Lecture notes of

CS273: Introduction to the Theory of Computation

Spring Semester, 2008

and CS373: Theory of Computation

Spring Semester, 2009

CS, UIUC

Margaret Fleck Sariel Har-Peled[1]

May 18, 2009

[1] Department of Computer Science; University of Illinois; 201 N. Goodwin Avenue; Urbana, IL, 61801, USA; sariel@uiuc.edu; http://www.uiuc.edu/~sariel/.

Contents

Preface – Spring 2009

> Finally: It was stated at the outset, that this system would not be here, and at once, perfected. You cannot but plainly see that I have kept my word. But I now leave my cetological System standing thus unfinished, even as the great Cathedral of Cologne was left, with the crane still standing upon the top of the uncompleted tower. For small erections may be finished by their first architects; grand ones, true ones, ever leave the copestone to posterity. God keep me from ever completing anything. This whole book is but a draft - nay, but the draft of a draft. Oh, Time, Strength, Cash, and Patience!
>
> – Moby Dick, Herman Melville

This manuscript is a collection of class notes used in teaching CS 373 (Theory of Computation), in the spring of 2009, in the Computer Science department in UIUC. The instructors were Sariel Har-Peled and Madhusudan Parthasarathy. They are based on older class notes – see second preface for details.

This class notes diverse from previous semesters in two main points:

(A) **Regular languages pumping lemma**. Although we still taught the pumping lemma for regular languages, we did not expected the students to use it to proving languages are not regular. Instead, we provided direct proofs that shoes that any automaton for these languages would require infinite number of states. This leads to much simpler proofs than using the pumping lemma, and it seems the students find them easier to understand. Naturally, we are not the first to come up with this idea, it is sometimes referred to as the "technique of many states".

The main problem with the pumping lemma is the large number of quantifiers involved in stating it. They seem to make it harder for the student to use it.

(B) **Recursive automatas.** Instead of teaching PDAs, we used an alternative machine model of ***Recursive automata*** (RA) for context-free languages. RAs are PDAs that do not manipulate the stack directly, but only through the calling stack. For a discussion of this issue, see Chapter 20 (page 127).

This lead to various changes later in the course. In particular, the fact that the intersect of CFL language and a regular language is still CFL, is proven directly on the grammar. Similarly, the proof that deciding if a grammar generates all words is undecidable now follows by a simpler but different proof, see relevant portion for details.

In particular, the alternative proof uses the fact that given two configurations of a TM written on top of each other, then a DFA can verify that the top configuration yields the bottom configuration. This is a cute observation that seems to be worthy of describing in class, and it leads naturally into the Cool-Levin theorem proof.

What remains to be done. Students suggested that more examples would be useful to have in the class notes. In future instances in the class it would be a good idea to allocate two lectures towards the end to teach the Cook-Levin Theorem properly. A more algorithmic emphasize might be a good preparation for later courses.

And of course, no class notes are prefect. These class notes can definitely be further improved.

Format. Every chapter corresponds to material covered in one lecture in the course. Every week we also had a discussion section run by the TAs. The TAs also wrote (most of) the notes included for the discussion section.

Acknowledgements

The chapters on recursive automata and the review chapter was written by Madhusudan Parthasarathy.

The TAs in the class (Reza Zamani, Aparna Sundar, and Micha Hadosh) provided considerable help in writing the exercises, and their solutions, and we thank them for their valuable work.

For further acknowledgements, see the older preface.

Getting the source for this work

These class notes would ultimately be available online. Stay tuned.

Copyright

Sariel Har-Peled
May 2009, Urbana, IL. USA.

Preface - Spring 2008

This manuscript is a collection of class notes used in teaching CS 273 (Introduction to The Theory of Computation), in the spring of 2008, in the Computer Science department in UIUC. The instructors were Margaret Fleck and Sariel Har-Peled.

These class notes are an initial effort to have class notes covering the material taught in this class, and is largely based on hand written class notes from previous semesters, and the book used in the class (Sipser [Sip05]).

Quality. We do not consider these class notes to be perfect, and in fact, far from it. However, it is our hope that people would improve these class notes in proceedings semesters, and bring them into acceptable quality. From previous experience, it takes 3–4 iterations before the class notes reach acceptable quality. Even getting the class notes to their current form required non-trivial amount of work.

Format. Every chapter corresponds to material covered in one lecture in the course. Every week we also had a discussion section run by the TAs. The TAs also wrote (most of) the notes included for the discussion section.

Why? We have no complaints about the book, but rather we prefer the form of class notes over the form a book. Writing a class notes is also an effective (if somewhat time consuming) way to prepare for lecture. And is usual at some points we preferred to present some material in our own way.

Ultimately, we hope that after several semesters of polishing these class notes they would be good enough to replace the required text book in the class.

Acknowledgements

We had the benefit of interacting with several people on the work in this class notes. Other instructors that taught this class and had contributed (directly or indirectly) to the material covered in this class are Chandra Chekuri Madhusudan Parthasarathy, Lenny Pitt, and Mahesh Viswanathan.

In addition, the TAs in the class (Reza Zamani, James Lai, and Raman Sharykin) provided considerable help in writing the exercises, and their solutions, and we thank them for their valuable work.

We would also like to thank the students in the class for their input, which helped in discovering numerous typos and errors in the manuscript.

Getting the source for this work

These class notes would ultimately be available online. Stay tuned.

Margaret Fleck and Sariel Har-Peled
May 2008, Urbana, IL. USA.

Part I

Lectures

Chapter 1

Lecture 1: Overview and Administrivia

20 January 2009

1.1 Course overview

The details vary from term to term. This is a rough outline of the course and a motivation for why we study this stuff.

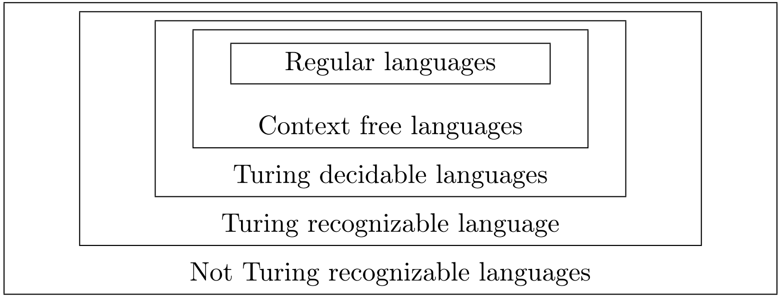
1. Theory of Computation.
   - Build formal mathematical models of computation.
   - Analyze the inherent capabilities and limitations of these models.
2. Course goals:
   - Simple practical tools you can use in later courses, projects, etc. The course will provide you with tools to model complicated systems and analyze them.
   - Inherent limits of computers: problems that no computer can solve.
   - Better fluency with formal mathematics (closely related to skill at debugging programs).
3. What is computable?
   (a) check if a number n is prime
   (b) compute the product of two numbers
   (c) sort a list of numbers
   (d) find the maximum number from a list
4. Computability, complexity, automata.
5. Example:

```
input n;
assume n>1;
while (n !=1) {
    if (n is even)
        n := n/2;
    else
        n := 3*n+1;
}
```

Does this program always stop? Not known.

6. Course divides into three sections
   - regular languages ———— practical tools
   - context-free languages
   - Turing machines.

7. Turing machines and decidability → limits of computation.


Regular languages
Context free languages
Turing decidable languages
Turing recognizable language
Not Turing recognizable languages

8. Regular languages and context-free languages:
   - Simple and computationally efficient.
   - heavily used in programming languages and compilers.
   - also in computational linguistics.
   - well-known application: `grep`

9. Difference in scope
   - regular languages describe tokens (e.g. what is a legal variable name?)
   - context-free languages describe syntax (whole program, whole sentence)

   Illustrate with your favorite example from programming languages or natural language.

10. State machines
    - widely used in other areas of CS (e.g. networking)
    - equivalent to regular languages (we will see later in course)
    - used to implement algorithms for regular languages e.g. `grep`.

    Illustrate with your favorite simple state machine, e.g. a vending machine.

11. Decidability:
    - Are there problems that computers can not solve? ⇒ yes!
    - By the end of the course, you will know why.
      Example: the CS 225 grader problem.
      - Given a random `C` program (maybe very badly written).
      - Will it stop or will it keep running forever?
      - Will it return the right answer for all possible inputs?

12. Models of mathematics
    - 19th century - major effort to formalize calculus.
    - Georg Cantor - starts set theory. Proves that the "number" of integers is strictly smaller than the number of integers, using the diagonalization argument.

- David Hilbert (1920's) tries to formalize all of math and prove it correct
- Kurt Gödel (1931) shows that one can not prove consistency of a mathematical formalism having non-trivial power.

13. Formal models of computation
    - Alonzo Church: lambda calculus (like LISP).
    - Alan Turing: Turing machines (very simple computers).
    - Church/Turing thesis: these models can do anything that any computer can do.
    - Both showed (1936) that their respective models contain undecidable problems.
14. It is mysterious and "cool" that some simple-looking problems are undecidable.
15. The proofs of undecidability are a bit abstract. Earlier parts of the course will help prepare you, so you can understand the last part.

1.2 Necessary Administrivia

This lecture mentions the highlights. Browse the web page for more details:

`http://www.cs.uiuc.edu/class/sp09/cs373/{}`.

- Prerequisites: CS 125, CS 173, CS 225 (or equivalents). Other experience can sometimes substitute (e.g. advanced math). Speak to us if you are not sure.
- Vital to join the class newsgroup (details on web page). Carries important announcements, e.g. exam times, hints and corrections on homeworks.

  Especially see the Lectures page for schedule of topics, readings, quiz/exam dates.
- Homework 1 should be available on the class website. Due next Thursday. (Normally they will be due Thursdays on 12:30, but next Monday is a holiday.) Browse chapter 0 and read section 1.1. Normally, homeworks and readings will not be announced in class and you must watch the website and newsgroups.
- Read and follow the homework format guidelines on the web page. Especially: each problem on a separate sheet, your name on each problem, your section time (e.g. 10) in upper-right corner. This makes a big difference grading and sorting graded homeworks.
- Course staff.
- Discussion sections. Office hours will be posted in the near future. Email and the newsgroup are always an option. Please do not be shy about contacting us.
- Problem sets, exams, etc are common to all sections. It may be easier to start with your lecture and discussion section instructors, but feel free to also talk to the rest of us.
- Sipser textbook: get a copy. We follow the textbook fairly closely. Our lecture notes only outline what was covered and don't duplicate the text. Used copies, international or first editions, etc are available cheap through Amazon.
- Graded work:

  (a) 30%: Final.

  (b) 20%: First midterm.

  (c) 20%: Second midterm.

(d) 25%: Homeworks and self-evaluations.

Worst homework will be dropped.

Self evaluations would be online quizes on the web.

(e) 5%: Attending discussion section.

- Late homeworks are not accepted, except in rare cases where you have a major excuse (e.g. serious illness, family emergency, weather unsafe for travel).
- Homeworks can be done in groups of ≤ 3 students. Write their names under your own on your homework. Also document any other major help you may have gotten. Each person turns in their own write-up **IN THEIR OWN WORDS**.
- Doing homeworks is vital preparation for success on the exams. Getting help from your partners is good, but don't copy their solutions blindly. Make sure you understand the solutions.
- See the web pages for details of our cheating policy. First offense → zero on the exam or assignment involved. Second offense or cheating on the final ⇒ fail the course. Please do not cheat.
- If you are not sure what is allowed, talk to us and/or document clearly what you did. That is enough to ensure it is not "cheating" (though you might lose points).
- Bugs happen, on homeworks and even in the textbook and on exams. If you think you see a bug, please bring it to our attention.
- Please tell us if you have any disabilities or other special circumstances that we should be aware of.

Chapter 2

Lecture 2: Strings, Languages, DFAs

17 January 2008

This lecture covers material on strings and languages from Sipser chapter 0. Also chapter 1 up to (but not including) the formal definition of computation (i.e. pages 31–40).

2.1 Alphabets, strings, and languages

2.1.1 Alphabets

An ***alphabet*** is any *finite* set of characters.

Here are some examples for such alphabets:

(i) $\{\texttt{0}, \texttt{1}\}$.

(ii) $\{\texttt{a}, \texttt{b}, \texttt{c}\}$.

(iii) $\{\texttt{0}, \texttt{1}, \#\}$.

(iv) $\{\texttt{a}, ...\texttt{z}, \texttt{A}, ...\texttt{Z}\}$: all the letters in the English language.

(v) ASCII - this is the standard encoding schemes used by computers mappings bytes (i.e., integers in the range 0..255) to characters. As such, `a` is 65, and the space character ␣ is 32.

(vi) $\{\texttt{moveforward}, \texttt{moveback}, \texttt{rotate90}, \texttt{reset}\}$.

2.1.2 Strings

This section should be recapping stuff already seen in discussion section 1.

A ***string*** over an alphabet Σ is a *finite* sequence of characters from Σ.

Some sample strings with alphabet (say) $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}\}$ are `abc`, `baba`, and `aaaabbbbccc`.

The ***length*** of a string x is the number of characters in x, and it is denoted by $|x|$. Thus, the length of the string $w = \texttt{abcdef}$ is $|w| = 6$.

The ***empty string*** is denoted by ϵ, and it (of course) has length 0. The empty string is the string containing zero characters in it.

The ***concatenation*** of two strings x and w is denoted by xw, and it is the string formed by the string x followed by the string w. As a concrete example, consider $x = \texttt{cat}$, $w = \texttt{nip}$ and the concatenated strings $xw = \texttt{catnip}$ and $wx = \texttt{nipcat}$.

Naturally, concatenating with the empty string results in no change in the string. Formally, for any string x, we have that $x\epsilon = x$. As such $\epsilon\epsilon = \epsilon$.

For a string w, the string x is a ***substring*** of w if the string x appears contiguously in w.

As such, for $w =$`abcdef`
we have that `bcd` is a substring of w,
but `ace` is not a substring of w.

A string x is a ***suffix*** of w if its a substring of w appearing in the end of w. Similarly, y is a ***prefix*** of w if y is a substring of w appearing in the beginning of w.

$$\begin{aligned} &\text{As such, for } w = \texttt{abcdef} \\ &\quad\text{we have that } \texttt{abc} \quad \text{is a prefix of } w, \\ &\qquad\qquad \text{and} \quad \texttt{def} \text{ is a suffix of } w. \end{aligned}$$

Here is a formal definition of prefix and substring.

Definition 2.1.1 The string x is a ***prefix*** of a string w, if there exists a string z, such that $w = xz$.
Similarly, x is a substring of w if there exist strings y and z such that $w = yxz$.

2.1.3 Languages

A ***language*** is a set of strings. One special language is Σ^*, which is the set of all possible strings generated over the alphabet Σ^*. For example, if

$$\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}\} \qquad \text{then} \qquad \Sigma^* = \{\epsilon, \texttt{a}, \texttt{b}, \texttt{c}, \texttt{aa}, \texttt{ab}, \texttt{ac}, \texttt{ba}, \ldots, \texttt{aaaaaabbbaababa}, \ldots\}.$$

Namely, Σ^* is the "full" language made of characters of Σ. Naturally, any language over Σ is going to be a subset of Σ^*.

Example 2.1.2 The following is a language

$$L = \{\texttt{b}, \texttt{ba}, \texttt{baa}, \texttt{baaa}, \texttt{baaaa}, ...\}.$$

Now, is the following a language?

$$\{\texttt{aa}, \texttt{ab}, \texttt{ba}, \epsilon\}.$$

Sure – it is not a very "interesting" language because its finite, but its definitely a language.
How about $\{\texttt{aa}, \texttt{ab}, \texttt{ba}, \emptyset\}$. Is this a language? No! Because $\emptyset$ is no a valid string (which comes to demonstrate that the empty word ϵ and $\emptyset$ are not the same creature, and they should be treated differently.

