Aalto University
School of Science

CS-C2160 Theory of Computation

Lecture 8: Turing Machines

Pekka Orponen
Aalto University
Department of Computer Science

Spring 2021

Topics:

- Turing machines
- Extensions of Turing machines
  - Multitrack machines
  - Multitape machines
  - Nondeterministic machines
- Excursion: The halting problem, first encounter

Material:

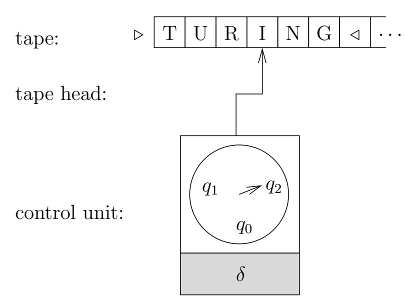
- In Finnish: Sections 4.1–4.2 and 6.1 in the Finnish lecture notes
- In English: Sections 3.1–3.3 in the Sipser book, multitrack machines on these slides

---

Turing Machines

8.1 Turing machines

Alan Turing 1935–36



- A Turing machine is like a finite automaton and has a tape ...
- but it can move both left and right on the tape
- and it can also write on the tape, not just read it.
- In addition, the tape is unbounded to the right.

The Church-Turing Thesis (∼1936)

Any mechanically (= physically) solvable computational problem can also be solved with a Turing machine.

Other models of computation that are equivalent to Turing machines:

- Recursive function systems by Gödel and Kleene (1936)
- Church's λ-calculus (1936)
- String rewriting systems by Post (1936) and Markov (1951)
- All current programming languages (when the amount of and access to the memory are not limited)

From the modern point of view:

- *Turing machines $\equiv$ (assembly-language) computer programs*

Definition 8.1

A *Turing machine*[a] is a tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}),$$

where

- Q is the finite set of *states*,
- Σ is the finite *input alphabet*,
- $\Gamma \supseteq \Sigma$ is the finite *tape alphabet* (we assume $\triangleright, \triangleleft \notin \Gamma$),
- $\delta : (Q - \{q_{\text{acc}}, q_{\text{rej}}\}) \times (\Gamma \cup \{\triangleright, \triangleleft\}) \to Q \times (\Gamma \cup \{\triangleright, \triangleleft\}) \times \{L, R\}$ is the *transition function*,
- $q_0 \in Q$ is the *start state* ($q_0 \neq q_{\text{acc}}$ and $q_0 \neq q_{\text{rej}}$),
- $q_{\text{acc}} \in Q$ is the *accept state*, and
- $q_{\text{rej}} \in Q$ is the *reject state* ($q_{\text{rej}} \neq q_{\text{acc}}$).

[a] Sipser's book uses a slightly different (but effectively equivalent) formalisation of Turing machines that (i) does not contain the start- and end-of-tape symbols $\triangleright$ and $\triangleleft$ but (ii) has a special "blank" symbol for the yet-unused positions on the tape.

The interpretation for a value

$$\delta(q, a) = (q', b, \Delta)$$

of the transition function is that when in state q and reading symbol a on the tape, the machine:

- moves to state q',
- writes symbol b at the same position on the tape, and
- moves the tape head one position in direction Δ ($L \sim$ "left", $R \sim$ "right").

The set of symbols the machine is allowed to write and the move directions are restricted in some cases:

- The transition function is not defined for the states q_{acc} and q_{rej}. When in either of these states, the machine *halts* immediately.
- For all transitions $\delta(q, a) = (q', b, \Delta)$ it is required that:
  - if $a = \triangleright$, then $b = \triangleright$ and $\Delta = R$
    That is, the start-of-tape symbol is never overwritten and the machine cannot move left beyond that symbol (i.e., off the tape).
  - $b = \triangleright$ is allowed only if $a = \triangleright$
    In other words, new start-of-tape symbols cannot be written.
  - $b = \triangleleft$ is allowed only if $a = \triangleleft$ and $\Delta = L$
    The machine does not explicitly write new end-of-tape symbols; they are introduced automatically when the machine moves past (and overwrites) the current end-of-tape symbol.

- A *configuration* of the machine is a tuple

$$(q,u,a,v) \in Q \times \Gamma^* \times (\Gamma \cup \{\varepsilon\}) \times \Gamma^*,$$

  where possibly $a = \varepsilon$ if also either $u = \varepsilon$ or $v = \varepsilon$.
- Interpretation: the machine is in state q and the contents of the tape are (i) from the beginning to the left of the tape head u, (ii) at the tape head position a and (iii) from the right of the tape head to the end of the tape v.
- When at the very start/end of the tape, $a = \varepsilon$ and $u = \varepsilon / v = \varepsilon$. In the "start" case $u = \varepsilon$, the machine is thought to read the symbol $\rhd$ and in the "end" case $v = \varepsilon$ the symbol $\lhd$.
- The *start configuration on input $x = a_1 a_2 \ldots a_n$* is the tuple

$$(q_0, \varepsilon, a_1, a_2 \ldots a_n).$$

- A configuration (q,u,a,v) is more compactly denoted as $(q, u\underline{a}v)$ and the start configuration on input x as $(q_0, \underline{x})$

