Linear bounded automata and context-sensitive grammars, Intro to computability theory

NTIN071 Automata and Grammars

Jakub Bulín (KTIML MFF UK)

Spring 2025

** Adapted from the Czech-lecture slides by Marta Vomlelová with gratitude. The translation, some modifications, and all errors are mine.*

Recap of Lecture 10

- Turing machine: two-way infinite tape, read, write, move head
- Accept iff in a final state; configurations
- TMs with output, computing a function
- Recursively enumerable vs. recursive languages (always halt).
- Construction tricks:
  - storage in state
  - multiple tracks (on a single tape)
- Variants of TMs:
  - multi-tape (independent heads),
  - nondeterministic (accept iff some choices lead to final state)

3.3 Turing Machines and grammars

Chomsky hierarchy: Type 0

Turing machines	↔	Formal grammars
Linear bounded automata	↔	Context-sensitive grammars
Pushdown automata	↔	Context-free grammars
Finite automata	↔	Regular grammars

Theorem

A language is recursively enumerable, if and only if it is generated by a Type 0 grammar.

Turing machine to grammar

- First generate the relevant portion of the tape and a copy of the input word (nonterminal $\underline{X}$ for each $x \in \Gamma$, in reverse)
- Why? TM can rewrite w, G must generate it, cannot modify
- We have $wB^n\underline{W}^RQ_0B^m$, where B^n, B^m is sufficient free space
- Then simulate moves (essentially reverse configs+free space)
- In a final state erase the simulated tape, keep only w

$G = (\{S, C, D, E\} \cup \{\underline{X}\}_{x\in\Gamma} \cup \{Q_i\}_{q_i \in Q}, \Sigma, \mathcal{P}, S)$ where $\mathcal{P}$ is:

(1)	$S \to DQ_0E$	simulation starts in initial state
	$D \to xD\underline{X} \mid E$	generate input word, reverse copy for simulation
	$E \to BE \mid \epsilon$	generate sufficient free space for simulation
(2)	$\underline{X}P \to Q\underline{X'}$	for all $\delta(p, x) = (q, x', R)$ [direction reversed!]
	$\underline{X}P\underline{Y} \to \underline{X'}\,\underline{Y}Q$	for all $\delta(p, x) = (q, x', L)$
(3)	$P \to C$	for all $p \in F$
	$C\underline{X} \to C, \underline{X}C \to C$	clean the tape
	$C \to \epsilon$	finish, generated w

Example: $L = \{a^{2n} \mid n \geq 0\}$

$M = (\{q_0, q_1, q_2, q_F\}, \{a\}, \{a\}, \delta, q_0, B, \{q_F\})$ where

$$\delta(q_0, a) = (q_1, a, R),$$
$$\delta(q_1, a) = (q_0, a, R),$$
$$\delta(q_0, B) = (q_F, B, L)$$

$G = (\{S, C, D, E, Q_0, Q_1, Q_F, \underline{a}\}, \{a\}, S, \mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_3)$

Initialize: $\mathcal{P}_1$

$S \to DQ_0E$

$D \to aD\underline{a} \mid E$

$E \to BE \mid \epsilon$

Simulate: $\mathcal{P}_2$

$\underline{a}Q_0 \to Q_1\underline{a}$

$\underline{a}Q_1 \to Q_0\underline{a}$

$BQ_0\underline{a} \to B\underline{a}Q_F$

Cleanup: $\mathcal{P}_3$

$Q_F \to C$

$C\underline{a} \to C$

$\underline{a}C \to C$

$BC \to C$

$C \to \epsilon$

For $w = aa$: initialize $aaB\underline{aa}Q_0$, simulate $aaB\underline{a}Q_F\underline{a}$, cleanup: aa

Proof

$L(M) \subseteq L(G)$

- For $w \in L(M)$ there is a finite accepting sequence of moves
- The grammar generates sufficient space
- Then we simulate the moves
- Finally clean non-input symbols

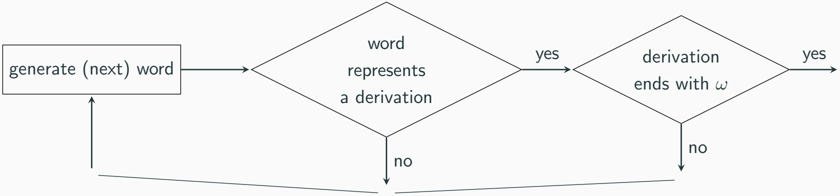
$L(G) \subseteq L(M)$

- Steps in a derivation for $w \in L(G)$ may be in different order
- But we can reorder them into the phases (1), (2), (3)
- Since we eliminated the underlined symbols, we must have generated the cleaning variable C
- In order to generate C we must have generated a final state
- A final state can only be generated from the initial state by a sequence of simulated moves $\square$

Grammar to Turing machine

Idea: The TM sequentially generates all possible derivations.
(Note: here we do not care about efficiency.)

