# CS21 — Decidability and Tractability
Umans

Yubo Su

# Contents

**1 Key Concepts** **2**

**2 05/01/15 — Introduction** **3**
2.1 What are problems/computations? . . . . . 3

**3 07/01/15 — Finite Automata** **5**
3.1 Characterising FA . . . . . 5
3.2 Nondeterministic FA . . . . . 6
3.3 NFA-FA equivalence . . . . . 6

**4 09/01/15 — Regular Expressions** **7**

**5 12/01/15 — Pumping Lemma** **8**
5.1 Pumping Lemma . . . . . 8
5.2 Pushdown automata . . . . . 9

**6 14/01/15 — Context-free grammar** **10**

**7 16/01/15 — Wworking with CFGs** **12**
7.1 Ambiguity and CFGs, Chomsky Normal Form . . . . . 12
7.2 Properties of CFGs . . . . . 13
7.3 NPDA and CFG Equivalence . . . . . 13
7.4 Non context-free languages . . . . . 13

# Chapter 1

# Key Concepts

- A language $L$ is regular = $L$ is recognized by an FA = $L$ is described by a regular expression.
- Pumping lemma $w = xyz \in L \to xy^i z \in L$? always holds for regular languages.

# Chapter 2

# 05/01/15 — Introduction

Website is `http://www.cs.caltech.edu/~umans/cs21`.

This course forms an introduction to the *theory of computation.* Computability/complexibility sits atop algorithms on the heirarchy of computer science, algorithms which sit on top of software design/implementation. To speak about algorithms, the heart of programs, we must be able to talk *mathematically* about problems, computers and algorithms.

Imagine a perfect world, where every problem would have a corresponding provably optimal algorithm. Then any problem would become a simple lookup problem, and CS would be very dull. Thankfully, that's not our world! In fact, we can prove that problems exist without algorithmic solutions, such as the halting problem. Even more interestingly, for many simple problems with algorithmic solutions we know nothing about their optimality, such as multiplying two integers or factoring integers.

Part one of the class comprises models of computation and characterizations of their solvability; part two discusses Turing machines, limits on their power and reductions between problems; and part three talks about complexity classes P and NP. Main points of the class comprise decidability (whether algorithms exist) and tractability (whether *efficient* algorithms exist).

## 2.1 What are problems/computations?

We define a problem as a *function that maps an input to an output, where the input and output are strings over a finite alphabet* $\Sigma$, or $f : \Sigma** \to \Sigma**$. Let's make our problems simpler and examine only *decision problems*, which map an input to an *accept* or *reject* problem. It turns out that all problems are equivalent to decision problems; we prove this on the homework. If we are examining only decision problems, we can consider a *language* $L$ for $L \subseteq \Sigma^*$ such that all of $L$ maps to *accept* under $f$.

A computation is then taking an input and mapping it either to *accept* or *reject* or, crucially, *looping forever.* We say that the language *recognized* by the machine is the set of inputs that map to *accept*, and if every other input leads to reject we say that the language is *decided* by the machine. This should make sense; recognize means we can only *accept* if accept, but *decide* means we can definitively reject if reject.

We consider a finite automaton, which reads input only sequentially and maintains a finitely large state (memory of what has been seen). We can represent these by diagrams such as

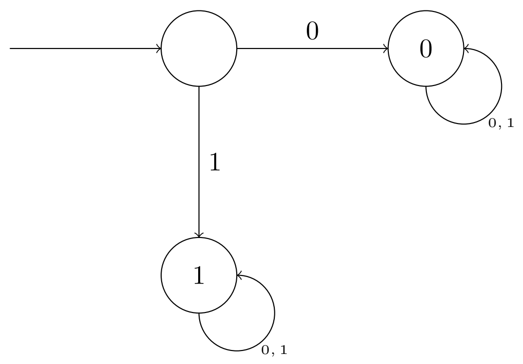

**Figure 2.1:** An example of a finite automaton; its language is the set of strings starting with a 1.

# Chapter 3

# 07/01/15 — Finite Automata

Recall that a machine accepts an input in a finite alphabet $\Sigma$. A subset of strings over $\Sigma$ is called the language *recognized* or *decided* (recall the difference) of the machine if the machine accepts this subset of strings. Recall also that a finite automaton reads left to right, one symbol at a time, and maintains a finite state.

