Griffith University

3130CIT Theory of Computation

(Based on slides by Harald Søndergaard of
The University of Melbourne)

Variants of Turing machines

Variants of Turing Machines

To try to make the Turing machine more powerful we could add to its features:

- Let its tape extend indefinitely in both directions.
- Let its tape have multiple tracks.
- Let there be several tapes, each with its independent tape head.
- Add nondeterminism.

It turns out that none of these increase a Turing machine's capabilities as a recogniser.

Multitape Machines

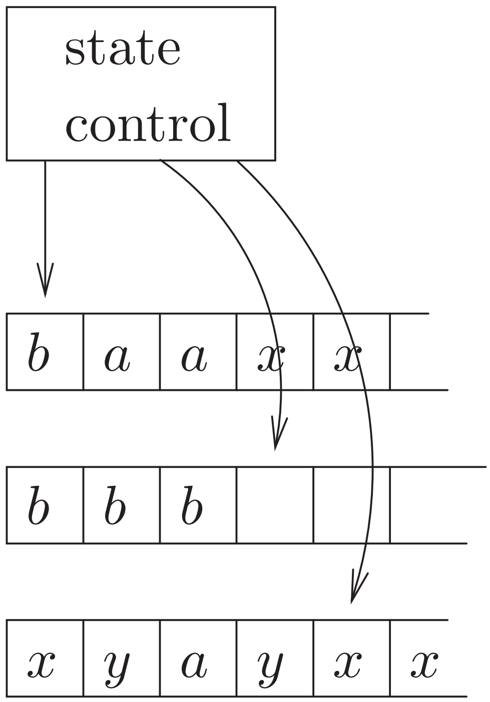
A multitape Turing machine has k tapes. It takes its input on tape 1, other tapes are initially blank.

The transition function now has type

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R\}^k$$

It specifies how the k tape heads behave when the machine is in state q_i, reading $a_1, \ldots a_k$:

$$\delta(q_i, a_1, \ldots, a_k) = (q_j, (b_1, \ldots, b_k), (d_1, \ldots, d_k))$$

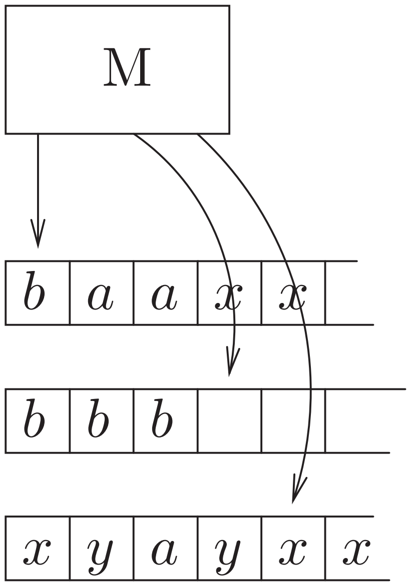


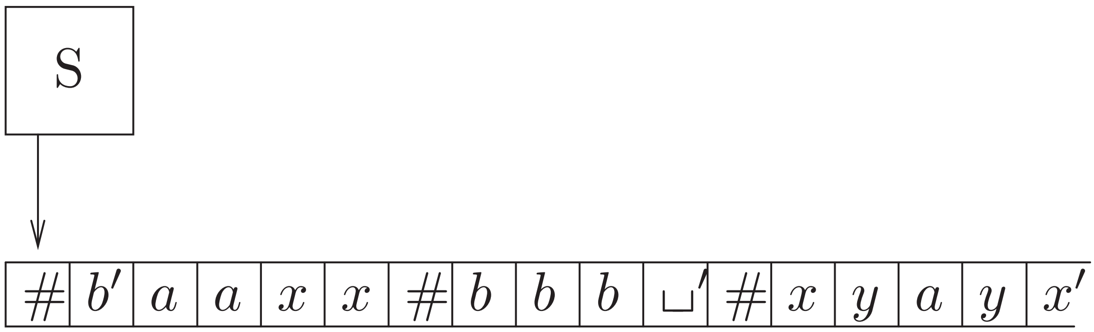
Multitape Machines (cont.)

Theorem: A language is Turing recognisable iff some multitape Turing machine recognises it.

Proof sketch: We show how to simulate a multitape machine M by a standard Turing machine S.

The standard machine has tape alphabet $\{\#\} \cup \Gamma \cup \Gamma'$ where $\#$ is a separator, not in $\Gamma \cup \Gamma'$, where there is a one-to-one correspondence between symbols in Γ and (marked) symbols in Γ'.





