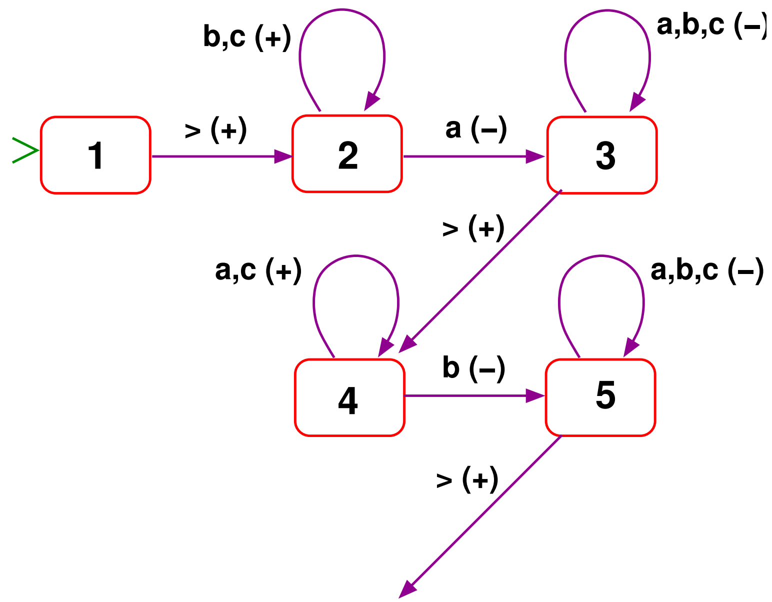
- Here is a 2DFA over $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$
  that recognizs the strings using all three letter.

A 2DFA for the "all-letters" language



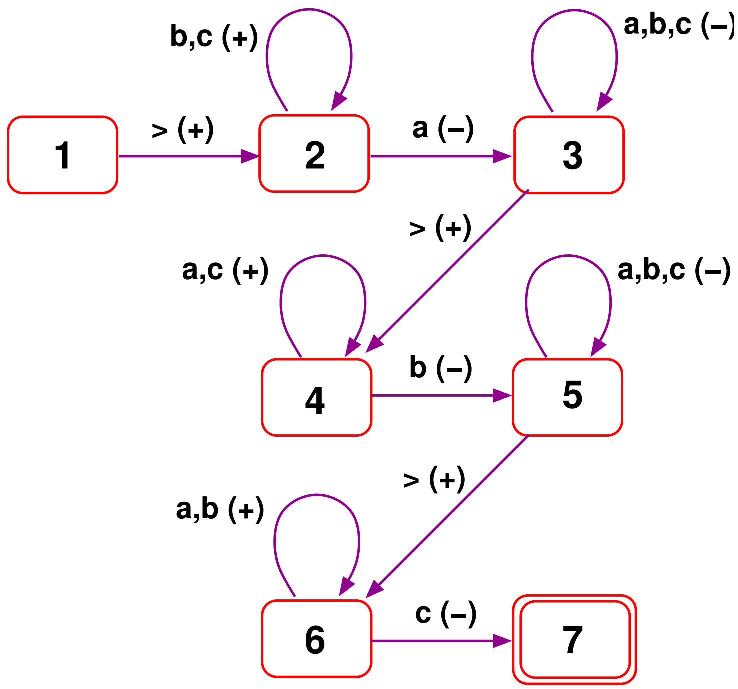
- Cycle through **b**'s and **c**'s until an **a** is found.
  If so, return to the gate;
  if not then then the blank end-maker is reached, for which there is no transition.
  The machine stops without accepting.

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 Σ–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: $(\mathtt{x}, >0101\underline{1}00\sqcup)$
- The ***initial cfg for input*** w is the cfg $(s, \underline{>}w\sqcup)$.

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 $q \xrightarrow{\gamma(-)} p$ 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 $q \xrightarrow{\gamma(+)} p$ then $(q, u\underline{\gamma}\tau v) \Rightarrow (p, u\gamma\underline{\tau}v)$
  - If $q \xrightarrow{\gamma(-)} p$ then $(q, u\tau\underline{\gamma}v) \Rightarrow (p, u\underline{\tau}\gamma v)$
- These clauses are the only ones in force.
  If a cfg ends with a cursored symbol, as in $(q, \mathbf{01101}\underline{\mathbf{0}})$,
  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 $\{\mathtt{a}^n\mathtt{b}^n\}$

- 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.

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.
- ***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 **a** (say by replacing it with $>$),
  then traverse the input and cross off final **b**.
- Stop and accept if and when neither **a** nor **b**
  are present for a new cycle.

Accepting $\mathbf{a^3b^3}$

$\underline{\triangleright}$ a a a b b b $\sqcup$

Accepting a^3b^3

$> \underline{a}\,a\,a\,b\,b\,b\,\sqcup$

Accepting $\mathbf{a^3b^3}$

$> \underline{>}$ a a b b b $\sqcup$

Accepting $\mathbf{a^3b^3}$

$> > \underline{a}\,a\,b\,b\,b\,\sqcup$

Accepting $\mathbf{a^3b^3}$

$> > \mathtt{a}\,\underline{\mathtt{a}}\,\mathtt{b}\,\mathtt{b}\,\mathtt{b}\,\sqcup$

Accepting $\mathbf{a}^3\mathbf{b}^3$

$\triangleright \triangleright \mathtt{a}\,\mathtt{a}\,\underline{\mathtt{b}}\,\mathtt{b}\,\mathtt{b}\,\sqcup$

Accepting $\mathtt{a}^3\mathtt{b}^3$

$> > \mathtt{a}\,\mathtt{a}\,\mathtt{b}\,\underline{\mathtt{b}}\,\mathtt{b}\,\sqcup$

Accepting $\mathtt{a}^3\mathtt{b}^3$

$> > \mathtt{a}\,\mathtt{a}\,\mathtt{b}\,\mathtt{b}\,\underline{\mathtt{b}}\,\sqcup$

Accepting $\mathtt{a}^3\mathtt{b}^3$

$> > \mathtt{a\,a\,b\,b\,b}\,\underline{\sqcup}$

Accepting $\mathtt{a}^3\mathtt{b}^3$

$> > \mathtt{a}\,\mathtt{a}\,\mathtt{b}\,\mathtt{b}\,\underline{\mathtt{b}}\,\sqcup$

Accepting $\mathbf{a^3b^3}$

$> > \mathtt{a\,a\,b\,b}\,\underline{\sqcup}\,\sqcup$

Accepting $\mathbf{a^3b^3}$

$> > \texttt{a}\,\texttt{a}\,\texttt{b}\,\underline{\texttt{b}}\,\sqcup\,\sqcup$

Accepting $\mathtt{a}^3\mathtt{b}^3$

$> > \mathtt{a}\ \mathtt{a}\ \underline{\mathtt{b}}\ \mathtt{b}\ \sqcup\ \sqcup$

Accepting $\mathtt{a}^3\mathtt{b}^3$

$> > \mathtt{a}\, \underline{\mathtt{a}}\, \mathtt{b}\, \mathtt{b} \sqcup \sqcup$

