Turing machines and linear bounded automata

Informatics 2A: Lecture 29

John Longley

School of Informatics
University of Edinburgh
jrl@inf.ed.ac.uk

25 November, 2011

The Chomsky hierarchy: summary

Level	Language type	Grammar rules	Accepting machines
3	Regular	$X \to \epsilon$, $X \to Y$, $X \to aY$	NFAs (or DFAs)
2	Context-free	$X \to \beta$	Nondeterministic pushdown automata
1	Context-sensitive	$\alpha \to \beta$ with $\lvert\alpha\rvert \leq \lvert\beta\rvert$	Nondet. linear bounded automata
0	Recursively enumerable	$\alpha \to \beta$ (unrestricted)	Turing machines

The length restriction in CSGs: some intuition

What's the motivation for the restriction $|\alpha| \leq |\beta|$ in context-sensitive rules?

Idea: in a context-sensitive derivation $S \Rightarrow \cdots \Rightarrow \cdots \Rightarrow s$, all the sentential forms are of length at most $|s|$.

This means that if L is context-sensitive, and we're trying to decide whether $s \in L$, we only need to consider possible sentential forms of length $\leq |s|$. So intuitively, we have the problem under control, at least in principle.

By contrast, without the length restriction, there's no upper limit on the length of intermediate forms that might appear in a derivation of s. So if we're searching for a derivation for s, how do we know when to stop looking? Intuitively, the problem here is wild and out of control.

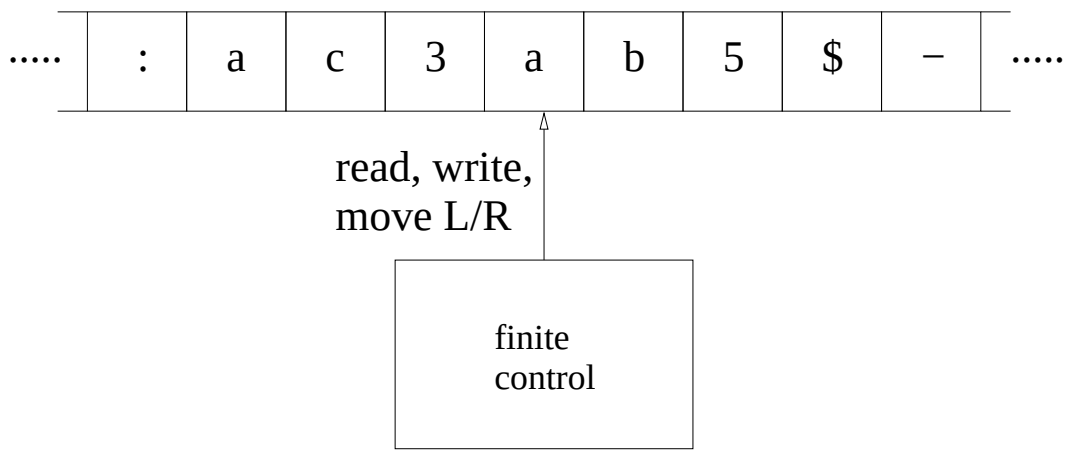
We'll see this intuition made more precise as we proceed.

Turing machines

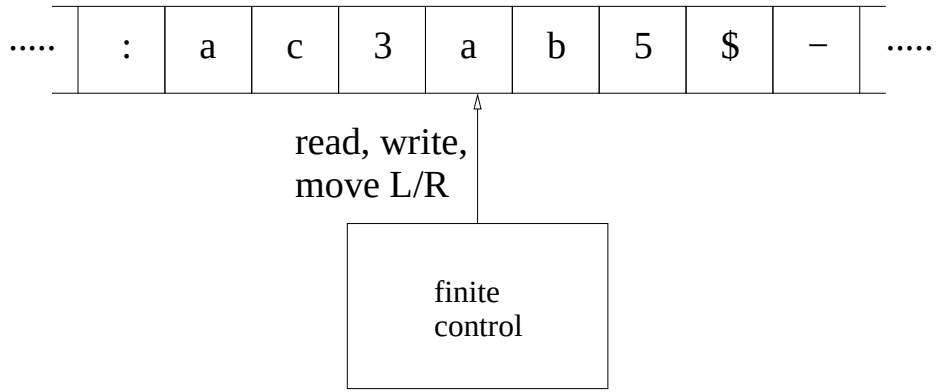
Recall that NFAs are 'essentially memoryless', whilst NPDAs are equipped with memory in the form of a stack.

To find the right kinds of machines for the top two Chomsky levels, we need to allow more general manipulation of memory.

A Turing machine essentially consists of a finite-state control unit, equipped with a memory tape, infinite in both directions. Each cell on the tape contains a symbol drawn from a finite alphabet Γ.



Turing machines, continued



At each step, the behaviour of the machine can depend on

- the current state of the control unit,
- the tape symbol at the current read position.

Depending on these things, the machine may then

- overwrite the current tape symbol with a new symbol,
- shift the tape left or right by one cell,
- jump to a new control state.

