Finite Automata Theory and Formal Languages

TMV027/DIT321– LP4 2016

Lecture 14
Ana Bove

May 19th 2016

Overview of today's lecture:

- Push-down automata;
- Turing machines.

Recap: Context-free Languages

- Closure properties for CFL:
  - Union, concatenation, closure, reversal and prefix;
  - Intersection and difference with a RL;
  - No closure under complement;
- Decision properties for CFL:
  - Is the language empty?
  - Does a word belong to the language of a certain grammar?
- The following problems are undecidable:
  - Is the CFG G ambiguous?
  - Is the CFL $\mathcal{L}$ inherently ambiguous?
  - If $\mathcal{L}_1$ and $\mathcal{L}_2$ are CFL, is $\mathcal{L}_1 \cap \mathcal{L}_2 = \emptyset$?
  - If $\mathcal{L}_1$ and $\mathcal{L}_2$ are CFL, is $\mathcal{L}_1 = \mathcal{L}_2$? is $\mathcal{L}_1 \subseteq \mathcal{L}_2$?
  - If $\mathcal{L}$ is a CFL and $\mathcal{P}$ a RL, is $\mathcal{P} = \mathcal{L}$? is $\mathcal{P} \subseteq \mathcal{L}$?
  - If $\mathcal{L}$ is a CFL over Σ, is $\mathcal{L} = \Sigma^*$?

Push-down Automata

Push-down automata (PDA) are essentially ϵ-NFA with a *stack* to store information.

The stack is needed to give the automata extra "memory".

Observe we can only access the last element that was added to the stack!

Example: To recognise the language 0^n1^n we proceed as follows:

- When reading the 0's, we push a symbol into the stack;
- When reading the 1's, we pop the symbol on top of the stack;
- We accept the word if when we finish reading the input then the stack is empty.

The languages accepted by the PDA are exactly the CFL.
See the book, sections 6.1–6.3.

Variation of Push-down Automata

DPDA = DFA + stack: Accepts a language that is between RL and CFL.
The lang. accepted by DPDA have unambiguous grammars.
However, not all languages that have unambiguous grammars can be accepted by these DPDA.

Example: The language generated by the unambiguous grammar

$$S \to 0S0 \mid 1S1 \mid \epsilon$$

cannot be recognised by a DPDA.
See section 6.4 in the book.

2 or more stacks: A PDA with at least 2 stacks is as powerful as a TM.
Hence these PDA can recognise the *recursively enumerable* languages (more on this later).
See section 8.5.2.

Undecidable Problems

Recall: An *undecidable problem* is a decision problem for which it is impossible to construct a single algorithm that always leads to a yes-or-no answer.

To prove that a certain problem P is undecidable one usually *reduces* an already known undecidable problem U to the problem P: instances of U become instances of P.

(Can be seen like one "transforms" U so it "becomes" P).

That is, $w \in U$ iff $w' \in P$ for certain w and w'.
Then, a solution to P would serve as a solution to U.

However, we know there are no solutions to U since U is known to be undecidable.
Then we have a contradiction.

Example of Undecidable Problem: Post's Correspondence

It is an undecidable decision problem introduced by Emil Post in 1946.

> *Given words $u_1, \ldots, u_n$ and $v_1, \ldots, v_n$ in $\{0, 1\}^*$, is it possible to find $i_1, \ldots, i_k$ such that $u_{i_1} \ldots u_{i_k} = v_{i_1} \ldots v_{i_k}$?*

Example: Given $u_1 = 1, u_2 = 10, u_3 = 001, v_1 = 011, v_2 = 11, v_3 = 00$ we have that $u_3 u_2 u_3 u_1 = v_3 v_2 v_3 v_1 = 001100011$.

We can use grammars to show that the Post's correspondence problem is undecidable by showing that a grammar is ambiguous iff the PCP has a solution.

(See Section 9.4 in the book.)

Undecidable and Intractable Problems

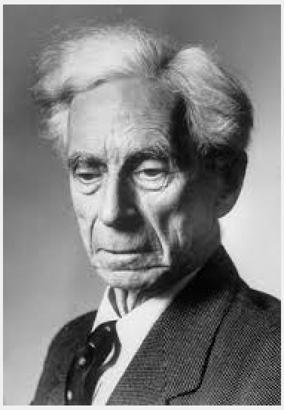
The theory of undecidable problems provides a guidance about what we may or may not be able to perform with a computer.

One should though distinguish between undecidable problems and *intractable problems*, that is, problems that are decidable but require a large amount of time to solve them.

