# Turing Machines

CENG 280

# Course outline

- Preliminaries: Alphabets and languages
- Regular languages
- Context-free languages
- Turing-machines
  - Turing machines, definition and examples
  - Extensions of TMS
  - Nondeterministic TMs
  - Unrestricted grammars
  - Church-Turing thesis, universal Turing machines
  - Halting problem

# Turing Machines

- PDA, more powerful than FSA but still not powerful enough to recognize some basic languages such as $a^n b^n c^n$
- Turing Machines invented by Alan Turing- stronger than PDA
- It will not be replaced by another automata
- If a numeric function is computable, there is a TM for it.
- Extensions (non-determinism, multiple tapes, multiple heads etc) do not generate additional computational power.
- What can be decided? Church-Turing thesis (recursive functions, Turing computable functions)
- Unrestricted grammars as language generators

# History

## Entscheidungsproblem

A challenge posed by David Hilbert and Wilhelm Ackerman in 1928. Is there an algorithm that takes as input a statement in a first order logic, and decides whether the statement is universally valid?

# History

## Entscheidungsproblem

A challenge posed by David Hilbert and Wilhelm Ackerman in 1928. Is there an algorithm that takes as input a statement in a first order logic, and decides whether the statement is universally valid?

Informal question: What is effectively computable?

# History

## Entscheidungsproblem

A challenge posed by David Hilbert and Wilhelm Ackerman in 1928. Is there an algorithm that takes as input a statement in a first order logic, and decides whether the statement is universally valid?

Informal question: What is effectively computable?

1930 Alonzo Church and his student Stephen Kleene introduced the notion of $\lambda-$definable functions.

# History

## Entscheidungsproblem

A challenge posed by David Hilbert and Wilhelm Ackerman in 1928. Is there an algorithm that takes as input a statement in a first order logic, and decides whether the statement is universally valid?

Informal question: What is effectively computable?

1930 Alonzo Church and his student Stephen Kleene introduced the notion of $\lambda-$definable functions.

1934 Kurt Gödel thought that Church's idea of effectively computable functions should be defined as $\lambda-$definable functions was unsatisfactory.

# History

## Entscheidungsproblem

A challenge posed by David Hilbert and Wilhelm Ackerman in 1928. Is there an algorithm that takes as input a statement in a first order logic, and decides whether the statement is universally valid?

Informal question: What is effectively computable?

1930 Alonzo Church and his student Stephen Kleene introduced the notion of $\lambda-$definable functions.

1934 Kurt Gödel thought that Church's idea of effectively computable functions should be defined as $\lambda-$definable functions was unsatisfactory.

1934 Gödel developed his version: recursive functions

# History

## Entscheidungsproblem

A challenge posed by David Hilbert and Wilhelm Ackerman in 1928. Is there an algorithm that takes as input a statement in a first order logic, and decides whether the statement is universally valid?

Informal question: What is effectively computable?

- 1930 Alonzo Church and his student Stephen Kleene introduced the notion of $\lambda-$definable functions.
- 1934 Kurt Gödel thought that Church's idea of effectively computable functions should be defined as $\lambda-$definable functions was unsatisfactory.
- 1934 Gödel developed his version: recursive functions
- 4-1935 Kleene with the help of Church and J. Barkley Rosser, proved that the two approaches, recursive functions and $\lambda-$calculi are equivalent

# History

## Entscheidungsproblem

A challenge posed by David Hilbert and Wilhelm Ackerman in 1928. Is there an algorithm that takes as input a statement in a first order logic, and decides whether the statement is universally valid?

Informal question: What is effectively computable?

- 1930 Alonzo Church and his student Stephen Kleene introduced the notion of $\lambda-$definable functions.
- 1934 Kurt Gödel thought that Church's idea of effectively computable functions should be defined as $\lambda-$definable functions was unsatisfactory.
- 1934 Gödel developed his version: recursive functions
- 4-1935 Kleene with the help of Church and J. Barkley Rosser, proved that the two approaches, recursive functions and $\lambda-$calculi are equivalent
- 1936 Church updated his methods to include recursion and proved that Entscheidungsproblem is unsolvable.

