Introduction to Theoretical Computer Science (ITCS)

Lecture 1: Finite Automata

Richard Mayr

University of Edinburgh

Semester 1, 2025/2026

Course Aims

- Understanding of computability, computational complexity and intractability;
- Lambda calculus, types, and type safety.

Course Outcomes

By the end of the course you should be able to

- Explain (non-)deterministic finite and pushdown automata and use the pumping lemma to show languages non-regular.
- Explain decidability, undecidability and the halting problem.
- Demonstrate the use of reductions for undecidability proofs.
- Explain the notions of P, NP, NP-complete.
- Use reductions to show problems to be NP-hard.
- Write short programs in lambda-calculus.

Course Outline

- Introduction. Finite automata.
- Regular languages and expressions.
- Context-free languages and pushdown automata.
- Register machines and their programming.
- Universal machines and the halting problem.
- Decision problems and reductions.
- Undecidability and semi-decidability.
- Complexity of algorithms and problems.
- The class P.
- Non-determinism and NP.
- NP-completeness.
- Beyond NP.
- Lambda-calculus.
- Recursion.
- Types.

Assessment

The course is assessed by a written examination (80%) and two coursework exercises, the first formative and the second summative (for the remaining 20%).
Coursework deadlines: End of weeks 6 and 10.

No tutorials, but weekly exercise sheets (starting in week 2).
Solutions one week later. Some exercises discussed in class.

Textbooks

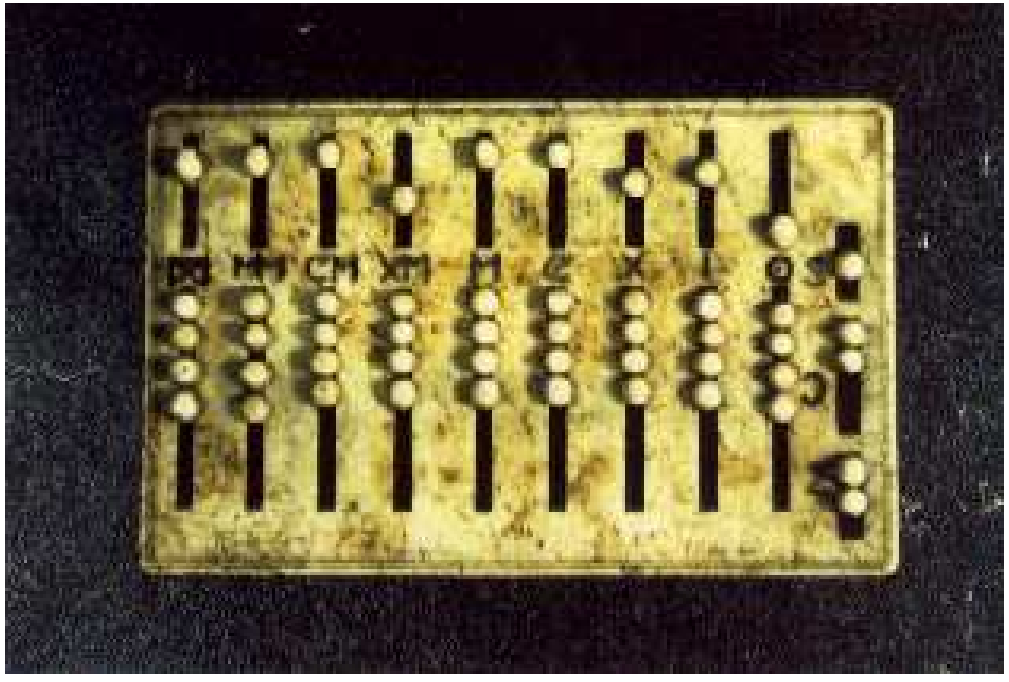
It will be useful, but not absolutely necessary, to have access to:

- Michael Sipser *Introduction to the Theory of Computation*, PWS Publishing (International Thomson Publishing)
- Benjamin C. Pierce *Types and Programming Languages*, MIT Press

There is also much information on the Web, and in particular Wikipedia articles are generally fairly good in this area.
Generally I will refer to textbooks for the detail of material I discuss on slides.

What is computation? What are computers?

Some computing devices:
The abacus – some millennia BP.

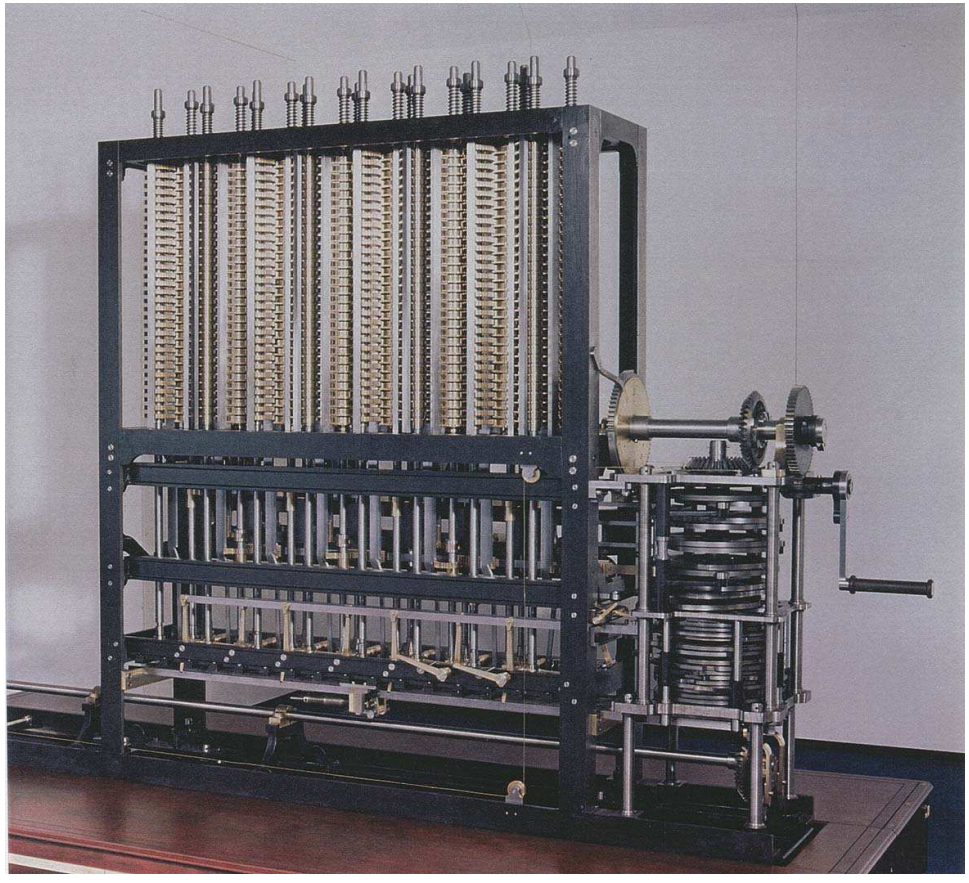


[Association pour le musée international du calcul de l'informatique et de l'automatique de Valbonne Sophia Antipolis (AMISA)]

First mechanical digital calculator – 1642 Pascal

[original source unknown]