Accepting $\mathbf{a^3b^3}$

$> > \underline{a}\, a\, b\, b \sqcup \sqcup$

Accepting $\mathbf{a^3b^3}$

$> \underline{>}$ **a a b b** $\sqcup \sqcup$

Accepting $\mathtt{a}^3\mathtt{b}^3$

$> > \underline{>} \, \mathtt{a} \, \mathtt{b} \, \mathtt{b} \sqcup \sqcup$

Accepting $\mathtt{a^3b^3}$

$> > > \underline{\mathtt{a}}\,\mathtt{b}\,\mathtt{b} \sqcup \sqcup$

Accepting $\mathtt{a}^3\mathtt{b}^3$

$> > > \mathtt{a}\ \underline{\mathtt{b}}\ \mathtt{b} \sqcup \sqcup$

Accepting $\mathtt{a}^3\mathtt{b}^3$

$> > > \mathtt{a}\ \mathtt{b}\ \underline{\mathtt{b}}\ \sqcup\ \sqcup$

Accepting $\mathbf{a}^3\mathbf{b}^3$

$> > > \mathtt{a}\ \mathtt{b}\ \mathtt{b}\ \underline{\sqcup}\ \sqcup$

Accepting $\mathbf{a^3b^3}$

$$> \; > \; > \; \mathtt{a} \; \mathtt{b} \; \underline{\mathtt{b}} \; \sqcup \; \sqcup$$

Accepting $\mathtt{a^3b^3}$

$> > > \mathtt{a}\,\mathtt{b}\,\underline{\sqcup}\,\sqcup\,\sqcup$

Accepting a^3b^3

$> > > a \underline{b} \sqcup \sqcup \sqcup$

Accepting $\mathtt{a}^3\mathtt{b}^3$

$> > > \underline{\mathtt{a}}\ \mathtt{b} \sqcup \sqcup \sqcup$

Accepting $\mathtt{a}^3\mathtt{b}^3$

$> > \underline{>} \mathtt{a}\,\mathtt{b} \sqcup \sqcup \sqcup$

Accepting $\mathtt{a}^3\mathtt{b}^3$

$> > > \underline{\mathtt{a}}\,\mathtt{b} \sqcup \sqcup \sqcup$

Accepting $\mathtt{a}^3\mathtt{b}^3$

$> > > \underline{>} \mathtt{b} \sqcup \sqcup \sqcup$

Accepting $\mathbf{a^3b^3}$

$> > > > \underline{b} \sqcup \sqcup \sqcup$

Accepting $\mathbf{a^3b^3}$

$> > > > \mathtt{b} \underline{\sqcup} \sqcup \sqcup$

Accepting $\mathbf{a^3b^3}$

$> > > > \underline{\mathtt{b}} \sqcup \sqcup \sqcup$

Accepting $\mathbf{a^3b^3}$

$> > > > \underline{\sqcup} \sqcup \sqcup \sqcup$

Accepting $\mathbf{a^3b^3}$

$> > > \underline{>} \sqcup \sqcup \sqcup \sqcup$

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:

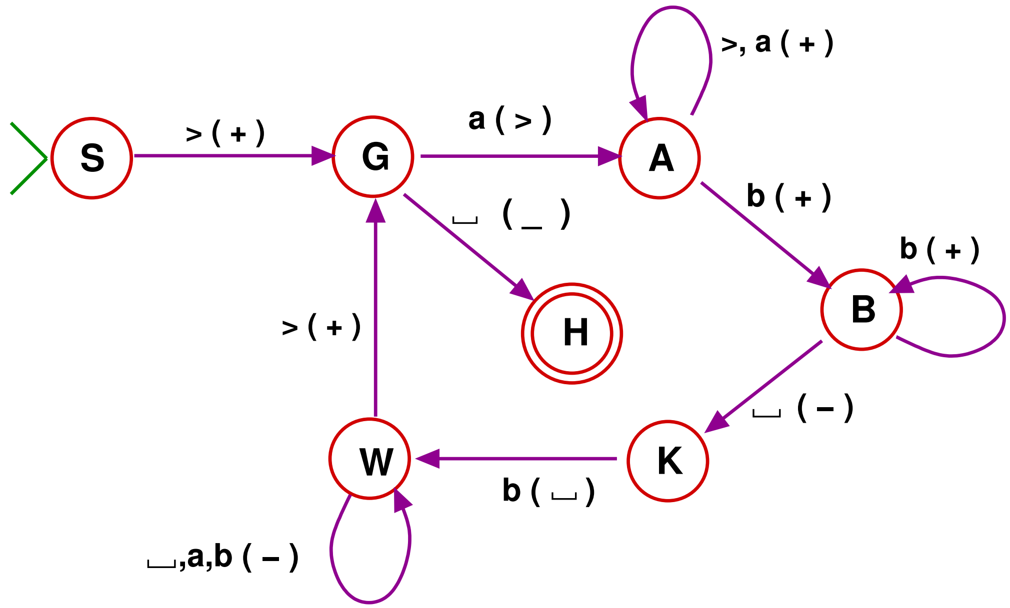
    $\delta : Q \times \Gamma \rightharpoonup Q \times \text{Act}$ where $\text{Act} = \{+, -\} \cup \Gamma$.

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:
    $$\delta : Q \times \Gamma \rightharpoonup Q \times \text{Act} \quad \text{where} \quad \text{Act} = \{+, -\} \cup \Gamma.$$
- 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 block 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 block 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 block 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: $(s,\ \underline{>}w\sqcup)$

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.

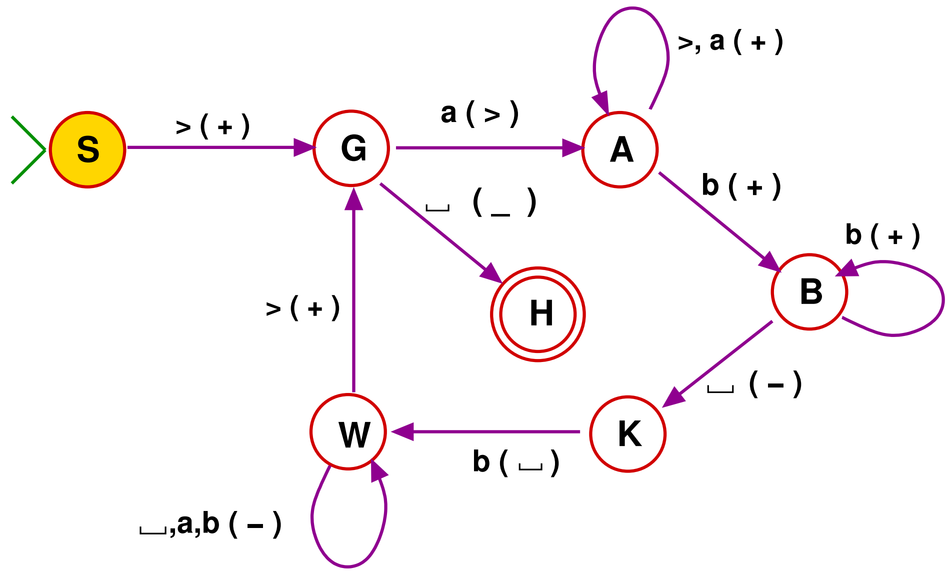
  The trace is ***accepting*** if c_n is accepting.