(In daily life, intractable problems are more common than undecidable ones.)

To reason about both kind of problems we need to have a basic notion of *computation*.

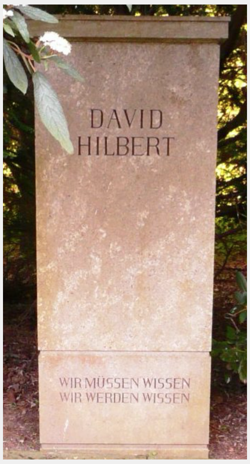
Once Upon a Time ...



In early 1900's, Bertrand Russell showed that formal logic can express large parts of mathematics.

In 1928, David Hilbert posed a challenge known as the Entscheidungsproblem (decision problem). This problem asked for an *effectively calculable* procedure to determine whether a given statement is provable from the axioms using the rules of logic.

To Prove or Not To Prove: That Is the Question!



The decision problem presupposed completeness: any statement or its negation can be proved.

"Wir müssen wissen, wir werden wissen"
("We must know, we will know")

In 1931, Kurt Gödel published the *incompleteness theorems*.

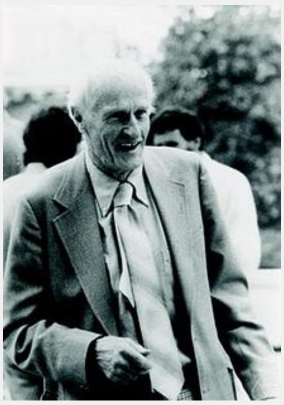
The first theorem shows that any consistent system capable of expressing arithmetic cannot be complete: there is a true statement that cannot be proved with the rules of the system.

The second theorem shows that such a system could not prove its own consistency.

λ-Calculus as a Language for Logic

In the '30s, Alonzo Church (and his students Stephen Kleene and John Barkley Rosser) introduced the λ-calculus as a way to define notations for logical formulas:

$$x \mid \lambda x.M \mid M\ N$$



In 1935, Kleene and Rosser proved the system inconsistent (due to self application).

λ-Calculus as a Language for Computations

Church discovered how to encode numbers in the λ-calculus.

For example, 3 is encoded as $\lambda f.\lambda x.f(f(f(x)))$.

Encoding for addition, multiplication and (later) predecesor were defined.

Thereafter Church and his students became convinced any *effectively calculable* function of numbers could be represented by a term in the λ-calculus.

Church's Thesis

Church proposed λ-definability as the definition of effectively calculable (known today as *Church's Thesis*).

He also demonstrated that the problem of whether a given λ-term has a *normal form* was not λ-definable (equivalent to the *Halting problem*).

A year later, he demonstrated there was no λ-definable solution to the Entscheidungsproblem.

General Recursive Functions

1933: Gödel was not convinced by Church's assertion that every effectively calculable function was λ-definable.

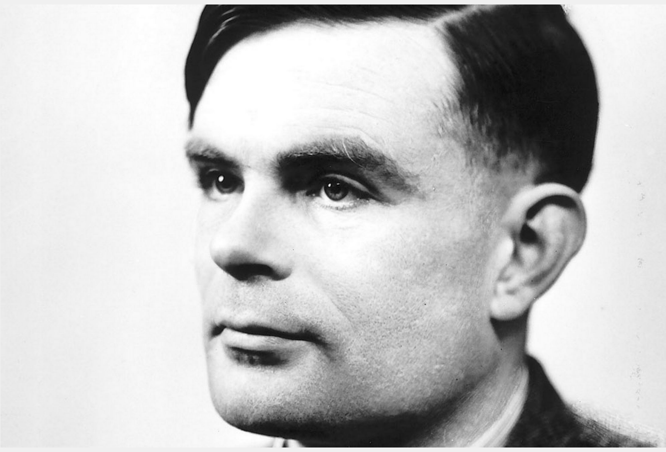
Church offered that Gödel would propose a different definition which he then would prove it was included in λ-definability.

1934: Gödel proposed the *general recursive functions* as his candidate for effective calculability (system which Kleene after developed and published).

Church and his students then proved that the two definitions were equivalent.

Now Gödel doubt his own definition was correct!

