Finite Automata Theory and Formal Languages

TMV027/DIT321– LP4 2014

Lecture 14
Ana Bove

May 15th 2014

Overview of today's lecture:

- Closure properties for CFL;
- Push-down automata;
- Turing machines.

Recap: Context-free Languages

- Decision properties for CFL
  - Is the language empty?
  - Does a word belong to the language of a certain grammar?
- Automata and grammars for programming language technology and natural language translation.

Closure under Union

Theorem: *Let $G_1 = (V_1, T, \mathcal{R}_1, S_1)$ and $G_2 = (V_2, T, \mathcal{R}_2, S_2)$ be CFG. Then $\mathcal{L}(G_1) \cup \mathcal{L}(G_2)$ is a context-free language.*

Proof: Let us assume $V_1 \cap V_2 = \emptyset$ (easy to get via renaming).

Let S be a fresh variable.

We construct $G = (V_1 \cup V_2 \cup \{S\}, T, \mathcal{R}_1 \cup \mathcal{R}_2 \cup \{S \to S_1 \mid S_2\}, S)$.

It is now easy to see that $\mathcal{L}(G) = \mathcal{L}(G_1) \cup \mathcal{L}(G_2)$ since a derivation will have the form

$$S \Rightarrow S_1 \Rightarrow^* w \text{ if } w \in \mathcal{L}(G_1)$$

or

$$S \Rightarrow S_2 \Rightarrow^* w \text{ if } w \in \mathcal{L}(G_2)$$

Closure under Concatenation

Theorem: *Let $G_1 = (V_1, T, \mathcal{R}_1, S_1)$ and $G_2 = (V_2, T, \mathcal{R}_2, S_2)$ be CFG. Then $\mathcal{L}(G_1)\mathcal{L}(G_2)$ is a context-free language.*

Proof: Again, let us assume $V_1 \cap V_2 = \emptyset$.

Let S be a fresh variable.

We construct $G = (V_1 \cup V_2 \cup \{S\}, T, \mathcal{R}_1 \cup \mathcal{R}_2 \cup \{S \to S_1 S_2\}, S)$.

It is now easy to see that $\mathcal{L}(G) = \mathcal{L}(G_1)\mathcal{L}(G_2)$ since a derivation will have the form

$$S \Rightarrow S_1 S_2 \Rightarrow^* uv$$

with

$$S_1 \Rightarrow^* u \text{ and } S_2 \Rightarrow^* v$$

for $u \in \mathcal{L}(G_1)$ and $v \in \mathcal{L}(G_2)$.

Closure under Closure

Theorem: *Let $G = (V, T, \mathcal{R}, S)$ be a CFG.*
Then $\mathcal{L}(G)^+$ and $\mathcal{L}(G)^*$ are context-free languages.

Proof: Let S' be a fresh variable.

We construct $G+ = (V \cup \{S'\}, T, \mathcal{R} \cup \{S' \to S \mid SS'\}, S')$ and $G* = (V \cup \{S'\}, T, \mathcal{R} \cup \{S' \to \epsilon \mid SS'\}, S')$.

It is easy to see that $S' \Rightarrow \epsilon$ in $G*$.

It is also easy to see that $S' \Rightarrow^* S \Rightarrow^* w$ if $w \in \mathcal{L}(G)$ is a valid derivation both in $G+$ and in $G*$.

In addition, if $w_1, \ldots, w_k \in \mathcal{L}(G)$, it is easy to see that the derivation

$$\begin{aligned} S' &\Rightarrow SS' \Rightarrow^* w_1S' \Rightarrow w_1SS' \Rightarrow^* w_1w_2S' \Rightarrow^* \ldots \\ &\Rightarrow^* w_1w_2\ldots w_{k-1}S' \Rightarrow^* w_1w_2\ldots w_{k-1}S \Rightarrow^* w_1w_2\ldots w_{k-1}w_k \end{aligned}$$

is a valid derivation both in $G+$ and in $G*$.

Non Closure under Intersection

Example: Consider the following languages over $\{a, b, c\}$:

$$\mathcal{L}_1 = \{a^kb^kc^m \mid k, m > 0\}$$

$$\mathcal{L}_2 = \{a^mb^kc^k \mid k, m > 0\}$$

It is easy to give CFG generating both $\mathcal{L}_1$ and $\mathcal{L}_2$, hence $\mathcal{L}_1$ and $\mathcal{L}_2$ are CFL.