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

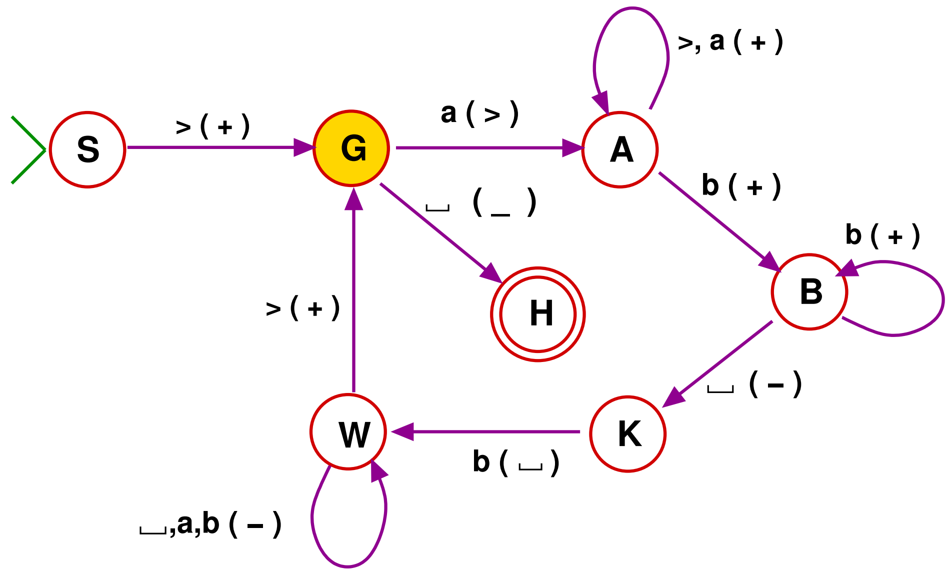
Example: Accepting trace for aabb

$(S,\ \underline{>}\mathbf{aabb}\sqcup)$



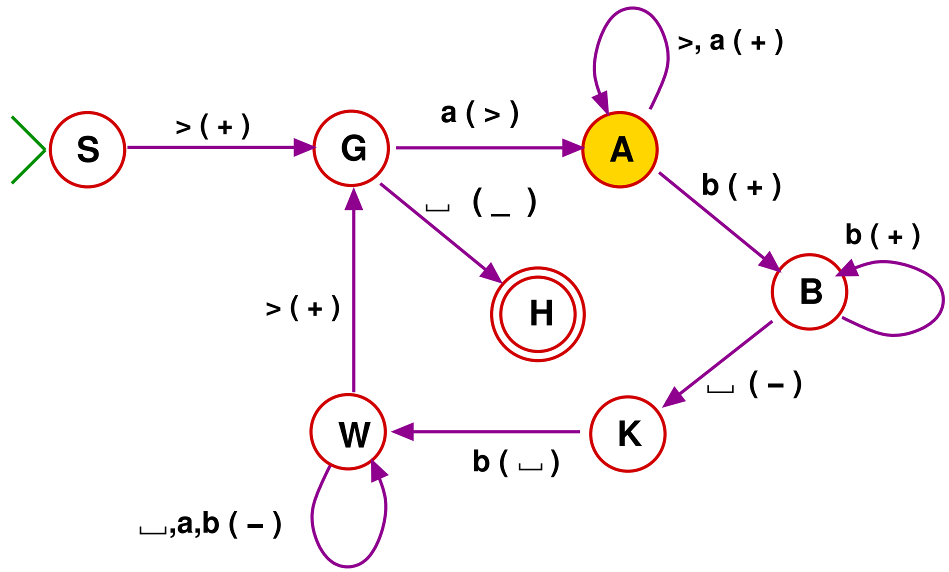
Example: Accepting trace for aabb

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



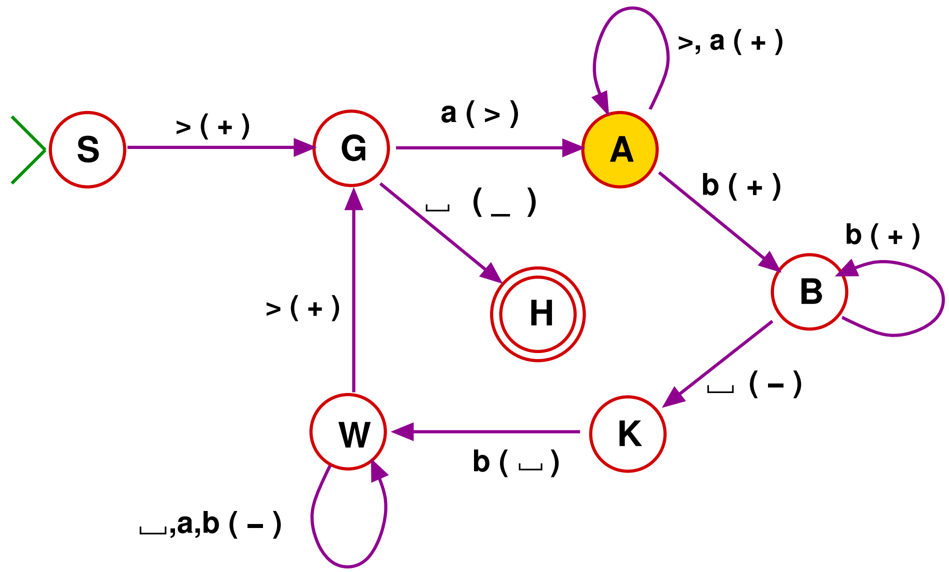
Example: Accepting trace for aabb

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



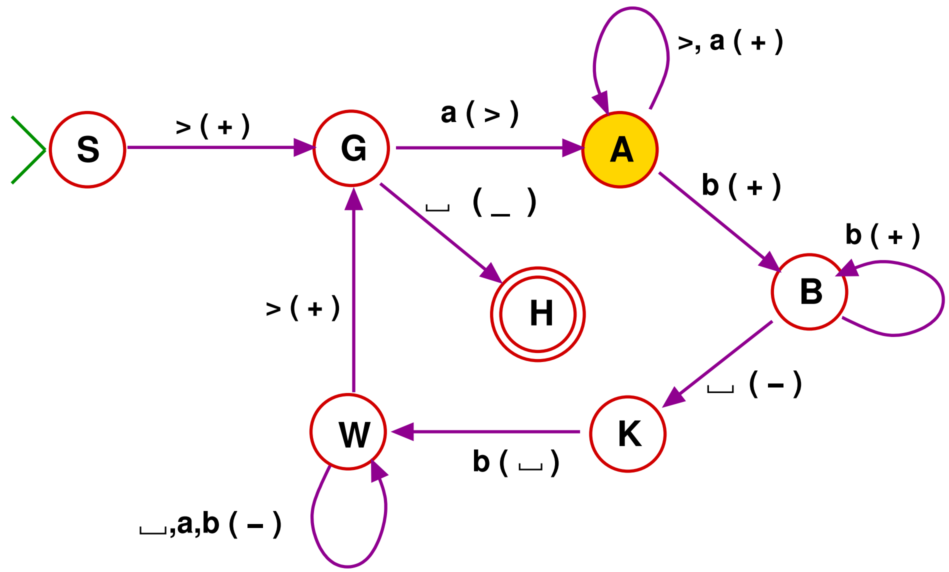
Example: Accepting trace for aabb

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



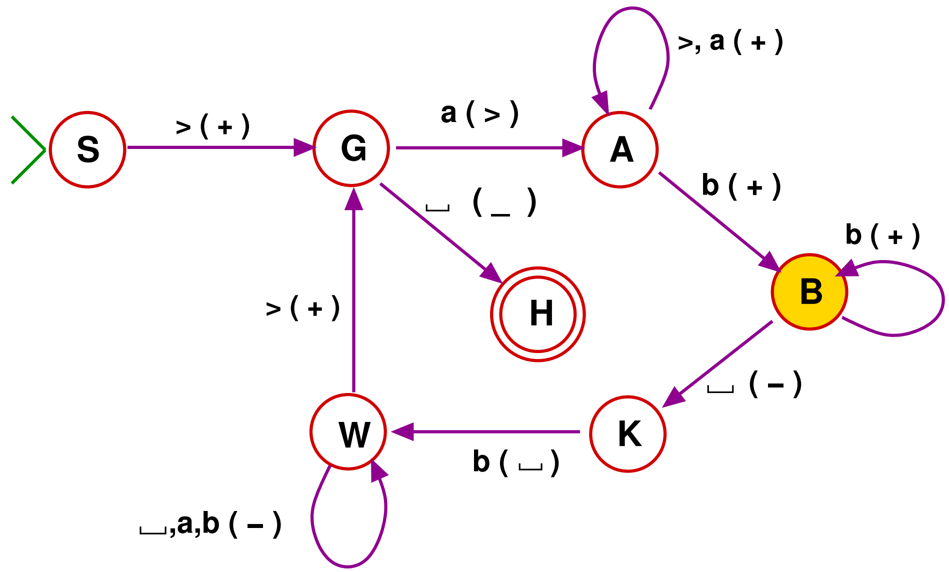
Example: Accepting trace for aabb