S reorganises its input $x_1 x_2 \cdots x_n$ into

$$\#x_1' x_2 \cdots x_n \underbrace{\#\sqcup'\# \cdots \#\sqcup'\#}_{k-1 \text{ times}}$$

Note how symbols of Γ' represent marked symbols from Γ, which denote the positions of the tape heads in the multitape machine.

Multitape Machines (cont.)

To simulate a move of M, S scans its tape to determine the marked symbols. S then scans the tape again, updating it according to M's transition function.

If a "virtual head" of M moves to a $\#$, S shifts that symbol, and every symbol after it, one cell to the right. In the vacant cell it writes $\sqcup$. It then continues to apply M's transition function.

Nondeterministic Turing Machines

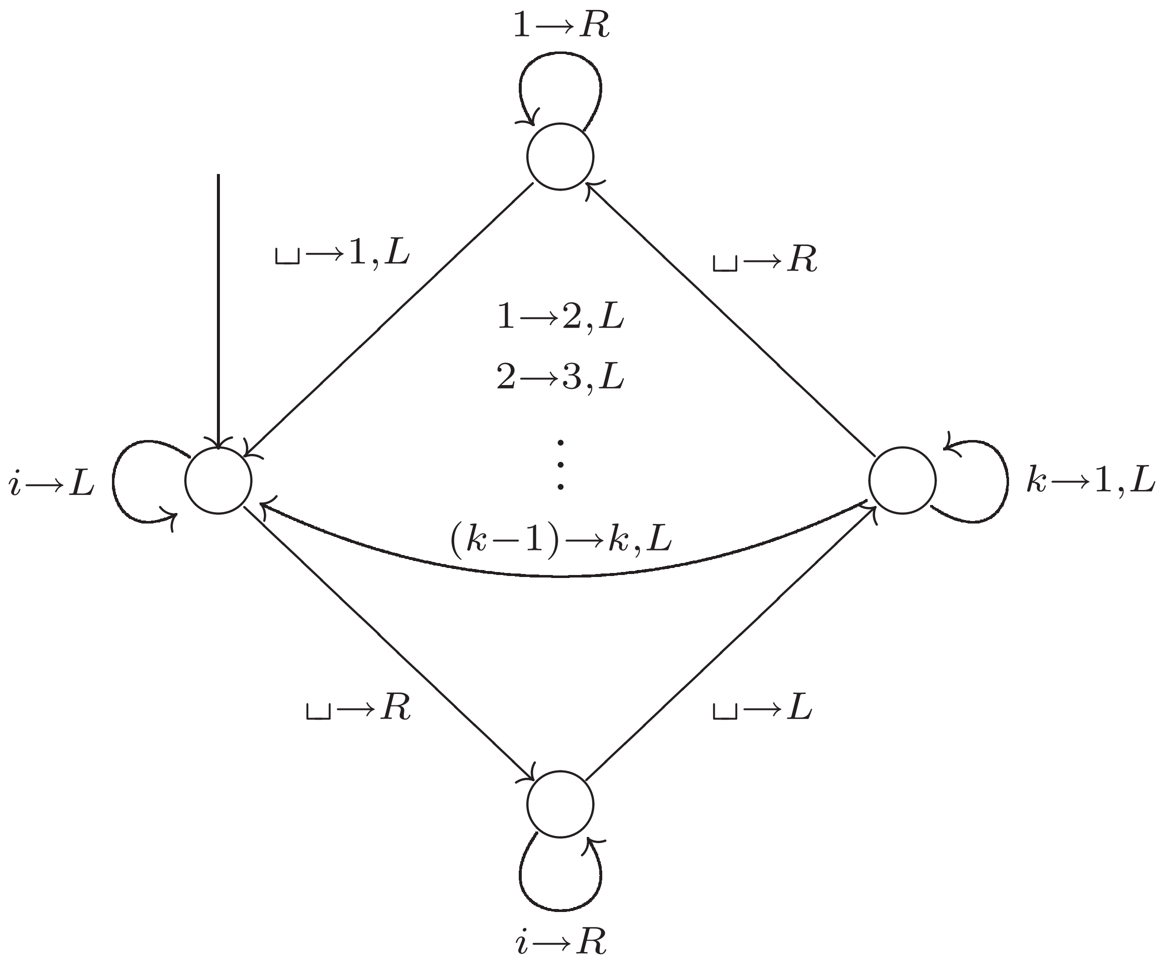
A nondeterministic Turing machine has a transition function of type

$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

If some computation branch lead to 'accept' then the machine accepts its input. This is the same type of nondeterminism as NFAs possess.

Nondet Turing Machines (cont.)

First, here is a deterministic machine to generate $\{1, \ldots, k\}^*$, in order of increasing length.



Try running this for $k = 3$.

Simulating Nondeterminism

Theorem: A language is Turing recognisable iff some nondeterministic Turing machine recognises it.