- A configuration (q,w) *leads in one step* to (or *yields*) configuration (q',w'), denoted as

$$(q,w) \underset{M}{\vdash} (q',w'),$$

  as determined by the following rules:
  - if $\delta(q,a) = (q',b,R)$, then $(q, u\underline{a}cv) \underset{M}{\vdash} (q', ub\underline{c}v)$;
  - if $\delta(q,a) = (q',b,L)$, then $(q, uc\underline{a}v) \underset{M}{\vdash} (q', u\underline{c}bv)$;
  - if $\delta(q,\rhd) = (q',\rhd,R)$, then $(q, \underline{\varepsilon}cv) \underset{M}{\vdash} (q', \underline{c}v)$;
  - if $\delta(q,\lhd) = (q',b,R)$, then $(q, u\underline{\varepsilon}) \underset{M}{\vdash} (q', ub\underline{\varepsilon})$;
  - if $\delta(q,\lhd) = (q',b,L)$, then $(q, uc\underline{\varepsilon}) \underset{M}{\vdash} (q', u\underline{c}b)$;
  - if $\delta(q,\lhd) = (q',\lhd,L)$, then $(q, uc\underline{\varepsilon}) \underset{M}{\vdash} (q', u\underline{c})$.

  where $q,q' \in Q$, $u,v \in \Gamma^*$, $a,b \in \Gamma$ and $c \in \Gamma \cup \{\varepsilon\}$.
- Configurations of form (q_{acc}, w) and (q_{rej}, w) do not yield any other configuration. In these configurations the machine *halts*.

- A configuration (q,w) *leads* to a configuration (q',w'), denoted as

$$(q,w) \underset{M}{\vdash}^* (q',w'),$$

  if there is a finite sequence of configurations (q_0,w_0), (q_1,w_1), $\ldots$, (q_n,w_n), $n \geq 0$, such that

$$(q,w) = (q_0,w_0) \underset{M}{\vdash} (q_1,w_1) \underset{M}{\vdash} \cdots \underset{M}{\vdash} (q_n,w_n) = (q',w').$$

- A Turing machine M *accepts* a string $x \in \Sigma^*$ if

$$(q_0,\underline{x}) \underset{M}{\vdash}^* (q_{\text{acc}}, w) \qquad \text{for some } w \in \Gamma^*;$$

  otherwise M *rejects* x.
- The language *recognised* by the machine M is

$$\mathcal{L}(M) = \{x \in \Sigma^\star \mid (q_0,\underline{x}) \underset{M}{\vdash}^* (q_{\text{acc}}, w) \text{ for some } w \in \Gamma^*\}.$$

Example:

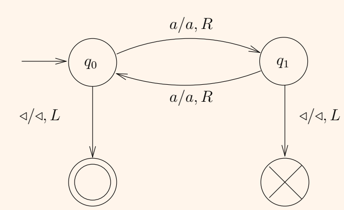
The (regular) language $\{a^{2k} \mid k \geq 0\}$ can be recognised with the Turing machine

$$M = (\{q_0, q_1, q_{\text{acc}}, q_{\text{rej}}\}, \{a\}, \{a\}, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}),$$

where

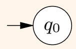
$$\begin{aligned}
\delta(q_0,a) &= (q_1,a,R),\\
\delta(q_1,a) &= (q_0,a,R),\\
\delta(q_0,\lhd) &= (q_{\text{acc}},\lhd,L),\\
\delta(q_1,\lhd) &= (q_{\text{rej}},\lhd,L).
\end{aligned}$$

State diagram representation:

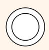


The notations used in the state diagram representation:

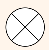
State q



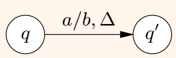
Start state



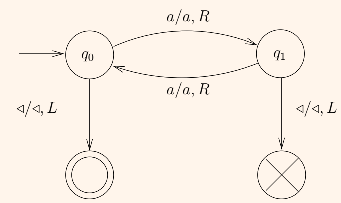
Accept state (q_{acc})



Reject state (q_{rej})



Transition $\delta(q,a) = (q', b, \Delta)$



The computation of machine M on input aaa:

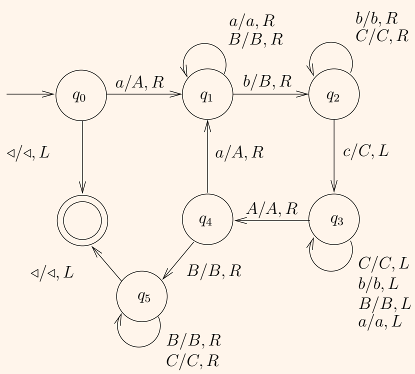
$$\begin{aligned}(q_0, \underline{a}aa) &\underset{M}{\vdash} (q_1, a\underline{a}a) \underset{M}{\vdash} (q_0, aa\underline{a}) \\ &\underset{M}{\vdash} (q_1, aaa\underline{\varepsilon}) \underset{M}{\vdash} (q_{\mathsf{rej}}, aa\underline{a}).\end{aligned}$$

The machine halts in state q_{rej} and thus $aaa \notin \mathcal{L}(M)$.

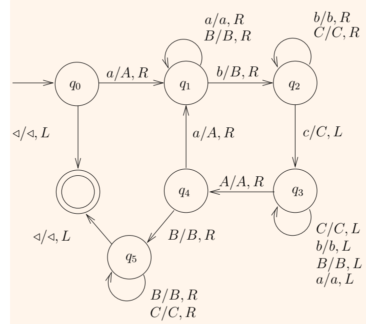
(*Note:* As this machine moves its tape head only to the right and accepts/rejects at the end of the string, it is effectively a finite automaton.)