Turing Machines



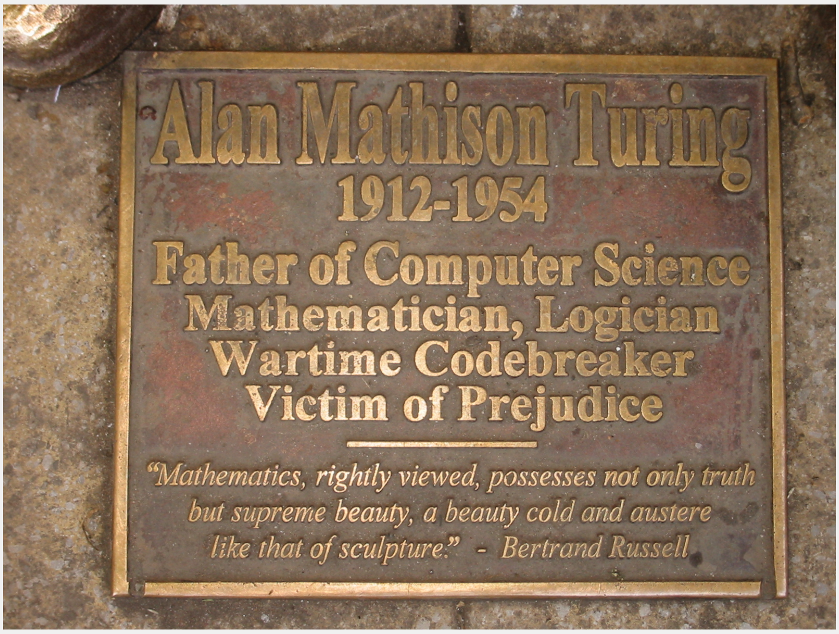
Simultaneously, Alan Mathison Turing formulated his notion of effectively calculable in terms of a *Turing machine*.

He used the Turing machines to show the *Entscheidungsproblem* undecidable by first showing that the *halting problem* was undecidable.

Turing also proved the equivalence of the λ-calculus and his machines. (*Church-Turing Thesis*)

Gödel is now finally convinced! :-)

Computer Science Was Born!



Turing's approach took into account the capabilities of a *(human) computer*: a human performing a computation assisted by paper and pencil.

Alan Mathison Turing (23 June 1912 – 7 June 1954)

- British computer scientist, mathematician, logician and cryptanalyst;
- Considered the father of theoretical computer science and artificial intelligence;
- Philosopher, theoretical biologist;
- Marathon and ultra distance runner;
- In the 50' he also became interested in chemistry.

Alan Mathison Turing

- He started studying at Cambridge and then moved to Princeton where he took his Ph.D. in 1938 with Alonzo Church;
- He invented the concept of a computer, called *Turing Machine* (TM);

  Turing showed that TM could perform any kind of *computation*;

  He also showed that his notion of *computable* was equivalent to Church's notion of *effective calculable*;
- During the WWII he helped Britain to break the German Enigma machines which shortened the war by 2-4 years and saved many lives!
- Since 1966, ACM annually gives the *Turing Award* for contributions to the computing community.

Turing Machines (1936)

- Theoretically, a TM is just as *powerful* as any other computer!

  Powerful here refers only to which computations a TM is capable of doing, not to how *fast* or *efficiently* it does its job.
- Conceptually, a TM has a finite set of states, a finite alphabet (containing a blank symbol), and a finite set of instructions;
- Physically, it has a *head* that can read, write, and move along an *infinitely long tape* (on both sides) that is divided into *cells*.
- Each cell contains a symbol of the alphabet (possibly the blank symbol):

$\cdots$		a_1	a_2	a_3	a_4	a_5		$\cdots$

$\uparrow$

Turing Machines: More Concretely

- Let $\Box$ represents the *blank* symbol and let Σ be a non-empty alphabet of symbols such that $\{\Box, \mathsf{L}, \mathsf{R}\} \cap \Sigma = \emptyset$.

  Now, we define $\Sigma' = \Sigma \cup \{\Box\}$;
- The read/write head of the TM is always placed over one of the cells. We said that that particular cell is being *read*, *examined* or *scanned*;
- At every moment, the TM is in a certain state $q \in Q$, where Q is a non-empty and finite set of states;
- In some cases, we consider a set F of final states.

Turing Machines: Transition Functions

In one *move*, the TM will:

1. Change to a (possibly) new state;
2. Replace the symbol below the head by a (possibly) new symbol;
3. Move the head to the left (denoted L) or to the right (denoted R).

The behaviour of a TM is given by a <u>possibly partial</u> *transition function*

$$\delta \in Q \times \Sigma' \to Q \times \Sigma' \times \{\mathsf{L}, \mathsf{R}\}$$

δ is such that for every $q \in Q$, $a \in \Sigma'$ there is *at most* one instruction.