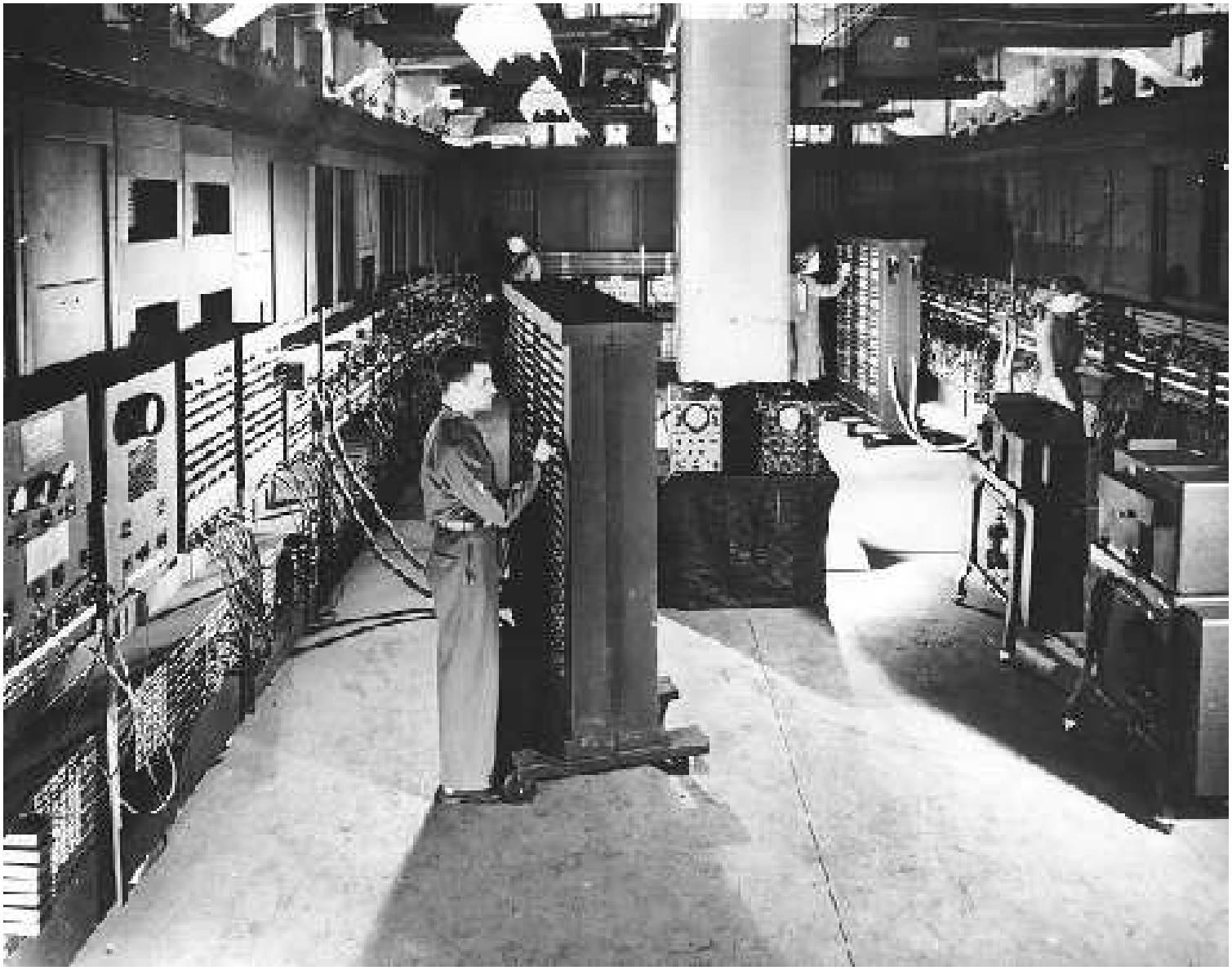
The Difference Engine, [The Analytical Engine] – 1812, 1832 Babbage / Lovelace.



[Science Museum ??]

Analytical Engine (never built) anticipated many modern aspects of computers. See `http://www.fourmilab.ch/babbage/`.

ENIAC – 1945, Eckert & Mauchley



[University of Pennsylvania]

What do computers manipulate?

Symbols? Numbers? Bits? Does it matter?
What about real numbers? Physical quantities? Proofs? Emotions?
Do we buy that numbers are enough? If we buy that, are bits enough?
How much memory do we need?

What can we compute?

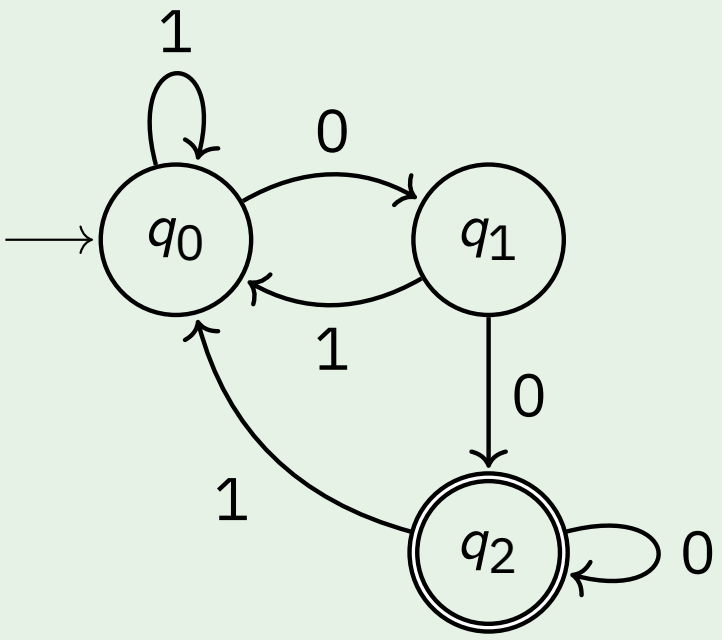
If we can cast a problem in terms that our computers manipulate, can we solve it? Always? Sometimes? With how much time? With how much memory?

In this course

We will seek mathematical answers to these questions. For that, we will need a model of computation.

Finite Automata

Example



A finite automaton takes a string as input and says "yes" or "no".

Define the *language* of a finite automaton A, written, $\mathcal{L}(A)$ to be the set of strings for which A says "yes".

A string is a (possibly-empty) sequence of *symbols* from a set called an *alphabet*, usually written Σ.

DFAs, formally

Definition

A *deterministic finite automaton* (DFA) is a quintuple $(Q, \Sigma, q_0, \delta, F)$ where:

- Q is a **finite** set of *states*,
- Σ is the *alphabet*, the set of *symbols*,
- $q_0 \in Q$ is the *initial state*
- $\delta : Q \times \Sigma \to Q$ is the *transition function*,
- $F \subseteq Q$ is the set of *final states*.

Exercise: What is the formal definition of our example?

Languages

Definition

A DFA *accepts* a string $w \in \Sigma^*$ iff $\delta^*(q_0, w) \in F$, where δ^* is δ applied successively for each symbol in w. The language of a DFA $\mathcal{L}(A) \subseteq \Sigma^*$ is the set of all strings accepted by A.

Exercise: What is the language of our example?

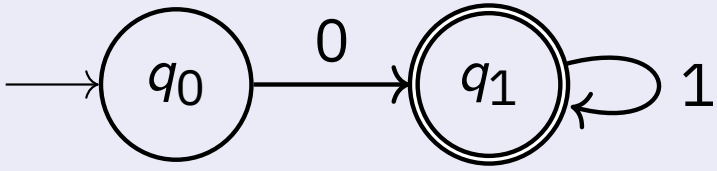
Determinism

In a DFA, the transition function is a total function which gives exactly one next state for each input symbol (it's *deterministic*).

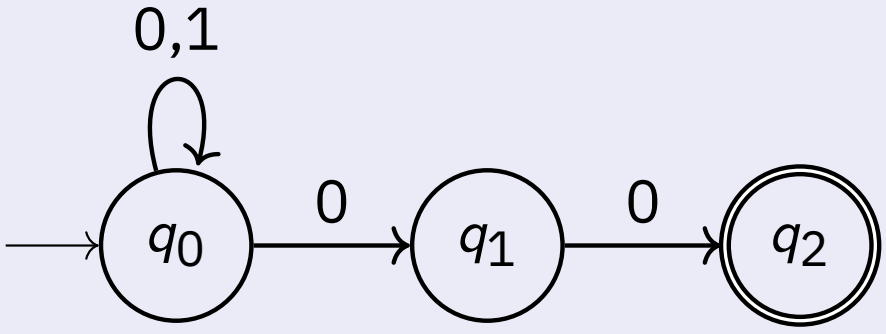
Questions

Does relaxing any of these requirements affect the set of languages we can recognise? How would we prove this?