Example:

A Turing machine recognising the (non-context-free) language $\{a^k b^k c^k \mid k \geq 0\}$:

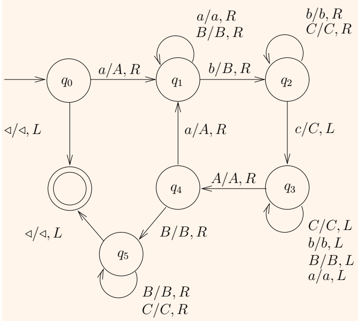


For the sake of clarity, the reject state is not drawn here, but again one interpretes that all the “missing transitions” implicitly lead to the reject state.



The computation of the machine on input $aabbcc$:

$(q_0, \underline{a}abbcc) \vdash (q_1, A\underline{a}bbcc) \vdash$
$(q_1, Aa\underline{b}bcc) \vdash (q_2, AaB\underline{b}cc) \vdash$
$(q_2, AaBb\underline{c}c) \vdash (q_3, AaB\underline{b}Cc) \vdash$
$(q_3, Aa\underline{B}bCc) \vdash (q_3, A\underline{a}BbCc) \vdash$
$(q_3, \underline{A}aBbCc) \vdash (q_4, A\underline{a}BbCc) \vdash$
$(q_1, AA\underline{B}bCc) \vdash (q_1, AAB\underline{b}Cc) \vdash$
$(q_2, AABB\underline{C}c) \vdash (q_2, AABBC\underline{c}) \vdash$
$(q_3, AABB\underline{C}C) \vdash (q_3, AAB\underline{B}CC) \vdash$
$(q_3, AA\underline{B}BCC) \vdash (q_3, A\underline{A}BBCC) \vdash$
$(q_4, AA\underline{B}BCC) \vdash (q_5, AAB\underline{B}CC) \vdash$
$(q_5, AABB\underline{C}C) \vdash (q_5, AABBC\underline{C}) \vdash$
$(q_5, AABBCC\underline{\varepsilon}) \vdash (q_{\mathsf{acc}}, AABBC\underline{C})$.



The computation of the machine on input $aabcbc$:

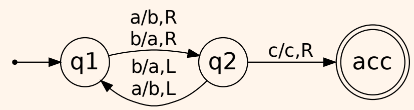
$(q_0, \underline{a}abcbc) \vdash (q_1, A\underline{a}bcbc) \vdash$
$(q_1, Aa\underline{b}cbc) \vdash (q_2, AaB\underline{c}bc) \vdash$
$(q_3, Aa\underline{B}Cbc) \vdash (q_3, A\underline{a}BCbc) \vdash$
$(q_3, \underline{A}aBCbc) \vdash (q_4, A\underline{a}BCbc) \vdash$
$(q_1, AA\underline{B}Cbc) \vdash (q_1, AAB\underline{C}bc) \vdash$
$(q_{\text{rej}}, AABC\underline{b}c)$.

Note

The definition of "language recognised by a machine" *does not require that the machine halts* on strings that do not belong to the language.

Example:

A Turing machine that enters an infinite loop on some inputs:

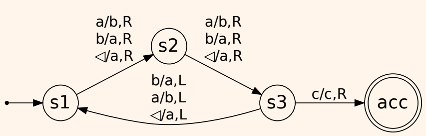


The computation on input abc:

$(q_1, \underline{a}bc) \vdash (q_2, b\underline{b}c) \vdash (q_1, \underline{b}ac) \vdash (q_2, a\underline{a}c) \vdash$
$(q_1, \underline{a}bc) \vdash (q_2, b\underline{b}c) \vdash (q_1, \underline{b}ac) \vdash (q_2, a\underline{a}c) \vdash \ldots$

Example:

A Turing machine that has an infinite computation using an unbounded amount of tape on some inputs:



The computation on input a:

$(q_1, \underline{a}) \vdash (q_2, b\underline{\varepsilon}) \vdash (q_3, ba\underline{\varepsilon}) \vdash (q_1, b\underline{a}a) \vdash (q_2, bb\underline{a}) \vdash (q_3, bbb\underline{\varepsilon}) \vdash$
$(q_1, bb\underline{b}a) \vdash (q_2, bba\underline{a}) \vdash (q_3, bbab\underline{\varepsilon}) \vdash (q_1, bba\underline{b}a) \vdash \ldots$

Extensions of Turing Machines

- Could we obtain even stronger models of computation if we extended the definition of Turing machines with new features?
- We could, for instance, allow multiple read/write tapes or non-determinism (like we did earlier with finite automata).
- In the following we study some such extensions and ...
- show that all the languages that one can recognise with such extended machines, can also be recognised with standard Turing machines.
- Witness the Church-Turing Thesis: *Any mechanically solvable computational problem can be solved with a (standard) Turing machine.*
- The extensions are also useful in designing machines for more complex purposes. (Cf. the uses of NFA as a design tool for DFA.)

8.2 Multitrack machines

- The tape of the Turing machine now consists of k parallel "tracks" that are all read and written in one computation step.