## 3.1 Characterising FA

We hereby formally define a finite automaton. An FA is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$ such that

- $Q$ is a finite set called the *states*.
- $\Sigma$ is a finite set called the *alphabet*.
- $\delta : Q \times \Sigma \to Q$ is a function called the *transition function*.
- $q_0$ is an element of $Q$ called the *start state*.
- $F$ is a subset of $Q$ called the *accept states*.

The FA's operation can then be described as $M = (Q, \Sigma, \delta, q_0, F)$ accepting a string $w = w_1 w_2 \dots w_n \in \Sigma^*$ if $\exists$ a sequence $\{r_i\}$ of states such that

- $r_0 = q_0$
- $\delta(r_i, w_{i+1}) = r_{i+1}$
- $r_n \in F$

We note that the set of languages recognized by an FA is closed under union, concatenation and star. Star is the operation defined on $A$ by

$$A^* = \{x_1 x_2 \dots x_k, x_i \in A\} \tag{3.1}$$

## 3.2 Nondeterministic FA

A deterministic FA updates a single state for every character of input read. A non-deterministic FA labels some transitions with $\varepsilon$ such that the next state going over one of these transitions is an ensemble of two or more possible states. An input is then accepted if the final ensemble of states contains an accept, or equivalently if there exists a set of not-necessarily-deterministic transitions that takes the initial state to the accept under the input.

Formally, a nondeterministic FA is the same 5-tuple as an FA, but the $\delta$ transition function changes. It is a mapping $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \wp(Q)$ called the "powerset of $Q$," the set of all subsets of $Q$.

## 3.3 NFA-FA equivalence

We claim that a language $L$ is recognized by an FA if and only if $L$ is recognized by an NFA. This entails a proof in both directions. One is trivial; every language recognized by an FA is also recognized by an NFA because FA are NFA without $\varepsilon$-transitions. The other direction proves a bit more tricky.

We strive to build an FA that simulates an NFA (and recognizes the same language). Given an NFA $M = (Q, \Sigma, \delta, q_0, F)$, we construct an FA $M' = (Q', \Sigma, \delta', q_0', F')$ with $Q' = \wp(Q)$ the powerset. Then if we are in a state $R \in Q'$ and we read symbol $a \in \Sigma$, what is the new state?

First we define $E(S) = \{q \in Q\}$ such that $q$ is reachable from $S$ by traveling along 0 or more $\varepsilon$-transitions. Then we define a new transition function $\delta'(R, a) = \displaystyle\bigcup_{r \in R} E(\delta(r, a))$. We also use start state $q_0' = E(q_0)$. This new FA then accepts the same states as does $M$ by construction, and we complete our equivalence proof.

# Chapter 4

# 09/01/15 — Regular Expressions

Currently we have two big theorems regarding finite automata and their languages

- The two sets of languages recognized by NFAs and FAs respectively are closed under union, concatenation and star.
- A language $L$ is recognized by an FA iff L is recognized by an NFA.

We next want to describe the set of languages that can be built up from union, concatenation and stars; these are called "patterns" or *regular expressions.* A theorem goes then that

A language $L$ is recognized by an FA iff $L$ is described by a regular expression.

First, we must define a regular expression. A regular expression is either an element of the alphabet, the empty string, the empty set or a combination of these aforementioned regular expressions by union, concatenation and star. We will then omit the proof because I am lazy.

Thus then a language recognized by an FA is called a *regular language.* These thus have three properties

- Closed under the 3 operations.
- Recognized exactly by NFA/FA.
- Described exactly by regular expressions.

