COMP30026 Models of Computation

Decidable Languages

Harald Søndergaard

Lecture 19

Semester 2, 2017

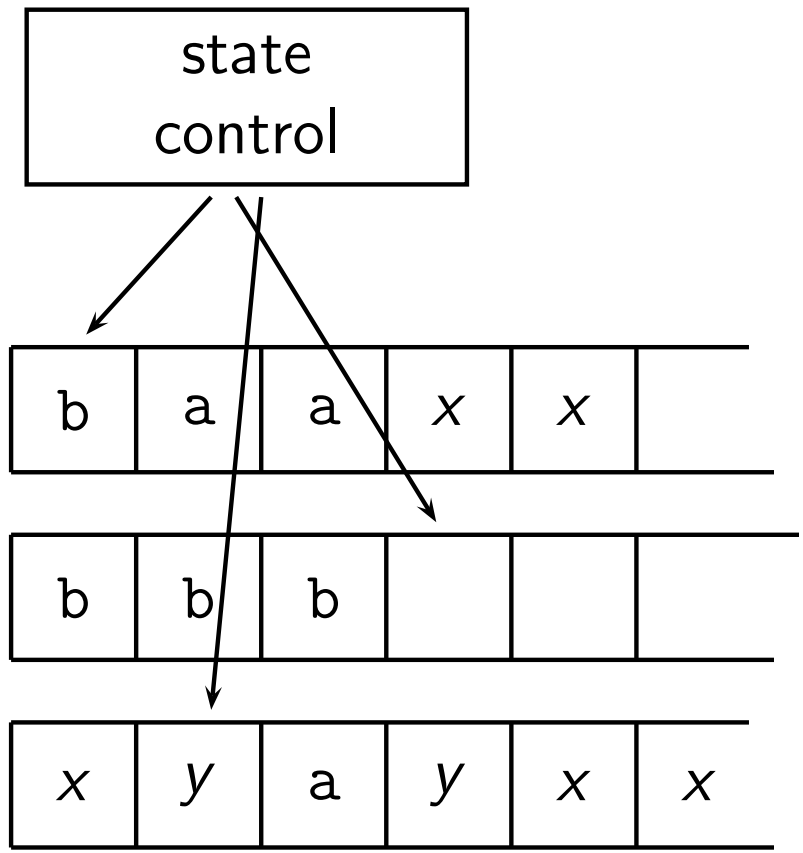
Multitape Machines

A multitape Turing machine has k tapes. It takes its input on tape 1, other tapes are 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 $a_1, \ldots a_k$:



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

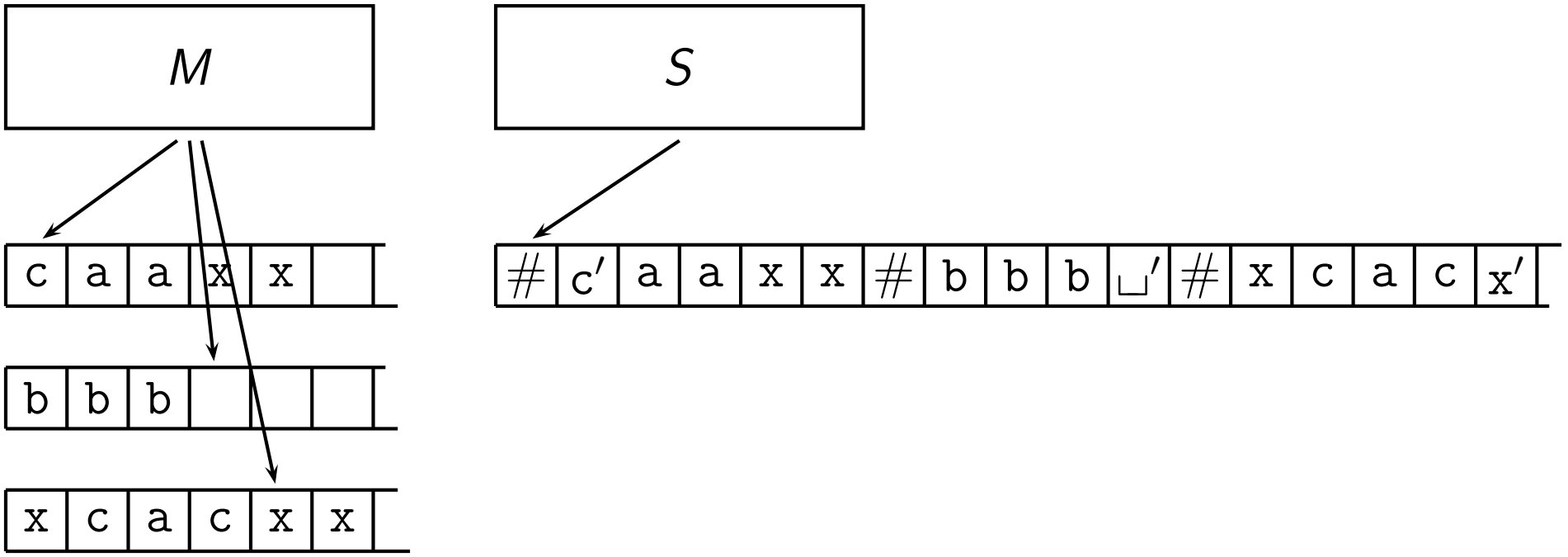
Simulating a Multitape Machine

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 $\Gamma \cup \Gamma'$, and there is some one-to-one correspondence between elements in Γ and elements in Γ'.