Lexicographic ordering of a set of strings is an ordering of strings that have shorter strings first, and sort the strings alphabetically within each length. Naturally, we assume that we have an order on the given alphabet.
Thus, for $\Sigma = \{\texttt{a}, \texttt{b}\}$, the Lexicographic ordering of Σ^* is

$$\epsilon, \texttt{a}, \texttt{b}, \texttt{aa}, \texttt{ab}, \texttt{ba}, \texttt{bb}, \texttt{aaa}, \texttt{aab}, \ldots.$$

Languages and set notation

Most of the time it would be more useful to use set notations to define a language; that is, define a language by the property the strings in this language posses.
For example, consider the following set of strings

$$L_1 = \left\{ x \;\middle|\; x \in \{\texttt{a}, \texttt{b}\}^* \text{ and } |x| \text{ is even } \right\}.$$

In words, L_1 is the language of all strings made out of $\texttt{a}, \texttt{b}$ that have even length.
Next, consider the following set

$$L_2 = \left\{ x \;\middle|\; \text{there is a } w \text{ such that } xw = \texttt{illinois} \right\}.$$

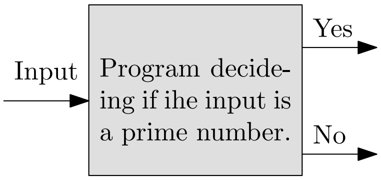
So L_2 is the language made out of all prefixes of L_2. We can write L_2 explicitly, but its tedious. Indeed,

$$L_2 = \{\epsilon, \texttt{i}, \texttt{il}, \texttt{ill}, \texttt{illi}, \texttt{illin}, \texttt{illino}, \texttt{illinoi}, \texttt{illinois}\}.$$

Why should we care about languages?

Consider the language L_{primes} that contains all strings over $\Sigma = \{0, 1, \ldots, 9\}$ which are prime numbers. If we can build a fast computer program (or an automata) that can tell us whether a string s (i.e., a number) is in L_{primes}, then we decide if a number is prime or not. And this is a very useful program to have, since most encryption schemes currently used by computers (i.e., RSA) rely on the ability to find very large prime numbers.

Let us state it explicitly: The ability to decide if a word is in a specific language (like L_{primes}) is equivalent to performing a computational task (which might be extremely non-trivial). You can think about this schematically, as a program that gets as input a number (i.e., string made out of digits), and decides if it is prime or not. If the input is a prime number, it outputs `Yes` and otherwise it outputs `No`. See figure on the right.



2.1.4 Strings and programs

An text file (i.e., source code of a program) is a long one dimensional string with special ⟨`NL`⟩ (i.e., newline) characters that instruct the computer how to display the file on the screen. That is, the special ⟨`NL`⟩ characters instruct the computer to start a new line. Thus, the text file

```
if x=y then
  jump up and down and scream.
```

Is in fact encoded on the computer as the string

`if␣x=y␣then`⟨`NL`⟩`␣␣jump␣up␣and␣down␣and␣scream.`

Here, ␣ denote the special space character and ⟨`NL`⟩ is the new-line character.

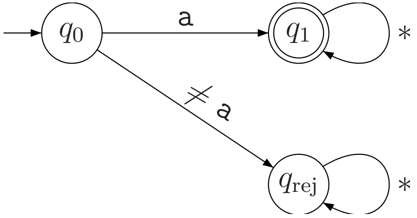
It would be sometime useful to use similar "complicated" encoding schemes, with sub-parts separated by # or $ rather than by ⟨`NL`⟩.

Program input and output can be consider to be files. So a standard program can be taught of as a function that maps strings to strings.[1] That is $P : \Sigma^* \to \Sigma^*$. Most machines in this class map input strings to two outputs: "yes" and "no". A few automatas and most real-world machines produce more complex output.

2.2 State machines

2.2.1 A simple automata

Here is a simple ***state machine*** (i.e., finite automaton) M that accepts all strings starting with `a`.



Here $*$ represents any possible character.

Notice key pieces of this machine: three states, q_0 is the start state (arrow coming in), q_1 is the final state (double circle), transition arcs.

To run the machine, we start at the start state. On each input character, we follow the corresponding arc. When we run out of input characters, we answer "yes" or "no", depending on whether we are in the final state.

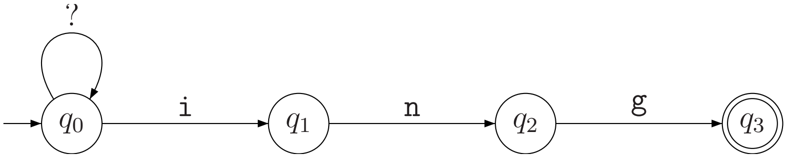
The language of a machine M is the set of strings it accepts, written $L(M)$. In this case $L(M) = \{\texttt{a}, \texttt{aa}, \texttt{ab}, \texttt{aaa}, \ldots\}$.

[1] Here, we are considering simple programs that just read some input, and print out output, without fancy windows and stuff like that.

2.2.2 Another automata

(This section is optional and can be skipped in the lecture.)

Here is a simple ***state machine*** (i.e., finite automaton) M that accepts all ASCII strings ending with `ing`.



Notice key pieces of this machine: four states, q_0 is the start state (arrow coming in), q_3 is the final state (double circle), transition arcs.

To run the machine, we start at the start state. On each input character, we follow the corresponding arc. When we run out of input characters, we answer "yes" or "no", depending on whether we are in the final state.

The language of a machine M is the set of strings it accepts, written $L(M)$. In this case $L(M) = \{\texttt{walking}, \texttt{flying}, \texttt{ing}, \ldots\}$.

2.2.3 What automatas are good for?

People use the technology of automatas in real-world applications:

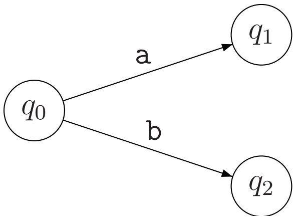
- Find all files containing `-ing` (grep)

- Translate each `-ing` into `-iG` (finite-state transducer)

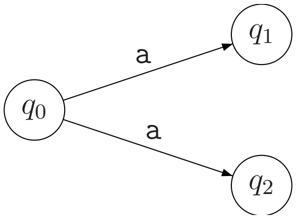
- How often do words in Chomsky's latest book end in `-ing`?

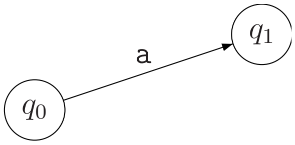
2.2.4 DFA - deterministic finite automata

We will start by studying ***deterministic finite automata*** (DFA). Each node in a deterministic machine has exactly one outgoing transition for each character in the alphabet. That is, if the alphabet is $\{\texttt{a}, \texttt{b}\}$, then all nodes need to look like

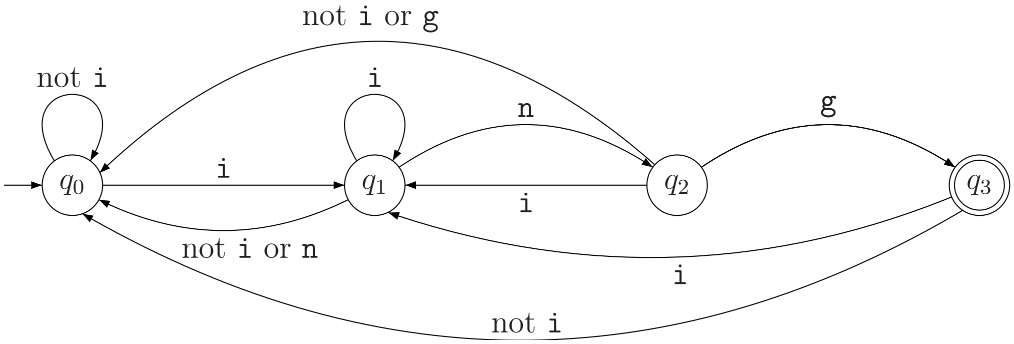


Both of the following are bad, where $q_1 \neq q_2$ and the right hand machine has no outgoing transition for the input character `b`.





So our `-ing` detector would be redrawn as:

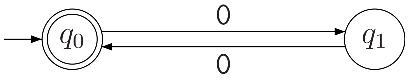


2.3 More examples of DFAs

2.3.1 Number of characters is even

Input: $\Sigma = \{0\}$.

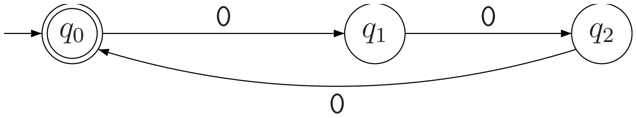
Accept: all strings in which the number of characters is even.



2.3.2 Number of characters is divisible by 3

Input: $\Sigma = \{0\}$.

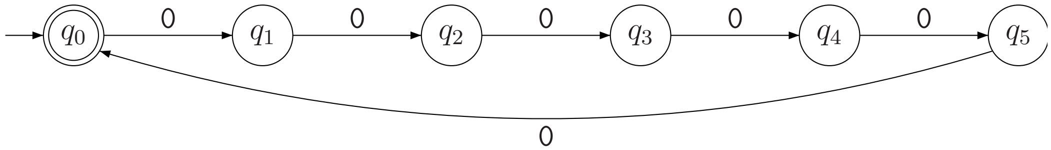
Accept: all strings in which the number of characters is divisible by 3.



2.3.3 Number of characters is divisible by 6

Input: $\Sigma = \{0\}$.

Accept: all strings in which the number of characters is divisible by 6.

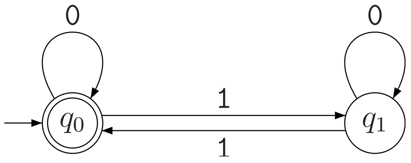


This example is especially interesting, because we can achieve the same purpose, by observing that $n \bmod 6 = 0$ if and only if $n \bmod 2 = 0$ and $n \bmod 3 = 0$ (i.e., to be divisible by 6, a number has to be divisible by 2 and divisible by 3 [a generalization of this idea is known as the Chinese remainder theorem]). So, we could run the two automatas of Section 2.3.1 and Section 2.3.2 in parallel (replicating each input character to each one of the two automatas), and accept only if both automatas are in an accept state. This idea would become more useful later in the course, as it provide a building operation to construct complicated automatas from simple automatas.

2.3.4 Number of ones is even

Input is a string over $\Sigma = \{\texttt{0}, \texttt{1}\}$.

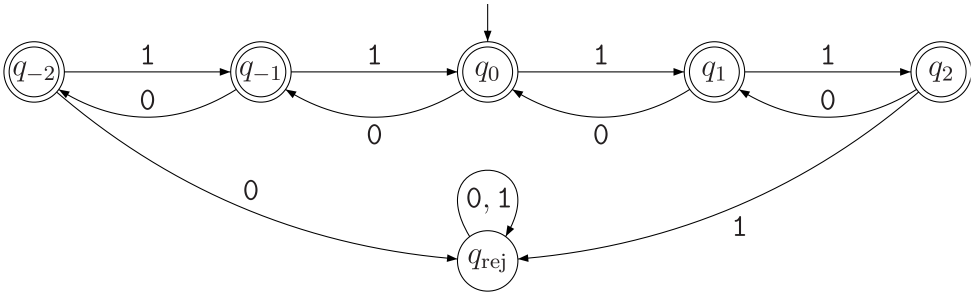
Accept: all strings in which the number of ones is even.



2.3.5 Number of zero and ones is always within two of each other

Input is a string over $\Sigma = \{\texttt{0}, \texttt{1}\}$.

Accept: all strings in which the difference between the number of ones and zeros in any prefix of the string is in the range $-2, \ldots, 2$. For example, the language contains ϵ, 0, 001, and 1101. You even have an extended sequence of one character e.g. 001111, but it depends what preceded it. So 111100 isn't in the language.



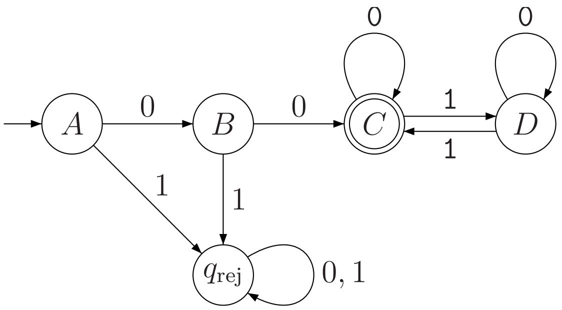
Notice that the names of the states reflect their role in the computation. When you come to analyze these machines formally, good names for states often makes your life much easier. BTW, the language of this DFA is

$$L(M) = \left\{ w \;\middle|\; w \in \{\texttt{0}, \texttt{1}\}^* \text{ and for every } x \text{ that is a prefix of } w, \, |\#\texttt{1}(x) - \#\texttt{0}(x)| \leq 2 \right\}.$$

2.3.6 More complex language

The input is strings over $\Sigma = \{\texttt{0}, \texttt{1}\}$.

Accept: all strings of the form $00w$, where w contains an even number of ones.



You can name states anything you want. Names of the form q_X are often convenient, because they remind you of what's a state. And people often make the initial state q_0. But this isn't obligatory.

2.4 The pieces of a DFA

To specify a DFA (***deterministic finite automata***), we need to describe

- a (finite) alphabet
- a (finite) set of states
- which state is the start state?
- which states are the final states?
- what is the transition from each state, on each input character?

Chapter 3

Lecture 3: More on DFAs

27 January 2009

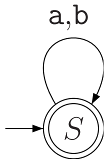
This lecture continues with material from section 1.1 of Sipser.

3.1 JFLAP demo

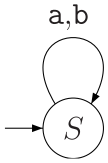
Go to `http://www.jflap.org`. Run the applet ("Try applet" near the bottom of the menu on the lefthand side). Construct some small DFA and run a few concrete examples through it.

3.2 Some special DFAs

For $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, consider the following DFA that accepts Σ^*:

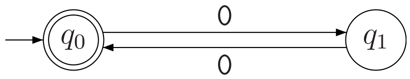


The DFA that accepts nothing, is just



3.3 Formal definition of a DFA

Consider the following automata, that we saw in the previous lecture:



We saw last class that the following components are needed to specify a DFA:

(i) a (finite) alphabet

(ii) a (finite) set of states

(iii) which state is the start state?

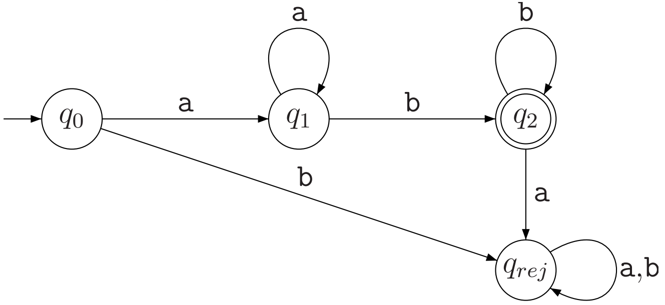
(iv) which states are the final states?

(v) what is the transition from each state, on each input character?

Formally, a ***deterministic finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where

- Q: A finite set (the set of ***states***).
- Σ: A finite set (the ***alphabet***)
- $\delta : Q \times \Sigma \to Q$ is the ***transition function***.
- q_0: The ***start*** state (belongs to Q).
- F: The set of ***accepting*** (or ***final***) states, where $F \subseteq Q$.

For example, let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and consider the following DFA M, whose language $L(M)$ contains strings consisting of one or more a's followed by one or more b's.



Then $M = (Q, \Sigma, \delta, q_0, F)$, $Q = \{q_0, q_1, q_2, q_{rej}\}$, and $F = \{q_2\}$. The transition function δ is defined by

δ	a	b
q_0	q_1	q_{rej}
q_1	q_1	q_2
q_2	q_{rej}	q_2
q_{rej}	q_{rej}	q_{rej}

We can also define δ using a formula

$$\delta(q_0, a) = q_1$$

$$\delta(q_1, a) = q_1$$

$$\delta(q_1, b) = q_2$$

$$\delta(q_2, b) = q_2$$

$$\delta(q, t) = q_{rej} \text{ for all other values of } q \text{ and } t.$$

Tables and state diagrams are most useful for small automata. Formulas are helpful for summarizing a group of transitions that fit a common pattern. They are also helpful for describing algorithms that modify automatas.