# Chapter 5

# 12/01/15 — Pumping Lemma

We want now to examine limits on the power of FAs; is every language describable by a sufficiently complex regular expression? For example, is the language

$$\{w : w\text{has an equal number of “01” and “10” substrings}\}$$

At first we suspect that FAs cannot count, so there is a good probability that we cannot do this. However, it turns out we can write it as $= 0\Sigma^*0 \cup 1\Sigma^*1$, because the fundamental equality of number of substrings doesn't require counting. The point stands though; how do we prove a language is not regular?

## 5.1 Pumping Lemma

Enter the pumping lemma, applying for *all* regular languages

Let $L$ be a regular language. There exists an integer $p$ (“pumping length”) for which every $w \in L$ with $|w| \geq p$ can be written as $w = xyz$ such that

(1) For every $i \geq 0, xy^iz \in L$

(2) $|y| > 0$

(3) $|xy| \leq p$

Using the pumping lemma to prove that $L$ is not regular is done by first assuming $L$ is regular. This implies a pumping length $p$ exists. We then select a string $w \in L$ of length at least $p$ and then argue that for every way of writing $w = xyz$ that satisfies (2), (3) of the lemma, pumping on $y$ yields a string not in $L$ which provides a contradiction.

As an example, prove $L = 0^n1^n$ is not regular. Let $p$ be the pumping length for $L$, so we choose $w = 0^p1^p$. We then examine all the ways we can write $w = xyz$ with $|y| > 0$ and $|xy| \leq p$. As one example, choose $x, y$ both still in the 0s (first half) of $w$, then repeating $y$ produces a string with more 0s than 1s which is no longer in $L$, so $L$ is not regular. A similar language provably irregular is $L = \{w : \text{equal number of 0s, 1s}\}$, for which we can use the same $w$ in fact.

Let's prove the pumping Lemma. Let $M$ be the machine that recognizes $L$, and call $p$ the number of states of $M$. Consider $w \in L, |w| \geq p$. Then $M$ must go through at least $p + 1$ states, which means there must be a repeated state. Given that this state is repeated, we can takee the segment of $w$ between this repeat and cycle it; this string is still recognized by $M$. Thus $xy^iz \in L$.

## 5.2 Pushdown automata

We can see an FA as an input tape and a finite control $q_0$ that is changed as it reads over the input. A limitation of FA is because it can only "remember" a bounded amount of information. To recognize a language such as $0^n 1^n$ though, we need memory that can *count*. We seek a simple modification that unbounds the accessible memory; this is accomplished precisely by introducing a stack of memory. This is called a *pushdown automaton*, one that in addition to the control $q_0$ can also push nd pop memory onto an infinite stack.

To represent the nondeterministic pushdown automaton (NPDA), we introduce a dollar sign \$ symbol that can be pushed and popped off the stack; notation such as $\varepsilon \to \$$ means to nondeterministically push a \$ to the stack, while something like $0 \to \varepsilon$ means to pop. We then exhibit an NPDA that accepts exactly input strings of form $0^n 1^n$ by pushing 0s and accepting 1s only while 0s can be popped, refer to lecture slides.

The formal definition of an NPDA is given by the tuple

- $Q$ is a finite set called the *states*.
- $\Sigma$ is a finite set called the *alphabet*.
- $\Gamma$ is a finite set called the *stack alphabet*.
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \to \wp(Q \times (\Gamma \cup \{\varepsilon\}))$ is a function called the *transition function*.
- $q_0$ is an element of $Q$ called the *start state*.
- $F$ is a subset of $Q$ called the *accept states*.

We omit the formal definition of acceptance because I'm lazy; it's in Lecture 4.

## Chapter 6

# 14/01/15 — Context-free grammar

Let's begin with an exercise; design an NPDA to recognize the language

$$\left\{a^i b^j c^k; i, j, k \geq 0 \text{ and } i = j \text{ or } i = k\right\}$$

We already know how to check whether $i = j$ *or* $i = k$, so we can just add an epsilon transition out front of two machines, one to recognize each of $i = j, i = k$.