- What if we made δ *partial*?



- What if we made δ *non-deterministic*?



Nondeterministic Finite Automata

Definition

A *nondeterministic finite automaton* (NFA) is a quintuple $(Q, \Sigma, q_0, \delta, F)$ where:

- Q is a **finite** set of *states*,
- Σ is the *alphabet*, the set of *symbols*,
- $q_0 \in Q$ is the *initial state*
- $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ is the *transition function*,
- $F \subseteq Q$ is the set of *final states*.

Note the only difference here is the transition function, which gives a set of next states for a given symbol.

Nondeterministic Finite Automata

Definition

A *run* of an NFA A on a string $w = a_1 a_2 \ldots a_k$ is a sequence of states $q_0 q_1 \ldots q_k$ in Q such that:

- q_0 is the initial state
- for all $i = 1 \ldots k$ we have $q_i \in \delta(q_{i-1}, a_i)$.

A run is *accepting* if the last state $q_k \in F$.

The nondeterminism means that we have multiple alternative computations. For our purposes we will use *angelic* non-determinism, which says that we achieve success if any of our alternatives succeed.

$$\mathcal{L}(A) = \{w \mid \text{there exists an accepting run of } A \text{ on } w\}$$

Exercise: Is `10100` in the language of our previous example?

NFA = DFA

Claim

Making finite automata non-deterministic does not change their expressivity. That is, for every non-deterministic automaton A there is a deterministic automaton D such that $\mathcal{L}(D) = \mathcal{L}(A)$ and vice versa.

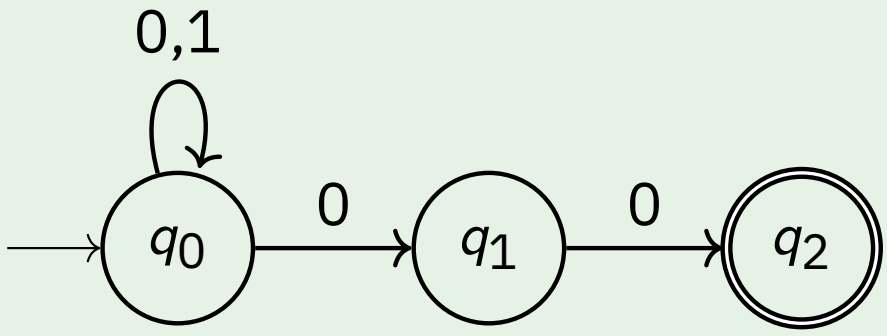
- **DFA** $\Rightarrow$ **NFA**:Easy, the DFA is already an NFA where the transition function always returns a singleton set.
- **NFA** $\Rightarrow$ **DFA**:We will use the *subset construction*.

Subset Construction

Key Idea

For an NFA A, the corresponding DFA D tracks the set of states that A could possibly be in, given the string read so far. So, each state of D is a set of states from A.

Example (From earlier)



Formally

The Subset Construction

Given an NFA $(Q_A, \Sigma, q_0, \delta_A, F_A)$, construct a DFA $(\mathcal{P}(Q_A), \Sigma, \{q_0\}, \delta_D, F_D)$ where:

$$\delta_D(S, a) = \bigcup_{q \in S} \delta_A(q, a) \quad \text{for each } S \subseteq Q$$

and

$$F_D = \{S \subseteq Q \mid S \cap F_A \neq \varnothing\}$$

Question: If our NFA has n states, how many states could our DFA have?

Proof?

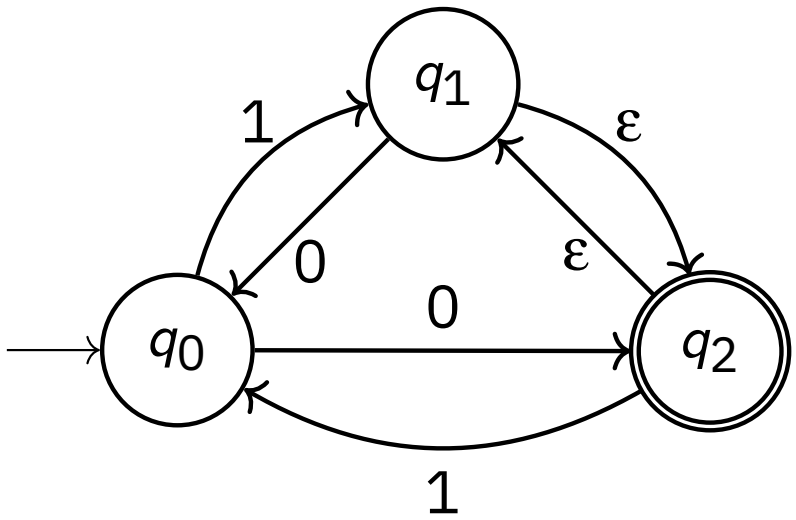
Proving that this is correct (i.e. that the DFA D obtained from an NFA A recognises the same language as A) relies on a proof by induction on the length of the input string w, that $\delta_D^*(\{q_0\}, w)$ is the set of all states q such that there exists a run of A on w from q_0 to q.
We will cover this if we have extra time.

-NFAs

Another Generalisation

What if we allow non-deterministic state changes that do not consume any input symbols?

We label these *silent moves* with ε (the empty string):



Exercise: Is `001` accepted above? Can we express this as a DFA?

-NFA to DFA

The subset construction also applies to ε-NFAs.

Definition

Define the *ε-closure* $E(q)$ of a state q as the set of all states reachable from q by silent moves.That is, $E(q)$ is the least set satisfying:

- $q \in E(q)$
- For any $s \in E(q)$, we also have $\delta(s, \varepsilon) \subseteq E(q)$.

We also extend this to sets, where $E(S) = \bigcup_{q \in S} E(q)$.

In our subset construction, everything is the same except that each subset (each state of our DFA) is ε-closed:

$$\delta_D(S, a) = E\left(\bigcup_{s \in S} \delta_A(s, a)\right)$$

Summary

DFAs, NFAs and ε-NFAs all recognise the same class of languages, called the *regular languages*. They are equal in expressive power, although some representations (NFAs) are more compact than others (DFAs).

Questions for next time[a]

[a] Or this time, if we have time.

- Are the regular languages closed under union? sequential composition? intersection? complement?
  (How would we prove this?)
- What languages are not regular?
  (How would we prove this?)

Introduction to Theoretical Computer Science

Lecture 2: Regular Languages

Richard Mayr

University of Edinburgh

Semester 1, 2025/2026

Recall..

DFAs, NFAs and ε-NFAs all recognise the same class of languages, called the *regular languages*. They are equal in expressive power, although some representations (NFAs) are more compact than others (DFAs).

Closure Properties

Definition

The *union* of two languages L_1 and L_2, written $L_1 \cup L_2$, is the language that includes all strings of L_1 and all strings of L_2.

Are the regular languages *closed* under union?

That is, if we have two regular languages L_1 and L_2, is $L_1 \cup L_2$ also regular?

Exercise: Prove this.

Closure Properties

Definition

The *sequential composition* of two languages L_1 and L_2, written $L_1 L_2$, is the language of strings that consist of a string in L_1 followed by a string in L_2.

$$L_1 L_2 = \{vw \mid v \in L_1, w \in L_2\}$$

Are the regular languages *closed* under sequential composition?

That is, if we have two regular languages L_1 and L_2, is $L_1 L_2$ also regular?

Exercise: Prove this.

Closure Properties

Notation

Similarly to arithmetic, define L^0 as $\{\varepsilon\}$ and $L^{n+1} = LL^n$.

Definition

The *Kleene closure* of a language L, written L^*, is the language of strings that consist wholly of zero or more strings in L.

$$L^* = \bigcup_{i \in \mathbb{N}} L^i$$

(n.b: in computer science, $0 \in \mathbb{N}$)

Are the regular languages *closed* under Kleene closure?