3.4 Formal definition of acceptance

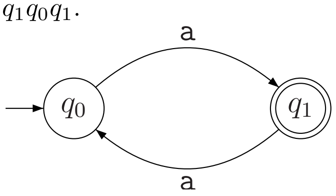
We've also seen informally how to run a DFA. Let us turn that into a formal definition. Suppose $M = (Q, \Sigma, \delta, q_0, F)$ is a given DFA and $w = w_1 w_2 \dots w_k \in \Sigma^*$ is the input string. Then M ***accepts*** w iff there exists a sequence of states $r_0, r_1, \dots r_k$ in Q, such that

1. $r_0 = q_0$
2. $\delta(r_i, w_{i+1}) = r_{i+1}$ for $i = 0, \dots, k-1$.
3. $r_k \in F$.

The ***language recognized*** by M, denoted by $L(M)$, is the set $\left\{w \;\middle|\; M \text{ accepts } w\right\}$.

For example, when our automaton above accepts the string `aabb`, it uses the state sequence $q_0q_1q_1q_2q_2$. (Draw a picture of the transitions.) That is $r_0 = q_0$, $r_1 = q_1$, $r_2 = q_1$, $r_3 = q_2$, and $r_4 = q_2$.

Note that the states do not have to occur in numerical order in this sequence, e.g. the following DFA accepts `aaa` using the state sequence $q_0q_1q_0q_1$.



A language (i.e. set of strings) is ***regular*** if it is recognized by some DFA.

3.5 Closure properties

Consider the set of odd integers. If we multiply two odd integers, the answer is always odd. So the set of odd integers is said to be ***closed*** under multiplication. But it is not closed under addition. For example, $3 + 5 = 8$ which is not odd.

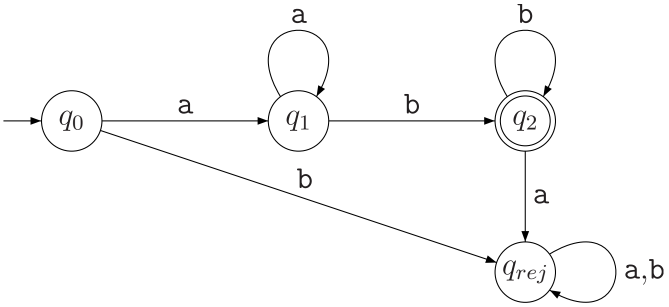
To talk about closure, you need two sets: a larger universe U and a smaller set $X \subseteq U$. The universe is often supposed to be understood from context. Suppose you have a function F that maps values in U to values in U. Then X is ***closed under f*** if F applied to values from X always produces an output value that is also in X.

For automata theory, U is usually the set of all languages and X contains languages recognized by some specific sort of machine, e.g. regular languages.

3.5.1 Closure under complement of regular languages

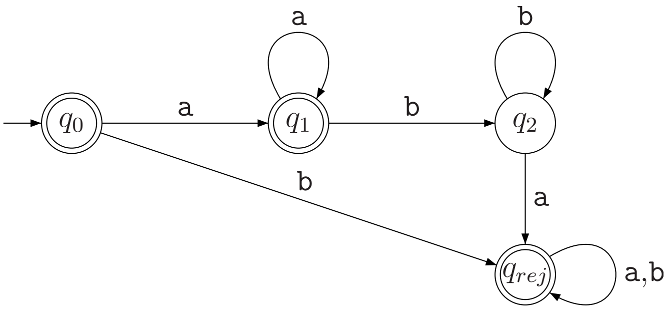
Here we are interested in the question of whether the regular languages are closed under set complement. (The complement language keeps the same alphabet.) That is, if we have a DFA $M = (Q, \Sigma, \delta, q_0, F)$ accepting some language L, can we construct a new DFA M' accepting $\overline{L} = \Sigma^* \setminus L$?

Consider the automata M from above, where L is the set of all strings of at least one `a` followed by at least one `b`.



The complement language $\overline{L}$ contains the empty string, strings in which some `b`'s precede some `a`'s, and strings that contain only `a`'s or only `b`'s.

Our new DFA M' should accept exactly those strings that M rejects. So we can make M' by swapping final/non-final markings on the states:

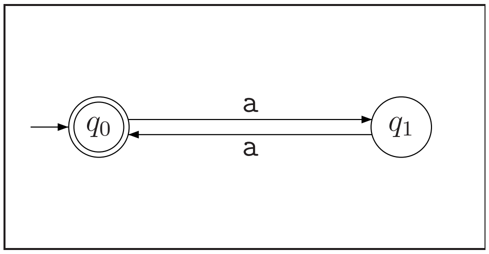


Formally, $M' = (Q, \Sigma, \delta, q_0, Q \setminus F)$.

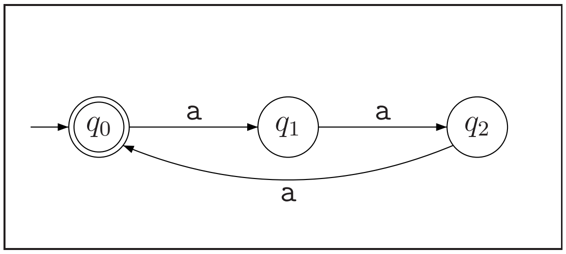
3.6 Closure under intersection

We saw in previous lecture an automatas that accepts strings of even length, or that their length is a product of 3. Here are their automatas:

M_1:

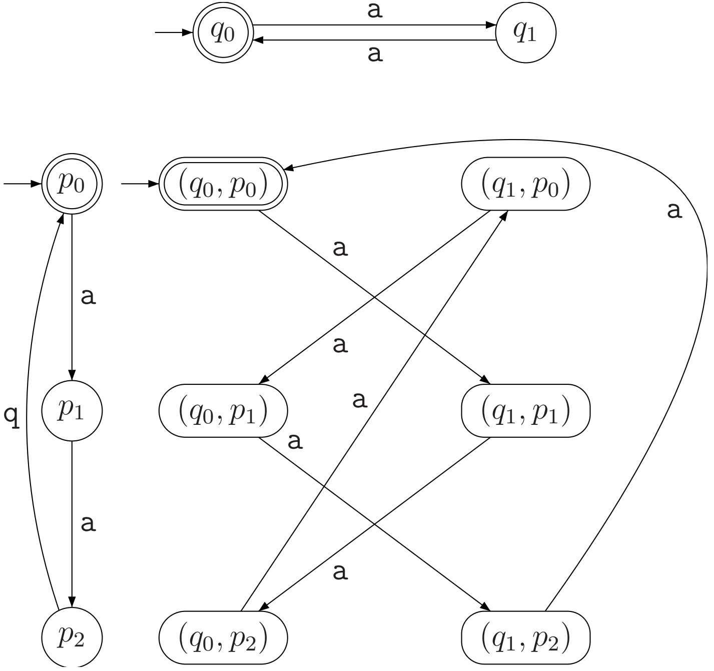


M_2:



Assume, that we would like to build an automata that accepts the language which is the intersection of the language of both automatas. That is, we would like to accept the language $L(M_1) \cap L(M_2)$. How do we build an automata for that?

The idea is to build a product automata of both automatas. See the following for an example.



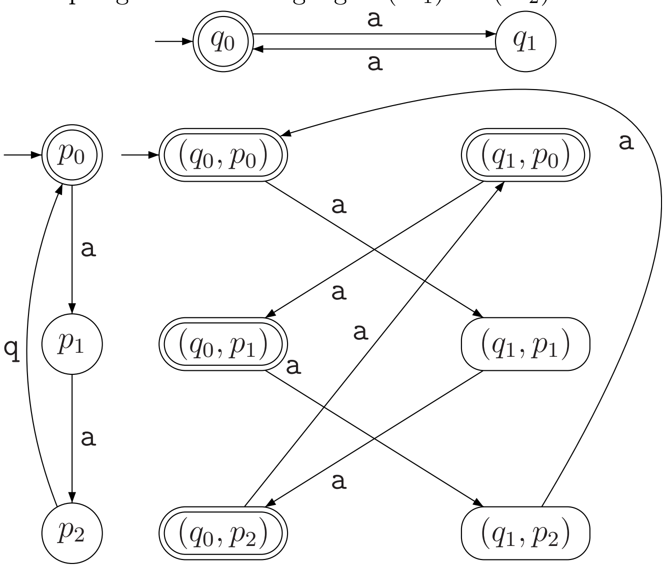
Given two automatas $M = (Q, \Sigma, \delta, q_0, F)$ and $M' = (Q', \Sigma', \delta', q'_0, F')$, their ***product automata*** is the automata formed by the product of the states. Thus, a state in the resulting automata $N = M \times M'$ is a pair (q, q'), where $q \in Q$ and $q' \in Q'$.

The key invariant of the product automata is that after reading a word w, its in the state (q, q'), where, q is that state that M is at after reading w, and q' is the state that M' is in after reading w.

As such, the intersection language $L(M) \cap L(M')$ is recognized by the product automata, where we set the pairs $(q, q') \in Q(N)$ to be an accepting state for N, if $q \in F$ and $q' \in F'$.

Similarly, the automata accepting the union $L(M) \cup L(M')$ is created from the product automata, by setting the accepting states to be all pairs (q, q'), such that either $q \in F$ or $q' \in F'$.

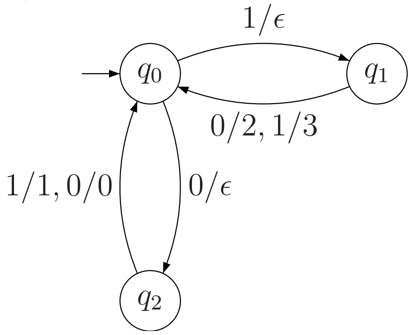
As such, the automata accepting the union language $L(M_1) \cup L(M_2)$ is the following.



3.7 (Optional) Finite-state Transducers

In many applications, transitions also perform actions. E.g. a transition reading `WANNATALK` from the network might also call some `C` code that opens a new HTTP connection.

Finite-state transducers are a simple case of this. An FST is like a DFA but each transition optionally writes an output symbol. These can be used to translate strings from one alphabet to another. For example, the following FST translates binary numbers into base-4 numbers. E.g. 011110 becomes 132. We'll assume that FSTs don't accept or reject strings, just translate them.



So, formally, an FST is a 5-tuple $(Q, \Sigma, \Gamma, \delta, q_0)$, where

- Q is a finite set (the states).

- Σ and Γ are finite sets (the input and output alphabets).

- $\delta : Q \times \Sigma \to Q \times \Gamma_\epsilon$ is the transition function.

- q_0 is the start state

Notation: $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$.
The transition table for our example FST might look like the following.

δ	0	1
q_0	(q_1, ϵ)	(q_2, ϵ)
q_1	$(q_0, 0)$	$(q_0, 1)$
q_2	$(q_0, 2)$	$(q_0, 3)$

Chapter 4

Lecture 4: Regular Expressions and Product Construction

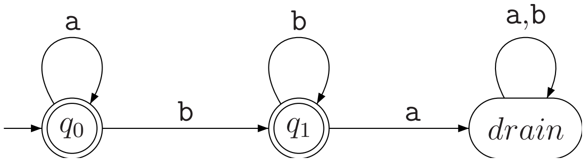
29 January 2009

This lecture finishes section 1.1 of Sipser and also covers the start of 1.3.

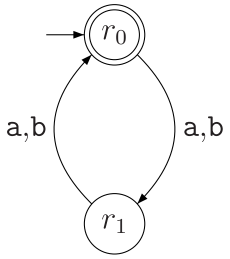
4.1 Product Construction

4.1.1 Product Construction: Example

Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and L is the set of strings in Σ^* that have the form $\mathtt{a}^*\mathtt{b}^*$ and have even length. L is the intersection of two regular languages $L_1 = \mathtt{a}^*\mathtt{b}^*$ and $L_2 = (\Sigma\Sigma)^*$. We can show they are regular by exhibiting DFAs that recognize them.

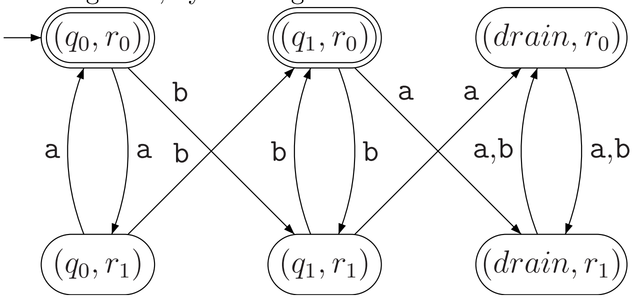


L_1



L_2

We can run these two DFAs together, by creating states that remember the states of both machines.



Notice that the final states of the new DFA are the states (q, r) where q is final in the first DFA ***and*** r is final in the second DFA. To recognize the union of the two languages, rather than the intersection, we mark all the states (q, r) such that either q or r are accepting states in the their respective DFAs.

State of a DFA after reading a word w. In the following, given a DFA $M = (Q, \Sigma, \delta, q_0, F)$, we will be interested in what state the DFA M is in, after reading the characters of a string $w = w_1 w_2 \ldots w_k \in \Sigma^*$. As in the definition of acceptance, we can just define the sequence of states that M would go through as it reads w. Formally, $r_0 = q_0$, and

$$r_i = \delta(r_{i-1}, w_i)\,, \quad \text{for } i = 1, \ldots, k.$$

As such, r_k is the state M would be after reading the string w. We will denote this state by $\delta(q_0, w)$. Note, that by definition

$$\delta(q_0, w) = \delta\Big(\delta(q_0, w_1 \ldots w_{k-1})\,,\ w_k\Big)\,.$$

In general, if the DFA is in a state q, and we want to know in what state it would be after reading a string w, we will denote it by $\delta(q, w)$.

4.2 Product Construction: Formal construction

We are given two DFAs $M = (Q, \Sigma, \delta, q_0, F)$ and $M' = (Q', \Sigma, \delta', q'_0, F')$ both working above the same alphabet Σ. Their ***product automata*** is the automata

$$N = \Big(\mathcal{Q},\ \Sigma,\ \delta_N,\ (q_0, q'_0)\,,\ F_N\Big)\,,$$

where $\mathcal{Q} = Q \times Q'$, and $\delta_N : \mathcal{Q} \times \Sigma \to \mathcal{Q}$. Here, for $q \in Q$, $q' \in Q'$ and $c \in \Sigma$, we define

$$\delta_N(\ \underbrace{(q, q')}_{\text{state of } N}\ , c\ \ ) = \Big(\delta(q, c),\ \delta'(q', c)\Big)\,. \tag{4.1}$$

The set $F_N \subseteq \mathcal{Q}$ of accepting states is free to be whatever we need it to be, depending on what we want N to recognize. For example, if we would like N to accept the intersection $L(M) \cap L(M')$ then we will set $F_N = F \times F'$. If we want N to recognize the union language $L(M) \cup L(M')$ then $F_N = (F \times Q') \cup \cup (Q \times F')$.

Lemma 4.2.1 *For any input word $w \in \Sigma^*$, the product automata N of the DFAs $M = (Q, \Sigma, \delta, q_0, F)$ and $M' = (Q', \Sigma, \delta', q'_0, F')$, is in state (q, q') after reading w, if and only if (i) M in the state q after reading w, and (ii) M' is in the state q' after reading w.*

Proof: The proof is by induction on the length of the word w.

If $w = \epsilon$ is the empty word, then N is initially in the state (q_0, q'_0) by construction, where q_0 (resp. q'_0) is the initial state of M (resp. M'). As such, the claim holds in this case.