Note: We have a *deterministic* TM.

How to Compute with a TM?

Before the execution starts, the tape of a TM looks as follows:

$\cdots$		a_1	a_2	$\cdots$	a_{n-1}	a_n		b_1	$\cdots$	b_m		$\cdots$
		$\uparrow$										

- The input data is placed on the tape, if necessary separated with blanks;
- There are infinitely many blank to the left and to the right of the input;
- The head is placed on the first symbol of the input;
- The TM is in a special *initial state* $q_0 \in Q$;
- The machine then proceeds according to the transition function δ.

Turing Machine: Formal Definition

Definition: A *TM* is a 6-tuple $(Q, \Sigma, \delta, q_0, \square, F)$ where:

- Q is a non-empty, finite set of states;
- Σ is a non-empty alphabet such that $\{\square, \mathsf{L}, \mathsf{R}\} \cap \Sigma = \emptyset$;
- $\delta \in Q \times \Sigma' \to Q \times \Sigma' \times \{\mathsf{L}, \mathsf{R}\}$ is a transition function, where $\Sigma' = \Sigma \cup \{\square\}$;
- $q_0 \in Q$ is the initial state;
- $\square$ is the blank symbol, $\square \notin \Sigma$;
- F is a non-empty, finite set of final or accepting states, $F \subseteq Q$.

Note: In some cases, the set F is not relevant (compare with FA).

Result of a Turing Machine

Definition: Let $M = (Q, \Sigma, \delta, q_0, \Box, F)$ be a TM.
We say that M *halts* if for certain $q \in Q$ and $a \in \Sigma$, $\delta(q, a)$ is undefined.

Whatever is written in the tape when the TM *halts* can be considered as the *result* of the computation performed by the TM.

If we are only interested in the result of a computation, we can omit F from the formal definition of the TM.

Examples

Example: Let $\Sigma = \{0, 1\}$, $Q = \{q_0\}$ and let δ be as follows:

$$\delta(q_0, 0) = (q_0, 1, \mathrm{R})$$
$$\delta(q_0, 1) = (q_0, 0, \mathrm{R})$$

What does this TM do?

Example: The execution of a TM might loop.

Consider the following set of instructions for Σ and Q as above.

$$\delta(q_0, a) = (q_0, a, \mathrm{R}) \qquad \text{with } a \in \Sigma \cup \{\Box\}$$

Recursive and Recursively Enumerable Languages

Let $M = (Q, \Sigma, \delta, q_0, \Box, F)$ be a TM.

Definition: The TM M accepts a word $w \in \Sigma^*$ if when we run M with w as input, the TM is in a final state when it halts.

Definition: The *language* accepted by a TM is the set of words that are accepted by the TM.

Definition: A language is called *recursively enumerable* if there is a TM accepting the words in that language.

Definition: A *Turing decider* is a TM that never loops, i.e. the TM halts.

Definition: A language is called *recursive* or *decidable* if there is a Turing decider accepting the words in the language.

Example of a Turing Decider

How to define a TM that accepts the language $\mathcal{L} = \{ww^r \mid w \in \{0,1\}^*\}$?

Let $\Sigma = \{0, 1, X, Y\}$, $Q = \{q_0, \ldots, q_7\}$ and $F = \{q_7\}$,

Let $a \in \{0, 1\}$, $b \in \{X, Y, \Box\}$, and $c \in \{X, Y\}$.

$$
\begin{array}{lll}
\delta(q_0, 0) = (q_1, X, \mathrm{R}) & \delta(q_0, 1) = (q_3, Y, \mathrm{R}) & \delta(q_0, \Box) = (q_7, \Box, \mathrm{R}) \\
\delta(q_1, a) = (q_1, a, \mathrm{R}) & \delta(q_3, a) = (q_3, a, \mathrm{R}) & \\
\delta(q_1, b) = (q_2, b, \mathrm{L}) & \delta(q_3, b) = (q_4, b, \mathrm{L}) & \\
\delta(q_2, 0) = (q_5, X, \mathrm{L}) & \delta(q_4, 1) = (q_5, Y, \mathrm{L}) & \\
\delta(q_5, a) = (q_6, a, \mathrm{L}) & & \delta(q_5, c) = (q_7, c, \mathrm{R}) \\
\delta(q_6, a) = (q_6, a, \mathrm{L}) & \delta(q_6, c) = (q_0, c, \mathrm{R}) &
\end{array}
$$

What happens with the input 0110?

And with the input 010?

Overview of Next Lecture

- More on Turing machines;
- Summary of the course.