Finite Automata Theory and Formal Languages

TMV027/DIT321– LP4 2014

Lecture 15

Ana Bove

May 19th 2014

Overview of today's lecture:

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

Recap: CFL, PDA, TM

- Closure properties for CFL:
  - Union, concatenation, closure, reversal, prefix and homomorphism;
  - Intersection and difference with a RL;
  - No closure under complement;
- Push-down automata;
- Turing machines:
  - Defined by a 6-tuple $(Q, \Sigma, \delta, q_0, \Box, F)$;
  - Has an infinite tape on both sides;
  - Has a head that reads/writes and moves to left or right;
  - Transition function $\delta \in Q \times \Sigma' \to Q \times \Sigma' \times \{\mathsf{L}, \mathsf{R}\}$;
  - Language accepted by a TM;
  - Turing decider;
  - Recursive and recursively enumerable languages.

Example of a Turing Decider

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 $c \in \{X, Y\}$.

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

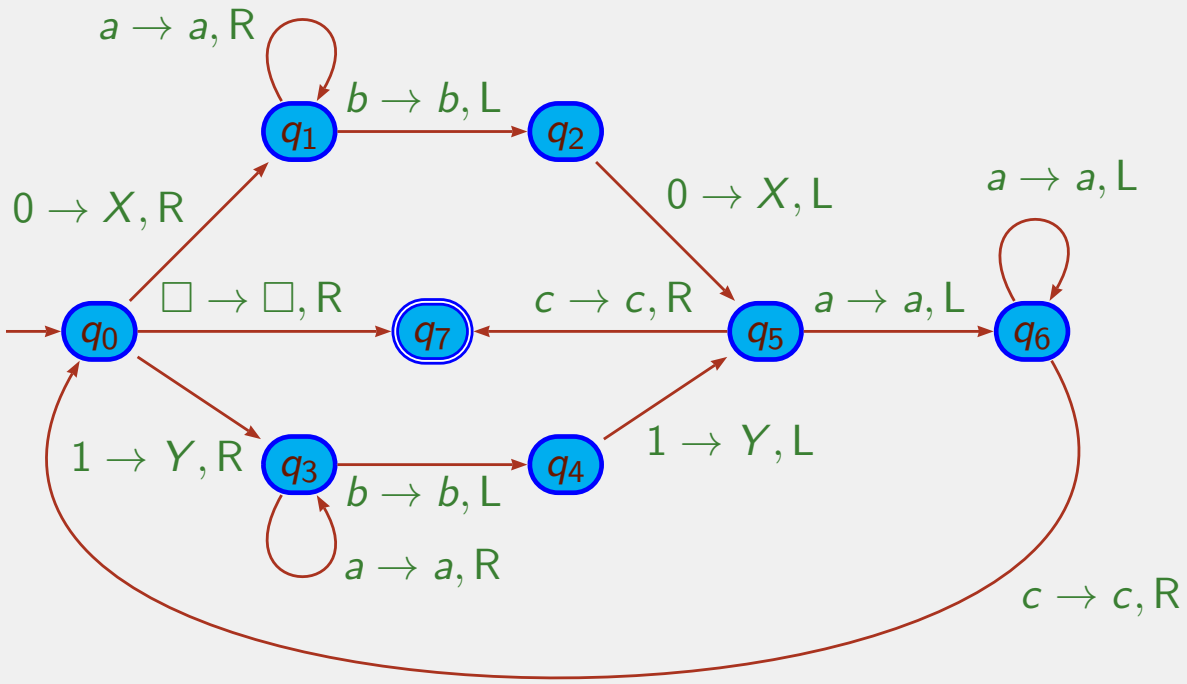
How easy is to understand the "program"?

High-level Description of a TM for $\{ww^r \mid w \in \{0,1\}^*\}$

1. If in the initial state we read 0 (resp. 1) then mark with an X (resp. Y), move R and change to the state that *searches* for the corresponding 0 (resp.1) at the end.
   If in the initial state we read $\square$, the word is empty and we accept;
2. When searching for the corresponding 0 (resp. 1) move R over 0's and 1's.
   If when searching for the corresponding 0 (resp. 1) we read X, Y or $\square$, move L and change to the state that *checks* if the symbol is a 0 (resp. 1);
3. If when checking that the symbol is a 0 (resp. 1) we read 0 (resp. 1), mark with X (resp. Y), and move L.
   Otherwise, halt;
4. If after moving L we read X or Y, then the word is of the form ww^r and we accept.
   If we instead read 0 or 1, then move L and change to the state that *goes* to the first unmark symbol;
5. When going to the first unmark symbol, move L over 0's and 1's;
   If when going to the first unmark symbol, we read X or Y, move R and change to the initial state to repeat the procedure with the rest of the input.

Transition Diagram of a TM for $\{ww^r \mid w \in \{0,1\}^*\}$

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



Coding the Natural Numbers

Unary Coding The number 0 is represented by the empty symbol $\square$ and a number $n \neq 0$ is represented with n consecutive 1's.
The number 5 is then represented as

$\cdots$		1	1	1	1	1		$\cdots$
		$\uparrow$						

Kleene's Coding The natural number n is represented with $n+1$ consecutive 1's.
The number 5 is then represented as

$\cdots$		1	1	1	1	1	1		$\cdots$
		$\uparrow$							

Examples

How can we write a TM that compute the following functions over the Natural numbers:

1. Successor and predecessor;
2. Addition and subtraction;
3. Multiplication.

Closure Properties

Recursive languages are closed under union, intersection and complement.

Recursive enumerable languages are closed under union and intersection, but not complement.

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.

Variants of Turing Machines

What follows are some variants, extensions and restrictions to the notion of TM that we presented, none of them modifying the power of the TM.