Otherwise, assume $w = w_1 w_2 \ldots w_{k-1} w_k$, and the claim is true by induction for all input words of length strictly smaller than k.

Let (q_{k-1}, q'_{k-1}) be the state that N is in after reading the string $\widehat{w} = w_1 \ldots w_{k-1}$. By induction, as $|\widehat{w}| = k - 1$, we know that M is in the state q_{k-1} after reading $\widehat{w}$, and M' is in the state q'_{k-1} after reading $\widehat{w}$.

Let $q_k = \delta(q_{k-1}, w_k) = \delta(\delta(q_0, \widehat{w})\,,\ w_k) = \delta(q_0, w)$ and

$$q'_k = \delta'(q'_{k-1}, w_k) = \delta'(\delta'(q'_0, \widehat{w})\,,\ w_k) = \delta'(q'_0, w)\,.$$

As such, by definition, M (resp. M') would in the state q_k (resp. q'_k) after reading w.

Also, by the definition of its transition function, after reading w the DFA N would be in the state

$$\begin{aligned}\delta_N((q_0, q'_0), w) &= \delta_N(\delta_N((q_0, q'_0), \widehat{w})\,, w_k) = \delta_N\big((q_{k-1}, q'_{k-1}),\ w_k\big)\\ &= \big(\delta(q_{k-1}, w_k),\ \delta(q'_{k-1}, w_k)\big) = (q_k,\ q'_k)\,,\end{aligned}$$

see Eq. (4.1). This establishes the claim. ■

Lemma 4.2.2 *Let $M = (Q, \Sigma, \delta, q_0, F)$ and $M' = (Q', \Sigma, \delta', q'_0, F')$ be two given DFAs. Let N be their produced automata, where its set of accepting states is $F \times F'$. Then $L(N) = L(M) \cap L(M')$.*

Proof: If $w \in L(M) \cap L(M')$, then $q_w = \delta(q_0, w) \in F$ and $q'_w = \delta'(q'_0, w) \in F'$. By Lemma 4.2.1, this implies that $\delta_N((q_0, q'_0), w) = (q_w, q'_w) \in F \times F'$. Namely, N accepts the word w, implying that $w \in L(N)$, and as such $L(M) \cap L(M') \subseteq L(N)$.

Similarly, if $w \in L(N)$, then $(p_w, p'_w) = \delta_N((q_0, q'_0), w)$ must be an accepting state of N. But the set of accepting states of N is $F \times F'$. That is $(p_w, p'_w) \in F \times F'$, implying that $p_w \in F$ and $p'_w \in F'$. Now, by Lemma 4.2.1, we know that $\delta(q_0, w) = p_w \in F$ and $\delta'(q'_0, w) = p'_w \in F'$. Thus, M and M' both accept w, which implies that $w \in L(M)$ and $w \in L(M')$. Namely, $w \in L(M) \cap L(M')$, implying that $L(N) \subseteq L(M) \cap L(M')$.

Putting the above together implies the claim. ■

4.3 Operations on languages

Regular operations on languages (sets of strings). Suppose L and K are languages.

- ***Union***: $L \cup K = \left\{ x \,\middle|\, x \in L \quad \text{or} \quad x \in K \right\}$.
- ***Concatenation***: $L \circ K = LK = \left\{ xy \,\middle|\, x \in L \quad \text{and} \quad y \in K \right\}$.
- ***Star*** (***Kleene star***):

$$L^* = \left\{ w_1 w_2 \dots w_n \,\middle|\, w_1, \dots, w_n \in L \text{ and } n \geq 0 \right\}.$$

We (hopefully) all understand what union does. The other two have some subtleties. Let

$$L = \{\texttt{under}, \texttt{over}\}, \quad \text{and} \quad K = \{\texttt{ground}, \texttt{water}, \texttt{work}\}.$$

Then

$$LK = \{\texttt{underground}, \texttt{underwater}, \texttt{underwork}, \texttt{overground}, \texttt{overwater}, \texttt{overwork}\}.$$

Similarly,

$$K^* = \left\{ \begin{array}{l} \epsilon, \texttt{ground}, \texttt{water}, \texttt{work}, \texttt{groundground}, \\ \texttt{groundwater}, \texttt{groundwork}, \texttt{workground}, \\ \texttt{waterworkwork}, \dots \end{array} \right\}.$$

For star operator, note that the resulting set *always* contains the empty string ϵ (because n can be zero).

Also, each of the substrings is chosen independently from the base set and you can repeat. E.g. `waterworkwork` is in K^*.

Regular languages are closed under many operations, including the three "regular operations" listed above, set intersection, set complement, string reversal, "homomorphism" (formal version of shifting alphabets). We have seen (last class) why regular languages are closed under set complement. We will prove the rest of these bit by bit over the next few lectures.

4.4 Regular Expressions

Regular expressions are a convenient notation to specify regular languages. We will prove in a few lectures that regular expressions can represent ***exactly*** the same languages that DFAs can accept.

Let us fix an alphabet Σ. Here are the basic regular expressions:

regex	conditions	set represented
$\texttt{a}$	$\texttt{a} \in \Sigma$	$\{\texttt{a}\}$
ϵ		$\{\epsilon\}$
$\emptyset$		$\{\}$

Thus, $\emptyset$ represents the empty language. But ϵ represents that language which has the empty word as its only word in the language.

In particular, for a regular expression $\langle \texttt{exp} \rangle$, we will use the notation $L(\langle \texttt{exp} \rangle)$ to denote the language associated with this regular expression. Thus,

$$L(\epsilon) = \{\epsilon\} \quad \text{and} \quad L(\emptyset) = \{\}\,,$$

which are two *different* languages.

We will slightly abuse notations, and write a regular expression $\langle \texttt{exp} \rangle$ when in reality what we refer to is the language $L(\langle \texttt{exp} \rangle)$. (Abusing notations should be done with care, in cases where it clarify the notations, and it is well defined. Naturally, as Humpty Dumpty did, you need to define your "abused" notations explicitly.[1])

Suppose that $L(R)$ is the language represented by the regular expression R. Here are recursive rules that make complex regular expressions out of simpler ones. (Lecture will add some randomly-chosen small concrete examples.)

regex	conditions	set represented
$R \cup S$ or $R + S$	R, S regexes	$L(R) \cup L(S)$
$R \circ S$ or RS	R, S regexes	$L(R)L(S)$
R^*	R a regex	$L(R)^*$

And some handy shorthand notation:

regex	conditions	set represented
R^+	R a regex	$L(R)L(R)^*$
Σ		Σ

Exponentiation binds most tightly, then multiplication, then addition. Just like you probably thought. Use parentheses when you want to force a different interpretation.

Some specific boundary case examples:

1. $R\epsilon = R = \epsilon R$.

2. $R\emptyset = \emptyset = \emptyset R$.

   This is a bit confusing, so let us see why this is true, recall that

   $$R\emptyset = \left\{ xy \;\middle|\; x \in R \text{ and } y \in \emptyset \right\}.$$

   But the empty set ($\emptyset$) does not contain any element, and as such, no concatenated string can be created. Namely, its the empty language.

3. $R \cup \emptyset = R$ (just like with any set).

4. $R \cup \epsilon = \epsilon \cup R$.

   This expression can not always be simplified, since ϵ might not be in the language $L(R)$.

5. $\emptyset^* = \{\epsilon\}$, since the empty word is always contain in the language generated by the star operator.

6. $\epsilon^* = \{\epsilon\}$.

[1] From *Through the Looking Glass*, by Lewis Carroll:

> 'And only one for birthday presents, you know. There's glory for you!'
>
> 'I don't know what you mean by "glory",' Alice said.
>
> Humpty Dumpty smiled contemptuously. 'Of course you don't – till I tell you. I meant "there's a nice knock-down argument for you!"'
>
> 'But "glory" doesn't mean "a nice knock-down argument",' Alice objected.
>
> 'When I use a word,' Humpty Dumpty said, in rather a scornful tone, 'it means just what I choose it to mean – neither more nor less.'
>
> 'The question is,' said Alice, 'whether you can make words mean so many different things.'
>
> 'The question is,' said Humpty Dumpty, 'which is to be master – that's all.'

4.4.1 More interesting examples

Suppose $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$.

1. $(\Sigma\Sigma)^*$ is the language of all even-length strings.

   (That is, the language associated with the regular expression $(\Sigma\Sigma)^*$ is made out of all the even-length strings over Σ.)

2. $\Sigma(\Sigma\Sigma)^*$ is all odd-length strings.

3. $\mathtt{a}\Sigma^*\mathtt{a} + \mathtt{b}\Sigma^*\mathtt{b} + \mathtt{c}\Sigma^*\mathtt{c}$ is all strings that start and end with the same character.

Regular expression for decimal numbers

Let $D = \{\mathtt{0}, \mathtt{1}, ..., \mathtt{9}\}$, and consider the alphabet $E = D \cup \{-, .\}$. Then decimal numbers have the form

$$(- \cup \epsilon)\, D^*\, (\epsilon \cup .)D^*.$$

But this does not force the number to contain any digits, which is probably wrong. As such, the correct expression is

$$(- \cup \epsilon)(D^+(\epsilon \cup .)D^* \cup D^*(\epsilon \cup .)D^+).$$

Notice that $\mathtt{a}^n$ is **not** a regular expression. Some things written with non-star exponents are regular and some are not. It depends on what conditions you put on n. E.g. $\left\{\mathtt{a}^{2n} \,\middle|\, n \geq 0\right\}$ is regular (even length strings of $\mathtt{a}$'s). But $\left\{\mathtt{a}^n\mathtt{b}^n \,\middle|\, n \geq 0\right\}$ is not regular.

However, $\mathtt{a}^3$ (or any other fixed power) is regular, as it just a shorthand for $\mathtt{aaa}$. Similarly, if R is a regular expression, then R^3 is regular since its a shorthand for RRR.

Chapter 5

Lecture 5: Nondeterministic Automata

February 3, 2009

This lecture covers the first part of section 1.2 of Sipser, through p 54.

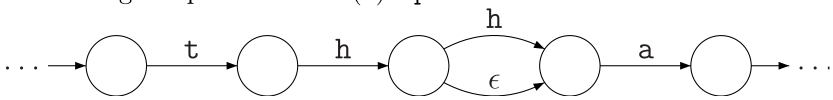
5.1 Non-deterministic finite automata (NFA)

A ***non-deterministic finite automata*** (***NFA***) is like a DFA but with three extra features. These features make them easier to construct, especially because they can be composed in a modular fashion. Furthermore, they are easier to read, and they tend to be much smaller and as such easier to describe. Computationally, they are equivalent to DFAs, in the sense that they recognize the same languages.

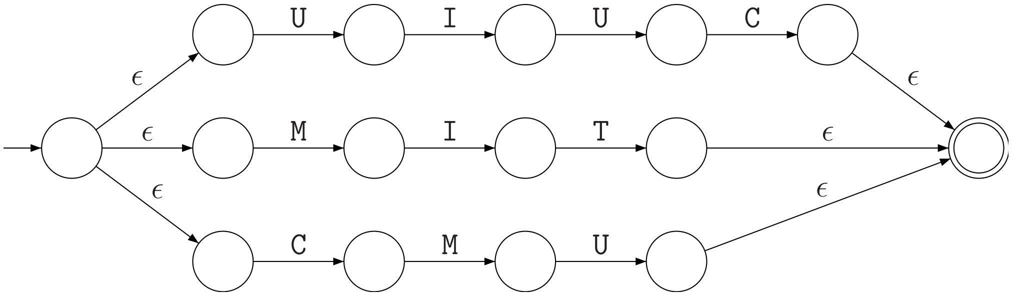
For practical applications of any complexity, users can write NFAs or regular expressions (trivial to convert to NFAs). A computer algorithm might compile these to DFAs, which can be executed/simulated quickly.

5.1.1 NFA feature #1: Epsilon transitions

An NFA can do a state transition without reading input. This makes it easy to represent optional characters. For example, "Northampton" is commonly misspelled as "Northhampton". A web search type application can recognize both variants using the pattern North(h)ampton.



Epsilon transitions also allow multiple alternatives (set union) to be spliced together in a nice way. E.g. we can recognize the set {UIUC, MIT, CMU} with the following automaton. This allows modular construction of large state machines.



How do we execute an NFA?

Assume a NFA N is in state q, and the next input character is c. The NFA N may have multiple transitions it could take. That is, multiple possible next states. An NFA accepts if there is *some* path through its state diagram that consumes the whole input string and ends in an accept state.

Here are two possible ways to think about this:

(i) the NFA magically guesses the right path which will lead to an accept state.

(ii) the NFA searches all paths through the state diagram to find such a path.

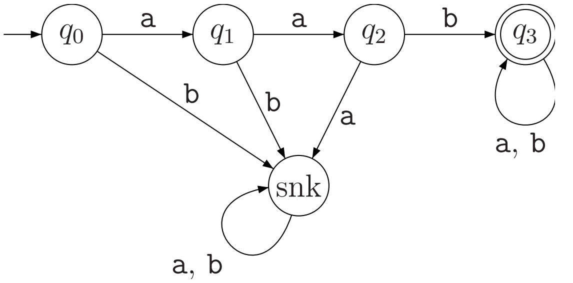
The first view is often the best for mathematical analysis. The second view is one reasonable approach to implementing NFAs.

5.1.2 NFA Feature #2: Missing transitions

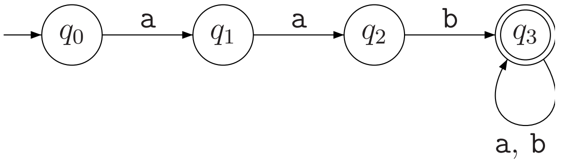
Assume a NFA N is in state q, and the next input character is c. The NFA may have no outgoing transition from q that corresponds to the input character c.

This means that you can not get to an accepting state from this point on. So the NFA will reject the input string unless there is some other alternative path through the state diagram. You can think of the missing transitions as going to an implicit sink state. Visually, diagrams of NFAs are much simpler by not having to put in the sink state explicitly.

Example. Consider the DFA that accepts all the strings over $\{\mathtt{a}, \mathtt{b}\}$ that starts with aab. Here is the resulting DFA.

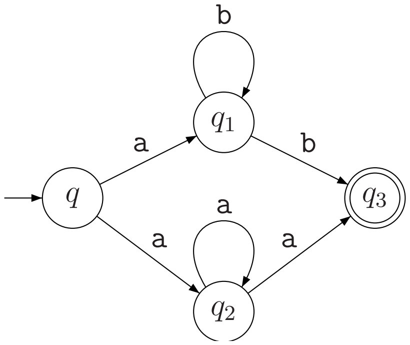


The NFA for the same language is even simpler if we omit transitions, and the sink state. In particular, the NFA for the above language is the following.



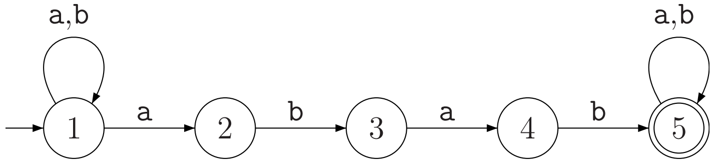
5.1.3 NFA Feature #3: Multiple transitions

A state q in a NFA may have more than one outgoing transition for some character t. This means that the NFA needs to "guess" which path will accept the input string. Or, alternatively, search all possible paths. This complicates deciding if a string is accepted by a NFA, but it greatly simplifies the resulting machines. Thus, the automata on the right accepts all strings in the language $\mathtt{a}b^*\mathtt{b} + \mathtt{a}\mathtt{a}^*\mathtt{a}$. Of course, its not too hard to build a DFA for this language, but even here the description of the NFA is simpler.