- code $S \Rightarrow \beta_1 \Rightarrow \ldots \Rightarrow \beta_n = \omega$ as a string $\#S\#\beta_1\#\ldots\#\omega\#$
- construct a TM accepting exactly $\#\alpha\#\beta\#$ where $\alpha \Rightarrow \beta$
- construct a TM accepting $\#\beta_1\#\ldots\#\beta_k\#$ where $\beta_1 \Rightarrow^* \beta_k$
- construct a TM generating sequentially all possible strings
- check if the string is a valid derivation ending with ω



3.4 Linear bounded automata and context-sensitive grammars

Chomsky hierarchy: Type 1

Turing machines	Formal grammars
Linear bounded automata	Context-sensitive grammars
Pushdown automata	Context-free grammars
Finite automata	Regular grammars

Context-sensitive languages

Theorem

The following are equivalent for a language L:

(i) *L is generated by a context-sensitive grammar.*

(ii) *L is generated by a monotone grammar.*

(iii) *L is recognized by a Linear Bounded Automaton (LBA).*

- context-sensitive grammar: $\alpha_1 A \alpha_2 \to \alpha_1 \gamma \alpha_2$ where $A \in V$, $\gamma \in (V \cup T)^+$, $\alpha_1, \alpha_2 \in (V \cup T)^*$ ($S \to \epsilon$ if S not in bodies)
- monotone grammar: $\alpha \to \beta$ where $|\alpha| \leq |\beta|$
- Linear Bounded Automaton (LBA): a nondeterministic TM only using the input portion of the tape [we formalize later]

Note: Context-sensitive grammars are monotone, $(i) \Rightarrow (ii)$ trivial. Monotone grammars do not shorten sentential forms in a derivation

Example: $L = \{a^n b^n c^n \mid n \geq 1\}$ is context-sensitive

(Recall that L is not context-free.)

A monotone grammar:

$S \to aSBC \mid abC$	*right amount of* a, B, C
$CB \to BC$	*reorder to* $a^n b B^{n-1} C^n$
$bB \to bb$	$B \to b$ *only if preceded by* b
$bC \to bc$	$C \to c$ *only if preceded by* b
$cC \to cc$	*…or by* c

The rule $CB \to BC$ is not context-sensitive. But we can convert it to a chain of context-sensitive rules:

$$CB \to XB,\ XB \to XY,\ XY \to BY,\ BY \to BC$$

(Same for any monotone rule, as long as there are no terminals.)

From monotone to context-sensitive grammars $(ii) \Rightarrow (i)$

Recall: separated grammar means productions of the form $\alpha \to \beta$ where either $\alpha, \beta \in V^+$ or $\alpha \in V, \beta \in T \cup \{\epsilon\}$

Lemma

Every monotone grammar can be converted to an equivalent context-sensitive grammar.

Proof: First, convert to separated grammar (as for ChNF). This preserves monotonicity, $V_a \to a$ is monotone, context-sensitive.

Then, convert every production $A_1 \dots A_m \to B_1 \dots B_n$ ($m \leq n$) to the following chain (using new auxiliary variables C_i):

$$
\begin{aligned}
A_1 A_2 \dots A_m &\to C_1 A_2 \dots A_m & \qquad C_1 C_2 \dots C_m &\to B_1 C_2 \dots C_m \\
C_1 A_2 \dots A_m &\to C_1 C_2 \dots A_m & B_1 C_2 \dots C_m &\to B_1 B_2 \dots C_m \\
&\vdots & &\vdots \\
C_1 \dots C_{m-1} A_m &\to C_1 \dots C_{m-1} C_m & B_1 \dots B_{m-1} C_m &\to B_1 \dots B_{m-1} B_m \dots B_n
\end{aligned}
$$

Linear Bounded Automaton

Definition

A linear bounded automaton (LBA) is a *nondeterministic* Turing machine where the tape contains special symbols for left ($\underline{l}$) and right ($\underline{r}$) end. Those symbols cannot be rewritten and the head cannot move to the left of $\underline{l}$ or to the right of $\underline{r}$.