A	L	A	N	#	#	#	#			
M	A	T	H	I	S	O	N			...
T	U	R	I	N	G	#	#			

tape head:

- The transition function of such a machine is of form

$$\delta(q,(a_1,\dots,a_k)) = (q',(b_1,\dots,b_k),\Delta),$$

where $a_1,\dots,a_k$ are the symbols read on tracks $1,\dots,k$, $b_1,\dots,b_k$ the symbols written over them, and $\Delta \in \{L,R\}$ is the move direction as before.

- In the beginning of a computation, the input string is placed on the first track and the other tracks contain special "blank symbols" # in the same positions.

- Formally, a *k-track Turing machine* is a tuple

$$M = (Q,\Sigma,\Gamma,\delta,q_0,q_{\mathrm{acc}},q_{\mathrm{rej}}),$$

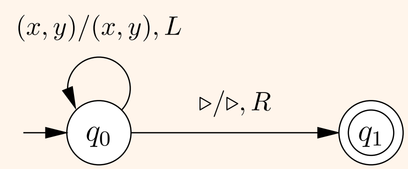
where the other components are as in the standard model but the transition function is:

$$\delta : (Q-\{q_{\mathrm{acc}},q_{\mathrm{rej}}\}) \times (\Gamma^k \cup \{\rhd,\lhd\}) \to Q \times (\Gamma^k \cup \{\rhd,\lhd\}) \times \{L,R\}.$$

- The "leads to" relation $\underset{M}{\vdash}$, start configuration etc. are defined similarly as in the standard model.

Example:

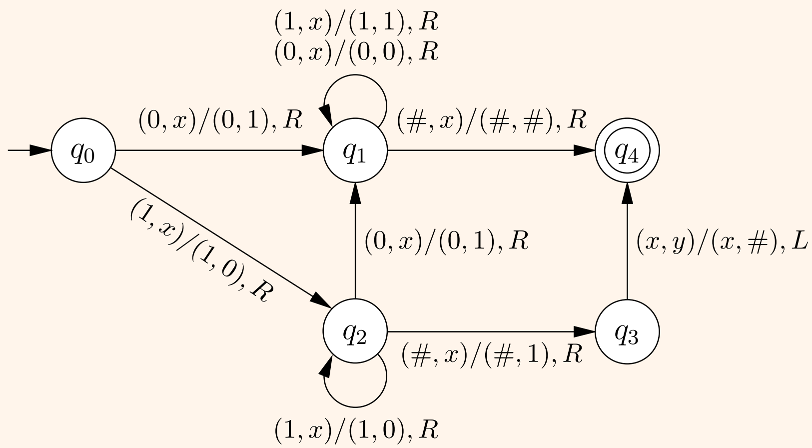
A 2-track Turing machine `rewind` that rewinds the tape head to the beginning of the tape:



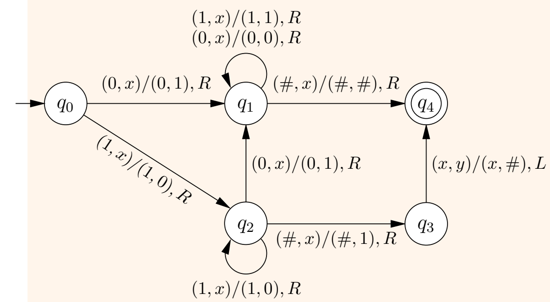
(The notation $(x,y)/(x,y),L$ is here a shorthand meant to cover all the transitions that can be obtained by replacing the variables x and y with some tape alphabet symbols.)

Example:

A 2-track Turing machine `succ` that computes the successor of a number given on track 1 to track 2 (the numbers are written in binary, least significant bit first):



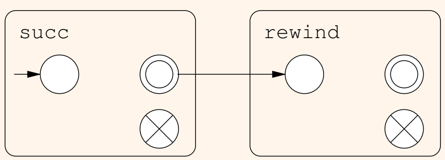
The computation of the machine on input number $3_{10} = 011_2$ (provided in lsb first representation on track 1):



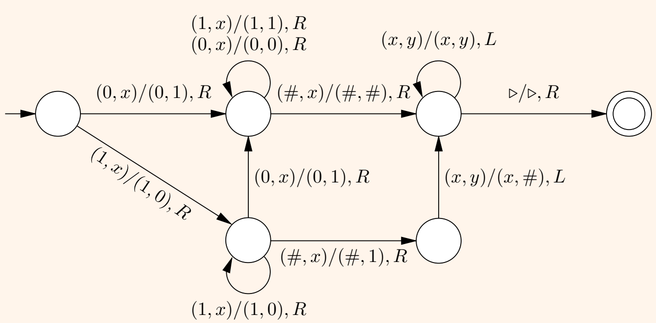
$$\left(q_0, \begin{matrix} 1 & 1 & 0 & \# \\ \underline{\#} & \# & \# & \# \end{matrix}\right) \vdash$$
$$\left(q_2, \begin{matrix} 1 & 1 & 0 & \# \\ 0 & \underline{\#} & \# & \# \end{matrix}\right) \vdash$$
$$\left(q_2, \begin{matrix} 1 & 1 & 0 & \# \\ 0 & 0 & \underline{\#} & \# \end{matrix}\right) \vdash$$
$$\left(q_1, \begin{matrix} 1 & 1 & 0 & \# \\ 0 & 0 & 1 & \underline{\#} \end{matrix}\right) \vdash$$
$$\left(q_4, \begin{matrix} 1 & 1 & 0 & \# \\ 0 & 0 & 1 & \# \end{matrix} \ \underline{\varepsilon}\right)$$

Example:

The sequential composition of the 2-track Turing machines `succ` and `rewind`:



which means the following:



Lemma 8.1

If a language L can be recognised with a k-track Turing machine, then it can be recognised with a standard Turing machine as well.