- Storage in the state;
- Multiple tracks in one tape;
- Subroutines;
- Multiple tapes;
- Non-deterministic TM;
- Semi-infinite tapes.

FA vs. PDA vs. TM

FA: Bounded input and finite set of states;
Can only read and move to the right;
After reading the word we decide whether to accept or not;

PDA: Bounded input and finite set of states;
Can only read and move to the right;
Stack with unbounded memory with LIFO access (last in-first out);
After reading the word we decide whether to accept or not;

TM: Unbound input and finite set of states;
Random access in unbounded memory;
If the TM is in a final state when there are no more movements, then the input is accepted.

Differences between FA vs. TM

- TM can read and write the input;
- TM have an infinite tape (on one or both directions);
- TM can move to the right and to the left;
- TM have a special states for accepting (and rejecting) independent of the content of the tape;
- TM can loop or get stack.

Hierarchy of Languages

RL: Accepted by a FA or generated by a RE;
Example: 0^n;

CFL: Accepted by a PDA or generated by a CFG;
Example: 0^n1^n;

Recursive: Accepted by a Turing decider;
Example: $0^n1^n2^n$; graph reachability (given two nodes in a graph, is there a path between them?);

Recursive enumerable: Accepted by a Turing machine;
Example: Halting problem, Hilbert tenth problem (given a multivariate polynomial equation, does it have an integer solution?).

Note: There is a proper inclusion between these languages!

Overview of the Course

We have covered chapters 1–5 + 7 + (8):

Formal proofs: mainly proofs by induction;

Regular languages: DFA, NFA, ϵ-NFA, RE;
Algorithms to transform one formalism to the other;
Pumping lemma for RL;
Closure and decision properties of RL;

Context-free languages: CFG;
Pumping lemma for CFL;
Closure and decision properties of CFL;

Turing machines: Just a bit.

Formal Proofs

We have used formal proofs along the course to prove our results.

Mainly proofs by induction:

- By induction on the structure of the input argument;
- By induction on the length of the input string;
- By induction on the length of the derivation;
- By induction on the height of a parse tree.

Finite Automata and Regular Expressions

FA and RE can be used to model and understand a certain situation/problem.

Example: Consider the problem with the man, the wolf, the goat and the cabbage.

Also the Gilbreath's principle. There we went from NFA $\rightarrow$ DFA $\rightarrow$ RE.

They can also be used to describe (parts of) a certain language.

Example: RE are used to specify and document the lexical analyser (*lexer*) in languages (the part of the compiler reading the input and producing the different *tokens*).

The implementation performs the steps RE $\rightarrow$ NFA $\rightarrow$ DFA $\rightarrow$ min DFA.

Example: Using Regular Expression to Identify the Tokens

```
Tokens = Space (Token Space)*
Token  = TInt | TId | TKey | TSpec
TInt   = Digit Digit*
Digit  = '0' | '1' | '2' | '3' | '4' | '5' | '6' |
         '7' | '8' | '9'
TId    = Letter IdChar*
Letter = 'A' | ... | 'Z' | 'a' | ... | 'z'
IdChar = Letter | Digit
TKey   = 'i''f' | 'e''l''s''e' | ...
TSpec  = '+''+' | '+' | ...
Space  = (' ' | '\n' | '\t')*
```

Regular Languages

Intuitively, a language is regular when a machine needs only limited amount of memory to recognise it.

We can use the Pumping lemma for RL to show that a certain language is not regular.

There are many decision properties we can answer for RL.
Some of them are:

$$\mathcal{L} \neq \emptyset? \qquad w \in \mathcal{L}? \qquad \mathcal{L} \subseteq \mathcal{L}'? \qquad \mathcal{L} = \mathcal{L}'?$$

Context-free Grammars

CFG play an important role in the description and design of programming languages and compilers.

CFG are used to define the syntax of most programming languages.

Parse trees reflect the structure of the word.

In a compiler, the parser takes the input into its abstract syntax tree (which also reflects the structure of the word but abstracts from some concrete features).

A grammar is ambiguous if a word in the language has more than one parse tree.

Context-free Languages

These languages are generated by CFG.

It is enough to provide a stack to a ϵ-NFA in order to recognise these languages.

We can use the Pumping lemma for CFL to show that a certain language is not context-free.

There are only a couple of decision properties we can answer for CFL. Mainly:

$$\mathcal{L} \neq \emptyset? \qquad w \in \mathcal{L}?$$

However there are no algorithms to determine whether $\mathcal{L} \subseteq \mathcal{L}'$ or $\mathcal{L} = \mathcal{L}'$.

There is no algorithm either to decide if a grammar is ambiguous or a language is inherently ambiguous.

Turing Machines

Simple but powerful devices.

They can be thought of as a DFA plus a tape which we can read and write.

Define the recursively enumerated languages.

It allows the study of *decidability*: what can or cannot be done by a computer (halting problem).

Computability vs *complexity* theory: we should distinguish between what can or cannot be done by a computer, and the inherent difficulty of the problem (*tractable* (polynomial)/*intractable* (NP-hard) problems).

Learning Outcome of the Course

- Explain and manipulate the different concepts in automata theory and formal languages;
- Have a clear understanding about the equivalence between (non-)deterministic finite automata and regular expressions;
- Acquire a good understanding of the power and the limitations of regular languages and context-free languages;
- Prove properties of languages, grammars and automata with rigorously formal mathematical methods;
- Design automata, regular expressions and context-free grammars accepting or generating a certain language;
- Describe the language accepted by an automata or generated by a regular expression or a context-free grammar;
- Simplify automata and context-free grammars;
- Determine if a certain word belongs to a language;
- Define Turing machines performing simple tasks;
- Differentiate and manipulate formal descriptions of languages, automata and grammars.