A word w is accepted if $q_0\underline{l}w\underline{r} \vdash^* \alpha p \beta$ for some $p \in F$

- The space for computation is given by the input word, we cannot exceed its length.
- Not a problem for context-sensitive/monotone grammars: sentential forms in a derivation cannot shorten.
- Nondeterminisim is crucial!

Construction trick: 'draw' several tape symbols into one cell (as in multi-track tape), increase space by constant factor; hence 'linear'

From context-sensitive grammar to LBA $(i) \Rightarrow (iii)$

Track 1: a copy of the input w, read-only

Track 2: simulate the derivation of w

l	w		r
	S		

- initialize with S in first field (the rest blank)
- at the end it should contain w, compare to Track 1

- to simulate one derivation step (apply rule $\alpha X \beta \to \alpha\gamma\beta$):

u	α	X	β	v

u	α	γ	β	v

- rewrite the sentential form using production rules
- nondeterministically choose which rule and where to apply it
- rewrite head to body (move the rest to the right)
- if only terminals, compare with Track 1, accept if match $\square$

From LBA to monotone grammar $(iii) \Rightarrow (ii)$

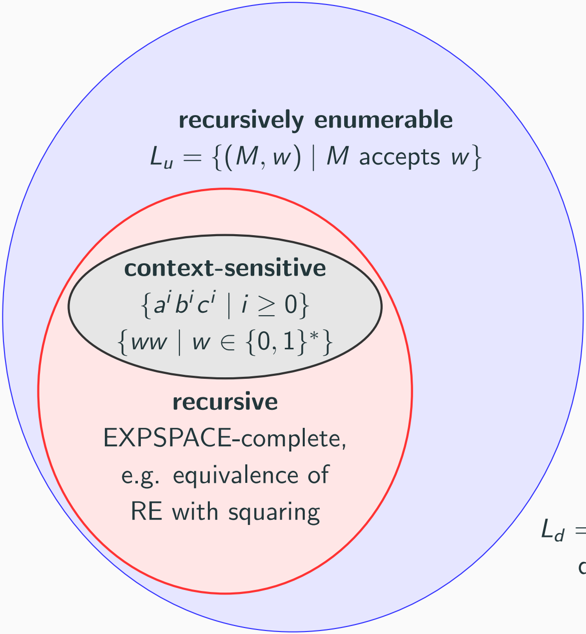
- the grammar cannot generate any 'extra' symbols
- we hide the computation in 'two-track' variables
- generate a word of the form

$$(a_0, [q_0, \underline{l}, a_0]), (a_1, a_1), \ldots, (a_n, [a_n, \underline{r}])$$

w		
$q_0, \underline{l}, a_0$		$a_n, \underline{r}$

- simulate computation in the 2nd track (as for TMs)
  - for $\delta(p, x) \ni (q, x', R)$: $P\underline{X}\,\underline{Y} \to \underline{X'}Q\underline{Y}$
  - for $\delta(p, x) \ni (q, x', L)$: $\underline{Y}P\underline{X} \to Q\underline{Y}\,\underline{X'}$
- if the state is accepting, 'erase' the 2nd track
- special production for generating ϵ (if $\epsilon \in L$) $\square$

Hierarchy of languages: context-sensitive and above



$L_d = \{w \mid \text{TM with code } w$
$\text{does not accept input } w\}$

Chapter 4: Intro to computability theory

First, a brief overview in 4 slides, without technical details

Languages and decision problems

A decision problem P: given input w (usually a 0-1 string), answer YES or NO (e.g. 'Is the given number prime?', 'Is the given picture classified as cat by the given neural net?')

$$L_P = \{w \mid P(w) \text{ answers 'YES'}\}$$

- P is (algorithmically) decidable $\Leftrightarrow$ L_P is recursive $\Leftrightarrow$ there is a TM deciding L_P (halting on every input, answering correctly)
- P is partially decidable $\Leftrightarrow$ L_P is recursively enumerable $\Leftrightarrow$ there is a TM that accepts every YES-instance w but for NO-instances it may either reject or run an infinite loop