Exercise: Prove this.

Regular Expressions

Regular expressions are an algebraic notation for regular languages. Many of you will have already used (some variant of) regular expressions in your text editors.

Syntax	**Semantics**			
a	$[\![a]\!]$	$=$	$\{a\}$	$(a \in \Sigma)$
$\varnothing$	$[\![\varnothing]\!]$	$=$	$\varnothing$	
ε	$[\![\varepsilon]\!]$	$=$	$\{\varepsilon\}$	
$R_1 \cup R_2$	$[\![R_1 \cup R_2]\!]$	$=$	$[\![R_1]\!] \cup [\![R_2]\!]$	
$R_1 \circ R_2$	$[\![R_1 \circ R_2]\!]$	$=$	$[\![R_1]\!][\![R_2]\!]$	
$R*$	$[\![R*]\!]$	$=$	$[\![R]\!]^*$	

Regular Expressions

The notation used for regexes here may differ from the "regular" expressions you may have seen in text editors. Please note that sometimes these editors contain extensions that recognise non-regular languages, so intuitions from text editors may not apply here.

Questions

- How do we write "at least one `0`"? What about "at least one `0` and at least one `1`?"
- How do we write $R+ = R^1 \cup R^2 \cup R_3 \cup \cdots$ using existing operators?
- How do we write $R?$, the *optional* R, using existing operators?

Regular Expressions vs Finite Automata

Regular expressions *exactly characterise* the regular languages, just as finite automata do. This means that every regular language can be represented as a regular expression.

How do we prove this?

- **RE** $\rightarrow$ **DFA** – apply the constructions used in our closure proofs, then the subset construction.
- **DFA** $\rightarrow$ **RE** – convert to a *generalised NFA*, then reduce to a single transition.

A note

The DFAs we get from our **RE** $\rightarrow$ **DFA** translation are not very space-efficient. Most implementations use more advanced techniques to minimise the DFA.

Generalised NFAs

Definition

A *generalised NFA*, or GNFA, is an NFA where:

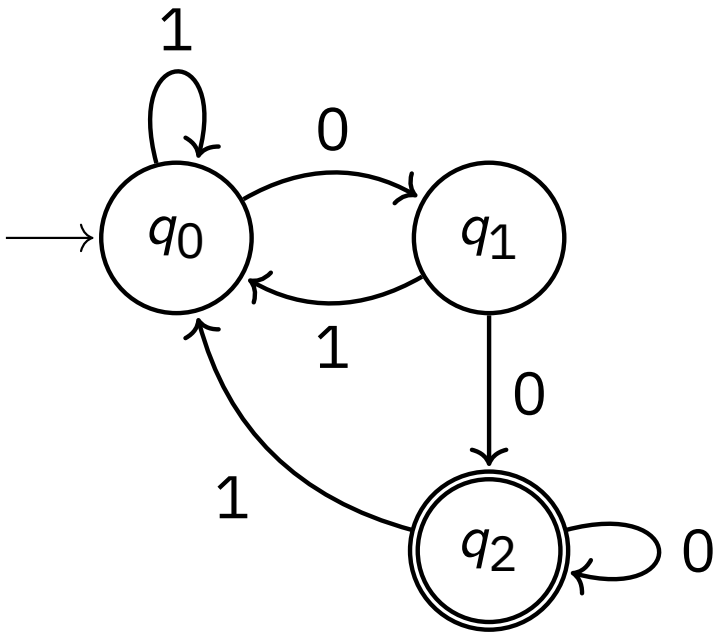
- Transitions have regular expressions on them instead of symbols.
- There is only one unique final state.
- The transition relation is *full*, except that the initial state has no incoming transitions, and the final state has no outgoing transitions.

(n.b: transitions can be labelled with $\varnothing$)

What do we need to do to convert a DFA to a GNFA?

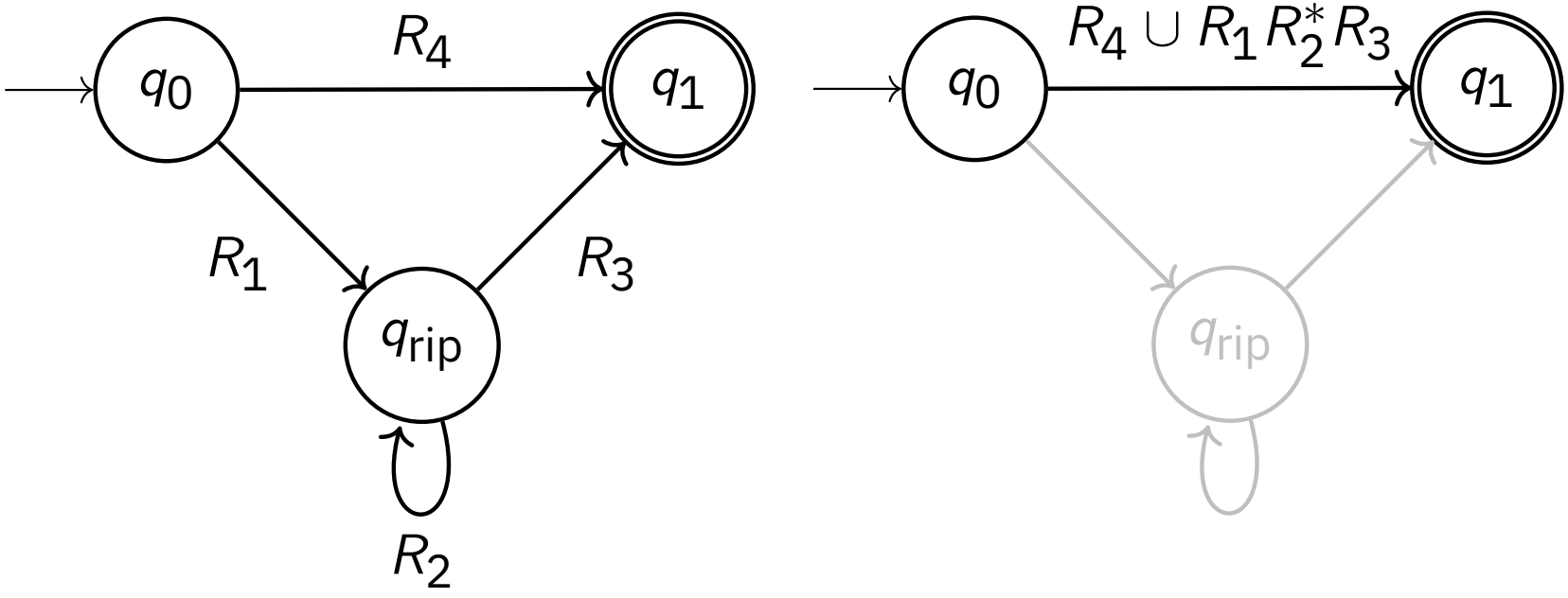
DFA to GNFA

1. Add a new start state, connect via ε moves to the old one.
2. Add a new final state, connect via ε moves from the old final state(s).
3. If two states q_0 and q_1 have two transitions between them $q_0 \xrightarrow{a} q_1$ and $q_0 \xrightarrow{b} q_1$, replace them with $q_0 \xrightarrow{a \cup b} q_1$.
4. Introduce $\varnothing$-labelled transitions where needed to make the transition relation full.



GNFA to RE

We will eliminate each of the inner states of the GNFA one by one. When all of them are gone, only the initial and final state will remain, with one transition between them. The label on this transition will be our regular expression.



Exercise: Let's reduce our example to a single RE.

Introduction to Theoretical Computer Science

Lecture 3: Beyond the Regular Languages

Richard Mayr

University of Edinburgh

Semester 1, 2025/2026

Non-regular languages

What are some examples of *non-regular* languages?