This happens repeatedly until (let's say) the control unit enters some final state.

Turing machines as acceptors

To use a Turing machine T as an acceptor for a language over Σ, assume $\Sigma \subseteq \Gamma$, and set up the tape with the test string $s \in \Sigma^*$ written left-to-right starting at the read position, and with blank symbols everywhere else.

Then let the machine run (maybe overwriting s), and if it enters the final state, declare that the original string s is accepted.

The language accepted by T (written $\mathcal{L}(T)$) consists of all strings s that are accepted in this way.

Theorem: A set $L \subseteq \Sigma^*$ is generated by some unrestricted (Type 0) grammar if and only if $L = \mathcal{L}(T)$ for some Turing machine T. So both Type 0 grammars and Turing machines lead to the same class of recursively enumerable languages.

Turing machines, a bit more formally

A Turing machine T consists of:

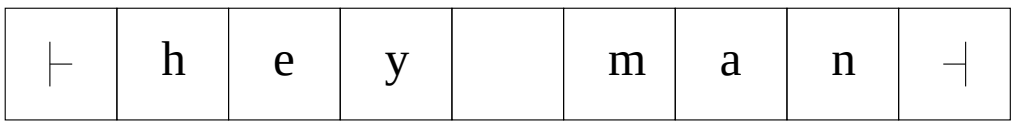
- A set Q of control states
- An initial state $i \in Q$
- A final (accepting) state $f \in Q$
- A tape alphabet Γ
- An input alphabet $\Sigma \subseteq \Gamma$
- A blank symbol $- \in \Gamma - \Sigma$
- A transition function $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$.

Numerous minor variations possible — differences unimportant.

Can then define formally what's meant by an accepting computation for a string $s \in \Sigma^*$ ...

Linear bounded automata

Suppose we modify our model to allow just a finite tape, initially containing just the test string s with endmarkers on either side:



The machine therefore has just a finite amount of memory, determined by the length of the input string. We call this a linear bounded automaton.

(LBAs are sometimes defined as having tape length bounded by a constant multiple of length of input string — doesn't make any difference in principle.)

Theorem: A language $L \subseteq \Sigma^*$ is context-sensitive if and only if $L = \mathcal{L}(T)$ for some non-deterministic linear bounded automaton T.

Rough idea: we can guess at a derivation for s. We can check each step since each sentential form fits within the tape.

Determinism vs. non-determinism: a curiosity

- At the bottom level of the Chomsky hierarchy, it makes no difference: every NFA can be simulated by a DFA.
- At the top level, the same happens. We've defined a 'deterministic' version of Turing machines — but any 'non-det Turing machine' can be simulated by a det one.
- At the context-free level, there is a difference: we need NPDAs to account for all context-free languages.

  (Example: $\Sigma^* - \{ss \mid s \in \Sigma^*\}$ is a context-free language whose complement isn't context-free, see last lecture. However, if L is accepted by a DPDA then so is its complement — can just swap accepting and non-accepting states.)

- What about the context-sensitive level? Are NLBAs strictly more powerful than DLBAs? This is still an open question! (Can't use the same argument because it turns out that CSLs are closed under complementation — only shown in 1988.)

Detecting non-acceptance: LBAs versus TMs

Suppose T is an LBA. How might we detect that s is not in $\mathcal{L}(T)$?

Clearly, if there's an accepting computation for s, there's one that doesn't pass through exactly the same machine configuration twice (if it did, we could shorten it).

Since the tape is finite, the total number of machine configurations is finite (though large). So in theory, if T runs for long enough without reaching the final state, it will enter the same configuration twice, and we may as well abort.

Note that on this view, repeated configurations would be spotted not by T itself, but by 'us watching', or perhaps by some super-machine spying on T.

For Turing machines with unlimited tape space, this reasoning doesn't work. Is there some general way of spotting that a computation isn't going to terminate ?? See next lecture ...

Wider significance of Turing machines

Turing machines are important because (it's generally believed that) anything that can be done by any mechanical procedure or algorithm can in principle be done by a Turing machine (Church-Turing thesis). E.g.:

- Any language $L \subseteq \Sigma^*$ that can be 'recognized' by some mechanical procedure can be recognized by a TM.
- Any mathematical function $f : \mathbb{N} \to \mathbb{N}$ that can be computed by a mechanical procedure can be computed by a TM (e.g. representing integers in binary, and requiring the TM to write the result onto the tape.)

Status of Church-Turing Thesis

The CT Thesis is a somewhat informal statement insofar as the general notion of a mechanical procedure isn't formally defined (although we have a pretty good idea of what we mean by it).

Although a certain amount of philosophical hair-splitting is possible, the broad idea behind CTT is generally accepted.

At any rate, anything that can be done on any present-day computer (even disregarding time/memory limitations) can in principle be done on a TM.

So if we buy into CTT, theorems about what TMs can/can't do can be interpreted as fundamental statements about what can/can't be accomplished by mechanical computation in general.

We'll see some examples of such theorems next time.