# History

**Entscheidungsproblem** Informal question: What is effectively computable?

4-1935 Kleene with the help of Church and J. Barkley Rosser, proved that the two approaches, recursive functions and $\lambda-$calculi are equivalent

1936 Church updated his methods to include recursion and proved that Entscheidungsproblem is unsolvable.

1936 Alan Turing orally presented his work on "automatic machine"

ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

By A. M. TURING.

[Received 28 May, 1936.—Read 12 November, 1936.]

The "computable" numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means. Although the subject of this paper is ostensibly the computable *numbers*, it is almost equally easy to define and investigate computable functions of an integral variable or a real or computable variable, computable predicates, and so forth. The fundamental problems involved are, however, the same in each case, and I have chosen the computable numbers for explicit treatment as involving the least cumbrous technique. I hope shortly to give an account of the relations of the computable numbers, functions, and so forth to one another. This will include a development of the theory of functions of a real variable expressed in terms of computable numbers. According to my definition, a number is computable if its decimal can be written down by a machine.

In §§ 9, 10 I give some arguments with the intention of showing that the computable numbers include all numbers which could naturally be regarded as computable. In particular, I show that certain large classes of numbers are computable. They include, for instance, the real parts of all algebraic numbers, the real parts of the zeros of the Bessel functions, the numbers $\pi$, $e$, etc. The computable numbers do not, however, include all definable numbers, and an example is given of a definable number which is not computable.

Although the class of computable numbers is so great, and in many

# History

**Entscheidungsproblem** Informal question: What is effectively computable?

-1935 Kleene with the help of Church and J. Barkley Rosser, proved that the two approaches, recursive functions and $\lambda-$calculi are equivalent

1936 Church updated his methods to include recursion and proved that Entscheidungsproblem is unsolvable.

1936 Alan Turing orally presented his work on "automatic machine"

1937 His paper was printed, "On Computable Numbers, with an Application to the Entscheidungsproblem". He showed that the $\lambda-$calculi and his machine coincide.

# Turing Machine

## Definition

A Turing machine is a quintuple $M = (K, \Sigma, \delta, s, H)$ where

- $K$ is a finite set of states,
- $\Sigma$ is an alphabet containing the blank symbol $\sqcup$, and the left end symbol $\triangleright$, but not containing the symbols $\leftarrow$ and $\rightarrow$.
- $s \in K$ is the initial state
- $H \subseteq K$ is the set of halting states, and,
- $\delta$ is the transition function $\delta : (K \setminus H) \times \Sigma \rightarrow K \times (\Sigma \cup \{\leftarrow, \rightarrow\})$ such that
  - if $\delta(q, \triangleright) = (p, b)$ then $b = \rightarrow$
  - if $\delta(q, a) = (p, b)$ then $b \neq \triangleright$

The machine takes transition $\delta(q, a) = (p, b)$ when it is in state $q \in K \setminus H$, and reads $a$. After the transition it moves to state $p$, and if $b \in \{\leftarrow, \rightarrow\}$ it moves the reading head in the corresponding direction. If $b \in \Sigma$, then it writes $b$ over $a$. The operation of $M$ is deterministic. It will stop when $M$ enters a halting state.

## Example

$M = (K, \Sigma, \delta, s, H)$, where $K = \{q_0, q_1, h\}$, $\Sigma = \{a, \sqcup, \triangleright\}$, $s = q_0$,

$H = \{h\}$, and $\delta$ :

| $q,$ | $\sigma$ | $\delta(q, \sigma)$ |
|---|---|---|
| $q_0$ | $a$ | $(q_1, \sqcup)$ |
| $q_0$ | $\sqcup$ | $(h, \sqcup)$ |
| $q_0$ | $\triangleright$ | $(q_0, \rightarrow)$ |
| $q_1$ | $a$ | $(q_0, a)$ |
| $q_1$ | $\sqcup$ | $(q_0, \rightarrow)$ |
| $q_1$ | $\triangleright$ | $(q_1, \rightarrow)$ |

# Turing Machine

## Example

$M = (K, \Sigma, \delta, s, H)$, where $K = \{q_0, h\}$, $\Sigma = \{a, \sqcup, \triangleright\}$, $s = q_0$, $H = \{h\}$,

and $\delta$ :

| $q$, | $\sigma$ | $\delta(q, \sigma)$ |
|---|---|---|
| $q_0$ | $a$ | $(q_0, \leftarrow)$ |
| $q_0$ | $\sqcup$ | $(h, \sqcup)$ |
| $q_0$ | $\triangleright$ | $(q_0, \rightarrow)$ |