Proof

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ be a k-track Turing machine recognising the language L. An equivalent standard Turing machine $\widehat{M}$ can be constructed as follows:

$$\widehat{M} = (\widehat{Q}, \Sigma, \hat{\Gamma}, \hat{\delta}, \hat{q}_0, q_{\text{acc}}, q_{\text{rej}}),$$

where $\widehat{Q} = Q \cup \{\hat{q}_0, \hat{q}_1, \hat{q}_2\}$, $\hat{\Gamma} = \Sigma \cup \Gamma^k$ and for all $q \in Q$ we have

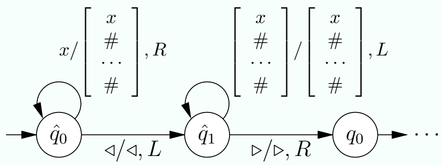
$$\hat{\delta}(q, \begin{bmatrix} a_1 \\ \vdots \\ a_k \end{bmatrix}) = (q', \begin{bmatrix} b_1 \\ \vdots \\ b_k \end{bmatrix}, \Delta),$$

when $\delta(q, (a_1, \ldots, a_k)) = (q', (b_1, \ldots, b_k), \Delta)$.

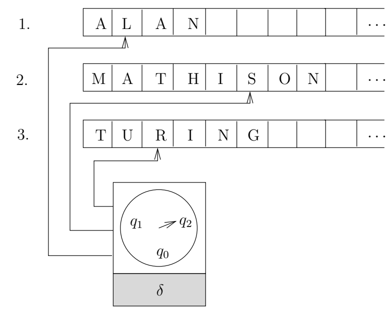
In the beginning of its computation, the machine $\widehat{M}$ must "lift" the input string to the (simulating) first track, meaning that it replaces the input $a_1a_2\ldots a_n$ with

$$\begin{bmatrix} a_1 \\ \# \\ \vdots \\ \# \end{bmatrix}\begin{bmatrix} a_2 \\ \# \\ \vdots \\ \# \end{bmatrix}\cdots\begin{bmatrix} a_n \\ \# \\ \vdots \\ \# \end{bmatrix}.$$

For this, the transition function of $\widehat{M}$ includes, in addition to the transitions copied from M, a small "preprocessor" sub-machine



8.3 Multitape machines



- We now allow a Turing machine to have k independent tapes, each with its own tape head.
- The machine reads and writes all the tapes in each step.
- In the beginning of the computation, the input is on the first tape, the other tapes are empty, and all the tape heads point to the beginning of their tapes.

- The transitions of such a machine are of form

$$\delta(q, a_1, \ldots, a_k) = (q', (b_1, \Delta_1), \ldots, (b_k, \Delta_k)),$$

  where
  - $a_1, \ldots, a_k$ are the symbols read from the tapes $1, \ldots, k$,
  - $b_1, \ldots, b_k$ are the symbols written on the tapes $1, \ldots, k$, and
  - $\Delta_1, \ldots, \Delta_k \in \{\mathsf{L}, \mathsf{R}, \mathsf{S}\}$ are the move directions of the tape heads (S means "stay", i.e., the tape head is not moved).
- Formally, a *k-tape Turing machine* is a tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}),$$

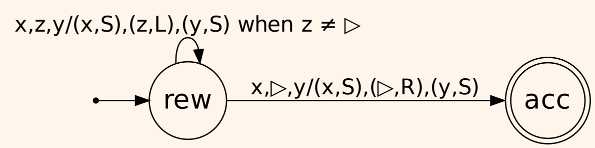
  where all the other components are as in the standard model, but the transition function is of form:

$$\delta : (Q - \{q_{\text{acc}}, q_{\text{rej}}\}) \times (\Gamma \cup \{\triangleright, \triangleleft\})^k \to Q \times ((\Gamma \cup \{\triangleright, \triangleleft\}) \times \{\mathsf{L}, \mathsf{R}, \mathsf{S}\})^k.$$

- The "leads to" relations and other concepts are defined similarly as in the standard model.

Example:

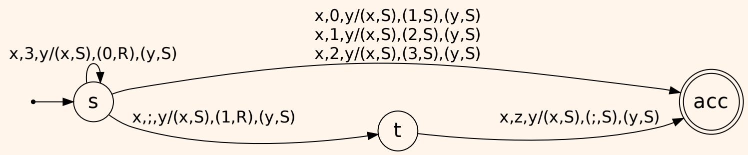
A 3-tape Turing machine `Rew2` that moves the tape head of the second tape to the beginning (other tape heads stay in their original places):



Here x, y, and z are again parameters for abbreviations and, for instance, the transition "$x, z, y/(x, \mathsf{S}), (z, \mathsf{L}), (y, \mathsf{S})$ when $z \neq \triangleright$" represents all the possible transitions that can be obtained by replacing x, y and z with any tape alphabet symbols (excluding the case $z = \triangleright$).