Simulating a Multitape Machine



S reorganises 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 elements of Γ' represent "marked" elements from Γ.

Simulating a Multitape Machine

Simulating an M move, S scans its tape to determine the marked symbols. On a second scan it updates the tape 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 $\sqcup$.

Nondeterministic Turing Machines

A nondeterministic Turing machine has a transition function of type

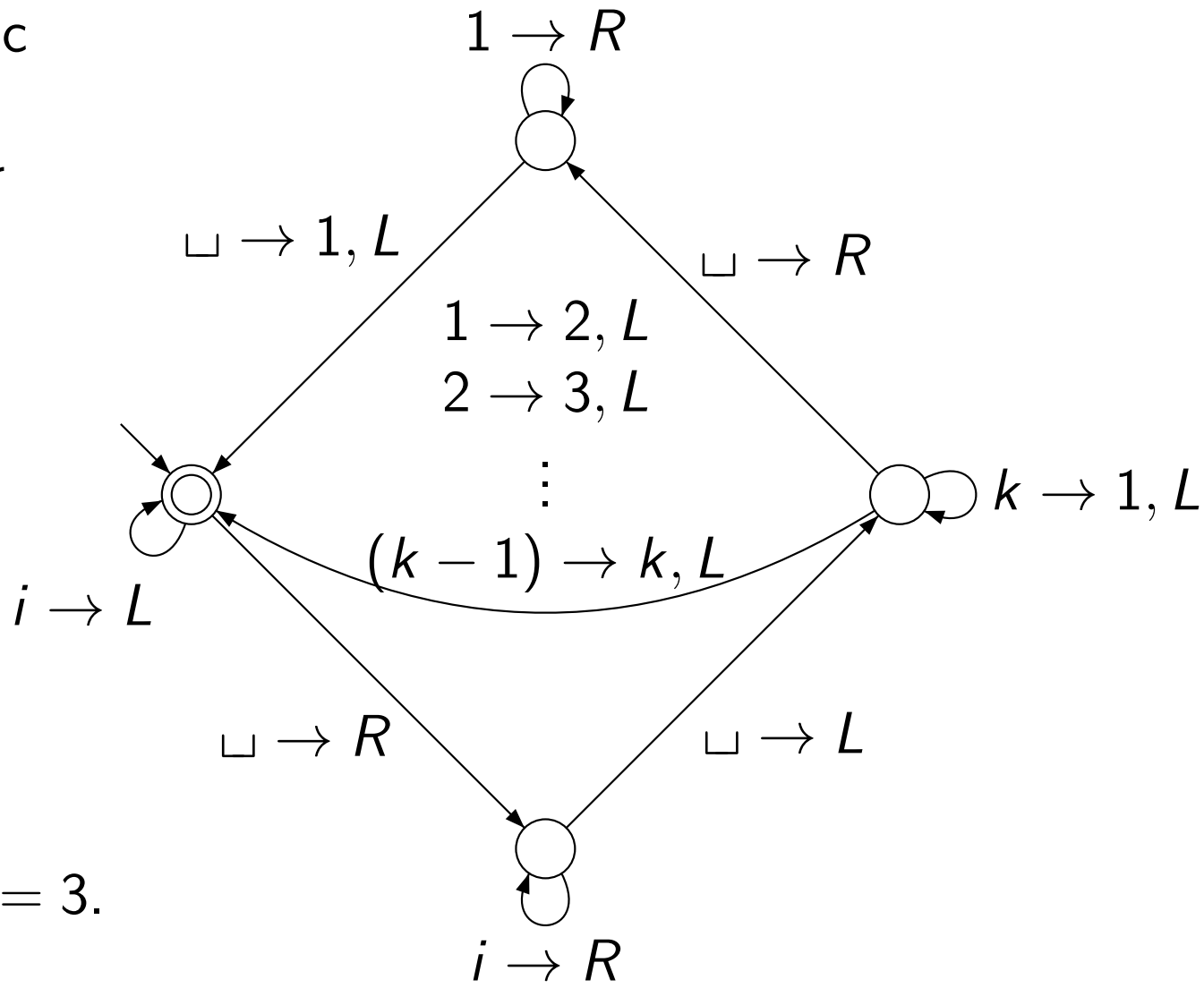
$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

If some computation branch leads to 'accept' then the machine accepts its input.