# Turing Machine: Configuration

- A **configuration** of a machine $M$ is a member of

$$K \times \triangleright\Sigma^{\star} \times (\Sigma^{\star}(\Sigma \setminus \{\sqcup\}) \cup \{e\})$$

# Turing Machine: Configuration

- A **configuration** of a machine $M$ is a member of

$$K \times \triangleright\Sigma^{\star} \times (\Sigma^{\star}(\Sigma \setminus \{\sqcup\}) \cup \{e\})$$

- A configuration is shown as $(q, wa, u)$ or $(q, w\underline{a}u)$ where $w \in \Sigma^{\star}$ shows the part to the left of the scanned square, $a$ is the symbol in the scanned square, and $u$ is the string to the right of the scanned square.

# Turing Machine: Configuration

- A **configuration** of a machine $M$ is a member of

$$K \times \triangleright\Sigma^{\star} \times (\Sigma^{\star}(\Sigma \setminus \{\sqcup\}) \cup \{e\})$$

- A configuration is shown as $(q, wa, u)$ or $(q, w\underline{a}u)$ where $w \in \Sigma^{\star}$ shows the part to the left of the scanned square, $a$ is the symbol in the scanned square, and $u$ is the string to the right of the scanned square.
- A configuration $(q, w\underline{a}u)$ is called halted configuration when $q \in H$.

# Turing Machine: Configuration

- A **configuration** of a machine $M$ is a member of

$$K \times \triangleright\Sigma^\star \times (\Sigma^\star(\Sigma \setminus \{\sqcup\}) \cup \{e\})$$

- A configuration is shown as $(q, wa, u)$ or $(q, w\underline{a}u)$ where $w \in \Sigma^\star$ shows the part to the left of the scanned square, $a$ is the symbol in the scanned square, and $u$ is the string to the right of the scanned square.
- A configuration $(q, w\underline{a}u)$ is called halted configuration when $q \in H$.
- A configuration $(q_1, w_1\underline{a_1}u_1)$ yields configuration $(q_2, w_2\underline{a_2}u_2)$ in one step, shown with $(q_1, w_1\underline{a_1}u_1) \vdash_M (q_2, w_2\underline{a_2}u_2)$. if and only if for some $b \in \Sigma \cup \{\leftarrow, \rightarrow\}$, $\delta(q_1, a_1) = (q_2, b)$ and either
    - $b \in \Sigma$, $w_1 = w_2$, $u_1 = u_2$ and $a_2 = b$ or
    - $b = \rightarrow$, $w_2 = w_1a_1$ , and, if $a_1 = \sqcup$ and $u_1 = e$ then $u_2 = e$, otherwise $u_1 = a_2u_2$
    - $b = \leftarrow$, $w_1 = w_2a_2$ and, if $a_1 = \sqcup$ and $u_1 = e$ then $u_2 = e$ else $u_2 = a_1u_1$

# Turing Machine: Computation

- $\vdash_M^\star$ is the reflexive transitive closure of $\vdash_M$. We say a configuration $C_1$ yields configuration $C_2$ if $C_1 \vdash_M^\star C_2$.

# Turing Machine: Computation

- $\vdash_M^\star$ is the reflexive transitive closure of $\vdash_M$. We say a configuration $C_1$ yields configuration $C_2$ if $C_1 \vdash_M^\star C_2$.
- A computation of a machine $M$ is a sequence of computations $C_0, \ldots, C_n$ such that $n \geq 1$ and

$$C_0 \vdash_M C_1 \vdash_M \ldots \vdash_M C_n$$

The given computation is length $n$ and it is also written as $C_0 \vdash_M^n C_n$.

# A Notation for Turing Machines

**A notation for Turing Machines**: The main idea is to use a hierarchical notation. Define complex machines by combining simple machines.

# A Notation for Turing Machines

**A notation for Turing Machines**: The main idea is to use a hierarchical notation. Define complex machines by combining simple machines.