Example:

A 3-tape machine `4Succ2` that computes the successor of the number given on tape 2. Numbers are in base 4, in “least significant digit first” order and terminated by a semicolon. The machine shall be started in a situation where the tape head of the second tape is in the beginning.



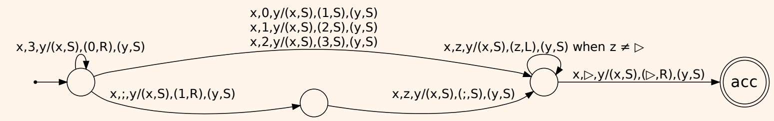
E.g., computing the successor 100_4 of 33_4 (i.e., 15_{10}):

$$\begin{pmatrix} & ab\underline{b}a \\ \mathrm{s}, & \underline{3}3; \\ & \underline{1}11; \end{pmatrix} \vdash \begin{pmatrix} & ab\underline{b}a \\ \mathrm{s}, & 0\underline{3}; \\ & \underline{1}11; \end{pmatrix} \vdash \begin{pmatrix} & ab\underline{b}a \\ \mathrm{s}, & 00\underline{;} \\ & \underline{1}11; \end{pmatrix} \vdash \begin{pmatrix} & ab\underline{b}a \\ \mathrm{t}, & 001\underline{\varepsilon} \\ & \underline{1}11; \end{pmatrix} \vdash$$

$$\begin{pmatrix} & ab\underline{b}a \\ \mathrm{acc}, & 001\underline{;} \\ & \underline{1}11; \end{pmatrix}$$

Example:

The sequential composition of machines `4Succ2` and `Rew2`. It computes the successor of the number given on tape 2 (in base 4, “least significant digit first” order and ;-terminated) and rewinds the second tape head to the beginning after the computation.

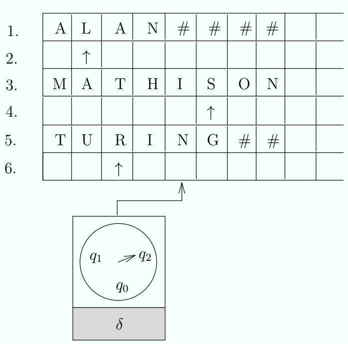


The machine shall be started in a situation where the tape head of the second tape is in the beginning.

Lemma 8.2

If a language L can be recognised with a k-tape Turing machine, then it can be recognised with a standard Turing machine as well.

Proof



Let

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\mathrm{acc}}, q_{\mathrm{rej}})$$

be a k-tape Turing machine recognising the language L. We can simulate it with a $2k$-track Turing machine $\widehat{M}$ as follows. The odd tracks $1, 3, 5, \ldots, 2k-1$ of $\widehat{M}$ correspond to the tapes $1, 2, \ldots, k$ of M and for each odd track, the following even track contains exactly one $\uparrow$ symbol that indicates the tape head position on the tape of the odd track.

- In the beginning, the input string is placed on the first track as usual, and in its first move $\widehat{M}$ writes the $\uparrow$ symbols to the first positions of the even tracks.
- After this, $\widehat{M}$ operates by “sweeping” across the tape forwards and backwards.
- On each forward sweep from the beginning to the end, $\widehat{M}$ collects information about which symbols are at the positions indicated by the $\uparrow$ symbols, i.e., at the tape head positions of the simulated machine M.
- Based on this information, $\widehat{M}$ then performs a backward sweep to the beginning and makes the changes on its multitrack tape (writes tape symbols, moves tape head markers $\uparrow$) that correspond to the changes made by a single transition of the simulated machine M.

The multitrack machine $\widehat{M}$ can then be simulated with a standard Turing machine, as presented in Lemma 8.1.

8.4 Nondeterministic machines

- Formally, a *nondeterministic Turing machine* is a tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}),$$

where the other components are as in the standard model but the transition function is of form:

$$\delta : (Q - \{q_{\text{acc}}, q_{\text{rej}}\}) \times (\Gamma \cup \{\triangleright, \triangleleft\}) \to \mathcal{P}(Q \times (\Gamma \cup \{\triangleright, \triangleleft\}) \times \{L, R\}).$$

- The interpretation of a value

$$\delta(q, a) = \{(q_1, b_1, \Delta_1), \ldots, (q_k, b_k, \Delta_k)\}$$

of the transition function is that, when in state q and reading tape symbol a, the machine can act according to *some* triple (q_i, b_i, Δ_i) in the list.

- For nondeterministic machines, the configurations, "leads to" relations etc. are defined as for the standard deterministic machines, except that the condition $\delta(q, a) = (q', b, \Delta)$ is replaced with the nondeterminic version $(q', b, \Delta) \in \delta(q, a)$.
- Because of this, the "leads to" relation $\underset{M}{\vdash}$ is no longer single-valued, meaning that a configuration (q, w) can now have many possible successor configurations (q', w') (i.e., those for which $(q, w) \underset{M}{\vdash} (q', w')$ holds).
- The language recognised by the machine M is now

$$\mathcal{L}(M) = \{x \in \Sigma^\star \mid (q_0, \underline{x}) \underset{M}{\vdash}^* (q_{\text{acc}}, w) \text{ for some } w \in \Gamma^*\}.$$

- That is, a string x belongs to the language recognised by a nondeterministic machine M if *some* finite sequence of configurations leads from the start configuration to an accepting configuration.