This is the same type of nondeterminism that an NFA possesses.

Simulating a Nondeterministic Turing Machine

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



Try running it for $k = 3$.

Simulating a Nondeterministic Turing Machine

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.

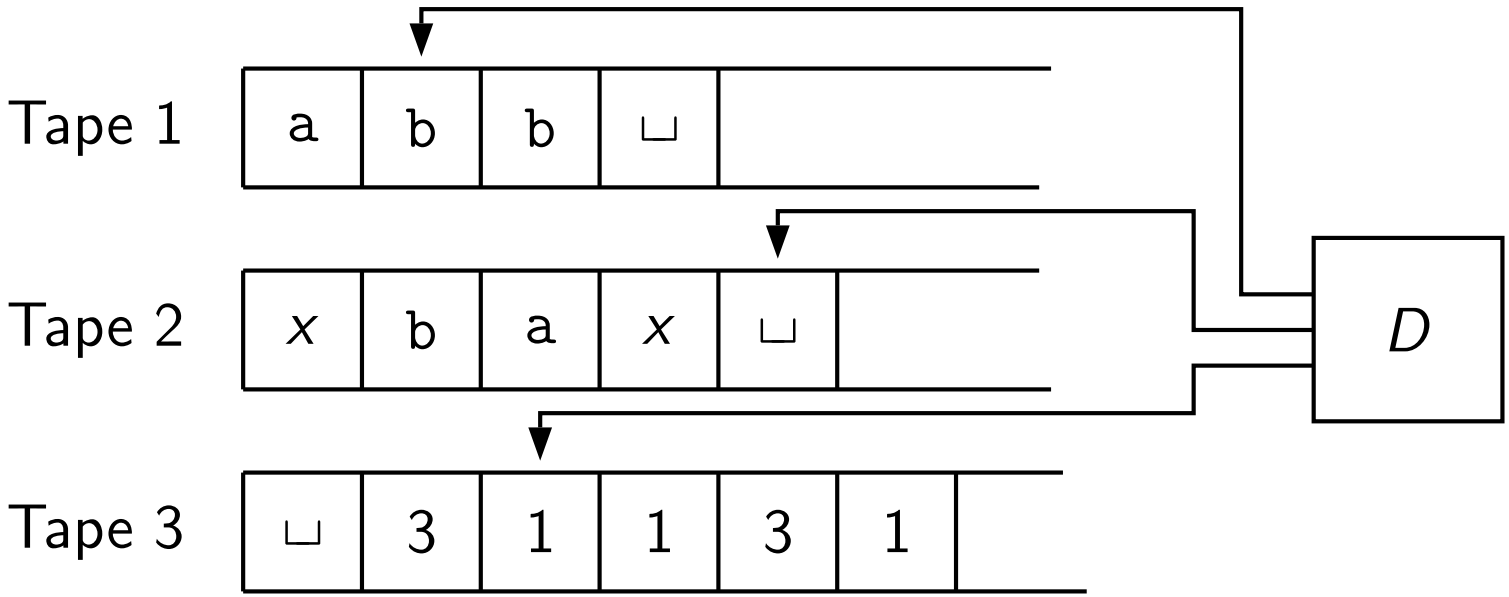
Let k be the largest number of choices, according to N's transition function, for any state/symbol combination.

Tape 1 contains the input.

Tape 3 holds 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 a Nondeterministic Turing Machine



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.

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

Enumerators

Thm: 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.

Enumerators

Thm: 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.

Decidable Problems

Problems regarding regular languages tend to be decidable.

We can phrase these problems as language decidability problems.

For example, the acceptance problem for DFAs is whether, given a DFA D and a string w, D accepts input w.

Since we can encode the DFA as a string, the acceptance problem can be seen as testing for membership of the language

$$A_{DFA} = \{\langle D, w\rangle \mid D \text{ is a DFA that accepts } w\}$$

By $\langle D, w\rangle$ we mean a (string) encoding of the pair D, w.

DFA Acceptance Is Decidable

Theorem: A_{DFA} is a decidable language.

Proof sketch: The crucial point is that it is possible for a Turing machine M to simulate a DFA D.

M finds on its tape, say

$$\underbrace{1\ldots n}_{Q}\#\#\underbrace{\mathtt{ab}\ldots\mathtt{z}}_{\Sigma}\#\#\underbrace{1\mathtt{a}2\#\ldots\#n\mathtt{b}n}_{\delta}\#\#\underbrace{1}_{q_0}\#\#\underbrace{3\ 7}_{F}\#\#\underbrace{\mathtt{baa}\ldots}_{w}\$$$

First M checks that the first five components represent a valid DFA, and if not, rejects.