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



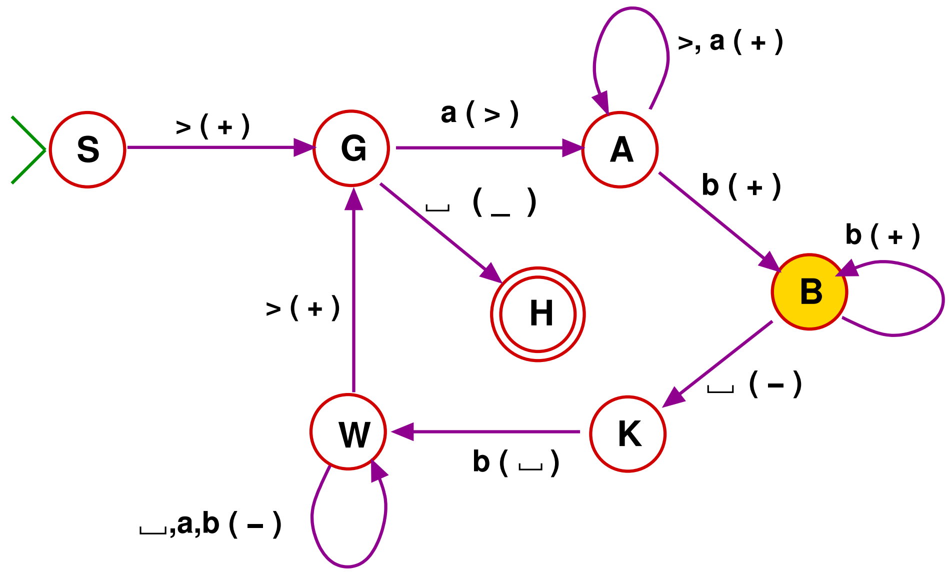
Example: Accepting trace for aabb

$(B,\ >>\mathtt{ab}\underline{\mathtt{b}}\sqcup)$



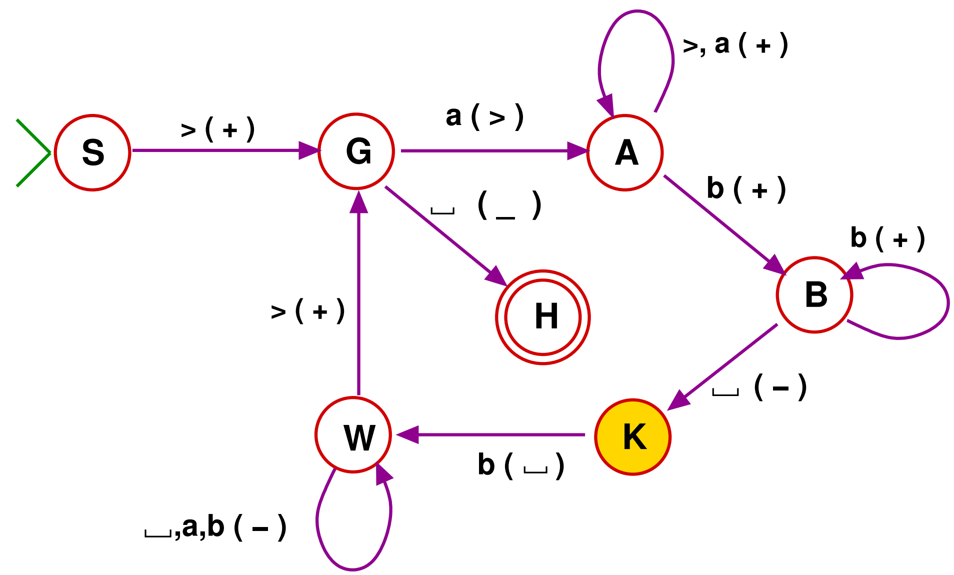
Example: Accepting trace for aabb

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



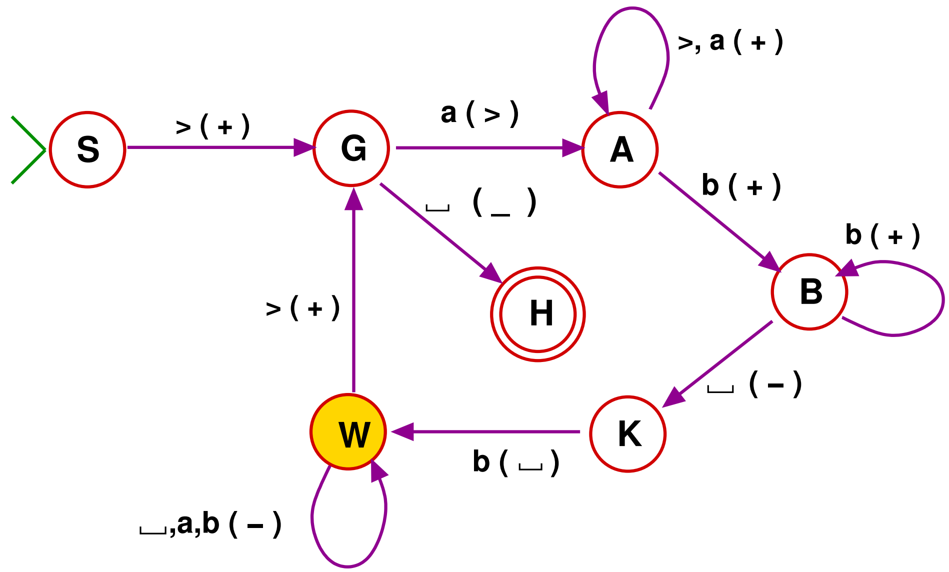
Example: Accepting trace for aabb

$(K,\ >>\mathtt{a}\,\mathtt{b}\,\underline{\mathtt{b}})$



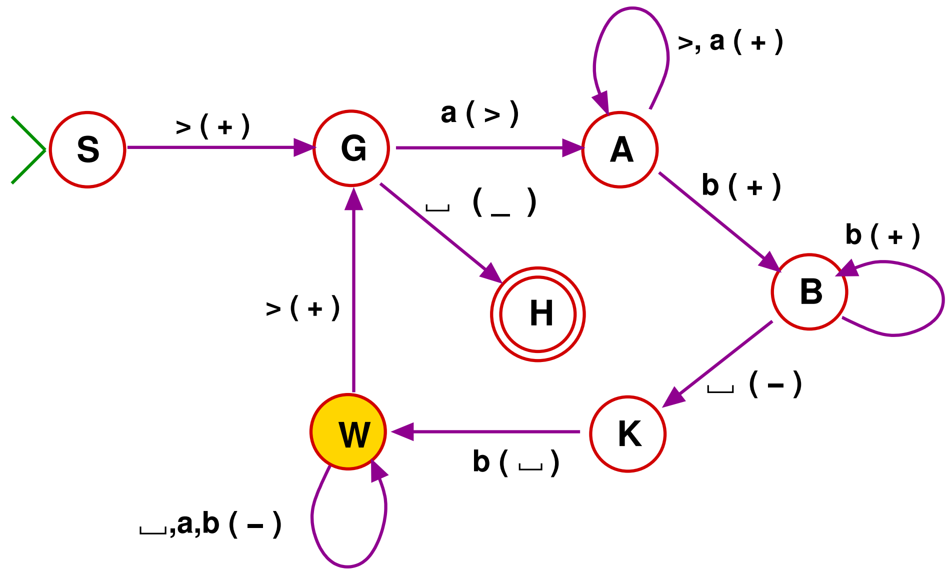
Example: Accepting trace for aabb

$(W, \ >>\mathtt{a\,b}\sqcup\sqcup)$



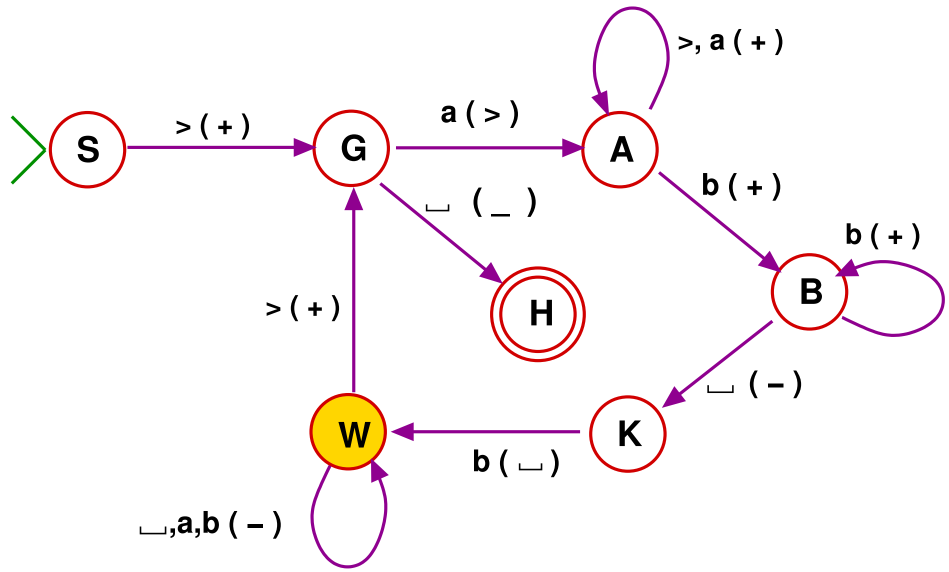
Example: Accepting trace for aabb

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

