Introduction to Theoretical Computer Science

Lecture 5: Starting on Computability

Richard Mayr

University of Edinburgh

Semester 1, 2025/2026

More Pigeonholes

Suppose a CFG has n non-terminals, and we have a parse tree of height $k > n$. **What must have happened?**.
The same non-terminal V must have appeared as its own descendant in the tree.

Pumping for CFLs

Pumping down Cut the tree at the higher occurrence of V and replace it with the subtree at the lower occurrence of V.

Pumping up Cut at the lower occurrence and replace it with a fresh copy of the higher occurrence.

Pumping Lemma for CFLs

Theorem

If L is context-free then there exists a $p \in \mathbb{N}$ (the pumping length) such that if $w \in L$ with $|w| \geq p$ then w may be split into **five** pieces $w = uvxyz$ such that:

1. $uv^ixy^iz \in L$ for all $i \in \mathbb{N}$.
2. $|vy| > 0$ and
3. $|vxy| \leq p$

It can be useful to think of it like a game:

1. **You** pick a language L
2. **Adversary** picks a pumping length p
3. **You** pick a word $w \in L$ with $|w| \geq p$.
4. **Adversary** splits it into $uvxyz$ s.t. $|vxy| \leq p$ and $vy \neq \varepsilon$.
5. **You** win if you can find $i \in \mathbb{N}$ such that $uv^ixy^iz \notin L$. Your prize is a proof of L not being context-free.

Examples

Example

Let $L = \{\mathtt{a}^i\mathtt{b}^i\mathtt{c}^i \mid i > 0\}$. If L is a CFL it must have a pumping length p. Consider the word $w = \mathtt{a}^p\mathtt{b}^p\mathtt{c}^p$. Then, we cannot avoid contradiction no matter how we split $w = uvxyz$:

If vxy is in $\mathtt{a}^*\mathtt{b}^*$ then uxz (i.e. uv^0xy^0z) is not in L because **condition 2** says vy contains at least one symbol. So uxz has fewer than p copies of $\mathtt{a}$ or $\mathtt{b}$ but still p copies of $\mathtt{c}$. Similarly if vxy is in $\mathtt{b}^*\mathtt{c}^*$.

There are no other cases due to **condition 3**.

Another example

Consider $L = \{ww \mid w \in \{0,1\}^*\}$. If it is context free it must have a pumping length $p > 0$.

A rule of thumb

Pick a string w that allows as few cases for partitions of $w = uvxyz$ as possible, to reduce the number of case distinctions.

Consider the word $0^p 1^p 0^p 1^p$. Let $uvxyz = w$ such that $|vxy| \leq p$ and $vy \neq \varepsilon$. vxy can range over at most two of the four regions:

- If vxy is in a single one of the regions i.e. $vxy \in 0^* \cup 1^*$ then pumping either way takes us out of L.
- Otherwise, if vxy spans some part of the first two or last two regions, i.e. a substring of $0^p 1^p$, pumping down will take us out of L.
- If vxy straddles the midpoint of w, pumping down will remove 1s from the first half but 0s from the second half, taking us out of L.

Chomsky Grammars

CFGs are a special case of *Chomsky Grammars*. Chomsky Grammars are much like CFGs except that the left-hand side of a production may be any string that includes at least one non-terminal:

Example

$$
\begin{array}{lcl}
S & \to & \mathtt{abc} \mid \mathtt{a}A\mathtt{bc} \\
A\mathtt{b} & \to & \mathtt{b}A \\
A\mathtt{c} & \to & \mathtt{B}bcc \\
\mathtt{b}B & \to & \mathtt{B}b \\
\mathtt{a}B & \to & \mathtt{aa}A \mid \mathtt{aa}
\end{array}
$$

This grammar is called **context-sensitive**

The Chomsky Hierarchy

Definition

A grammar $G = (N, \Sigma, P, S)$ is of *type*:

0 (or *computably enumerable*) in the general case.

1 (or *context-sensitive*) if $|\alpha| \leq |\beta|$ for all productions $\alpha \rightarrow \beta$, except we also allow $S \rightarrow \varepsilon$ if S does not occur on the RHS of any rule.

2 (or *context-free*) if all productions are of the form $A \rightarrow \alpha$ (i.e. a CFG).

3 (or *right-linear*) if all productions are of the form $A \rightarrow w$ or $A \rightarrow wB$ where $w \in \Sigma$ and $B \in N$.