Example: Recognising composite numbers with nondeterministic Turing machines

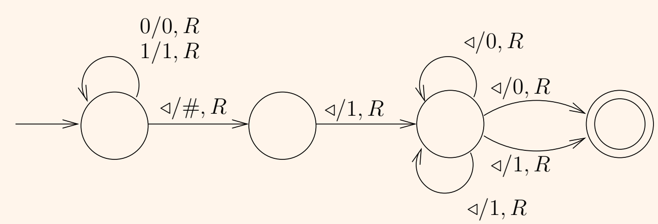
A non-negative integer n is a *composite* number if it has integer factors $p, q \geq 2$ s.t. $pq = n$. A non-negative integer that is not composite is either *unit* (1) or a *prime* number.

Assume that we already have a deterministic Turing machine `check_mult` that recognises the language

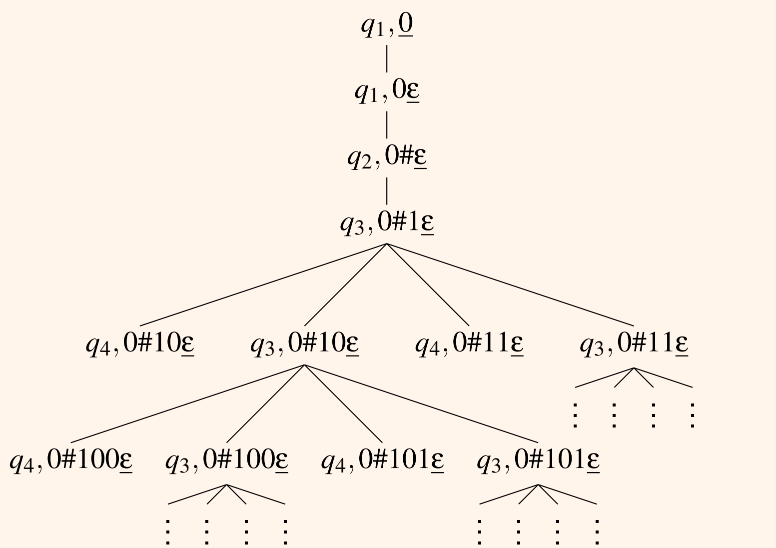
$$\mathcal{L}(\texttt{check\_mult}) = \{n\#p\#q \mid n, p, q \text{ are binary numbers and } n = pq\}.$$

In addition, let `go_start` be a deterministic Turing machine that moves the tape head to the beginning of the tape.

Furthermore, let `gen_int` be the following nondeterministic machine. It writes an *arbitrary* binary number (in the most-significant-bit-first order) that is greater than 1 at the end of the tape:



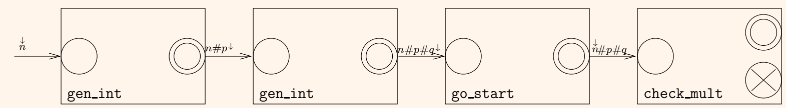
Some computations of machine `gen_int` on input string '0':



A nondetermistic Turing machine that recognises the language

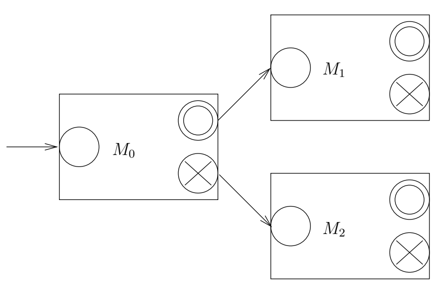
$$\mathcal{L}(\texttt{test\_composite}) = \{n \mid n \text{ is a binary compound number}\}$$

can now be constructed by combining the above mentioned machines:



The resulting machine accepts an input binary string n if and only if there exist binary numbers $p, q \geq 2$ for which $n = pq$ holds — that is, if and only if n is a composite number.

Note. A common diagram notation for an "if-then-else" combination of Turing machines:



Theorem 8.3

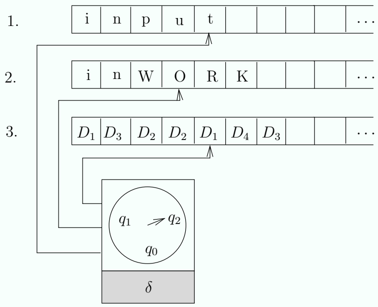
If a language L can be recognised with a nondeterministic Turing machine, then it can be recognised with a standard deterministic Turing machine as well.

Proof (idea)

- Let

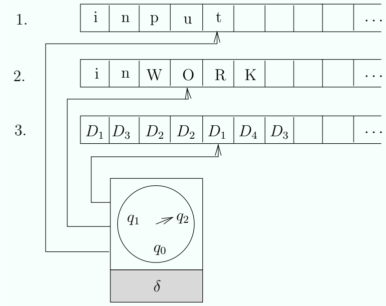
$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$$

  be a nondeterministic Turing machine recognising the language L.
- One can simulate M with a 3-tape deterministic machine $\widehat{M}$ that systematically explores all the computations of M until it finds a computation that ends in an accepting configuration — if such a computation exists.
- The 3-tape machine $\widehat{M}$ can then be transformed into a standard deterministic machine as presented in Lemmas 8.1 and 8.2.