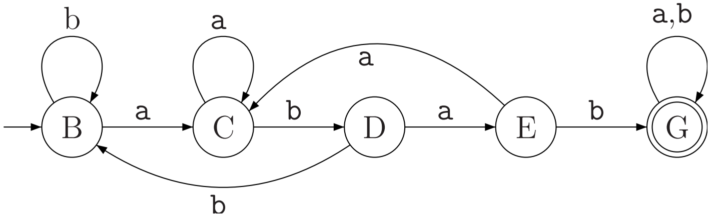


As another example, the automata below accepts strings containing the substring abab.

(N1)



The respective DFA, shown below, needs a lot more transitions and is somewhat harder to read.



5.2 More Examples

5.2.1 Running an NFA via search

Let us run an explicit search for the above NFA (N1) on the input string ababa. Initially, at time $t = 0$, the only possible state is the start state 1. The search is depicted in table on the right. When the input is exhausted, one of the possible states (E) is an accept state, and as such the NFA (N1) accepts the string ababa.

Time	Possible states	Remaining input
$t = 0$	$\{1\}$	ababa
$t = 1$	$\{1, 2\}$	baba
$t = 2$	$\{1, 3\}$	aba
$t = 3$	$\{1, 2, 4\}$	ba
$t = 4$	$\{1, 3, 5\}$	a
$t = 5$	$\{1, 2, 4, 5\}$	ϵ

5.2.2 Interesting guessing example

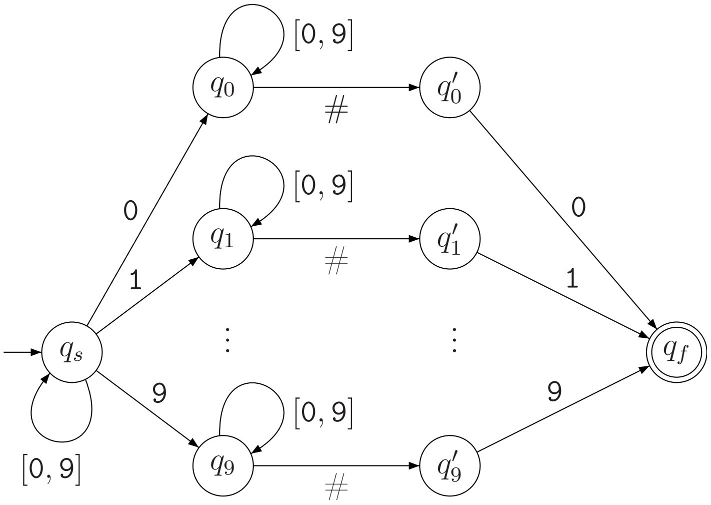
Some NFAs are easier to construct and analyze if you take the "guessing" view on how they work.

Let $\Sigma = \{0, 1, \ldots, 9\}$, denote this as $[0, 9]$ in short form. Let

$$L = \left\{ w\#c \;\middle|\; c \in \Sigma, w \in \Sigma^*, \text{ and } c \text{ occurs in } w \right\}.$$

For example, the word 314159#5 is in L, and so is 314159#3. But the word 314159#7 is not in L.

Here is the NFA M that recognizes this language.



The NFA M scans the input string until it "guesses" that it is at the character c in w that will be at the end of the input string. When it makes this guess, M transitions into a state q_c that "remembers" the value c. The rest of the transitions then confirm that the rest of the input string matches this guess.

A DFA for this problem is considerably more taxing. We will need a state to remember each digit encountered in the string read so far. Since there are 2^{10} different subsets, we will require an automata with at least 1024 states! The NFA above requires only 22 states, and is much easier to draw and understand.

5.3 Formal definition of an NFA

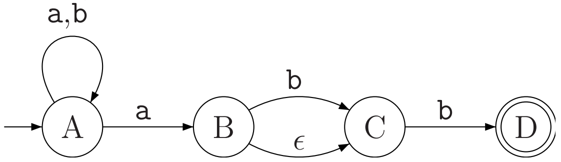
An NFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$. Similar to a DFA except that the type signature for δ is

$$\delta : Q \times \Sigma_\epsilon \to \mathbb{P}(Q),$$

where $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ and $\mathbb{P}(Q)$ is the power set of Q (i.e., all possible subsets of Q). As such, the input character for $\delta(\cdot)$ can be either a real input character or ϵ (in this case the NFA does not eat [or drink] any input character when using this transition). The output value of δ is a *set* of states (unlike a DFA).

Example 5.3.1 Consider the language $L = (\mathtt{a} + \mathtt{b})^*\mathtt{a}(\mathtt{b} + \epsilon)\mathtt{b}$.

Its respective NFA is the following:



Here

$\delta(A, \mathtt{a}) = \{A, B\}$

$\delta(B, \mathtt{a}) = \emptyset$ (NB: not $\{\emptyset\}$)

$\delta(B, \epsilon) = \{C\}$ (NB: not just C)

$\delta(B, \mathtt{b}) = \{C\}$ (NB: just follows one transition arc).

The trace for recognizing the input `abab`:

$t = 0$: state $= A$, remaining input `abab`.

$t = 1$: state $= A$, remaining input `bab`.

$t = 2$: state $= A$, remaining input `abi`.

$t = 3$: state $= B$, remaining input `b`.

$t = 4$: state $= C$, remaining input `b` (ϵ transition used, and no input eaten).

$t = 5$: state $= D$, remaining input ϵ.

Is every DFA an NFA? Technically, no (why?[1]). However, it is easy to convert any DFA into an NFA. If δ is the transition function of the DFA, then the corresponding transition of the NFA is going to be $\delta'(q, t) = \{\delta(q, t)\}$.

5.3.1 Formal definition of acceptance

Let $M = \Big(Q, \Sigma, \delta, q_0, F\Big)$ be an NFA. Let w be a string in Σ^*.

The NFA M accepts w if and only if there is a sequence of states $r_0, r_1, \ldots, r_n$ and a sequence of inputs $x_1, x_2, \ldots, x_n$, where each x_i is either a character from Σ or ϵ, such that

(i) $w = x_1 x_2 \ldots x_n$.

(The input string "eaten" by the NFA is the input string w.)

[1] Because, the transition function is defined differently.

(ii) $r_0 = q_0$.

(The NFA starts from the start state.)

(iii) $r_n \in F$.

(The final state in the trance in an accepting state.)

(iv) $r_{i+1} \in \delta(r_i, x_{i+1})$ for every i in $[0, n-1]$.

(The transitions in the trace are all valid. That is, the state r_{i+1} is one of the possible states one can go from r_i, if the NFA consumes the character x_{i+1}.

So, in the above example, $n = 6$, our state sequence is $AAABCD$, and our sequence of inputs is abaϵb. Key differences the notation of acceptance from DFA are

(i) Inserting/allowing ϵ into input character sequence.

(ii) Output of δ is a set, so in condition (iv) above, r_{i+1} is a member of δ's output. (For a DFA, in this case, we just had to check that the new state is equal to δ's output.)

Chapter 6

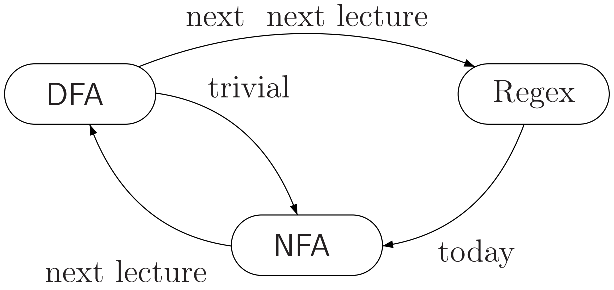
Lecture 6: Closure properties

February 5, 2009

This lecture covers the last part of section 1.2 of Sipser (pp. 58–63), part of 1.3 (pp. 66–69), and also closure under string reversal and homomorphism.

6.1 Overview

We defined a language to be ***regular*** if it is recognized by some DFA. The agenda for the new few lectures is to show that three different ways of defining languages, that is NFAs, DFAs, and regexes, and in fact all equivalent; that is, they all define regular languages. We will show this equivalence, as follows.



One of the main properties of languages we are interested in are closure properties, and the fact that regular languages are closed under union, intersection, complement, concatenation, and star (and also under homomorphism).

However, closure operations are easier to show in one model than the other. For example, for DFAs showing that they are closed under union, intersection, complement are easy. But showing closure of DFA under concatenation and $*$ is hard.

Here is a table that lists the closure property and how hard it is to show it in the various models of regular languages.

Model					
Property	$\cap$ intersection	$\cup$ union	$\overline{L}$ complement	$\circ$ concatenation	$*$ star
DFA	Easy (done)	Easy (done)	Easy (done)	Hard	Hard
NFA	Doable (hw?)	Easy: Lemma 6.3.1	Hard	Easy: Lemma 6.3.2	Easy: Lemma 6.3.3
regex	Hard	Easy	Hard	Easy	Easy

Recall what it means for regular languages to be closed under an operation op. If L_1 and L_2 are regular, then L_1 op L_2 is regular. That is, if we have an NFA recognizing L_1 and an NFA recognizing L_2, we can construct an NFA recognizing L_1 op L_2.

The extra power of NFAs makes it easy to prove closure properties for NFAs. When we know all DFAs, NFAs, and regexes are equivalent, these closure results then apply to all three representations. Namely, they would imply that regular languages have these closure properties.

6.2 Closure under string reversal for NFAs

Consider a word w, we denote by w^R the ***reversed*** word. It is just w in the characters in reverse order. For example, for $w =$ barbados, we have $w^R =$ sodabrab. For a language L, the ***reverse*** language is

$$L^R = \left\{ w^R \,\middle|\, w \in L \right\}.$$

We would like to claim that if L is regular, then so is L^R. Formally, we need to be a little bit more careful, since we still did not show that a language being regular implies that it is recognized by an NFA.

Claim 6.2.1 *If L is recognized by an NFA, then there is an NFA that recognizes L^R.*

Proof: Let M be an NFA recognizing L. We need to construct an NFA N recognizing L^R.

The idea is to reverse the arrows in the NFA $M = (Q, \Sigma, \delta, q_0, F)$, and swap final and initial states. There is a bug in applying this idea in a naive fashion. Indeed, there is only one initial state but multiple final states.

To overcome this, let us modify M to have a single final state q_S, connected to old ones with epsilon transitions. Thus, the modified NFA accepting L, is

$$M' = \Big(Q \cup \{q_S\}, \Sigma, \delta', q_0, \{q_S\} \Big),$$

where q_S is the only accepting state for M. Note, that δ' is identical to δ, except that

$$\forall q \in F \quad \delta'(q, \epsilon) = q_S. \tag{6.1}$$

Note, that $L(M) = L(M') = L$.

As such, q_S will become the start state of the "reversed" NFA.

Now, the new "reversed" NFA N, for the language L^R, is

$$N = \Big(Q \cup \{q_S\}, \Sigma, \delta'', q_S, \{q_0\} \Big).$$

Here, the transition function δ'' is defined as

(i) $\delta''(q_0, t) = \emptyset$ for every $t \in \Sigma$.

(ii) $\delta''(q, t) = \left\{ r \in Q \,\middle|\, q \in \delta'(r, t) \right\}$, for every $q \in Q \cup \{q_S\}$, $t \in \Sigma_\epsilon$.

(iii) $\delta''(q_S, \epsilon) = F$ (the reversal of Eq. (6.1)). [1]

Now, we need to prove formally that if $w \in L(M)$ then $w^R \in L(N)$, but this is easy induction, and we omit it. ■

Note, that this will not work for a DFA. First, we can not force a DFA to have a single final state. Second, a state may have two incoming transitions on the same character, resulting in non-determinism when reversed.

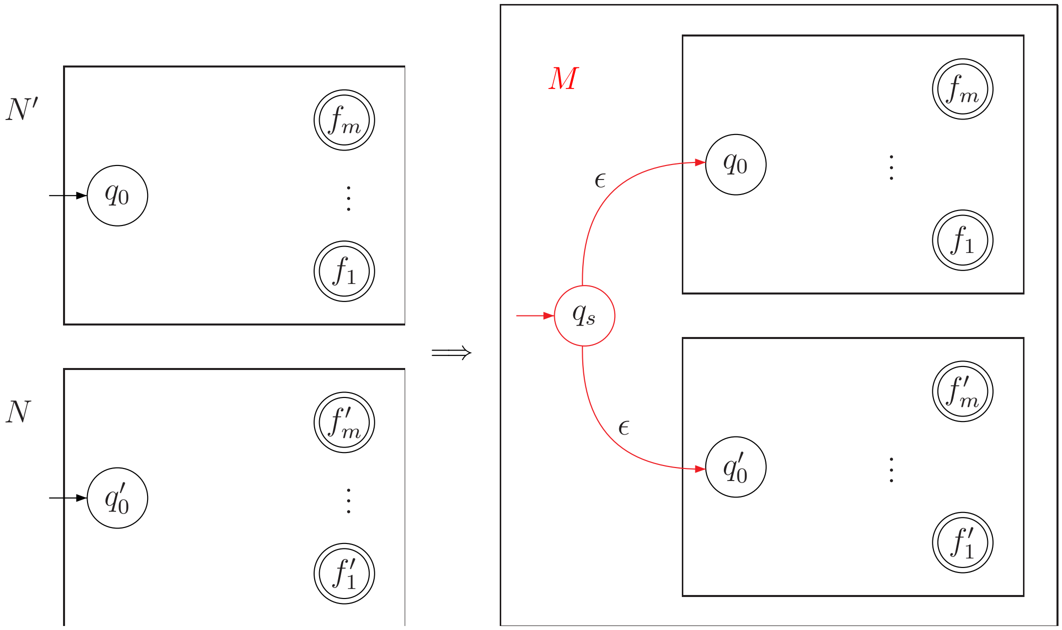
6.3 Closure of NFAs under regular operations

We consider the ***regular operations*** to be union, concatenation, and the star operator.

Advice to instructor: Do the following constructions via pictures, and give detailed tuple notation for only one of them. **!!!**

6.3.1 NFA closure under union

Given two NFAs, say N and N, we would like to build an NFA for the language $L(N) \cup L(N)$. The idea is to create a new initial state q_s and connect it with an ϵ-transition to the two initial states of N and N. Visually, the resulting NFA M looks as follows.



Formally, we are given two NFAs $N = (Q, \Sigma, \delta, q_0, F)$ and $N' = (Q', \Sigma, \delta', q_0', F')$, where $Q \cap Q' = \emptyset$ and the new state q_s is not in Q or Q'. The new NFA M is

$$M = \left(Q \cup Q' \cup \{q_s\}, \Sigma, \delta_M, s_q, F \cup F'\right),$$

where

$$\delta_M(q, c) = \begin{cases} \delta(q, c) & q \in Q, c \in \Sigma_\epsilon \\ \delta'(q, c) & q \in Q', c \in \Sigma_\epsilon \\ \{q_0, q_0'\} & q = q_s, c = \epsilon \\ \emptyset & q = q_s, c \neq \epsilon. \end{cases}$$

We thus showed the following.