Canonical examples: Matching parentheses, $L_1 = \{\mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N}\}$, $L_2 = \{\mathtt{c}^i\mathtt{a}^j\mathtt{b}^k \mid i = 1 \Rightarrow j = k + 1\}$.

Intuition

Recognising L_1 requires counting the number of $\mathtt{a}$s in the string, which is an unbounded natural number, which requires unbounded memory (not a finite amount of states).

How would we prove this?

Pumping

Suppose a DFA with k states accepts a word of length greater than k.
What must have happened?
$\Rightarrow$ The DFA must have visited a state more than once
$\Rightarrow$ There is a loop.

Therefore, if we go through that loop any number of times, the DFA should accept those words also. We call this *pumping*.

The Pumping Lemma

Theorem (Pumping Lemma)

If $L \subseteq \Sigma^*$ is regular then there exists a *pumping length* $p \in \mathbb{N}$ such that for any $w \in L$ where $|w| \geq p$, we may split w into three pieces $w = xyz$ satisfying three conditions:

1. $xy^i z$ for all $i \in \mathbb{N}$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

The proof of this relies on the pigeonhole principle.

We can prove a language is non-regular by taking the contrapositive of this.

$$\text{can't be pumped} \quad \Rightarrow \quad \text{not regular}$$

.

Using the Pumping Lemma

To prove a negation (e.g. non-regularity), a common technique is to assume to the contrary that the proposition holds and show that it would lead to a contradiction.

Example (For L_1)

Consider $L_1 = \{\mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N}\}$. Assume to the contrary that L_1 *is* regular and that p is its pumping length. We know $\mathtt{a}^p\mathtt{b}^p$ is $\in L_1$. No matter how we split this word into xyz, none of these splits satisfies the three conditions of the Pumping Lemma.

Case y consists only of as: Then $xyyz$ contains more as than bs, violating condition 1.

Case y contains bs: Then $|xy| > p$ violating condition 3.

Case y is empty (ε): Then $|y| = 0$ violating condition 2.

Another Non-Regular Language

Recall the language $L_2 = \{c^i a^j b^k \mid i = 1 \Rightarrow j = k + 1\}$.

Definition

Define the *left quotient* of a language L, written $w \backslash L$ to be the set of suffixes that can be added to w to produce a word in L:

$$w \backslash L = \{v \mid wv \in L\}$$

Exercise: Prove that $w \backslash L$ is regular when L is regular.

Observe that $ca \backslash L_2 = \{a^n b^n \mid n \in \mathbb{N}\} = L_1$, which is not regular.
Therefore L_2 is also **not regular**.

Limitations of the Pumping Lemma

We have seen that $L_2 = \{c^i a^j b^k \mid i = 1 \Rightarrow j = k + 1\}$ is not regular, but it is possible to pump this.

Assume that L_2 is regular and that p is its pumping length, and that $w \in L_2$ where $|w| \geq p$. We choose x, y (and implicitly z) based on the number of c's in w, written C:

Case $C = 0$: Choose $x = \varepsilon$ and $y =$ first letter of w

Case $0 < C \leq 3$: Choose $x = \varepsilon$ and $y = c^C$

Case $C > 3$: Choose $x = \varepsilon$ and $y = cc$

In each case, we can pump (i.e. repeat y arbitrarily many times and stay in L_2.)

So, the converse of the pumping lemma does not hold:

$$\text{can't be pumped} \quad \not\Leftarrow \quad \text{not regular}$$

.

Beyond the Pumping Lemma

The pumping lemma is useful, but not satisfying, because it is not an exact characterisation.

Definition

Let $L \subseteq \Sigma^*$ and $x, y \in \Sigma^*$. If there exists a suffix string z such that $xz \in L$ but $yz \notin L$ (or vice-versa), then x and y are *distinguishable* by L. If x and y are not distinguishable by L, we say $x \equiv_L y$. This is an *equivalence relation*.

The Myhill-Nerode Theorem

A language L is regular iff the number of $\equiv_L$ equivalence classes is finite.

Proof Sketch if time allows.

Using Myhill-Nerode

To use Myhill-Nerode to show that L is non-regular, we must show that there are infinite $\equiv_L$ equivalence classes.

In detail

More specifically, we find an infinite sequence $u_0 u_1 u_2 \ldots$ of strings such that for any i and j (where $i \neq j$), there is a string w_{ij} such that $u_i w_{ij} \in L$ but $u_j w_{ij} \notin L$ (or vice-versa).

Example

- $L_1 = \{\mathtt{a}^n \mathtt{b}^n \mid n \in \mathbb{N}\}$, choose $u_i = \mathtt{a}^i$ and $w_{ij} = \mathtt{b}^i$.
- $L_2 = \{\mathtt{c}^i \mathtt{a}^j \mathtt{b}^k \mid i = 1 \Rightarrow j = k + 1\}$, choose $u_i = \mathtt{c}\mathtt{a}^{i+1}$ and $w_{ij} = b^i$.

Context-Free Languages

What would happen if we added recursion to regexps?

Definition

A *Context-free grammar* (CFG) is a 4-tuple (N, Σ, P, S) where:

- N is a finite set of *variables* or *non-terminals*,
- Σ is a finite set of *terminals*
- $P \subseteq N \times (N \cup \Sigma)^*$ is a finite set of *rules* or *productions*. Typically productions are written like:

  $$A \to \mathtt{a}B\mathtt{c}$$

  Productions with common heads can be combined:

  $$A \to \mathtt{a} \mid A\mathtt{a} \mid \mathtt{b}A\mathtt{b}$$

- $S \in N$ is the *start variable*.

Context-Free Grammars

Notation: We use α, β, γ etc. to refer to *sequences of terminals*.

Definition (Derivations)

We make a *derivation step* $\alpha A \beta \Rightarrow_G \alpha\gamma\beta$ whenever $(A \to \gamma) \in P$.
The language of a CFG G is:

$$\mathcal{L}(G) = \{w \in \Sigma^* \mid S \Rightarrow_G^* w\}$$

Where $\Rightarrow_G^*$ is the *reflexive transitive closure* of $\Rightarrow_G$.

Example

Given the CFG G:

$$G = (\{S\}, \{0, 1, \{S \to \varepsilon \mid 0S1\}, S)$$

What is the language of G?

Introduction to Theoretical Computer Science

Lecture 4: Context-Free Languages

Richard Mayr

University of Edinburgh

Semester 1, 2025/2026

A language is *context-free* (a CFL) iff it is recognised by a context-free grammar.
Exercise: Are all regular languages context-free?

Uses of CFLs

Many programming languages are syntactically context-free. Even the syntax we defined last lecture for regular expressions is context free. Suppose $\Sigma = \{\mathtt{a}, \mathtt{b}\}$.

$$S \to \emptyset \mid \varepsilon \mid \mathtt{a} \mid \mathtt{b} \mid S \cup S \mid S \circ S \mid S^* \mid (S)$$

Exercise: Derive with this grammar that $(\mathtt{a} \cup \mathtt{b} \circ \mathtt{a})^*$ is a regular expression.
Always replace the leftmost remaining non-terminal at each step, giving a *leftmost derivation*.