Just as we called the languages recognized by NFAs *regular*, we call the languages recognized by NPDAs *context-free grammars.* What then is a context-free grammar? Define a non-terminal symbol to be simply a symbol that cannot exist in the final output string, and define a set of substitution rules of non-terminal symbols to terminal and non-terminal symbols (the left-hand-side of these rules must be a single non-terminal symbol). We follow the steps

- Start with a start symbol (non-terminal)
- Replace a non-terminal with the right-hand-side of a rule that has non-terminal as its left-hand-side
- Repeat until there are no more non-terminals.

For example, we can build a context-free grammar with the following rules $A \to 0A1, A \to B, B \to \#$. We then start with $A$ and can eventually build the output string such as $000\#111$, a string of terminal symbols that is in the language of the context-free grammar. The set of all strings that can be generated this way is the *language of the grammar.*

The official definition of a context-free grammar is a 4-tuple

- $V$ is a finite set called the *non-terminals.*
- $\Sigma$ is a finite set called the *terminals.*
- $R$ is a set of substitution rules from non-terminals to terminals/non-terminals called *productions.*
- $S \in V$ is the single *start variable.*

The notation goes then that $uAv$ *yields* $uwv$, or notated $u \Rightarrow v$. If it is a multiple step process, $u \Rightarrow^k v$, and if $u \Rightarrow^* v$ then we say the particular procedure is called a *derivation* of $v$.

Let's do an example of a CFG, balanced parentheses. We first specify that *balancing* means that number of open = number of closed, but also that no parentheses is ended before it is begun. Let's try to build a CFG for this. We use the rules

- $A \to (A)|AA$
- $A \to \epsilon$

We next need to show that this set of rules generates the context-free grammar bidirectionally. We first work that every string derived by this grammar is balanced; this is easy to do by induction on the number of steps of the derivation.

Suppose $A \Rightarrow^k s$, and we want to prove that $s$ is balanced. We induct over $k$; for $k = 1$, we end up with only the empty string, which is balanced. We then induct on $k$; the two cases we want to examine are

- $A \Rightarrow (A) \Rightarrow^{k-1} (s') = s$ where we define $s'$. But by the strong inductive hypothesis $s'$ is balanced, and so must then be $s$.
- $A \Rightarrow AA \Rightarrow^{k-1} s's'' = s$, where each of the $A$s derives one of $s', s''$. By the strong inductive hypothesis these are balanced and thus so must be $s$.

We then want to prove the other direction, that all balanced parentheses are derived by this context-free grammar. We will instead induct on the length of the string. Base case is length 0, which we can derive with our CFG. The inductive step comprises showing a string of length $n$ is derivable; we begin by identifying the shortest prefix of the string that is balanced. There are again two cases

- This comprises the whole string — $s = (s')$. We assume that $A$ derives $s'$, which is of length shorter than $n$, and so since we can do $A \Rightarrow (A) \Rightarrow^* (s')$ we show that $A$ also derives $s$.
- This does not comprise the whole string — $s = s's''$. We assume again that $A$ derives $s', s''$ which are of length shorter than $n$, and since we can do $A \Rightarrow AA \Rightarrow^* s's''$ we show that $A$ also derives $s$.

QED!

# Chapter 7

# 16/01/15 — Wworking with CFGs

## 7.1 Ambiguity and CFGs, Chomsky Normal Form

Recall the definition of a CFG. An example of a CFG is the set of arithmetic expressions, over the alphabet $\{+, *, (, ), a\}$. Let's describe this via a CFG. We exhibit the production rules $S \to (S)|S + S|S * S|a$, which seems to work, yay!

An easy way to visualise CFGs is to see a parse tree, where expressions expand down a tree into a parse tree. In this particular case, this isn't the best CFG, if we examine the corresponding parse tree. Order of operations isn't necessarily respected in going up our parse tree. Instead, we can make a much more complicated CFG, given by

- $E \to E + T|T$
- $T - -ET * F|F$
- $F \to T * F$
- $F \to (E)|a$

with $E, T, F$ representing expressions, terms, and factors respectively. This way, the parse tree is completely unambiguous; there is always exactly a single *left-most derivation* (where we always replace the left-most non-terminal first at any given step). We can sometimes rewrite a CFG to have a non-ambiguous parse tree, but some CFGs cannot be dis-ambigu-ified.