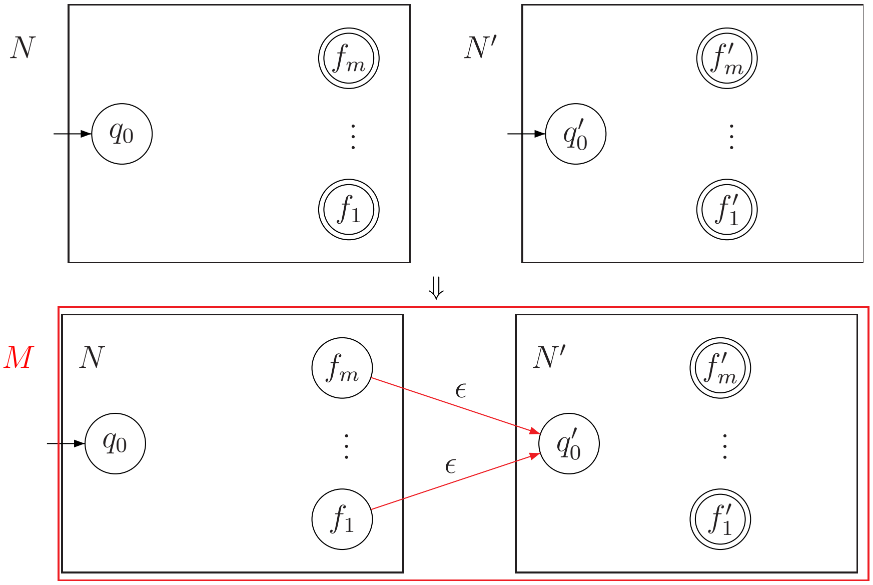
Lemma 6.3.1 *Given two NFAs N and N' one can construct an NFA M, such that $L(M) = L(N) \cup L(N')$.*

6.3.2 NFA closure under concatenation

Given two NFAs N and N', we would like to construct an NFA for the concatenated language $L(N) \circ L(N') = \left\{ xy \;\middle|\; x \in L(N) \text{ and } y \in L(N') \right\}$. The idea is to concatenate the two automatas, by connecting the final states of the first automata, by ϵ-transitions, into the start state of the second NFA. We also make the accepting states of N not-accepting. The idea is that in the resulting NFA M, given input w, it "guesses"

[1] This can be omitted, since it is implied by the (ii) rule.

how to break it into two strings $x \in L(N)$ and $y \in L(N')$, so that $w = xy$. Now, there exists an execution trace for N accepting x, then we can jump into the starting state of N' and then use the execution trace accepting y, to reach an accepting state of the new NFA M. Here is how visually the resulting automata looks like.



Formally, we are given two NFAs $N = (Q, \Sigma, \delta, q_0, F)$ and $N' = (Q', \Sigma, \delta', q'_0, F')$, where $Q \cap Q' = \emptyset$. The new automata is

$$M = (Q \cup Q', \Sigma, \delta_M, q_0, F'),$$

where

$$\delta_M(q, c) = \begin{cases} \delta(q, \epsilon) \cup \{q'_0\} & q \in F, c = \epsilon \\ \delta(q, c) & q \in F, c \neq \epsilon \\ \delta(q, c) & q \in Q \setminus F, c \in \Sigma_\epsilon \\ \delta'(q, c) & q \in Q', c \in \Sigma_\epsilon. \end{cases}$$

Lemma 6.3.2 *Given two NFAs N and N' one can construct an NFA M, such that $L(M) = L(N) \circ L(N') = L(N)L(N')$.*

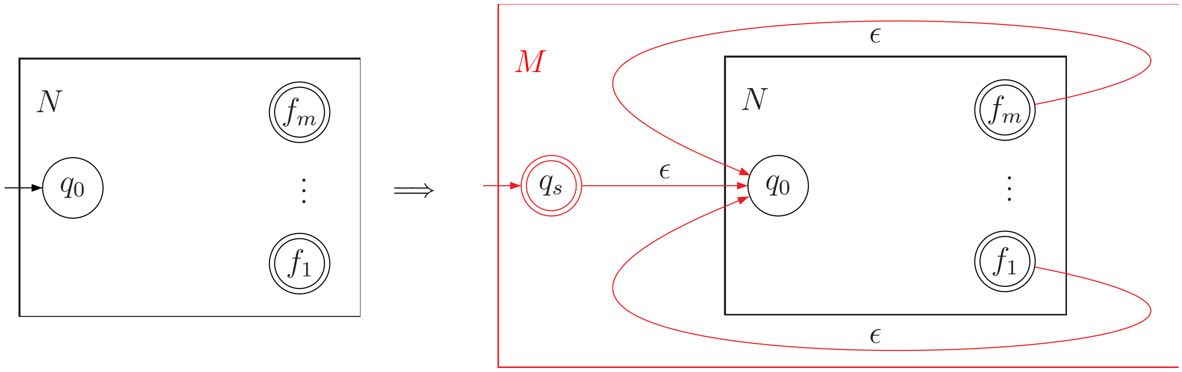
Proof: The construction is described above, and the proof of the correctness (of the construction) is easy and sketched above, so we skip it. You might want to verify that you know how to fill in the details for this proof (wink, wink). ■

6.3.3 NFA closure under the (Kleene) star

We are given a NFA N, and we would like to build an NFA for the Kleene star language

$$(L(N))^* = \left\{ w_1 w_2 \ldots w_k \;\middle|\; w_1, \ldots, w_k \in L(N), k \geq 0 \right\}.$$

The idea is to connect the final states of N back to the initial state using ϵ-transitions, so that it can loop back after recognizing a word of $L(N)$. As such, in the ith loop, during the execution, the new NFA M recognized the word w_i. Naturally, the NFA needs to guess when to jump back to the start state of N. One minor technicality, is that $\epsilon \in (L(N))^*$, but it might not be in $L(N)$. To overcome this, we introduce a new start state q_s (which is accepting), and its connected by (you guessed it) an ϵ-transition to the initial state of N. This way, $\epsilon \in L(M)$, and as such it recognized the required language. Visually, the transformation looks as follows.



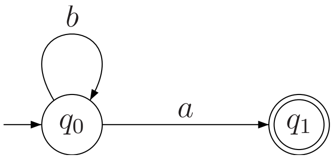
Formally, we are given the NFA $N = (Q, \Sigma, \delta, q_0, F)$, where $q_s \notin Q$. The new NFA is

$$M = \Big(\, Q \cup \{q_s\}\,,\ \Sigma,\ \delta_M,\ q_s,\ F \cup \{q_s\}\,\Big)\,,$$

where

$$\delta_M(q, c) = \begin{cases} \delta(q, \epsilon) \cup \{q_0\} & q \in F, c = \epsilon \\ \delta(q, \epsilon) & q \in F, c \neq \epsilon \\ \delta(q, c) & q \in Q \setminus F \\ \{q_0\} & q = q_0, c = \epsilon \\ \emptyset & q = q_0, c \neq \epsilon. \end{cases}$$

Why the extra state? The construction for star needs some explanation. We add arcs from final states back to initial state to do the loop. But then we need to ensure that ϵ is accepted. It's tempting to just make the initial state final, but this doesn't work for examples like the following. So we need to add a new initial state to handle ϵ.



Notice that it also works to send the loopback arcs to the new initial state rather than to the old initial state.

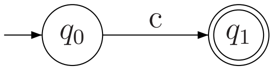
Lemma 6.3.3 *Given an NFA N, one can construct an NFA M that accepts the language $(L(N))^*$.*

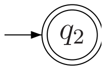
6.3.4 Translating regular expressions into NFAs

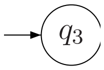
Lemma 6.3.4 *For every regular expression R over alphabet Σ, there is a NFA N_R such that $L(R) = L(N_R)$.*

Proof: The proof is by induction on the structure of R (can be interpreted as induction over the number of operators in R)

The base of the induction is when R contains no operator (i.e., the number operators in R is zero), then R must be one of the following:

(i) If $R = c$, where $c \in \Sigma$, then the corresponding NFA is .

(ii) If $R = \epsilon$ then the corresponding NFA is .

(iii) If $R = \emptyset$ then the corresponding NFA is .

As for induction step, assume that we proved the claim for all expressions having at most $k-1$ operators, and R has k operators in it. We consider if R can be written in any of the following forms:

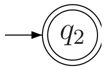
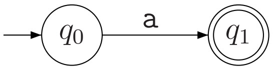
(i) $R = R_1 + R_2$. By the induction hypothesis, there exists two NFAs N_1 and N_2 such that $L(N_1) = L(R_1)$ and $L(N_2) = L(R_2)$. By Lemma 6.3.1, there exists an NFA M that recognizes the union; that is $L(M) = L(N_1) \cup L(N_2) = L(R_1) \cup L(R_2) = L(R)$.

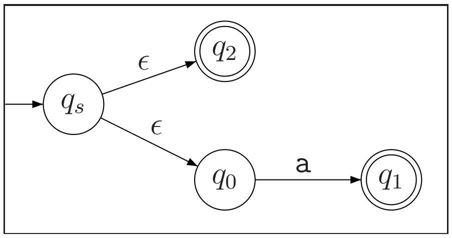
(ii) $R = R_1 \circ R_2 \equiv R_1 R_2$. By the induction hypothesis, there exists two NFAs N_1 and N_2 such that $L(N_1) = L(R_1)$ and $L(N_2) = L(R_2)$. By Lemma 6.3.2, there exists an NFA M that recognizes the concatenated language; that is, $L(M) = L(N_1) \circ L(N_2) = L(R_1) \circ L(R_2) = L(R)$.

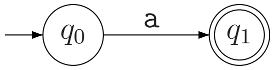
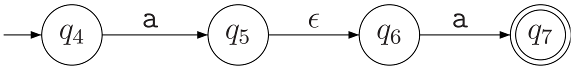
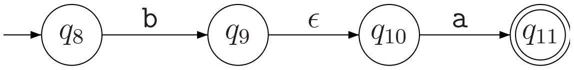
(iii) $R = (R_1)^*$. By the induction hypothesis, there exists a NFA N_1, such that $L(N_1) = L(R_1)$. By Lemma 6.3.3, there exists an NFA M that recognizes the star language; that is, $L(M) = (L(N_1))^* = (L(R_1))^* = L(R)$.

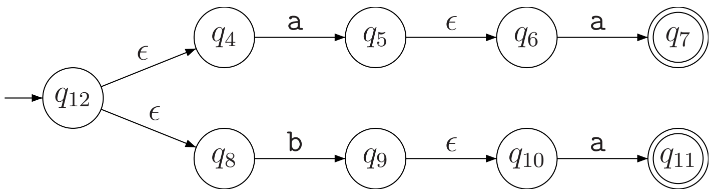
This completes the proof of the lemma, since we showed for all possible regular expressions with k operators how to build a NFA for them. ■

Example: From regular expression into NFA

Consider the regular expression $R = (\mathtt{a} + \epsilon)(\mathtt{aa} + \mathtt{ba})^*$. We have that $R = R_1 \circ R_2$, where $R_1 = \mathtt{a} + \epsilon$ and $R_2 = (\mathtt{aa} + \mathtt{ba})^*$. Let use first build an NFA for $R_1 = \mathtt{a} + \epsilon$. The NFA for ϵ is . and for a is . By Lemma 6.3.1, the NFA for their union, and thus of R_1, is



Now, $R_2 = (R_3)^*$, where $R_3 = \mathtt{aa} + \mathtt{ba}$. The NFA for a is , and as such the NFA for aa is , by Lemma 6.3.2. Similarly, the NFA for ba is . As such, by Lemma 6.3.1, the NFA for $R_3 = \mathtt{aa} + \mathtt{ba}$ is



By Lemma 6.3.3, the NFA for $R_2 = (R_3)^*$ is

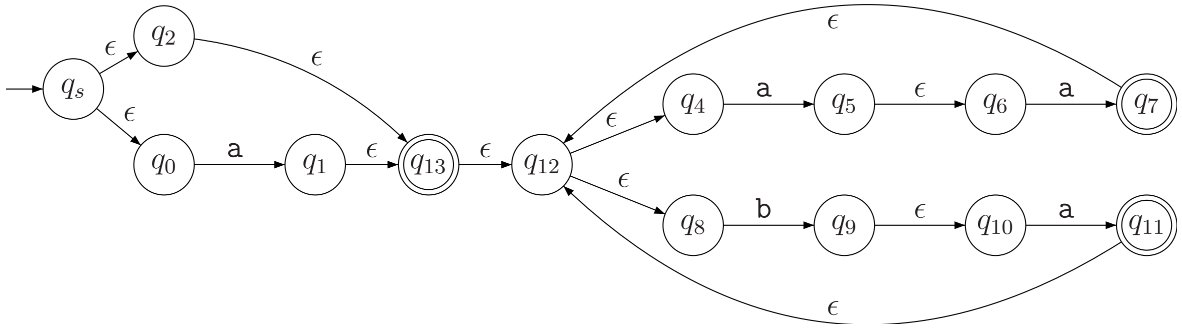
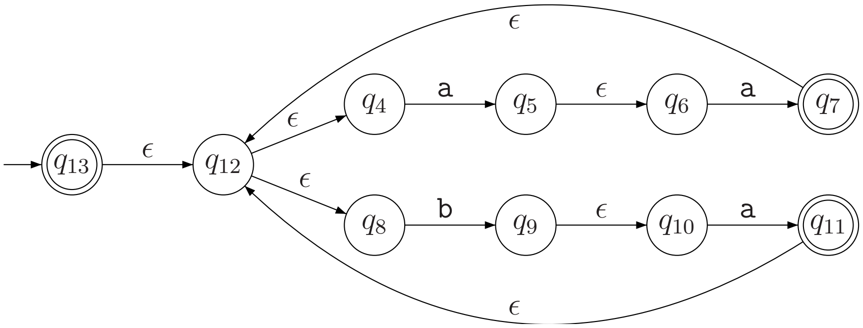


Figure 6.1: The NFA constructed for the regular expression $R = (\mathtt{a} + \epsilon)(\mathtt{aa} + \mathtt{ba})^*$.



Now, $R = R_1R_2 = R_1 \circ R_2$, and by Lemma 6.3.2, the NFA for R is depicted in Figure 6.1.

Note, that the resulting NFA is by no way the simplest and more elegant NFA for this language (far from it), but rather the NFA we get by following our construction carefully.

Chapter 7

Lecture 7: NFAs are equivalent to DFAs

10 February 2009

7.1 From NFAs to DFAs

7.1.1 NFA handling an input word

For the NFA $N = (Q, \Sigma, \delta, q_0, F)$ that has no ϵ-transitions, let us define $\Delta_N(X, c)$ to be the set of states that N might be in, if it was in a state of $X \subseteq Q$, and it handled the input c. Formally, we have that

$$\Delta_N(X, c) = \bigcup_{x \in X} \delta(x, c).$$

We also define $\Delta_N(X, \epsilon) = X$. Given a word $w = w_1, w_2, \ldots, w_n$, we define

$$\Delta_N(X, w) = \Delta_N\Big(\Delta_N(X, w_1 \ldots w_{n-1}), w_n\Big) = \Delta_N(\Delta_N(\ldots \Delta_N(\Delta_N(X, w_1), w_2) \ldots), w_n).$$

That is, $\Delta_N(X, w)$ is the set of all the states N might be in, if it starts from a state of X, and it handles the input w.

The proof of the following lemma is by an easy induction on the length of w.

Lemma 7.1.1 *Let $N = (Q, \Sigma, \delta, q_0, F)$ be a given NFA with no ϵ-transitions. For any word $w \in \Sigma^*$, we have that $q \in \Delta_N(\{q_0\}, w)$, if and only if, there is a way for N to be in q after reading w (when starting from the start state q_0).*

More details. We include the proof for the sake of completeness, but the reader should by now be able to fill in such a proof on their own.

Proof: The proof is by induction on the length of $w = w_1 w_2 \ldots w_k$.

If $k = 0$ then w is the empty word, and then N stays in q_0. Also, by definition, we have $\Delta_N(\{q_0\}, w) = \{q_0\}$, and the claim holds in this case.

Assume that the claim holds for all word of length at most n, and let $k = n + 1$ be the length of w. Consider a state q_{n+1} that N reaches after reading $w_1 w_2 \ldots w_n w_{n+1}$, and let q_n be the state N was before handling the character w_{n+1} and reaching q_{n+1}. By induction, we know that $q_n \in \Delta_N(\{q_0\}, w_1 w_2 \ldots w_n)$. Furthermore, we know that $q_{n+1} \in \delta(q_n, w_{n+1})$. As such, we have that

$$\begin{aligned} q_{n+1} \in \delta(q_n, w_{n+1}) &\subseteq \bigcup_{q \in \Delta_N(\{q_0\}, w_1 w_2 \ldots w_n)} \delta(q, w_{n+1}) \\ &= \Delta_N(\Delta_N(\{q_0\}, w_1 w_2 \ldots w_n), w_{n+1}) = \Delta_N(\{q_0\}, w_1 w_@ \ldots w_{n+1}) \\ &= \Delta_N(\{q_0\}, w). \end{aligned}$$

Thus, $q_{n+1} \in \Delta_N(\{q_0\}, w)$.