Parse Trees

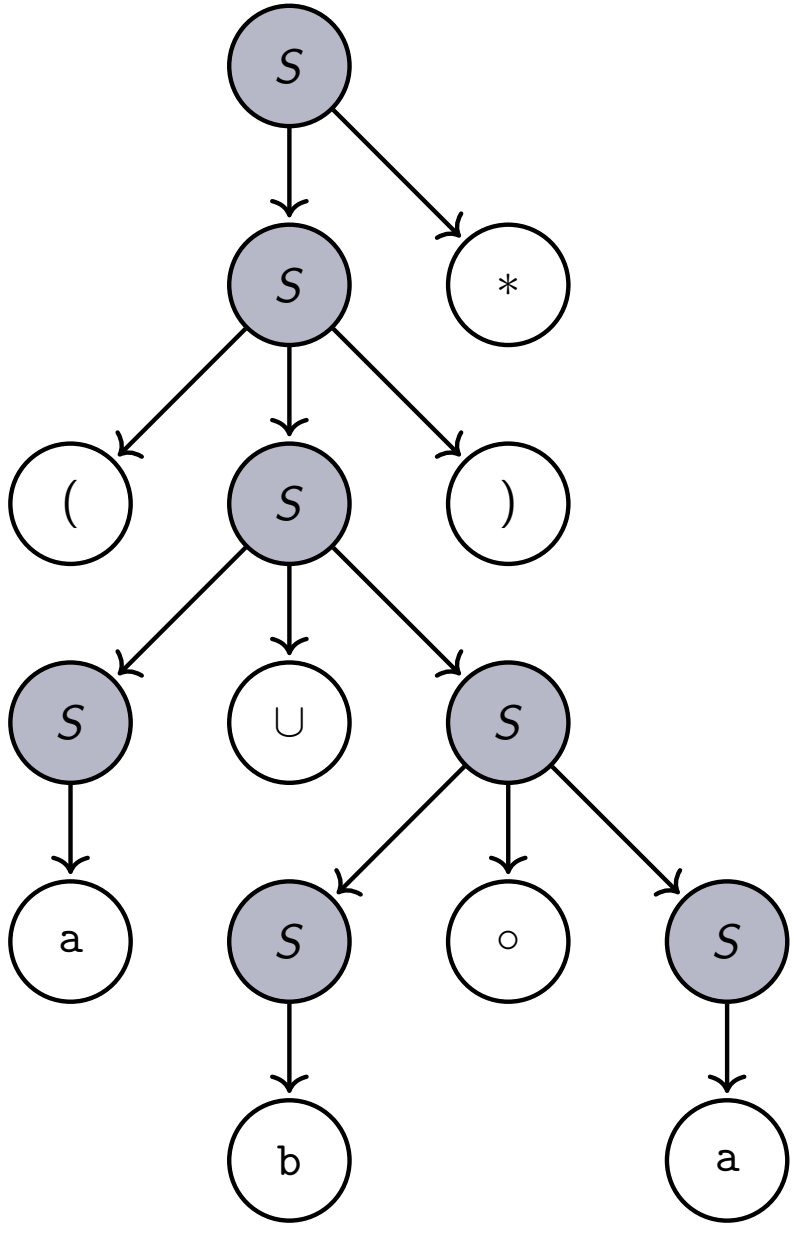
A *parse tree* is a tree that shows how to derive a string from a non-terminal.

The *yield* of a parse tree is the concatenation of all symbols at the leaves of the tree. If the root of the tree is S then the yield $x \in \mathcal{L}(G)$.

Exercise: Are there multiple parse trees possible for our example?

Ambiguity

A grammar is *ambiguous* if there is more than one parse tree (or leftmost derivation) for a given string. This can cause problems with parsing and with interpretation.



Eliminating Ambiguity

We want to eliminate ambiguity while still accepting all strings we accepted before. This is possible for our regular expressions language. Define first the atomic expressions:

$$A \to (S) \mid \emptyset \mid \varepsilon \mid \mathtt{a} \mid \mathtt{b}$$

Then expressions that may include Kleene star:

$$K \to A \mid A*$$

Then the expressions that may include composition (but left-associatively):

$$C \to K \mid C \circ K$$

Lastly, expressions that may include union:

$$S \to C \mid S \cup C$$

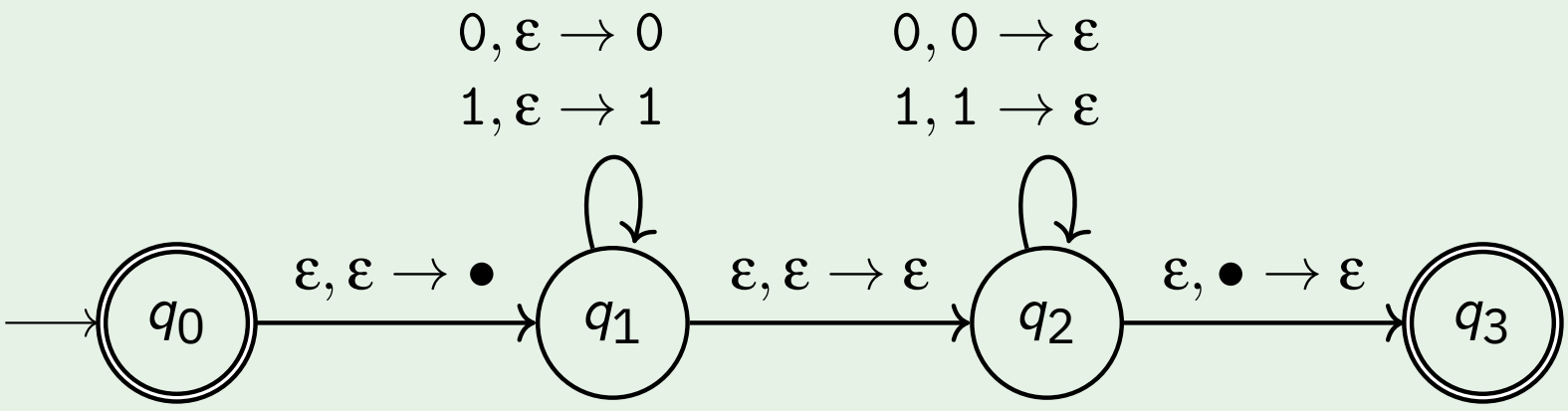
Question: What order of operations is assumed here?

Pushdown Automata

Pushdown Automata (PDAs) are to CFGs what Finite Automata are to regexps. Just as recursion is implemented with a stack in computer programming, a PDA is a ε-NFA with an additional *stack*.
It is more powerful than an NFA as it has infinite memory, but can only use it by pushing and popping symbols.

Pushdown Automata

Example (Pushdown Automaton)



Read $x, y \to z$ as consuming input x, popping y off the top of the stack, and pushing z on to the stack. The transition may only fire if y is on top of the stack.

In the above example, the input alphabet Σ is $\{0, 1\}$ and the *stack alphabet* Γ is $\{0, 1, \bullet\}$.

Exercise: What language is accepted here? Derive the string 1001.

Formally

Definition

A *pushdown automaton* is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where Q, Σ, Γ are all finite sets. Γ is the stack alphabet, and δ now may take a stack symbol as input or return one as output:

$$\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$$

All other components are as with ε-NFAs.

Acceptance

A string w is accepted by a PDA if it ends in a final state, i.e. $\delta^*(q_0, w, \varepsilon)$ gives a state q and a stack γ such that $q \in F$.

Claim

Theorem

A language is context-free iff it is recognised by a pushdown automaton.

- Think about why this might be.
- Can you think about languages that might not be context-free?
- Next lecture: beyond the context-free languages.

Claim

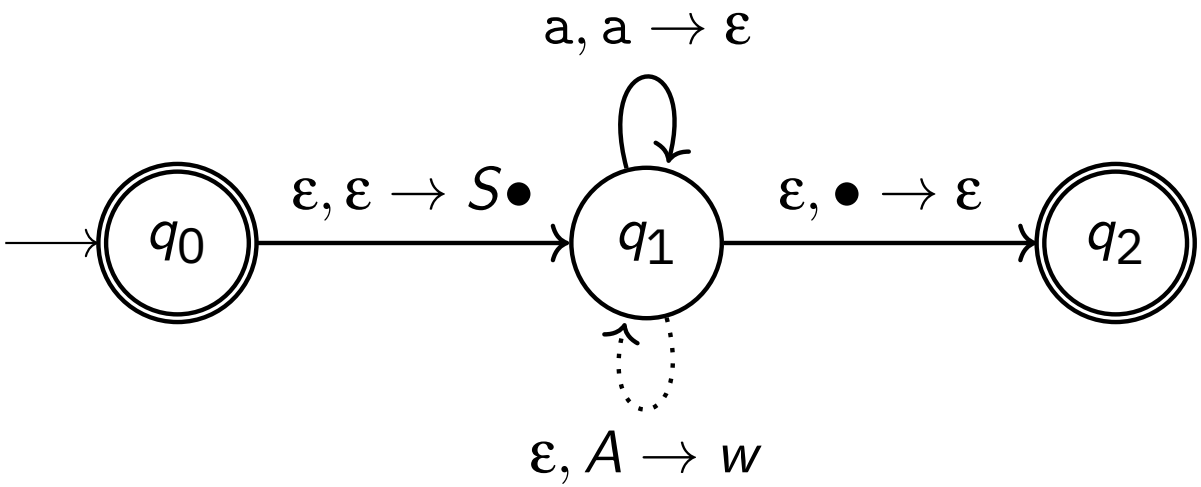
Theorem

A language is context-free iff it is recognised by a pushdown automaton.