Then M simulates the moves of D, keeping track of D's state and the current position in w, by writing these details on its tape, after \$.

When the last symbol in w has been processed, M accepts if D is in a state in F, and rejects otherwise.

TMs as Interpreters

We won't give the details of how the Turing machine simulates the DFA. Many tedious low-level programming steps are involved.

However, it should be clear that it is possible for a Turing machine to mimic DFA behaviour this way.

The description of D is nothing but a "program" and the claim is that a Turing machine can act as an interpreter for this language.

Turing machines themselves can be encoded as strings, and then a Turing machine can interpret Turing machines.

This is no more strange than the fact that we can write an interpreter for Haskell, say, in Haskell.

NFA Acceptance Is Decidable

Theorem:

$$A_{NFA} = \{\langle N, w\rangle \mid N \text{ is an NFA that accepts } w\}$$

is a decidable language.

Proof sketch: The procedure we gave for translating an NFA to an equivalent DFA was mechanistic and terminating, so a halting Turing machine can do that job.

Having written the encoding of the DFA on its tape, the Turing machine can then "run" the machine M from the previous proof.

DFA Emptiness Is Decidable

Theorem:

$$E_{DFA} = \{\langle D \rangle \mid D \text{ is a DFA and } L(D) = \emptyset\}$$

is decidable.

Proof sketch: We can design a Turing machine which takes $\langle D \rangle = (Q, \Sigma, \delta, q_0, F)$ as input and performs a reachability analysis:

1. Set $\mathit{reachable} = \{q_0\}$, D's start state.
2. Set $\mathit{new} = \{q \mid \delta(m, x) = q, m \in \mathit{reachable}\} \setminus \mathit{reachable}$.
3. If $\mathit{new} \neq \emptyset$, set $\mathit{reachable} = \mathit{reachable} \cup \mathit{new}$ and go to step 2.
4. If $\mathit{reachable} \cap F = \emptyset$, accept, otherwise reject.

DFA Equivalence Is Decidable

Theorem:

$$EQ_{DFA} = \{\langle A, B\rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$$

is decidable.

Proof sketch: We previously saw how it is possible to construct, from DFAs A and B, DFAs for $A \cap B$, $A \cup B$, and A^c.

These procedures are mechanistic and finite—a halting Turing machine M can perform them.

Hence from A and B, M can produce a DFA C to recognise

$$L(C) = \big(L(A) \cap L(B)^c\big) \cup \big(L(A)^c \cap L(B)\big)$$

Note that $L(C) = \emptyset$ iff $L(A) = L(B)$.

So M just needs to use the emptiness checker from before on C.

Generation by CFGs Is Decidable

Theorem:

$$A_{CFG} = \{\langle G, w\rangle \mid G \text{ is a CFG that generates } w\}$$

is decidable.

The proof relies on the fact that we can rewrite any CFG to a particular equivalent form, Chomsky Normal Form.

In Chomsky Normal Form, each production takes one of two forms:

$$A \to B\ C \qquad \text{or} \qquad A \to a$$

(With one exception: We also allow $S \to \epsilon$, where S is the grammar's start variable.)

Generation by CFGs Is Decidable

For every grammar in Chomsky Normal Form form, if string w can be derived then its derivation has exactly $2|w| - 1$ steps.

So to decide A_{CFG}, we can simply try out all possible derivations of that length, in finite time, and see if one generates w.

CFG Emptiness Is Decidable

Theorem:

$$E_{CFG} = \{\langle G \rangle \mid G \text{ is a CFG and } L(G) = \emptyset\}$$

is decidable.

Proof: We can design a Turing machine which takes $\langle G \rangle = (V, \Sigma, R, S)$ as input and performs a "producer" analysis:

1. Set *producers* $= \Sigma$, all of G's terminals.
2. Set $new = \left\{ A \;\middle|\; \begin{array}{l} \ulcorner A \to U_1 \cdots U_n \urcorner \in R, \\ \{U_1, \ldots, U_n\} \subseteq producers \end{array} \right\} \setminus producers$.
3. If $new \neq \emptyset$, set $producers = producers \cup new$ and go to step 2.
4. If $S \in producers$, reject, otherwise accept.

Every CFL Is Decidable

Two slides back we saw that it is decidable whether a CFG G generates a string w.

The decider, call it S, took $\langle G, w \rangle$ as input.

Now we are saying that any particular CFL L_0 is decidable:

Theorem: Every context-free language L_0 is decidable.

Proof: This is just saying that we can specialise the decider S. Let G_0 be a CFG for L_0. The decider for L_0 simply takes input w and runs S on $\langle G_0, w \rangle$.

The Hierarchy of Language Classes

The diagram shows the relations amongst language classes established so far.

But are there Turing recognisable languages that are not decidable?

As it turns out, yes.