However $\mathcal{L}_1 \cap \mathcal{L}_2 = \{a^kb^kc^k \mid k > 0\}$ is not a CFL (see slide 26 lecture 12).

Closure under Intersection with Regular Language

Theorem: *If $\mathcal{L}$ is a CFL and $\mathcal{P}$ is a RL then $\mathcal{L} \cap \mathcal{P}$ is a CFL.*

Proof: See Theorem 7.27 in the book.

(It uses *push-down automata* which we have not seen.)

Example: Consider the following language over $\Sigma = \{0, 1\}$:

$$\mathcal{L} = \{ww \mid w \in \Sigma^*\}$$

Consider now $\mathcal{L}' = \mathcal{L} \cap \mathcal{L}(0^*1^*0^*1^*) = \{0^n1^m0^n1^m \mid n, m \geqslant 0\}$.

$\mathcal{L}'$ is not a CFL (see exercise 6 on exercises for week 7).

Hence $\mathcal{L}$ cannot be a CFL since $\mathcal{L}(0^*1^*0^*1^*)$ is a RL.

Non Closure under Complement

Theorem: *CFL are not closed under complement.*

Proof: Notice that

$$\mathcal{L}_1 \cap \mathcal{L}_2 = \overline{\overline{\mathcal{L}_1} \cup \overline{\mathcal{L}_2}}$$

If CFL are closed under complement then they should be closed under intersection (since they are closed under union).

Then CFL are in general not closed under complement.

Closure under Difference?

Theorem: *CFL are not closed under difference.*

Proof: Let $\mathcal{L}$ be a CFL over Σ.

It is easy to give a CFG that generates Σ^*.

Observe that $\overline{\mathcal{L}} = \Sigma^* - \mathcal{L}$.

Then if CFL are closed under difference they would also be closed under complement.

Theorem: *If $\mathcal{L}$ is a CFL and $\mathcal{P}$ is a RL then $\mathcal{L} - \mathcal{P}$ is a CFL.*

Proof: Observe that $\overline{\mathcal{P}}$ is a RL and $\mathcal{L} - \mathcal{P} = \mathcal{L} \cap \overline{\mathcal{P}}$.

Closure under Reversal and Prefix

Theorem: *If $\mathcal{L}$ is a CFL then so is $\mathcal{L}^{\mathrm{r}} = \{\mathrm{rev}(w) \mid w \in \mathcal{L}\}$.*

Proof: Given a CFG $G = (V, T, \mathcal{R}, S)$ for $\mathcal{L}$ we construct the grammar $G^{\mathrm{r}} = (V, T, \mathcal{R}^{\mathrm{r}}, S)$ where $\mathcal{R}^{\mathrm{r}}$ is such that, for each rule $A \to \alpha$ in $\mathcal{R}$, then $A \to \mathrm{rev}(\alpha)$ is in $\mathcal{R}^{\mathrm{r}}$.

One should show by induction on the length of the derivations in G and G^{r} that $\mathcal{L}(G^{\mathrm{r}}) = \mathcal{L}^{\mathrm{r}}$.

Theorem: *If $\mathcal{L}$ is a CFL then so is* $\mathrm{Prefix}(\mathcal{L})$.

Proof: For closure under prefix see exercise 7.3.1 part a) in the book.

Closure under Homomorphisms

Theorem: *CFL are closed under homomorphisms.*

Proof: See Theorem 7.24 point 4 in the book.

(It uses the notion of *substitution* which we have not seen.)

Push-down Automata

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

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

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 the RL and the 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.
See section 8.5.2.

Undecidable Problems

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

Example: Halting problem: does this program terminate?

There are many undecidable for CFL:

- 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 $\mathcal{L} = \Sigma^*$?

Undecidable Problems

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: We can use grammars to show that the Post's correspondence problem is undecidable (Emil Post, 1946) by showing that a grammar is ambiguous iff the PCP has a solution.

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*.

Entscheidungsproblem (Decision Problem)

The *Entscheidungsproblem* (David Hilbert 1928) asks for an *algorithm* to decide whether a given statement is provable from the axioms using the rules of first-order logic.

To answer the question, the notion of *algorithm* had to be formally defined.

In 1936, Alonzo Church defined the concept of *effective calculable* based on his λ-calculus.

Also in 1936, Alan Turing presented the *Turing machines*.

(It was then proved that these are equivalent *models of computation*.)

In 1936, both published independent papers showing that a general solution to the Entscheidungsproblem is impossible.