The details of the proof of this are in Sipser's book, but I will give a sketch here.

CFG to PDA

The upper self-loop is added for every terminal a in the CFG. The lower self-loop is a shorthand for a looping sequence of states added for each production $A \to w$ that builds up w on the stack one symbol at a time.



PDA to CFG

First, we make sure that the PDA has only one accept state, empties its stack before terminating, and has only transitions that either push or pop a symbol (but not transitions that do both or neither).

Given such a PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$, we provide a CFG (V, Σ, R, S) with V containing a non-terminal A_{pq} for every pair of states $(p, q) \in Q \times Q$. The non-terminal A_{pq} generates all strings that go from p with an empty stack to q with an empty stack. Then S is just $A_{q_0 q_{\text{accept}}}$. R consists of:

- $A_{pq} \to \mathrm{a} A_{rs} \mathrm{b}$ if $p \xrightarrow{\mathrm{a}, \varepsilon \to t} r$ and $s \xrightarrow{\mathrm{b}, t \to \varepsilon} q$ (for intermediate states r, s and stack symbol t).
- $A_{pq} \to A_{pr} A_{rq}$ for all intermediate states r.
- $A_{pp} \to \varepsilon$

Proofs of why this works are in Sipser.

Closure properties

Are context-free languages closed under:

- Union? **Yes**
- Concatenation? **Yes**
- Kleene Star? **Yes**
- Intersection? **No**

Example

Consider $L_1 = \{a^i b^i c^j \mid i, j \in \mathbb{N}\}$ and $L_2 = \{a^j b^i c^i \mid i, j \in \mathbb{N}\}$.

- Complementation? **No** (via de Morgan's laws)

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. $uv^i xy^i z \in L$ 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 $uv^i xy^i z \notin L$. 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 $0^p1^p0^p1^p$. 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^p1^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

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

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 $\alpha \to \beta$, except we also allow $S \to \varepsilon$ if S does not occur on the RHS of any rule.

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

3 (or *right-linear*) if all productions are of the form $A \to w$ or $A \to wB$ 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?

Introduction to Theoretical Computer Science

Lecture 6: Universal RMs, Halting, and Turing

Richard Mayr

University of Edinburgh

Semester 1, 2025/2026

Universality

We've seen *register machines*, a computational model that I have claimed is universal.

Key Question

Can an RM *simulate* an RM?

That is, can we write an RM that, given some encoding of a machine M, written $\ulcorner M \urcorner$, computes the result (if any) of the machine M?

Firstly, we need an *encoding*.

Encoding an RM

We have registers $R_0 \ldots R_{m-1}$ and the program $I_0 \ldots I_{n-1}$.

Pairing

Recall that pairing functions allow us to pack multiple numbers $\langle a, b \rangle$ into one number.

$$
\begin{array}{lcl}
\ulcorner \mathrm{INC}(i) \urcorner & = & \langle 0, i \rangle \\
\ulcorner \mathrm{DECJZ}(i,j) \urcorner & = & \langle 1, i, j \rangle \\
\ulcorner P \urcorner & = & \langle \ulcorner I_0 \urcorner, \ldots, \ulcorner I_{n-1} \urcorner \rangle \\
\ulcorner R \urcorner & = & \langle R_0, \ldots, R_{m-1} \rangle \\
\ulcorner M \urcorner & = & \langle \ulcorner P \urcorner, \ulcorner R \urcorner \rangle
\end{array}
$$

Exercise: Write an RM program that, given such an encoding, computes its result, if any (very tedious but achievable).

Halting

The Halting Problem

Given an RM encoding $\ulcorner M \urcorner$, can we write a program to determine if the simulated machine halts or not?

- Suppose H is such an RM, which takes a machine coding $\ulcorner M \urcorner$ in R_0 and halts with 1 if M halts, and halts with 0 if M doesn't halt.
- Construct a new machine $L = (P_L, R_0 \dots)$ which, given a program $\ulcorner P \urcorner$, runs H on [the program with itself as input], i.e. the machine $(P, \ulcorner P \urcorner)$, and loops iff it halts.
- What happens if we run L with input P_L?

Contradiction!

If L halts on $\ulcorner P_L \urcorner$ that means that H says that $(P_L, \ulcorner P_L \urcorner)$ loops.
If L loops on $\ulcorner P_L \urcorner$ that means that H says that $(P_L, \ulcorner P_L \urcorner)$ halts.

Diagonalization

We saw Cantor's proof of the uncountability of infinite-length binary strings in the last lecture. This proof is another example of the same principle, which is called *diagonalization*.

Example (Gödel's first incompleteness theorem)

If a logic is capable of expressing basic (Peano) arithmetic, we can encode the provability of statements in the logic itself. Then, by the same diagonalisation trick, we can encode the statement *"This statement is not provable"* in the logic. If it is true, then it is not provable and thus the logic is *incomplete*. If it is false, then it is provable and thus the logic is *inconsistent*.

Consequences

We have sketched an argument that there are some programs that *cannot* be decided by register machines.

But what about *other* machines?

Turing Machines Reprisal

Recall a Turing Machine from prior courses. It is a machine with finite control, like an NFA or PDA, but with access to an unbounded *tape* $t_0 t_1 \ldots$ for storage. In each transition, we read and write to the tape, and move the tape head left or right.

Definition

A *Turing Machine* is a 7-tuple $\left(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}}\right)$:

- Q : states
- Σ : input symbols
- $\Gamma \supseteq \Sigma$: *tape* symbols, including a blank symbol $\sqcup$.
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{-1, +1\}$
- $q_0, q_{\text{accept}}, q_{\text{reject}} \in Q$: start, accept, reject states.

Exercise: Construct a TM to recognise $\{0^n 1^n 2^n \mid n \in \mathbb{N}\}$

Programming Turing Machines

More examples are given in Sipser, ch. 3.

Question

How do RMs compare to TMs?

I claim they are equivalent in power. How would we demonstrate this?

Exercise: Design a TM to simulate an RM
Exercise: Design a RM to simulate an TM

Upshot

The halting argument applies to TMs just as to RMs.

Extensions to Turing Machines

Do these modifications affect the expressivity of the machine?

- Adding the ability to stay put, i.e.:

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{-1, 0, +1\}$$

- Making the tape infinite in both directions?
- Restricting to only two symbols?
- Allowing multiple tapes?
- Allowing non-deterministic TMs? i.e.:

$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{-1, +1\})$$

NO

Summary

We have found one problem (halting) that we *cannot* compute in either RMs or TMs.

The Church-Turing Thesis

Any problem is computable by any model of computation iff it is computable by a Turing Machine.

Confirmed for: RMs, TMs, λ-calculus, combinator calculus, general recursive functions, pointer machines, counter machines, cellular automata, queue automata, enzyme-based DNA computers etc. etc. etc.

This means that for any model of computation we can think of, there are limits to what we can compute. Some problems are fundamentally *uncomputable* by any means. More on this next week.