**Symbol writing and head moving machine:** $M_a = (\{s, h\}, \Sigma, \delta, s, \{h\})$.

- $\delta(s, \triangleright) = (s, \rightarrow)$,
- $\delta(s, b) = (h, a)$ for each $b \in \Sigma \setminus \{\triangleright\}$.

# A Notation for Turing Machines

**A notation for Turing Machines**: The main idea is to use a hierarchical notation. Define complex machines by combining simple machines.

**Symbol writing and head moving machine:** $M_a = (\{s, h\}, \Sigma, \delta, s, \{h\})$.

- $\delta(s, \triangleright) = (s, \rightarrow)$,
- $\delta(s, b) = (h, a)$ for each $b \in \Sigma \setminus \{\triangleright\}$.
- If $a \in \Sigma$, write $a$ and halt.

# A Notation for Turing Machines

**A notation for Turing Machines**: The main idea is to use a hierarchical notation. Define complex machines by combining simple machines.

**Symbol writing and head moving machine:** $M_a = (\{s, h\}, \Sigma, \delta, s, \{h\})$.

- $\delta(s, \triangleright) = (s, \rightarrow)$,
- $\delta(s, b) = (h, a)$ for each $b \in \Sigma \setminus \{\triangleright\}$.

- If $a \in \Sigma$, write $a$ and halt.
- If $a \in \{\leftarrow, \rightarrow\}$, move the head in the given direction and halt.

# A Notation for Turing Machines

**A notation for Turing Machines**: The main idea is to use a hierarchical notation. Define complex machines by combining simple machines.

**Symbol writing and head moving machine:** $M_a = (\{s, h\}, \Sigma, \delta, s, \{h\})$.

- $\delta(s, \triangleright) = (s, \rightarrow)$,
- $\delta(s, b) = (h, a)$ for each $b \in \Sigma \setminus \{\triangleright\}$.

- If $a \in \Sigma$, write $a$ and halt.
- If $a \in \{\leftarrow, \rightarrow\}$, move the head in the given direction and halt.
- Only exception is the $\triangleright$ symbol.

# A Notation for Turing Machines

**A notation for Turing Machines**: The main idea is to use a hierarchical notation. Define complex machines by combining simple machines.
**Symbol writing and head moving machine:** $M_a = (\{s, h\}, \Sigma, \delta, s, \{h\})$.

- $\delta(s, \triangleright) = (s, \rightarrow)$,
- $\delta(s, b) = (h, a)$ for each $b \in \Sigma \setminus \{\triangleright\}$.

- If $a \in \Sigma$, write $a$ and halt.
- If $a \in \{\leftarrow, \rightarrow\}$, move the head in the given direction and halt.
- Only exception is the $\triangleright$ symbol.

- $M_a$ or $a$: $a-$writing machine for $a \in \Sigma$, write $a$ and halt
- $R = M_{\rightarrow}$: move right and halt
- $L = M_{\leftarrow}$: move left and halt

# A Notation for Turing Machines: Rules for combining machines

Connect machines via arrows ($\longrightarrow$), the connection will not be pursued until the first machine halts. Define $M = (K, \Sigma, \delta, s, H)$ from $M_i = (K_i, \Sigma, \delta_i, s_i, H_i), i = 1, 2, 3$.

$$
\begin{tikzcd}
M_1 \arrow[r, "a"] \arrow[d, "b"] & M_2 \\
M_3 &
\end{tikzcd}
$$

# A Notation for Turing Machines: Rules for combining machines

Connect machines via arrows ($\longrightarrow$), the connection will not be pursued until the first machine halts. Define $M = (K, \Sigma, \delta, s, H)$ from $M_i = (K_i, \Sigma, \delta_i, s_i, H_i), i = 1, 2, 3.$

$$\begin{tikzcd} M_1 \arrow[r, "a"] \arrow[d, "b"] & M_2 \\ M_3 & \end{tikzcd}$$

- Start in the initial state of $M_1$, operate as $M_1$ until it halts.
- When $M_1$ halts (if it ever does), if the currently scanned symbol is $a$ then initiate $M_2$ and operate as $M_2$.
- When $M_1$ halts, if the currently scanned symbol is $b$ then initiate $M_3$ and operate as $M_3$.