- Recursively enumerable is also called *Turing-recognisable*.
- Right-linear is also called...*regular*!

Emptiness

Can we write a computer program to determine if a given regular language is empty?

Emptiness for regular languages

Given a **finite automaton**, this is an instance of *graph reachability* — can we reach a final state? Can be done via depth-first search.
Given a **regular expression**, we can work inductively (see board).

Emptiness Continued

Can we write a computer program to determine if a given context-free language is empty?

Emptiness of CFLs

Given a CFG for our language:

1. Mark the terminals and ε as generating.
2. Mark as generating all non-terminals which have a production with only generating symbols in their RHS.
3. Repeat until nothing new is marked generating.
4. Check whether S is marked as generating.

Equivalence

Can we write a computer program to determine if two given DFAs are equivalent?

Equivalence of Regular Languages

Given two DFAs for L_1 and L_2 we can use our standard constructions to produce a DFA of the symmetric set difference:

$$(L_1 \cap \overline{L_2}) \cup (L_2 \cap \overline{L_1})$$

(Constructions for complement and intersection are in coursework 1, not lectures.)

If this DFA is empty, then the two languages are equal.

Equivalence Continued

Later we'll develop a theory that allows us to prove rigorously that there are problems that cannot be solved by any algorithm that can be implemented as a conventional computer program.

Such problems are called *undecidable*.

Many undecidable problems exist for CFLs:

- Are two CFGs equivalent?
- Is a given CFG ambiguous?
- Is there a way to make a CFG unambiguous?
- Is the intersection of two CFLs empty?
- Does a CFG generate all strings Σ^* (also called *universality*)

Register Machines

Key Insight

There is a general model of computation

You may have heard of the *Turing Machine*, but we will first focus on something closer to our understanding of programs.

Definition

A *register machine*, or RM, consists of:

- A **fixed** number m of *registers* $R_0 \ldots R_{m-1}$, which each hold a natural number.
- A **fixed** *program* P which is a sequence of n *instructions* $I_0 \ldots I_{n-1}$

Each instruction is either: INC(i), which increments register R_i, or DECJZ(i, j) which decrements R_i unless $R_i = 0$ in which case it jumps to I_j.

Questions of RMs

What can we compute with RMs? What is unrealistic about them?

Claim

RMs can compute anything any other computer can.

RM ASM

Problem

Programming in RMs directly is very tedious and programs can be overlong.

We will use some simple notation similar to assembly language to simplify it.

Macros

- We'll write them in English, e.g. "add R_i to R_j clearing R_i".
- When defining a macro, we'll number instructions from zero, but the instructions are renumbered when macros are expanded. We also use symbolic labels for jumps.
- Macros can use special, negative-indexed registers, guaranteed not to be used by normal programs.

Goto I_j using R_{-1} as temp

0	DECJZ	$(-1, j)$

Clear R_i

0	DECJZ	$(i, 2)$	
1	GOTO	0	(using macro above)

Copy R_i to R_j using R_{-2} as temp

	0	CLEAR	R_j
$loop_1$:	2	DECJZ	$(i, loop_2)$
	3	INC	(j)
	4	INC	(-2)
	5	GOTO	$loop_1$
$loop_2$:	6	DECJZ	$(-2, \text{end})$
	7	INC	(i)
	8	GOTO	$loop_2$
end	9		

RM Programming Exercises

- Addition and subtraction of registers
- Comparison of registers
- Multiplication of registers
- Division/Remainder of registers

How many registers?

So far, we've just assumed we had as many registers as we needed. But how many do we actually need?

Pairing functions

A *pairing function* is an injective function $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$.
An example is $f(x, y) = 2^x 3^y$.
We write $\langle x, y \rangle_2$ for $f(x, y)$. If $z = \langle x, y \rangle_2$, let $z_0 = x$ and $z_1 = y$.

Exercise: Program a pairing function and unpairing functions on a RM.
Exercise: Design (or look up) a surjective pairing function.

Generalising

Just a 2-tuple pairing function is enough to cram an arbitrary sequence of natural numbers into one $\mathbb{N}^* \to \mathbb{N}$.

Conclusion

With pairing functions, we can simulate any number of registers using just the registers we need to compute the pairing and unpairing functions, and one user register.

Question

So, how many registers do we actually need?