Introduction to Theoretical Computer Science

Lecture 7: Undecidability

Richard Mayr

University of Edinburgh

Semester 1, 2025/2026

Computable Functions

Definitions

A (total) function $\mathbb{N} \to \mathbb{N}$ is *computable*[a] if there is an RM/TM which computes f, i.e., given an x in R_0, leaves $f(x)$ in R_0.

A *decision problem* is a set D and a query subset $Q \subseteq D$. A problem is *decidable* or *computable* if $d \in Q$ is characterised by a computable function $f : D \to \{0, 1\}$, i.e., $d \in Q \Leftrightarrow f(d) = 1$.

[a] sometimes confusingly called *recursive*, but this is old terminology.

Note that our *language* problems, for DFAs and CFGs etc., are decision problems where $D = \Sigma^*$ and Q is the language in question.
Also, consider $D = \mathbb{N}$ and $Q = Primes$.
Or $D =$ RMs and $Q =$ the halting RMs.

Closure Properties

Are the decidable languages closed under:

- Union?
- Intersection?
- Complement?

(yes)

Undecidability

We know that undecidable problems exist, like H.

Another Example

$A_{\text{RM}} = \{\langle \ulcorner M \urcorner, w \rangle \mid M \text{ accepts } w\}$
The proof, in Sipser for TMs, is analogous to our proof for H.

Aside

We can also use a counting argument. The set of RMs is enumerable, but the set of languages is uncountable. So there are languages that are not decided (or even recognised) by any RM.

How would we show that other problems are undecidable?

Reductions

A *reduction* is a transformation from one problem to another.

To prove that a problem P_2 is *hard*, show that there is an *easy* reduction from a known hard problem P_1 to P_2.

Therefore

To prove that a problem P_2 is undecidable, show that there is a computable reduction from a known undecidable P_1 to P_2.

Pay close attention to the direction of the proof!

A correct example

Suppose it is well known that James cannot lift a car.

Theorem

James cannot lift a loaded truck.

Proof

By reduction from the car-lifting problem (P_1). Suppose James could lift a loaded truck. Then, he could lift a car by putting the car in the truck and then lifting the truck.
But, it is known that James cannot lift a car.

Known Hard Problem $\longrightarrow$ New Problem

An incorrect example

Suppose it is well known that James cannot lift a car.

Theorem

James cannot lift a feather.

Proof

By reduction to the car-lifting problem. We can reduce the feather-lifting problem to the car-lifting problem by putting the feather in the car.
It is known that James cannot lift a car. Therefore, James cannot lift a feather (**???!**).

Reductions

A Turing Transducer is a RM (or TM) which takes an instance d of a problem $P_1 = (D_1, Q_1)$ in R_0 and halts with an instance $d' = f(d)$ of $P_2 = (D_2, Q_2)$ in R_0. Thus, f is a computable function $D_1 \to D_2$.

Definition

A *mapping reduction* (or *many-one* reduction) from P_1 to P_2 is a Turing transducer f as above such that $d \in Q_1$ iff $f(d) \in Q_2$

If A is mapping reducible to B (written $A \leq_m B$), and A is undecidable, then B is undecidable.

Example

$$\text{NotEmpty}_{\text{TM}} = \{\ulcorner M \urcorner \mid \mathcal{L}(M) \neq \emptyset\}$$

Example (Proof)

We sketch a mapping reduction from A_{TM} to $\text{NotEmpty}_{\text{TM}}$. Given an instance $\langle M, w \rangle$ of A_{TM}, our reduction constructs a machine M' whose language is either $\{w\}$ or $\emptyset$. Given input x, it will reject if $x \neq w$, else run M on w.

Note that $\langle M, w \rangle \in A_{\text{TM}}$ iff $M' \in \text{NotEmpty}_{\text{TM}}$.

Thus, if we could solve $\text{NotEmpty}_{\text{TM}}$ we could solve A_{TM}, which we know is undecidable. Thus $\text{NotEmpty}_{\text{TM}}$ too is undecidable.

Uniform Halting

$$UH = \{\ulcorner M \urcorner \mid M \text{ halts on all inputs}\}$$

Example (Proof)

We reduce from H to UH. Given a machine M and input w, build a machine M' which ignores its input, writes w to the tape, and then behaves as M. Then M' halts on any input iff M halts on w.

The Looping Problem

Let *L* be the subset of RMs (or TMs) that go into an infinite loop. Show that *L* is undecidable.

Since *L* is the complement of *H*, this seems easy, but we can't fit it neatly into our definition of a mapping reduction.

Oracles

Definition

Given a decision problem (D, Q), an *oracle* for Q is a 'magic' RM instruction $\text{ORACLE}_Q(i)$ which, given an encoding of $d \in D$ in R_i, sets R_i to contain 1 iff $d \in Q$.

Consider RMs augmented with an oracle for halting H, sometimes written RM^H. We'll return to this.

If a problem P is decidable, is a machine RM^P more powerful than a standard RM?
No. No point in having decidable oracles!

Turing Reductions

Definition

A *Turing reduction* from P_1 to P_2 is an RM (or TM) equipped with an Oracle for P_2 that solves P_1.

Decidability results carry across Turing reductions just as with mapping reductions. But mapping reductions make *finer* distinctions of computing power.

Observe that H is Turing-reducible to L, and thus L is also undecidable.

Rice's Theorem

- A *property* is a set of RM (or TM) descriptions.
- A property is *nontrivial* if it contains some but not all descriptions.
- A property P is *semantic* if

$$\mathcal{L}(M_1) = \mathcal{L}(M_2) \Rightarrow (\ulcorner M_1 \urcorner \in P \Leftrightarrow \ulcorner M_2 \urcorner \in P)$$

  In other words, it concerns the language and not the particular implementation of the machine.

Rice's Theorem

All nontrivial semantic properties of TM/RM are undecidable.

Proof

Assume to the contrary that a nontrivial semantic property P is decidable, and it is decided by an RM M_P. W.l.o.g. a RM $T_\emptyset$ that always rejects is not in P — otherwise we shall proceed with the complement of P instead.
Let T be a RM with $\ulcorner T \urcorner \in P$. We build an M_P oracle-equipped RM S to decide A_{RM}.
On input $\langle M, w \rangle$:

1. Build a RM $N_{M,w}$ which on input x, simulates M on w. If M halts and rejects, it rejects. Otherwise, it simulates T on x, and accepts if T accepts.
2. Use M_P to answer if $\ulcorner N_{M,w} \urcorner \in P$.

Note the language of $N_{M,w}$ is $\mathcal{L}(T)$ if w is accepted by M and $\emptyset$ otherwise.
We know A_{TM} is undecidable, so P must also be undecidable.

Applications of Rice's Theorem

The following are all undecidable by Rice's theorem:

- Whether a language (of an RM/TM) is empty.
- Whether a language (of an RM/TM) is non-empty.
- Whether a language (of an RM/TM) is regular.
- Whether a language (of an RM/TM) is context-free.

Note

Sometimes we can prove these properties for particular machines, but it is not decidable in general.

Wrong applications of Rice's Theorem

Rice's theorem cannot be used for these:

- Whether a TM has less than 7 states.
- Whether a TM has a final state.
- Whether a TM has a start state.

(Note how these are properties of machines, not languages. I.e., they are not semantic.)

- Whether a language (of an RM/TM) is a subset of Σ^*.
- Whether a language of an RM is a language of a TM.

(These properties are trivial).

Far-reaching Consequence

We cannot write a program that answers a non-trivial question about black-box semantic properties of programs.

(However, there can exist decidable non-trivial non-semantic properties of programs.)

Next time..

We have developed a theory of undecidable problems, and shown how reductions can be used to show more problems are undecidable. We also saw the daisy cutter of undecidability results, Rice's theorem.

Next time

We will address semi-decidable problems. What about machines where we always halt if we accept, but if we do not accept, we may loop forever?