# A Notation for Turing Machines: Rules for combining machines

Connect machines via arrows ($\longrightarrow$), the connection will not be pursued until the first machine halts. Define $M = (K, \Sigma, \delta, s, H)$:
$M_1 \xrightarrow{a} M_2, M_1 \xrightarrow{b} M_3$, from $M_i = (K_i, \Sigma, \delta_i, s_i, H_i), i = 1, 2, 3$.

$$
\begin{tikzcd}
M_1 \arrow[r, "a"] \arrow[d, "b"] & M_2 \\
M_3 &
\end{tikzcd}
$$

- $K = K_1 \cup K_2 \cup K_3$
- $s = s_1$
- $H = H_2 \cup H_3$
- For each $\sigma \in \Sigma$, $q \in K \setminus H$, define $\delta(q, \sigma)$ as
  - If $q \in K_i \setminus H_i$, then $\delta(q, \sigma) = \delta_i(q, \sigma)$ for $i = 1, 2, 3$
  - If $q \in H_1$: if $\sigma$ is $a$ then $\delta(q, \sigma) = (s_2, \sigma)$, if $\sigma$ is $b$ then $\delta(q, \sigma) = (s_3, \sigma)$, if $\sigma \notin \{a, b\}$ then $\delta(q, \sigma) = (h, \sigma)$ for some $h \in H$.

# Examples of basic machines

Move right, read a symbol and move right

$$R \xrightarrow{a, b, \rhd, \sqcup} R$$

# Examples of basic machines

Move right, read a symbol and move right

$R \xrightarrow{a, b, \triangleright, \sqcup} R$ $\qquad\qquad$ $R \longrightarrow R$

# Examples of basic machines

Move right, read a symbol and move right

$R \xrightarrow{a, b, \triangleright, \sqcup} R$ $\qquad R \longrightarrow R \qquad RR, R^2$

# Examples of basic machines

Move right, read a symbol and move right

$R \xrightarrow{a, b, \triangleright, \sqcup} R$ $\qquad R \longrightarrow R$ $\qquad RR, R^2$

$\bar{a}$ : any symbol except $a$

# Examples of basic machines

Move right, read a symbol and move right

$R \xrightarrow{a, b, \triangleright, \sqcup} R$ $\qquad$ $R \longrightarrow R$ $\qquad$ $RR, R^2$

$\bar{a}$ : any symbol except $a$

$R_\sqcup$ :
$$\begin{tikzcd} R \arrow[loop above, "\bar{\sqcup}"] \end{tikzcd}$$

# Examples of basic machines

Move right, read a symbol and move right

$R \xrightarrow{a, b, \triangleright, \sqcup} R$ $\qquad\qquad$ $R \longrightarrow R$ $\qquad\qquad$ $RR, R^2$

$\bar{a}$ : any symbol except $a$

$R_\sqcup$ :
$$\begin{tikzcd} R \arrow[loop above, "\bar{\sqcup}"] \end{tikzcd} \qquad\qquad \begin{tikzcd} R \arrow[loop above, "a \neq \sqcup"] \end{tikzcd}$$

# Examples of basic machines

Move right, read a symbol and move right

$R \xrightarrow{a,\, b,\, \triangleright,\, \sqcup} R \qquad\qquad R \longrightarrow R \qquad\qquad RR,\, R^2$

$\bar{a}$ : any symbol except $a$

$R_\sqcup$ : 

$$\begin{tikzcd} R \arrow[loop above, "\bar{\sqcup}"] \end{tikzcd} \qquad \begin{tikzcd} R \arrow[loop above, "a \neq \sqcup"] \end{tikzcd} \qquad \begin{tikzcd} R \arrow[loop above, "\sqcup"] \arrow[r, "a \neq \sqcup"] & L_a \end{tikzcd}$$

# Examples of basic machines

- $R_{\sqcup}$ : $\overset{\bar{\sqcup}}{\overset{\circlearrowright}{R}}$ Find the first blank square to the right of the currently scanned square.

# Examples of basic machines

- $R_{\sqcup}$ : $\overset{\overline{\sqcup}}{\underset{R}{\circlearrowright}}$ Find the first blank square to the right of the currently scanned square.
- $R_{\overline{\sqcup}}$ : $\overset{\sqcup}{\underset{R}{\circlearrowright}}$ Find the first non-blank square to the right of the currently scanned square.