Proof sketch: We need to show that every nondeterministic Turing machine N can be simulated by a deterministic Turing machine D.

We show how it can be simulated by a 3-tape machine.

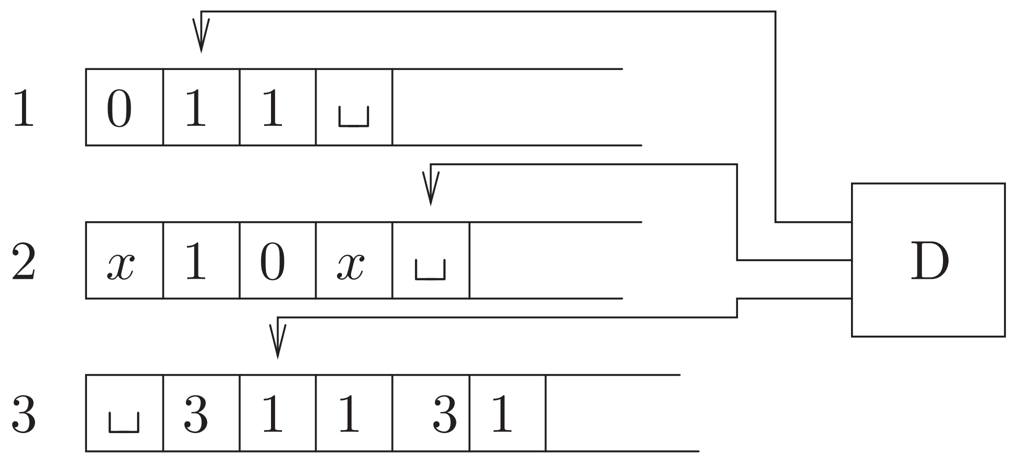
Let k be the largest number of choices, according to N's transition function, for any state/symbol combination.

Tape 1 contains the input.

Tape 3 holds progressively longer and longer sequences from $\{1, \ldots, k\}^*$.

Tape 2 is used to simulate N's behaviour for each fixed sequence of choices given by tape 3.

Simulating Nondeterminism (cont.)



1. Initially tape 1 contains input w.
   The other two tapes are empty.

2. Overwrite tape 2 by w.

3. Use tape 2 to simulate N. Tape 3 dictates how N should make its choices. If tape 3 gets exhausted, go to step 4. If N says *accept*, accept.

4. Generate the next "choice" string on tape 3.
   Go to step 2.

Enumerators

The Turing machine we built to generate all strings in $\{1, \ldots, k\}^*$ is an example of an *enumerator*.

We could imagine it being attached to a printer, and it would print all the strings in $\{1, \ldots, k\}^*$, one after the other, never terminating.

For an enumerator to enumerate a language L, for each $w \in L$, it must eventually print w. It is allowed to print w as often as it wants, and the strings can come in any order.

The reason why we also call Turing recognisable languages *recursively enumerable* is the following theorem.

Enumerators (cont.)

Theorem: L is Turing recognisable iff some enumerator enumerates L.

Proof: Let E enumerate L. Then we can build a Turing machine recognising L as follows:

1. Let w be the input.
2. Simulate E. For each string s output by E, if $s = w$, accept.

Conversely, let M recognise L. Then we can build an enumerator E by elaborating the enumerator from a few slides back: We can enumerate Σ^*, producing $s_1, s_2, \ldots$ Here is what E does:

1. Let $i = 1$.
2. Simulate M for i steps on each of $s_1, \ldots s_i$.
3. For each accepting computation, print that s.
4. Increment i and go to step 2.

Back to Algorithms

Hilbert's tenth problem (1900): Find an algorithm that determines whether a polynomial has an integral root.

As it turned out (Matijasevič 1970) no such algorithm exists.

This fact, however, can only be shown once we have a formal definition of what an algorithm *is*.

We need to argue that

$$\{p \mid p \text{ is a polynomial with integral root}\}$$

is not *decidable*.

The Church-Turing Thesis

$$\boxed{\begin{array}{c}\text{Computable}\\=\\\text{what a Turing machine can compute}\end{array}}$$

Note that we cannot hope to *prove* the Church-Turing thesis.

On the other hand, advances in physics could conceivably make the thesis false, in that some weird physical device might decide Turing machine halting, say.

The Church-Turing Thesis (cont.)

Note that

$$\{p \mid p \text{ is a polynomial with integral root}\}$$

is Turing *recognisable*.

To see this, here is how we can build a Turing machine M to recognise it.

Say the variables in p are x, y, and z.

M can enumerate all integer triples (i, j, k).

So M can evaluate p on each value triple (i, j, k) in turn.

If any of these evaluations give 0, M says *accept*.