As for the other direction, if $p_{n+1} \in \Delta_N(\{q_0\}, w)$, then there must exist a state $p_n \in \Delta_N(\{q_0\}, w_1 \ldots w_n)$, such that $p_{n+1} \in \delta(p_n, w_{n+1})$. By induction, this implies that there is execution trace for N starting at q_0 and ending at p_n, such that N reads $w_1 \ldots w_n$ to reach p_n. As such, appending the transition from p_n to p_{n+1} (that read the character w_{n+1} to this trace, results in a trace for N that starts at q_0, reads w, and end up in the state p_{n+1}.

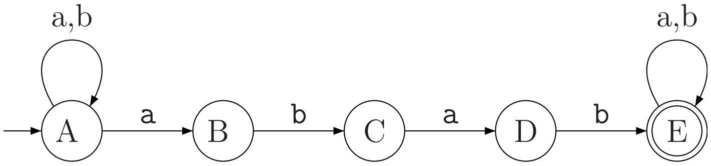
Putting these two arguments together, imply the claim. ■

7.1.2 Simulating NFAs with DFAs

One possible way of thinking about simulating NFAs is to consider each state to be a “light” that can be either on or off. In the beginning, only the initial state is on. At any point in time, all the states that the NFA might be in are turned on. As a new input character arrives, we need to update the states that are on.

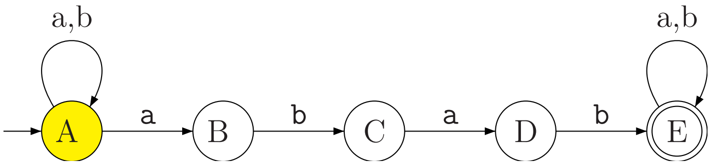
As a concrete examples, consider the automata below (which you had seen before), that accepts strings containing the substring `abab`.

(N1)



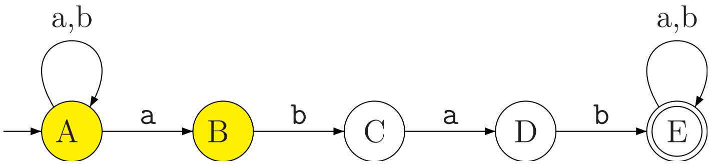
Let us run an explicit search for the above NFA (N1) on the input string `ababa`.

$t = 0$:



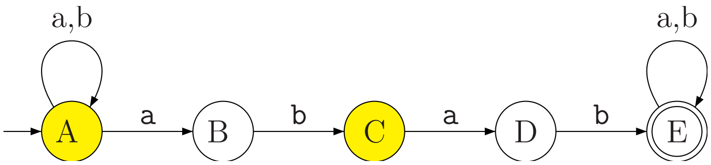
Remaining input: `ababa`.

$t = 1$:



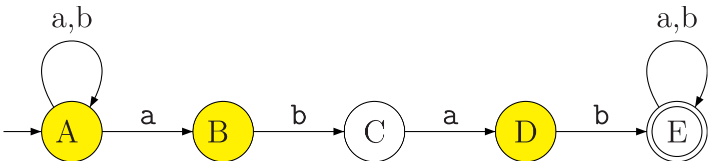
Remaining input: `baba`.

$t = 2$:



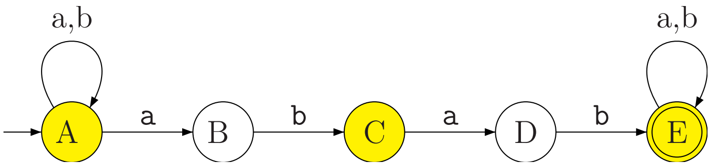
Remaining input: `aba`.

$t = 3$:



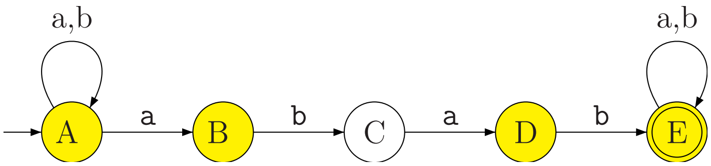
Remaining input: `ba`.

$t = 4$:



Remaining input: `a`.

$t = 5$:



Remaining input: ϵ.

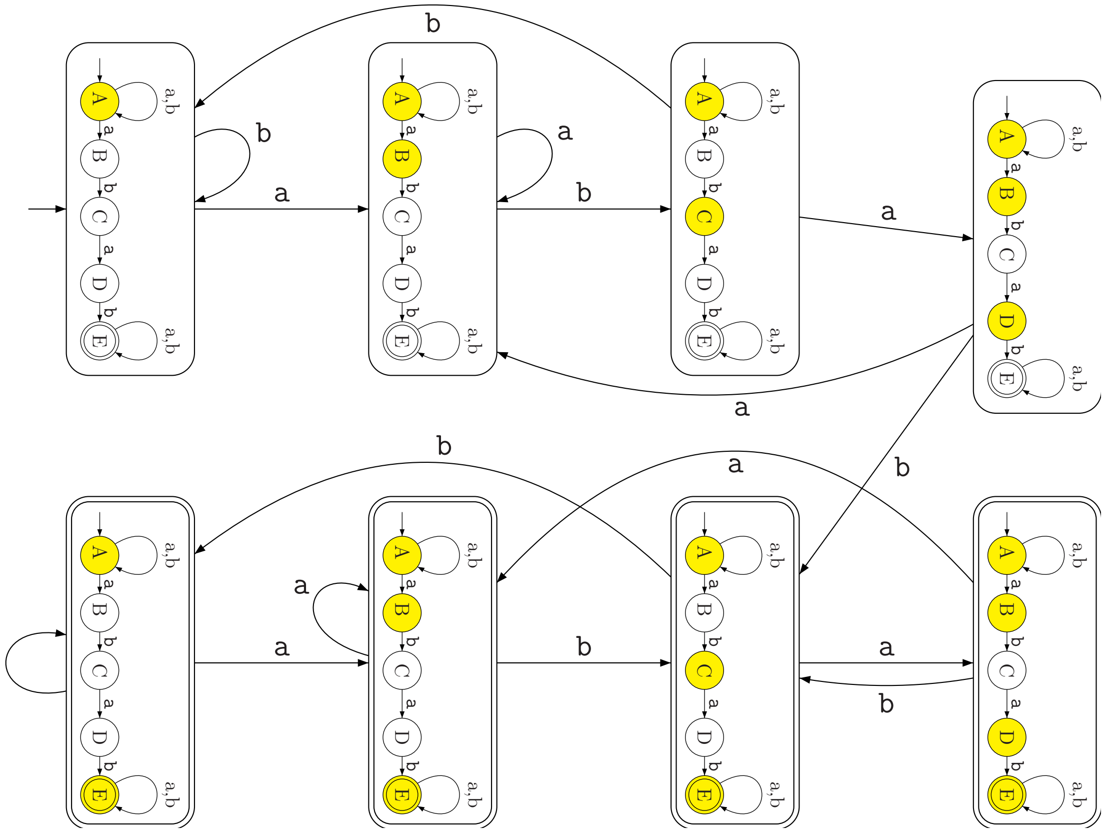


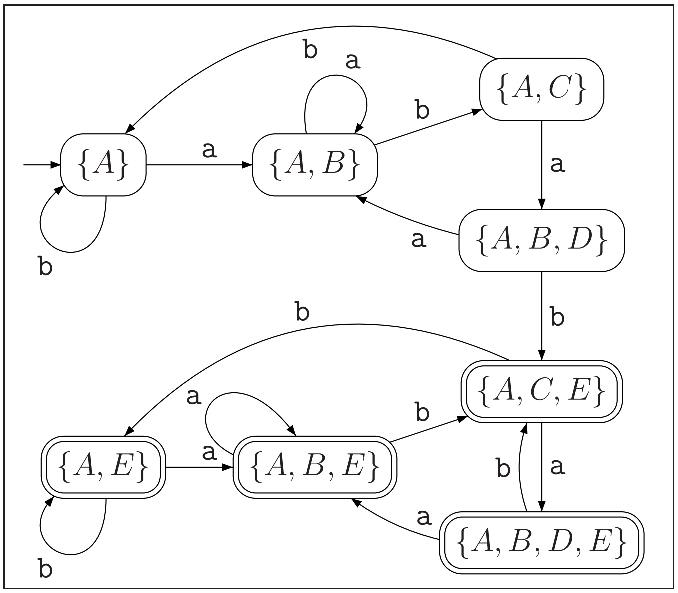
Figure 7.1: The resulting DFA

Note, that (N1) accepted `ababa` because when its done reading the input, the accepting state is on.

This provide us with a scheme to simulate this NFA with a DFA: (i) Generate all possible configurations of states that might be turned on, and (ii) decide for each configuration what is the next configuration, what is the next configuration. In our case, in all configurations the first state is turned on. The initial configuration is when only state A is turned on. If this sounds familiar, it should, because what you get is just a big nasty, hairy DFA, as shown on the last page of this class notes. The same DFA with the unreachable states removed is shown in Figure 7.1.

Every state in the DFA of Figure 7.1 can be identified by the subset of the original states that is turned on (namely, the original automata might be any of these states).

Thus, a more conventional drawing of this automata is shown on the right.



Thus, to convert an NFA N with a set of states Q into a DFA, we consider all the subsets of Q that N might be realized as. Namely, every subset of Q (i.e., a member of $\mathbb{P}(Q)$ – the power set of Q) is going to be a state in the new automata. Now, consider a subset $X \subseteq Q$, and for every input character $c \in \Sigma$, let us figure out in what states the original NFA N might be in if it is in one of the states of X, and it handles the characters c. Let Y be the resulting set of such states.

Clearly, we had just computed the transition function of the new (equivalent) DFA, showing that if the NFA is in one of the states of X, and we receive c, then the NFA now might be in one of the states of Y.

Now, if the initial state of the NFA N is q_0, then the new DFA M_{DFA} would start with the state (i.e., configuration) $\{q_0\}$ (since the original NFA might be only in q_0 at this point in time).

Its important that our simulation is *faithful*: At any point in time, if we are in state X in M_{DFA} then there is a path in the original NFA N, with the given input, to reach each state of Q that is in X (and similarly, X includes all the states that are reachable with such an input).

When does M_{DFA} accepts? Well, if it is in state X (here $X \subseteq Q$), then it accepts only if X includes one of the accepting states of the original NFA N.

Clearly, the resulting DFA M_{DFA} is equivalent to the original NFA.

7.1.3 The construction of a DFA from an NFA

Let $N = (Q, \Sigma, \delta, q_0, F)$ be the given NFA that does not have any ϵ-transitions. The new DFA is going to be

$$M_{\mathsf{DFA}} = \left(\mathbb{P}(Q)\,, \Sigma, \widehat{\delta}, \widehat{q_0}, \widehat{F}\right),$$

where $\mathbb{P}(Q)$ is the power set of Q, and $\widehat{\delta}$ (the transition function), $\widehat{q_0}$ the initial state, and the set of accepting states $\widehat{F}$ are to be specified shortly. Note that the states of M_{DFA} are subsets of Q (which is slightly confusing), and as such the starting state of M_{DFA}, is $\widehat{q_0} = \{q_0\}$ (and not just q_0).

We need to specify the transition function, so consider $X \in \mathbb{P}(Q)$ (i.e., $X \subseteq Q$), and a character c. For a state $s \in X$, the NFA might go into any state in $\delta(s, c)$ after reading q. As such, the set of all possible states the NFA might be in, if it started from a state in X, and received c, is the set

$$Y = \bigcup_{s \in X} \delta(s, c).$$

As such, the transition of M_{DFA} from X receiving c is the state of M_{DFA} defined by Y. Formally,

$$\widehat{\delta}(X, c) = Y = \bigcup_{s \in X} \delta(s, c). \tag{7.1}$$

As for the accepting states, consider a state $X \in \mathbb{P}(Q)$ of M_{DFA}. Clearly, if there is a state of F in X, then X is an accepting state; namely, $F \cap X \neq \emptyset$. Thus,

$$\widehat{F} = \left\{ X \;\middle|\; X \in \mathbb{P}(Q)\,, X \cap F \neq \emptyset \right\}.$$

Proof of correctness

Claim 7.1.2 *For any $w \in \Sigma^*$, the set of states reached by the NFA N on w is precisely the state reached by M_{DFA} on w. That is $\Delta_N(\{q_0\}, w) = \widehat{\delta}(\{q_0\}, w)$.*

Proof: The proof is by induction on the length of w.

If w is the empty word, then N is at q_0 after reading ϵ (i.e., $\Delta_N(\{q_0\}, \epsilon) = \{q_0\}$), and the M_{DFA} is still in its initial state which is $\{q_0\}$.

So assume that the claim holds for all words of length at most k.

Let $w = w_1 w_2 \ldots w_{k+1}$. Let X be the set of states that N might reach from q_0 after reading $w' = w_1 \ldots w_n$; that is $X = \Delta_N(\{q_0\}, w')$. By the induction hypothesis, we have that M_{DFA} is in the state X after reading w' (formally, we have that $\widehat{\delta}(\{q_0\}, w') = X$).

Now, the NFA N, when reading the last character w_{k+1}, can start from any state of X, and use any transition from such a state that reads the character w_{k+1}. Formally, the NFA N is in one of the states of

$$Z = \Delta_N(X, w_{k+1}) = \bigcup_{s \in X} \delta(s, w_{k+1}).$$

Similarly, by the definition of M_{DFA}, we have that from the state X, after reading w_{k+1}, the DFA M_{DFA} is in the state

$$Y = \widehat{\delta}(X, w_{k+1}) = \bigcup_{s \in X} \delta(s, w_{k+1}),$$

see Eq. (7.1). But clearly, $Z = Y$, which establishes the claim. ■

Lemma 7.1.3 *Any NFA N, without ϵ-transitions, can be converted into a DFA M_{DFA}, such that M_{DFA} accepts the same language as N.*

Proof: The construction is described above.

So consider a word $w \in \Sigma^*$, and observe that $w \in L(N)$ if and only if, the set of states N might be in after reading w (that is $\Delta_N(\{q_0\}, w)$), contains an accepting state of N. Formally, $w \in L(N)$ if and only if

$$\Delta_N\Big(\{q_0\}, w\Big) \cap F \neq \emptyset.$$

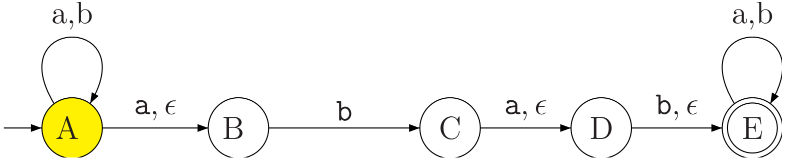
The DFA M_{DFA} is in the state $\widehat{\delta}(\{q_0\}, w)$ after reading w. Claim 7.1.2, implies that $Y = \widehat{\delta}(\{q_0\}, w) = \Delta_N(\{q_0\}, w)$. By construction, the M_{DFA} accepts at this state, if and only if, $Y \in \widehat{F}$, which equivalent to that Y contains a final state of N. That is $Y \cap F \neq \emptyset$. Namely, M_{DFA} accepts w if

$$\widehat{\delta}\Big(\{q_0\}, w\Big) \cap F \neq \emptyset \iff \Delta_N\Big(\{q_0\}, w\Big) \cap F \neq \emptyset.$$

Implying that M_{DFA} accepts w if and only if N accepts w. ■

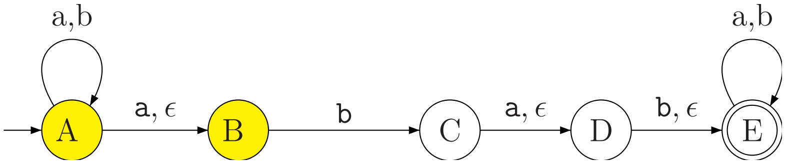
Handling ϵ-transitions

Now, we would like to handle a general NFA that might have ϵ-transitions. The problem is demonstrated in the following NFA in its initial configuration:

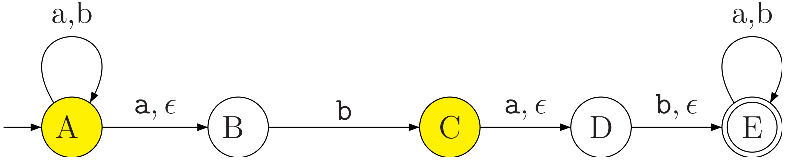


Clearly, the initial configuration here is $\{A, B\}$ (and not the one drawn above), since the automata can immediately jump to B if the NFA is already in A. So, the configuration $\{A\}$ should not be considered at all. As such, the true initial configuration for this automata is

(N2)

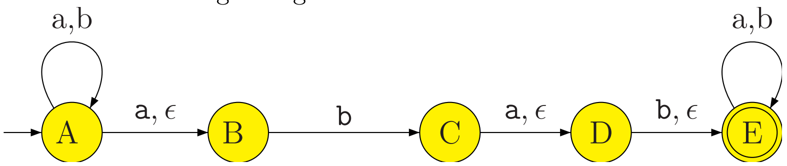


Next, consider the following more interesting configuration.