# Examples of basic machines

- $R_{\sqcup}$ : $\begin{tikzcd} R \arrow[loop above, "\overline{\sqcup}"] \end{tikzcd}$ Find the first blank square to the right of the currently scanned square.
- $R_{\overline{\sqcup}}$ : $\begin{tikzcd} R \arrow[loop above, "\sqcup"] \end{tikzcd}$ Find the first non-blank square to the right of the currently scanned square.
- $L_{\sqcup}$ : $\begin{tikzcd} L \arrow[loop above, "\overline{\sqcup}"] \end{tikzcd}$ Find the first blank square to the left of the currently scanned square.

# Examples of basic machines

- $R_{\sqcup}$ : $\overset{\overline{\sqcup}}{\overset{\circlearrowright}{R}}$ Find the first blank square to the right of the currently scanned square.
- $R_{\overline{\sqcup}}$ : $\overset{\sqcup}{\overset{\circlearrowright}{R}}$ Find the first non-blank square to the right of the currently scanned square.
- $L_{\sqcup}$ : $\overset{\overline{\sqcup}}{\overset{\circlearrowright}{L}}$ Find the first blank square to the left of the currently scanned square.
- $L_{\overline{\sqcup}}$ : $\overset{\sqcup}{\overset{\circlearrowright}{L}}$ Find the first non-blank square to the left of the currently scanned square.

## Example (The copying machine C)

If $C$ starts with input $w \in \{\Sigma \setminus \sqcup\}^\star$ to the left of the reading head, and blank spaces to the right, it copies $w$, i.e $\rhd \sqcup w\underline{\sqcup}$ to $\rhd \sqcup w \sqcup w\underline{\sqcup}$

# Computing with Turing Machines

- How TM will outperform all the machines introduced so far?

# Computing with Turing Machines

- How TM will outperform all the machines introduced so far?
- We need to define how it generates/recognizes a language

# Computing with Turing Machines

- How TM will outperform all the machines introduced so far?
- We need to define how it generates/recognizes a language

**new policy:** initial configuration: $(s, \triangleright\underline{\sqcup}w)$ , no blank symbol in $w$.

- How TM will outperform all the machines introduced so far?
- We need to define how it generates/recognizes a language

**new policy:** initial configuration: $(s, \triangleright \sqcup w)$ , no blank symbol in $w$.

**Definition**

Let $M = (K, \Sigma, \delta, s, H)$ be a TM with $H = \{y, n\}$. Any halting configuration whose state component is $y$ is called an **accepting configuration**, and any halting configuration whose state component is $n$ is called a **rejecting configuration**.

# Computing with Turing Machines

- How TM will outperform all the machines introduced so far?
- We need to define how it generates/recognizes a language

**new policy:** initial configuration: $(s, \triangleright\underline{\sqcup}w)$ , no blank symbol in $w$.

## Definition

Let $M = (K, \Sigma, \delta, s, H)$ be a TM with $H = \{y, n\}$. Any halting configuration whose state component is $y$ is called an **accepting configuration**, and any halting configuration whose state component is $n$ is called a **rejecting configuration**.

- A TM $M$ accepts $w \in (\Sigma \setminus \{\triangleright, \sqcup\})^{\star}$ if $(s, \underline{\sqcup}w) \vdash_M^{\star} (y, \triangleright u\underline{a}v)$

# Computing with Turing Machines

- How TM will outperform all the machines introduced so far?
- We need to define how it generates/recognizes a language

**new policy:** initial configuration: $(s, \triangleright\underline{\sqcup}w)$ , no blank symbol in $w$.

## Definition

Let $M = (K, \Sigma, \delta, s, H)$ be a TM with $H = \{y, n\}$. Any halting configuration whose state component is $y$ is called an **accepting configuration**, and any halting configuration whose state component is $n$ is called a **rejecting configuration**.

- A TM $M$ accepts $w \in (\Sigma \setminus \{\triangleright, \sqcup\})^\star$ if $(s, \underline{\sqcup}w) \vdash_M^\star (y, \triangleright u\underline{a}v)$
- A TM $M$ rejects $w \in (\Sigma \setminus \{\triangleright, \sqcup\})^\star$ if $(s, \underline{\sqcup}w) \vdash_M^\star (n, \triangleright u\underline{a}v)$

# Computing with Turing Machines

Let $\Sigma_0 \subseteq \Sigma \setminus \{\triangleright, \sqcup\}$ be the input alphabet. TM can use extra symbols $\Sigma \setminus \Sigma_0$ for computation.