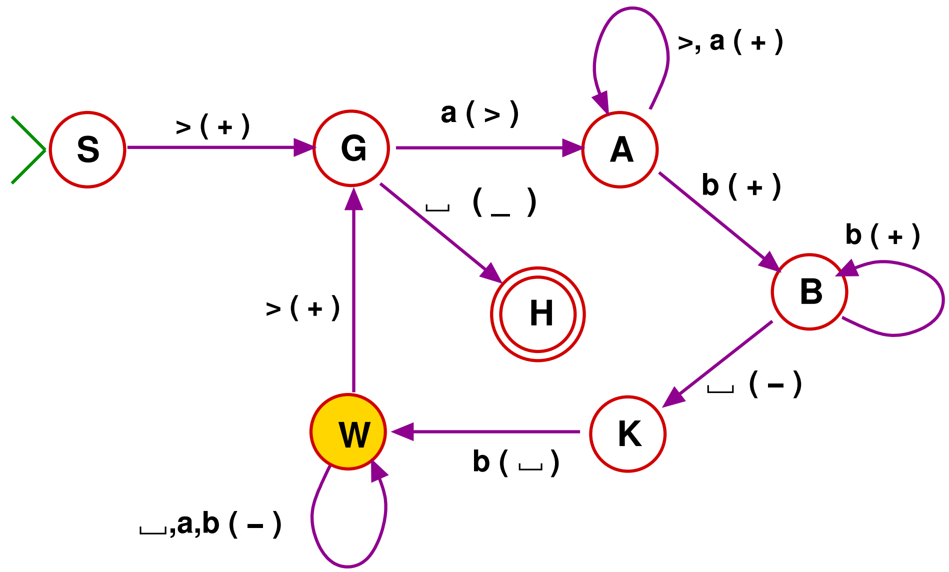


Example: Accepting trace for `aabb`

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

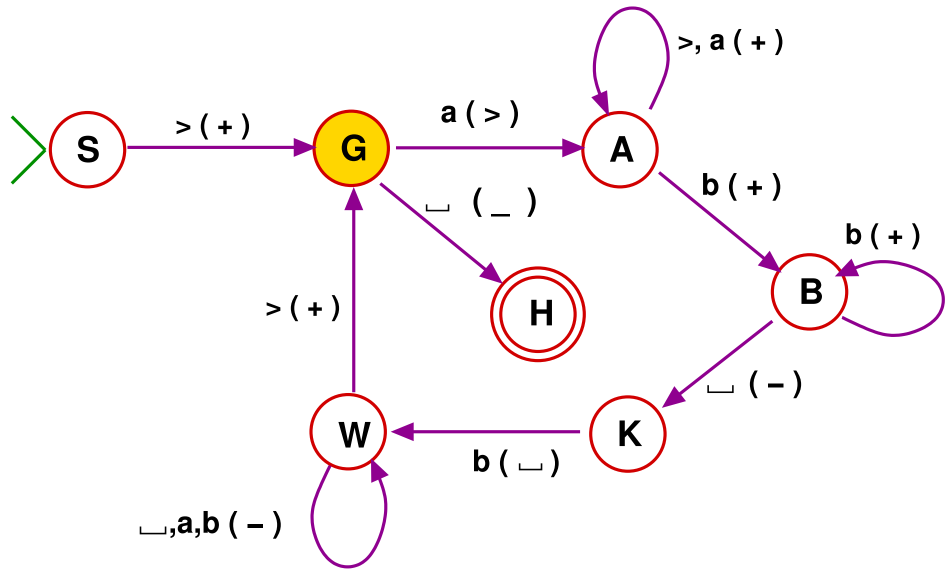


Example: Accepting trace for aabb



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

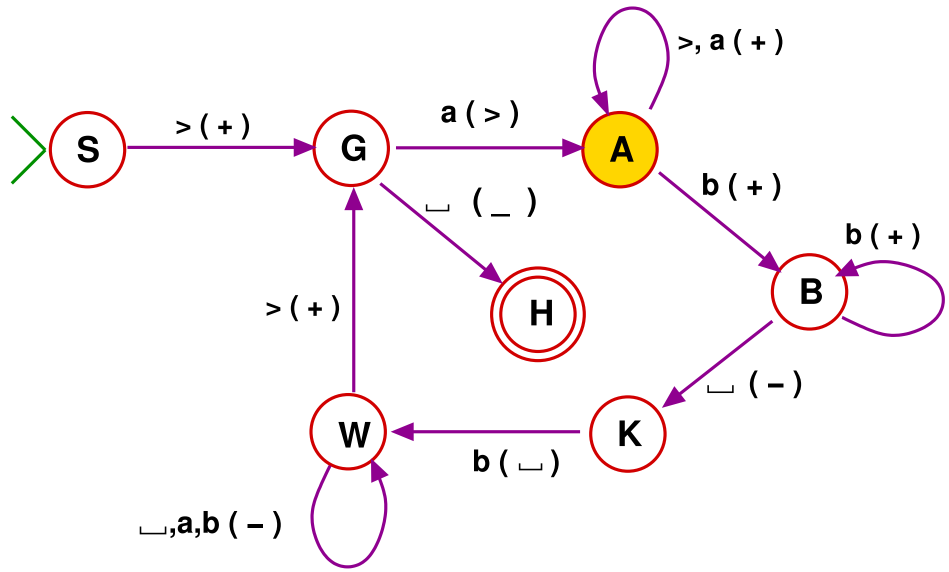
Example: Accepting trace for aabb



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

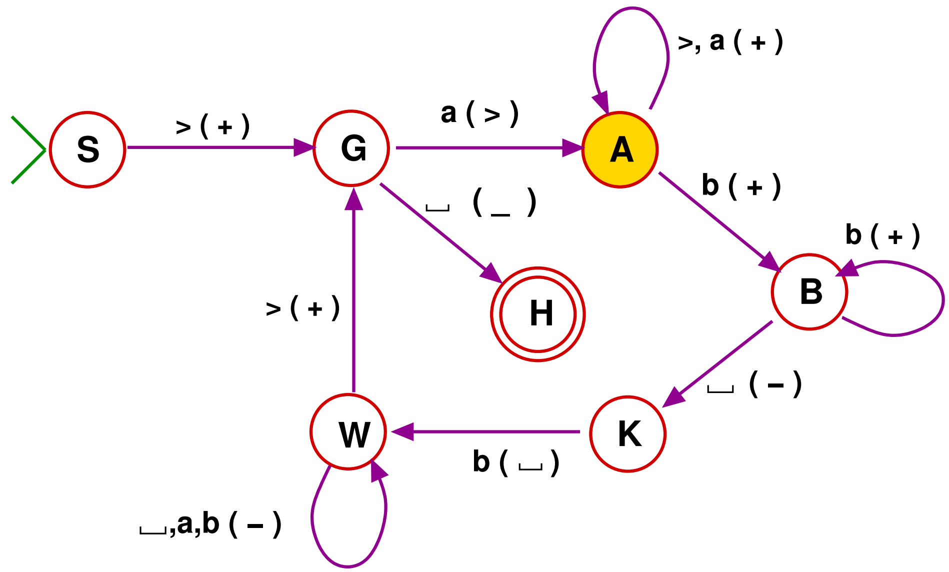
Example: Accepting trace for aabb

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

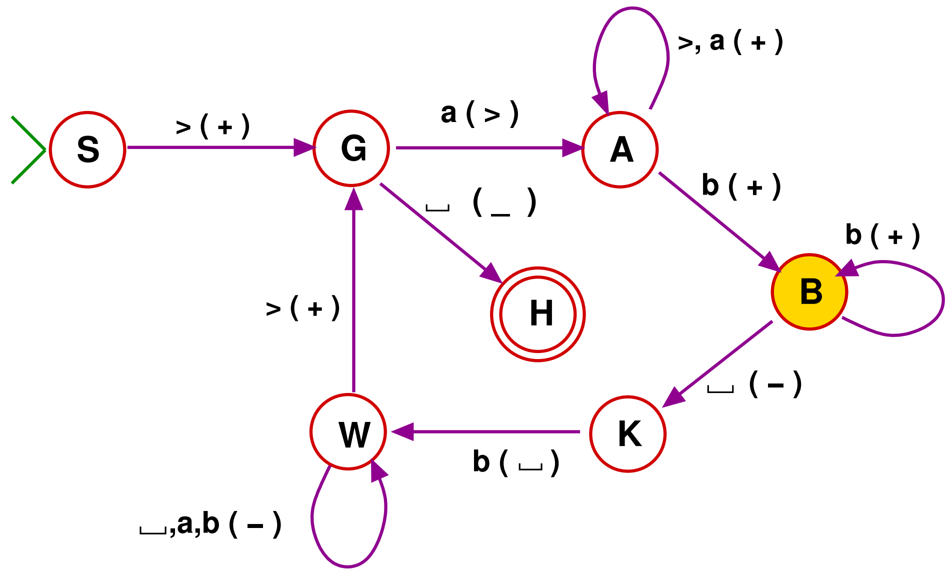


Example: Accepting trace for aabb