NB: almost all problems are not even partially decidable (TMs can be represented by finite strings, so only countably many TMs)

Coming up next: a concrete example, the diagonal language

Source code for a Turing Machine & how to execute it

Source code for TMs:

- encode TMs by 01-strings, $M \rightsquigarrow \mathrm{code}(M) \in \{0,1\}^*$
- if w is not well-formed code, then say it represents a TM with no transitions, so every $w \in \{0,1\}^*$ will represent some TM
- also encode a pair of 01-strings u, v as a 01-string $\langle u, v \rangle$

The Universal language: $L_U = \{\langle \mathrm{code}(M), w \rangle \mid M \text{ accepts } w\}$

"Does a given program return true on a given input?"

Theorem

The Universal language is recursively enumerable.

Proof idea: construct the Universal Turing Machine that can simulate any TM (using its code) on any input [details later]

Barber's paradox aka the diagonal argument

The Diagonal language:

$$L_D = \{w \mid M \text{ such that } w = \mathrm{code}(M) \text{ does *not* accept } w\}$$

> *"Return true if the given program does not return true when fed its own source code."*

Theorem

The Diagonal language is not recursively enumerable.

Proof idea: there cannot exist a TM recognizing L_D: running it on its own code would lead to Barber's paradox

> *"The program accepts all programs that don't accept themselves. Does the program accept itself?"*

Languages that are recursively enumerable, but not recursive

Post's theorem

A language L is recursive, if and only if both L and $\overline{L}$ are recursively enumerable.

Proof idea: simulate TMs for L and $\overline{L}$ in parallel, one must halt

Corollary

The language $\overline{L_D}$ is not recursive, but it is recursively enumerable.

"Does the given program return true when fed its own code?"

Corollary

The Universal language is not recursive.

(If a TM decided L_U, we could use it to decide $\overline{L_D}$: $w \rightsquigarrow \langle w, w \rangle$)

We can execute a program, but cannot test if it runs into a loop.

Now, the technical details

Machine-readable encoding of TMs (Gödel numbering)

To encode a TM as a binary string, we first assign integers to the states, tape symbols, and directions L, R. Assume:

- the start state is always q_1, the only final state is q_2
- the first tape symbol is always 0, the second 1, the third B (other tape symbols can be assigned arbitrarily)
- the direction L is 1, the direction R is 2

Each transition $\delta(q_i, X_j) = (q_k, X_l, D_m)$ is encoded by $0^i 1 0^j 1 0^k 1 0^l 1 0^m$. Since $i, j, k, l, m \geq 1$, substring 11 doesn't occur.

The entire encoding $\mathrm{code}(M)$ consists of codes for all transitions (in any order), separated by a pair of 1's: $C_1 11 C_2 11 \ldots C_{n-1} 11 C_n$.

Similarly, we encode a tuple of 01-strings as a 01-string: separate entries by 111. We also fix an order of 01-strings, by length + lexicographically ($w_0 = \epsilon$, $w_1 = 0$, $w_2 = 1$, $w_3 = 00$, $w_4 = 01$, $\ldots$)

Example

$$M = (\{q_1, q_2, q_3\}, \{0, 1\}, \{0, 1, B\}, \delta, q_1, B, \{q_2\})$$

δ	0	1	B
$\rightarrow q_1$		$(q_3, 0, R)$	
$*q_2$			
q_3	$(q_1, 1, R)$	$(q_2, 0, R)$	$(q_3, 1, L)$.

Codes for transitions:

C_1	C_2	C_3	C_4
0100100010100	0001010100100	00010010010100	0001000100010010

The full encoding $\mathrm{code}(M)$:

0100100010100110001010100100110001001001010011000100010001001O

Summary of Lecture 11

- Recursively enumerable languages are exactly those generated by (Type 0) grammars
  - TM to G: simulate moves on a reversed non-terminal copy of ω, generate sufficient space, cleanup if accepting state
  - G to TM: generate all strings, check if any of them represents a valid derivation of ω (sentential forms separated by $\#$)
- Context-sensitive languages:
  - context-sensitive grammars are equivalent to monotone grammars
  - Linear Bounded Automaton (LBA): nondeterministic TM with tape limited to the length of input
  - constructions: monotone grammar to LBA, LBA to monotone grammar
- Intro to computability: an overview
- decision problem $\leftrightsquigarrow$ the language of all 'YES' instances
- machine-readable encoding of TMs