# Computing with Turing Machines

Let $\Sigma_0 \subseteq \Sigma \setminus \{\triangleright, \sqcup\}$ be the input alphabet. TM can use extra symbols $\Sigma \setminus \Sigma_0$ for computation.

## Definition

A TM $M$ **decides** $L \subseteq \Sigma_0^\star$ if for any string $w \in \Sigma_0^\star$ the following is true:

- if $w \in L$, then $M$ accepts $w$,
- if $w \notin L$, then $M$ rejects $w$.

$L$ is called **recursive** if there is a TM $M$ that decides it.

## Example

$L = \{a^n b^n c^n \mid n \geq 0\}$. Write a TM $M$ that decides $L$. Show a computation over $aabbcc$.

# Computing with Turing Machines

- FSA, PDA : accept or reject
- TM in addition to accept or reject, there is a third option. It may fail to halt. (does not give an answer)

# Recursive Functions

## Definition

Let $M = (K, \Sigma, \delta, s, \{h\})$ be a TM with input alphabet $\Sigma_0 \subseteq \Sigma \setminus \{\triangleright, \sqcup\}$. Given $w \in \Sigma_0^\star$, suppose $M$ halts on $w$ with $y \in \Sigma_0^\star$ on the tape, i.e., $(s, \underline{\sqcup} w) \vdash_M^\star (h, \underline{\sqcup} y)$. Then $y$ is called the output of $M$ on $w$ and shown with $M(w) = y$.

A function $f : \Sigma_0^\star \to \Sigma_0^\star$ is called **recursive** if there is a Turing Machine $M$ such that $(s, \underline{\sqcup} w) \vdash_M^\star (h, \underline{\sqcup} f(w))$ for any $w \in \Sigma_0^\star$. If it halts for all inputs, then $M$ computes function $f$.

# Recursive Functions

**Example**

Is $K : \Sigma^\star \to \Sigma^\star$ with $K(w) = ww$ recursive?

# Recursive Functions

Strings in $\{0,1\}^\star$ can be used to represent integers in binary notation. Thus a TM $M$ computing functions from $\{0,1\}^\star$ to $\{0,1\}^\star$ can be thought as computing functions from natural numbers to natural numbers.

**Example**

Write a machine to compute $f(n) = n + 1$.

# Recursively Enumerable

## Definition

Let $M = (K, \Sigma, \delta, s, H)$ be a TM, $\Sigma_0 \subseteq \Sigma \setminus \{\triangleright, \sqcup\}$ be an alphabet and $L \subseteq \Sigma_0^\star$ be a language. M **semi-decides** $L$ if for any string $w \in \Sigma_0^\star$, M halts on $w$ if and only if $w \in L$.

A language $L$ is **recursively enumerable** if and only if there exists a TM $M$ that semidecides $L$.

# Recursively Enumerable

**Definition**

Let $M = (K, \Sigma, \delta, s, H)$ be a TM, $\Sigma_0 \subseteq \Sigma \setminus \{\triangleright, \sqcup\}$ be an alphabet and $L \subseteq \Sigma_0^\star$ be a language. M **semi-decides** $L$ if for any string $w \in \Sigma_0^\star$, M halts on $w$ if and only if $w \in L$.
A language $L$ is **recursively enumerable** if and only if there exists a TM $M$ that semidecides $L$.

**Theorem**

*If a language is recursive, then it is recursively enumerable.*

# Recursively Enumerable

## Definition

Let $M = (K, \Sigma, \delta, s, H)$ be a TM, $\Sigma_0 \subseteq \Sigma \setminus \{\triangleright, \sqcup\}$ be an alphabet and $L \subseteq \Sigma_0^\star$ be a language. M **semi-decides** $L$ if for any string $w \in \Sigma_0^\star$, M halts on $w$ if and only if $w \in L$.
A language $L$ is **recursively enumerable** if and only if there exists a TM $M$ that semidecides $L$.

## Theorem

*If a language is recursive, then it is recursively enumerable.*

## Theorem

*If a language is recursive, then its complement is also recursive.*