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



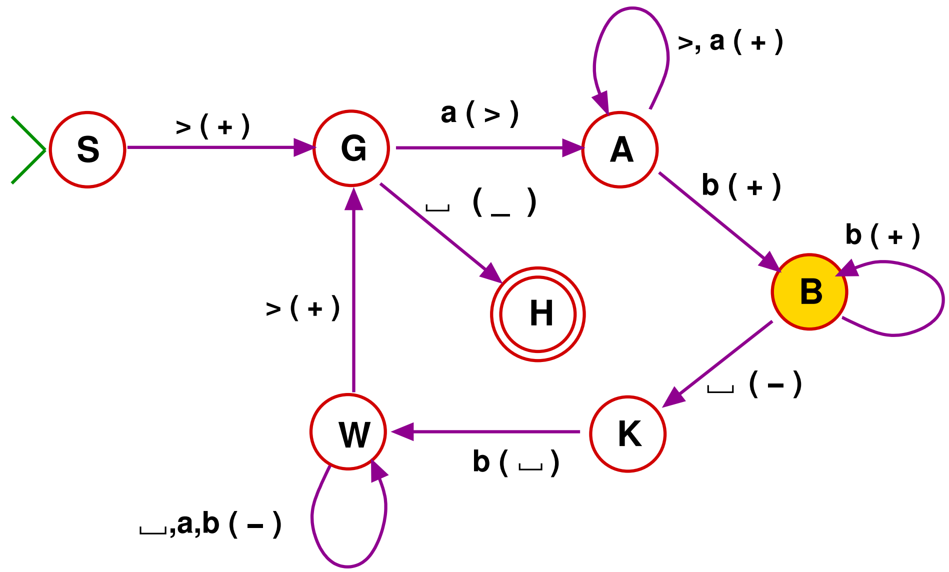
Example: Accepting trace for `aabb`



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

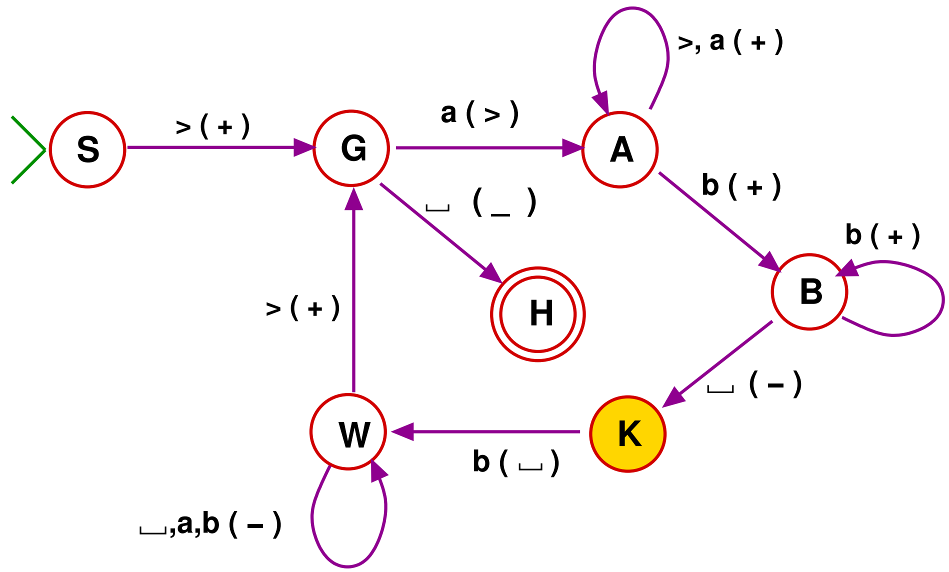
Example: Accepting trace for aabb

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



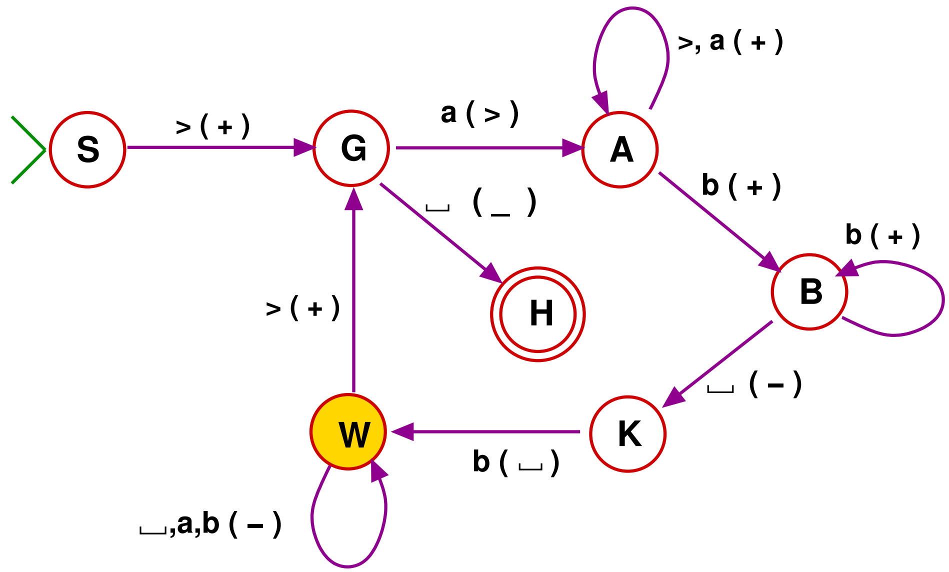
Example: Accepting trace for aabb

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



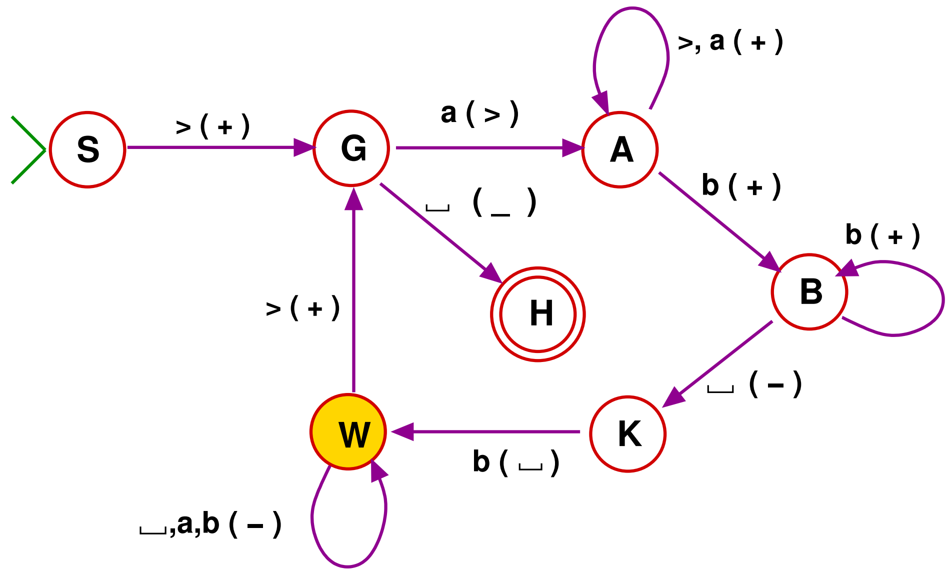
Example: Accepting trace for aabb

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



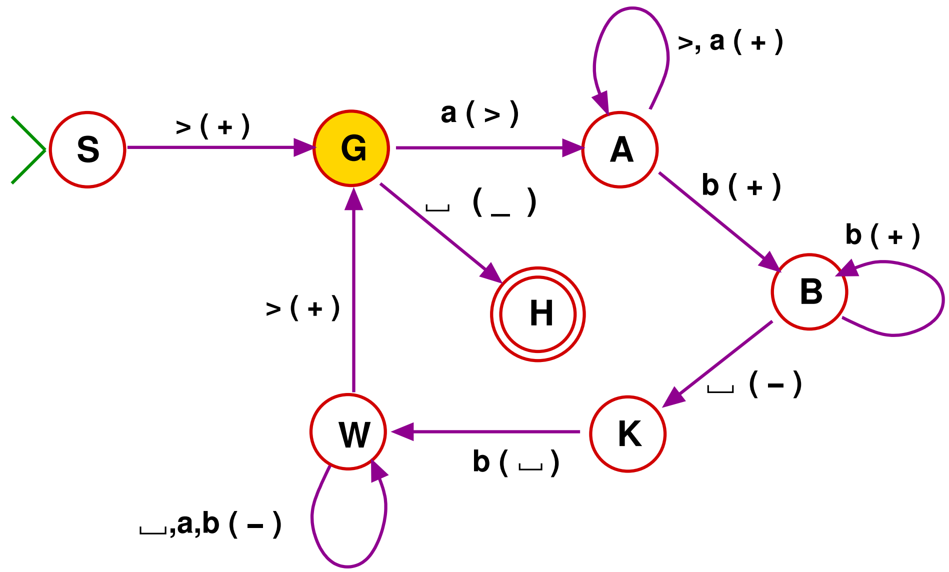
Example: Accepting trace for aabb