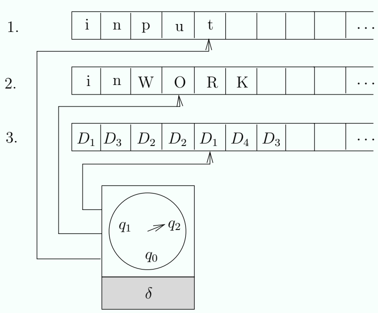


In more (but not full) detail:

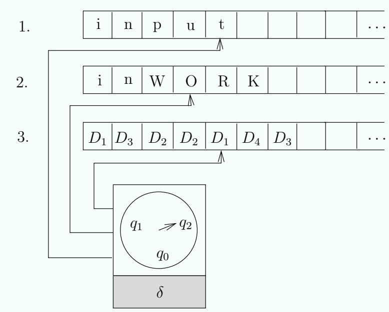
- On tape 1, $\widehat{M}$ stores the input string.
- On tape 2, $\widehat{M}$ simulates the work tape of machine M.
- In the beginning of each simulated computation, $\widehat{M}$ copies the input string from tape 1 to tape 2 and erases any spurious symbols that were left on tape 2 from the simulation of the previous computation.
- On tape 3, $\widehat{M}$ stores the "index" of the current computation of M to be simulated.



- Let r be the size of the biggest alternative-moves set in the transition function of M.
- Then machine $\widehat{M}$ has special tape symbols $D_1, \ldots, D_r$ and it enumerates all possible sequences of these on tape 3 in lexicographic ("shortlex") order: ε, D_1, D_2, …, D_r, D_1D_1, D_1D_2, …, D_1D_r, D_2D_1, …
- For each such generated sequence, $\widehat{M}$ simulates one (possibly incomplete) computation of machine M, in which the nondeterministic moves are made according to the sequence currently listed on tape 3.



- For instance, if tape 3 contains the sequence $D_1D_3D_2$, then the simulated computation takes the first choice in the first move, the third in the second, and the second in the third.
- If this computation did not end up in an accepting configuration, the next sequence $D_1D_3D_3$ is generated on tape 3 and a new simulation is performed.
- If the sequence on tape 3 is not valid because it contains a too large choice number at some point, the simulated computation is simply cancelled and the next sequence is generated.



Clearly this systematic simulation of computations of M leads $\widehat{M}$ to accept the input string if and only if M has an accepting computation. If M has no accepting computations on an input string, then machine $\widehat{M}$ does not halt.[a]

[a] With a bit more bookkeeping, the machine $\widehat{M}$ could also reject the input string (and halt) if for some n, all the computations of M of length n or less halt in a rejecting configuration. Even in this case, $\widehat{M}$ would obviously not halt if M had some nonhalting computations.

* Excursion: The Halting Problem, First Encounter

8.5 The halting problem

- As will be seen in Lecture 9, there are infinitely many more languages than Turing machines (or C/Python/Scala... programs).
- As languages correspond to decision problems, this means that *not all decision problems can be solved*.
- What about concrete examples of such undecidable problems?
- The best-known example is so-called *Turing's halting problem* (Alan Turing, 1936).
- In terms of C programs, we can formulate this result as follows:

Claim

There is no C function `halt(p,x)` that, given the source code string `p` of some C function and an input string `x` for `p`, outputs 1 if the execution of `p` on input `x` eventually terminates, and 0 if the execution of `p` on `x` never terminates. Here it is assumed that the programs can access an unlimited amount of memory.

Proof

Suppose, contrary to the claim, that such a function `halt` existed. By using this hypothetical function, we construct another function `confuse` as shown in the program code box below.
Let `c` denote the presented source code string of `confuse`, and study what happens if we run `confuse` on its own source code:

```
void confuse(char *q){
   int halt(char *p, char *x){
   ... /* Body of function 'halt' */
   }
   if (halt(q,q) == 1) while (1);
}
```

`confuse(c)` halts

$\Leftrightarrow$

`halt(c,c) == 1`

$\Leftrightarrow$

`confuse(c)` does not halt!

As we obtained a contradiction, we must deduce that the hypothetical halting testing function `halt` cannot exist.

In fact, as will be seen in the next lecture, there are *lots of such undecidable problems*.

The same in Python

The file `haltingTester.py` containing the hypothetical halting testing function `doesHalt`:

```
# (C) 2013 H. Ackerfrau
def doesHalt(sourceName, inputName):
  """Returns true if the program in file 'sourceName' halts
     when it is run on the input file 'inputName', false otherwise."""
  fs = open(sourceName, "r")
  fi = open(inputName, "r")
  ...
  return result

if __name__ == '__main__':
   source = sys.argv[1]
   input = sys.argv[2]
   halts = doesHalt(source, input)
   print(source+" halts on "+input+": "+halts)
```

The file `confuse.py`:

```
# (C) 2013 H. Ackerfrau
def doesHalt(sourceName, inputName):
  """Returns true if the program in file 'sourceName' halts
     when it is run on the input file 'inputName', false otherwise."""
  fs = open(sourceName, "r")
  fi = open(inputName, "r")
  ...
  return result

if __name__ == '__main__':
   sourceAndInput = sys.argv[1]
   halts = doesHalt(sourceAndInput, sourceAndInput)
   if halts:
     while True:
       pass
   print("I'll now halt :-)")
```

Does the execution of the command

`python confuse.py confuse.py`

terminate?