Alan Turing (1912 – 1954)

- *Alan Turing* was a mathematician, logician, cryptanalyst, and computer scientist.
  In the 50' he also became interested in chemistry;
- He took his Ph.D. in 1938 at Princeton 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 and saved many lives!
- Since 1966, ACM annually gives the *Turing Award* for contributions to the computing community.

Turing Machines (1936)

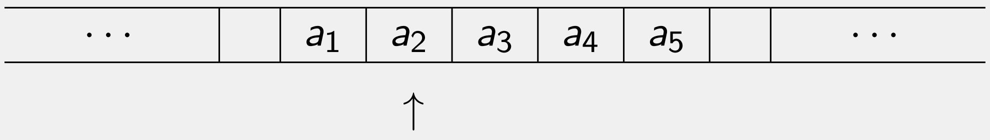
- Theoretically, a TM is just as *powerful* as any other computer!

  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):



Turing Machines: More Concretely

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

  Now, we define $\Sigma' = \Sigma \cup \{\square\}$;

- 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 by L) or to the right (denoted by R).

The behaviour of a TM is described by a (possibly partial) *transition function*

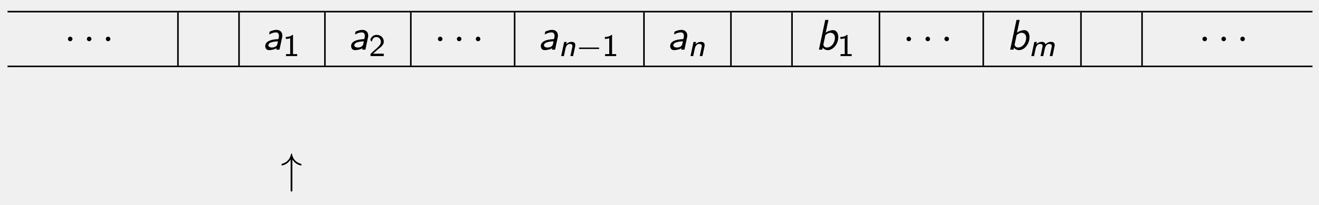
$$\delta \in Q \times \Sigma' \to Q \times \Sigma' \times \{\mathsf{L}, \mathsf{R}\}$$

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:



- 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 $(Q, \Sigma, \delta, q_0, \Box, F)$ where:

- Q is a non-empty, finite set of states;
- Σ is a non-empty alphabet such that $\{\Box, \mathrm{L}, \mathrm{R}\} \cap \Sigma = \emptyset$;
- $\delta \in Q \times \Sigma' \to Q \times \Sigma' \times \{\mathrm{L}, \mathrm{R}\}$ is a transition function, where $\Sigma' = \Sigma \cup \{\Box\}$;
- $q_0 \in Q$ is the initial state;
- $\Box$ is the blank symbol, $\Box \notin \Sigma$;
- 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:

$$\begin{aligned}\delta(q_0, 0) &= (q_0, 1, \mathrm{R})\\ \delta(q_0, 1) &= (q_0, 0, \mathrm{R})\end{aligned}$$

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 Recursive Enumerable Languages

Definition: Let $M = (Q, \Sigma, \delta, q_0, \Box, F)$ be a TM.
The TM M accepts a word $w \in \Sigma^*$ if when we run M with w as input data we reach a final state.

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

Definition: A languages 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, that is, the TM halts.

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

Example

The following TM accepts the language $\mathcal{L} = \{ww^r \mid w \in \{0,1\}^*\}$.
(One can prove using the Pumping lemma that this language is not context-free.)

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

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

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

What happens with the input 0110?
And with the input 010?

Turing Completeness

Definition: A collection of data-manipulation rules (for example, a programming language) is said to be *Turing complete* if and only if such system can simulate any single-taped Turing machine.

Example: Recursive functions (Stephen Kleene, 1936?) and λ-calculus (Alonzo Church, 1936).

The three models of computation were shown to be equivalent by Church, Kleene & (John Barkley) Rosser (1934–6) and Turing (1936-7).

Church-Turing Thesis (AKA Church Thesis)

A function is *algorithmically computable* if and only if it can be defined as a Turing Machine.

(Recall that the λ-calculus and Turing machines were shown to be computationally equivalent).

Note: This is not a theorem and it can never be one since there is no precise way to define what it means to be *algorithmically computable*.

However, it is strongly believed that both statements are true since they have not been refuted in the ca. 80 years which have passed since they were first formulated.

Overview of Next Lecture

- More on Turing machines;
- Summary of the course;
- (Exam exercises).