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



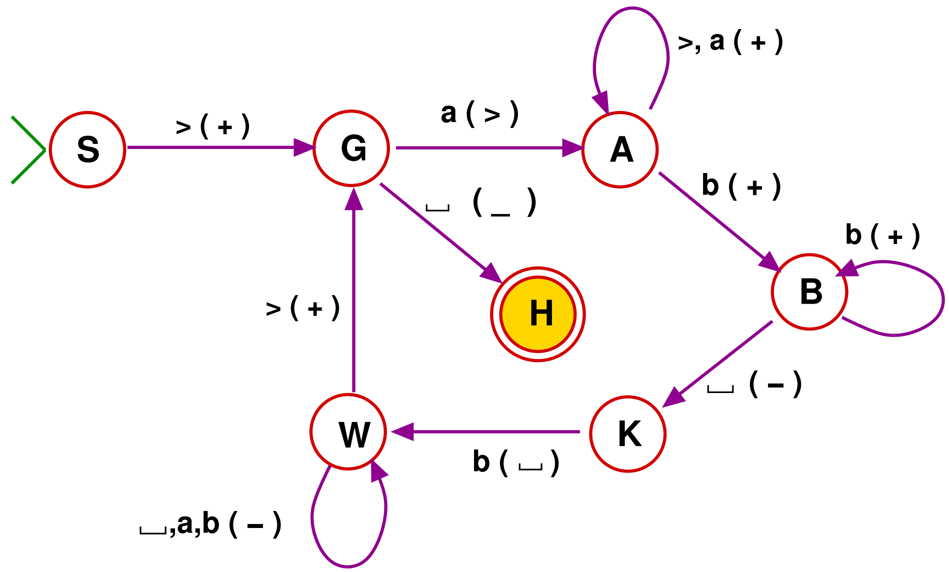
Example: Accepting trace for aabb

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



Example: Accepting trace for aabb

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



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 $q \xrightarrow{\gamma(+)} p$ 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 $q \xrightarrow{\gamma(+)} p$ 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...

Turing deciders

- A ***decider*** is an acceptor that ***terminates for all input***.
- A trace that ends with any state other than "accept" is consider to be a rejecting trace.

Turing deciders

- A ***decider*** is an acceptor that ***terminates for all input***.
- A trace that ends with any state other than "accept" is consider to be a rejecting trace.
- NOTE: The definition of deciders is ***not structural or syntactic:*** it is a condition on acceptors, for which no algorithm needs to be given.

Turing deciders

- A ***decider*** is an acceptor that ***terminates for all input***.
- A trace that ends with any state other than "accept"
  is consider to be a rejecting trace.
- NOTE: The definition of deciders is ***not structural or syntactic:***
  it is a condition on acceptors, for which
  no algorithm needs to be given.