It would be helpful if we could discuss all CFGs in a simple normal form; the most common one we use is the *Chomsky Normal Form.* This denotes that all production rules are of form $A \to BC, S \to \varepsilon, A \to a$, so no $\varepsilon$ are produced except for the starting non-terminal, and RHS's contain either one terminal or two non-terminals. We present an algorithm to turn any CFG into CNF

- Add a new start symbol — Add production $S_0 \to S$.
- Remove $\varepsilon$-productions — Say that $A \to \varepsilon$ is something we removed, then whenever $A$ is on RHS, we just delete it e.g we take $R \to uAV \Rightarrow R \to uv$.
- Eliminate unit productions — $A \to B$ with two non-terminals is illegal-ified. If we have a further rule $B \to u$ then simply $A \to u$.

- Convert remaining rules into proper form — Replace productions of form $A \to u_1 U_2 u_3 \ldots u_k$ to $A \to U_1 A_1, A_1 \to U_2 A_2 \ldots$ and then $U_{k-1} \to u_{k-1}, U_k \to u_k$ edge cases.

## 7.2 Properties of CFGs

Now that we have Chomsky Normal Form, we can try to prove some more properties about CFGs.

- Union — We introduce a new start symbol $S_0 \to S_1 | S_2$.
- Concatnation — We need a new start symbol $S_0 \to S_1 S_2$ the start symbols of the two constituent CFGs.
- Star — We introduce the rule $S_0 \to \varepsilon | S | S_0 S_0$.

We then want to show that every regular language is a CFL. Recall the definition of the regular languages, the union, concatnation, and star of $\varepsilon, a, \emptyset$. Since CFGs are closed as well, we simply need a start symbol that can map to these three and then by closure by the above properties we are equivalent to a regular language.

## 7.3 NPDA and CFG Equivalence

This is a very important part of this lecture. We first prove that $L$ described by a CFG implies $L$ is recognized by an NPDA. A slight trickiness is in that we can only access the top of the stack. The rough algorithm we will use for this proof is

- Place \$ then a start symbol $S$ on the stack. Repeat then the following
- If the top is a non-terminal, pick productions with $A$ on LHS and substitute onto the stack non-deterministically.
- If it is a terminal, read from the tape and pop from the stack.
- If at the end of the tape a \$ is on the stack, then we accept.

We can draw the NPDA for this but I'm too incompetend to do so, refer to lecture6 for this if you must.

We now examine the reverse direction, that a language accepted by an NPDA is described by a CFG. First, let's convert the NPDA to a normal form as well, so that

- Single accept states.
- Empties stack before accepting.
- Each transition either pushes or pops a symbol.

The main idea of the proof then hinges on the fact that a non-terminal $A_{p,q}$ generates exactly the strings that take the NPDA from state $p$ to state $q$, both with empty stacks. If we can get this to work, then the existence of $A_{start,end}$ generates all of the strings in the language recognized by the NPDA. Then a string taking the NPDA from $p$ to $q$ either empties the stack in between at some point $r$ (i.e. $A_{p,q} = A_{p,r} + A_{r,q}$ in some sense) or it never empties the stack. So then it becomes clear we want to examine the latter case; we must define our CFG now.

- Non-terminals $V = \{A_{p,q}\}$.
- Start variable $A_{start,accept}$.
- Production rule $A_{p,q} \to A_{p,r}A_{r,q}$.

Now let's examine that latter case; clearly the first and last operations push and pop the same string. Thus, the first and last intermediate states (call them $A_{r,s}$) between $p, q$ must also have a start symbol that takes from empty stack to empty stack. We are out of time, but the approach should already be clear; we add another production rule $A_{p,q} \to aA_{r,s}b, a, b \in \Sigma^*$ with $a, b$ on the input tape. We'll flesh this out formally next class.