But here, not only we can jump from A to B, but we can also jump from C to D, and from D to E. As such, this configuration is in fact the following configuration

(N3)



In fact, this automata can only be in these two configurations because of the ϵ-transitions.

So, let us formalize the above idea: Whenever the NFA N might be in a state s, we need to extend the configuration to all the states of the NFA reachable by ϵ-transitions from s. Let $R_\epsilon(s)$ denote the set of all states of N that are reachable by a sequence of ϵ-transitions from s (s is also in $R_\epsilon(s)$ naturally, since we can reach s without moving anywhere).

Thus, if N might be any state of $X \subseteq Q$, then it might be in any state of

$$\mathcal{E}(X) = \bigcup_{s \in X} R_\epsilon(s) .$$

As such, whenever we consider the set of states X for Q, in fact, we need to consider the *extended set of states* $\mathcal{E}(X)$. As such, for the above automata, we have

$$\mathcal{E}(\{A\}) = \{A, B\} \quad \text{and} \quad \mathcal{E}(\{A, C\}) = \{A, B, C, D, E\} .$$

Now, we can essentially repeat the above proof.

Theorem 7.1.4 *Any NFA N (with or without ϵ-transitions) can be converted into a DFA M_{DFA}, such that M_{DFA} accepts the same language as N.*

Proof: Let $N = (Q, \Sigma, \delta, q_0, F)$. The new DFA is going to be

$$M_{\mathsf{DFA}} = \Big(\mathbb{P}(Q), \Sigma, \delta_M, q_S, \widehat{F}\Big) .$$

Here, $\mathbb{P}(Q)$, Σ and $\widehat{F}$ are the same as above.

Now, for $X \in \mathbb{P}(Q)$ and $c \in \Sigma$, let

$$\delta_M(X) = \mathcal{E}\Big(\widehat{\delta}(X, c)\Big) ,$$

where $\widehat{\delta}$ is the old transition function from the proof of Lemma 7.1.3; namely, we always extend the new set of states to include all the states we can reach by ϵ-transitions. Similarly, the initial state is now

$$q_S = \mathcal{E}(\{q_0\}) .$$

It is now straightforward to verify that the new DFA is indeed equivalent to the original NFA, using the argumentation of Lemma 7.1.3. ■

Chapter 8

Lecture 8: From DFAs/NFAs to Regular Expressions

12 February 2009

In this lecture, we will show that any DFA can be converted into a regular expression. Our construction would work by allowing regular expressions to be written on the edges of the DFA, and then showing how one can remove states from this generalized automata (getting a new equivalent automata with the fewer states). In the end of this state removal process, we will remain with a generalized automata with a single initial state and a single accepting state, and it would be then easy to convert it into a single regular expression.

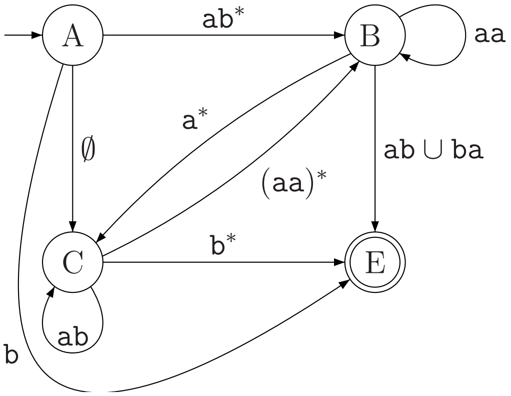
8.1 From NFA to regular expression

8.1.1 GNFA— A Generalized NFA

Consider an NFA N where we allowed to write any regular expression on the edges, and not only just symbols. The automata is allowed to travel on an edge, if it can matches a prefix of the unread input, to the regular expression written on the edge. We will refer to such an automata as a ***GNFA*** (***generalized non-deterministic finite automata*** [Don't you just love all these shortcuts?]).

Thus, the GNFA on the right, accepts the string abbbbaaba, since

$$A \xrightarrow{\texttt{abbbb}} B \xrightarrow{\texttt{aa}} B \xrightarrow{\texttt{ba}} E.$$



To simplify the discussion, we would enforce the following conditions:

(C1) There are transitions going from the initial state to all other states, and there are no transitions into the initial state.

(C2) There is a single accept state that has only transitions coming into it (and no outgoing transitions).

(C3) The accept state is distinct from the initial state.

(C4) Except for the initial and accepting states, all other states are connected to all other states via a transition. In particular, each state has a transition to itself.

When you can not actually go between two states, a GNFA has a transitions labelled with $\emptyset$, which will not match any string of input characters. We do not have to draw these transitions explicitly in the state diagrams.

8.1.2 Top-level outline of conversion

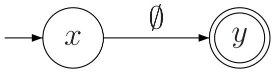
We will convert a DFA to a regular expression as follows:

(A) Convert DFA to a GNFA, adding new initial and final states.

(B) Remove all states one-by-one, until we have only the initial and final states.

(C) Output regex is the label on the (single) transition left in the GNFA. (The word ***regex*** is just a shortcut for regular expression.)

Lemma 8.1.1 *A DFA M can be converted into an equivalent GNFA G.*

Proof: We can consider M to be an NFA. Next, we add a special initial state q_{init} that is connected to the old initial state via ϵ-transition. Next, we add a special final state q_{final}, such that all the final states of M are connected to q_{final} via an ϵ-transition. The modified NFA M' has an initial state and a single final state, such that no transition enters the initial state, and no transition leaves the final state, thus M' comply with conditions (C1–C3) above. Next, we consider all pair of states $x, y \in Q(M')$, and if there is no transition between them, we introduce the transition 

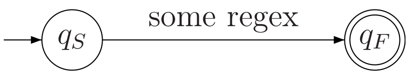
. The resulting GNFA G from M' is now compliant also with condition (C4).

It is easy now to verify that G is equivalent to the original DFA M. ■

We will remove all the intermediate states from the GNFA, leaving a GNFA with only initial and final states, connected by one transition with a (typically complex) label on it. The equivalent regular expression is obvious: the label on the transition.

Lemma 8.1.2 *Given a GNFA N with $k = 2$ states, one can generate an equivalent regular expression.*

Proof: A GNFA with only two states (that comply with conditions (C1)-(C4)) have the following form.



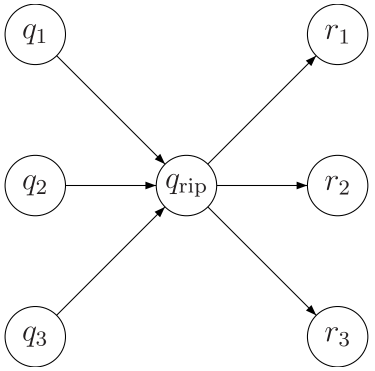
The GNFA has a single transition from the initial state to the accepting state, and this transition has the regular expression R associated with it. Since the initial state and the accepting state do not have self loops, we conclude that N accepts all words that matches the regular expression R. Namely, $L(N) = L(R)$. ■

8.1.3 Details of ripping out a state

We first describe the construction. Since $k > 2$, there is at least one state in N which is not initial or accepting, and let q_{rip} denote this state. We will "rip" this state out of N and fix the GNFA, so that we get a GNFA with one less state.

Transition paths going through q_{rip} might come from any of a variety of states q_1, q_2, etc. They might go from q_{rip} to any of another set of states r_1, r_2, etc.

For each pair of states q_i and r_i, we need to convert the transition through q_{rip} into a direct transition from q_i to r_i.

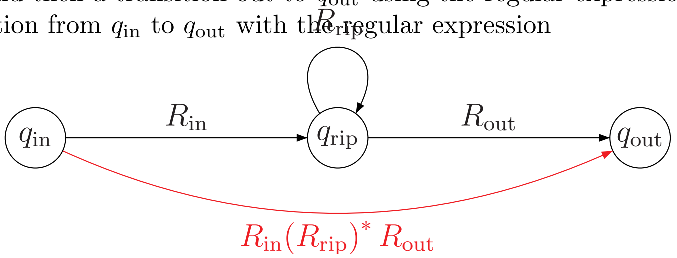


Reworking connections for specific triple of states

To understand how this works, let us focus on the connections between q_{rip} and two other specific states q_{in} and q_{out}. Notice that q_{in} and q_{out} might be the same state, but they both have to be different from q_{rip}.

The state q_{rip} has a self loop with regular expression R_{rip} associated with it. So, consider a fragment of an accepting trace that goes through q_{rip}. It transition into q_{rip} from a state q_{in} with a regular expression R_{in} and travels out of q_{rip} into state q_{out} on an edge with the associated regular expression being R_{out}. This trace, corresponds to the regular expression R_{in} followed by 0 or more times of traveling on the self loop (R_{rip} is used each time we traverse the loop), and then a transition out to q_{out} using the regular expression R_{out}. As such, we can introduce a direct transition from q_{in} to q_{out} with the regular expression

$$R = R_{\text{in}}(R_{\text{rip}})^* \, R_{\text{out}}.$$



Clearly, any fragment of a trace traveling $q_{\text{in}} \to q_{\text{rip}} \to q_{\text{out}}$ can be replaced by the direct transition $q_{\text{in}} \xrightarrow{R} q_{\text{out}}$. So, let us do this replacement for any two such stages, we connect them directly via a new transition, so that they no longer need to travel through q_{rip}.

Clearly, if we do that for all such pairs, the new automata accepts the same language, but no longer need to use q_{rip}. As such, we can just remove q_{rip} from the resulting automata. And let M' denote the resulting automata.

The automata M' is not quite legal, yet. Indeed, we will have now parallel transitions because of the above process (we might even have parallel self loops). But this is easy to fix: We replace two such parallel transitions $q_i \xrightarrow{R_1} q_j$ and $q_i \xrightarrow{R_2} q_j$, by a single transition

$$q_i \xrightarrow{R_1 + R_2} q_j.$$

As such, for the triple $q_{\text{in}}, q_{\text{rip}}, q_{\text{out}}$, if the original label on the direct transition from q_{in} to q_{out} was originally R_{dir}, then the output label for the new transition (that skips q_{rip}) will be

$$R_{dir} \; + \; R_{\text{in}}(R_{\text{rip}})^* \, R_{\text{out}}. \qquad (8.1)$$

Clearly the new transition, is equivalent to the two transitions it replaces. If we repeat this process for all the parallel transitions, we get a new GNFA M which has $k-1$ states, and furthermore it accepts exactly the same language as N.

8.1.4 Proof of correctness of the ripping process

Lemma 8.1.3 *Given a GNFA N with $k > 2$ states, one can generate an equivalent GNFA M with $k-1$ states.*

Proof: Since $k > 2$, N contains least one state in N which is not accepting, and let q_{rip} denote this state. We will "rip" this state out of N and fix the GNFA, so that we get a GNFA with one less state.

For every pair of states q_{in} and q_{out}, both distinct from q_{rip}, we replace the transitions that go through q_{rip} with direct transitions from q_{in} to q_{out}, as described in the previous section.

Correctness. Consider an accepting trace T for N for a word w. If T does not use the state q_{rip} than the same trace exactly is an accepting trace for M. So, assume that it uses q_{rip}, in particular, the trace looks like

$$T = \ldots q_i \xrightarrow{S_i} q_{\text{rip}} \overbrace{\xrightarrow{S_{i+1}} q_{\text{rip}} \ldots \xrightarrow{S_{j-1}} q_{\text{rip}}}^{\text{0 or more times}} \xrightarrow{S_{j-1}} q_j \ldots.$$

Where $S_i S_{i+1} \ldots, S_j$ is a substring of w. Clearly, $S_i \in R_{\text{in}}$, where R_{in} is the regular expression associated with the transition $q_i \to q_{\text{rip}}$. Similarly, $S_{j-1} \in R_{\text{out}}$, where R_{out} is the regular expression associated with the transition $q_{\text{rip}} \to q_j$. Finally, $S_{i+1} S_{i+2} \cdots S_{j-1} \in (R_{\text{rip}})^*$, where R_{rip} is the regular expression associated with the self loop of q_{rip}.

Now, clearly, the string $S_i S_{i+1} \dots S_j$ matches the regular expression $R_{\text{in}}(R_{\text{out}})^* \, R_{\text{out}}$. in particular, we can replace this portion of the trace of T by

$$T = \dots q_i \xrightarrow{\mathtt{S_i S_{i+1} \dots S_{j-1} S_j}} q_j \dots .$$

This transition is using the new transition between q_i and q_j introduced by our construction. Repeating this replacement process in T till all the appearances of q_{rip} are removed, results in an accepting trace $\widehat{T}$ of M. Namely, we proved that any string accepted by N is also accepted by M.

We need also to prove the other direction. Namely, given an accepting trace for M, we can rewrite it into an equivalent trace of N which is accepting. This is easy, and done in a similar way to what we did above. Indeed, if a portion of the trace uses a new transition of M (that does not appear in N), we can place it by a fragment of transitions going through q_{rip}. In light of the above proof, it is easy and we omit the straightforward but tedious details. ■

Theorem 8.1.4 *Any DFA can be translated into an equivalent regular expression.*

Proof: Indeed, convert the DFA into a GNFA N. As long as N has more than two states, reduce its number of states by removing one of its states using Lemma 8.1.3. Repeat this process till N has only two states. Now, we convert this GNFA into an equivalent regular expression using Lemma 8.1.2. ■

8.1.5 Running time

This is a relatively inefficient algorithm. Nevertheless, it establishes the equivalence between the automata and regular expressions. Fortunately, it is a conversion that you rarely need to do in practical applications. Usually, the input would be the regex and the application would convert it *into* an NFA or DFA. Converting in that direction is more efficient.

To realize the problem, note that the algorithm for ripping a single state has three nested loops in it.

```
For every state $q_{\text{rip}}$ do
    For every incoming state $q_{\text{in}}$ do
        For every outgoing state $q_{\text{out}}$ do
            Remove all transition paths from $q_{\text{in}}$ to $q_{\text{out}}$ via $q_{\text{rip}}$ by creating a direct transition between
            $q_{\text{in}}$ and $q_{\text{out}}$.
```

So, if the original DFA has n states, then the algorithm will do the inner step $O(n^3)$ times (which is not too bad). Worse, each time we remove a state, we replace the regex on each remaining transition with a regex that is potentially four times as large. (That is, we replace the regular expression R_{dir} associated with a transition, by a regular expression $R_{dir} \; + \; R_{\text{in}}(R_{\text{rip}})^* \, R_{\text{out}}$, see Eq. $(8.1)_{\text{p60}}$.)

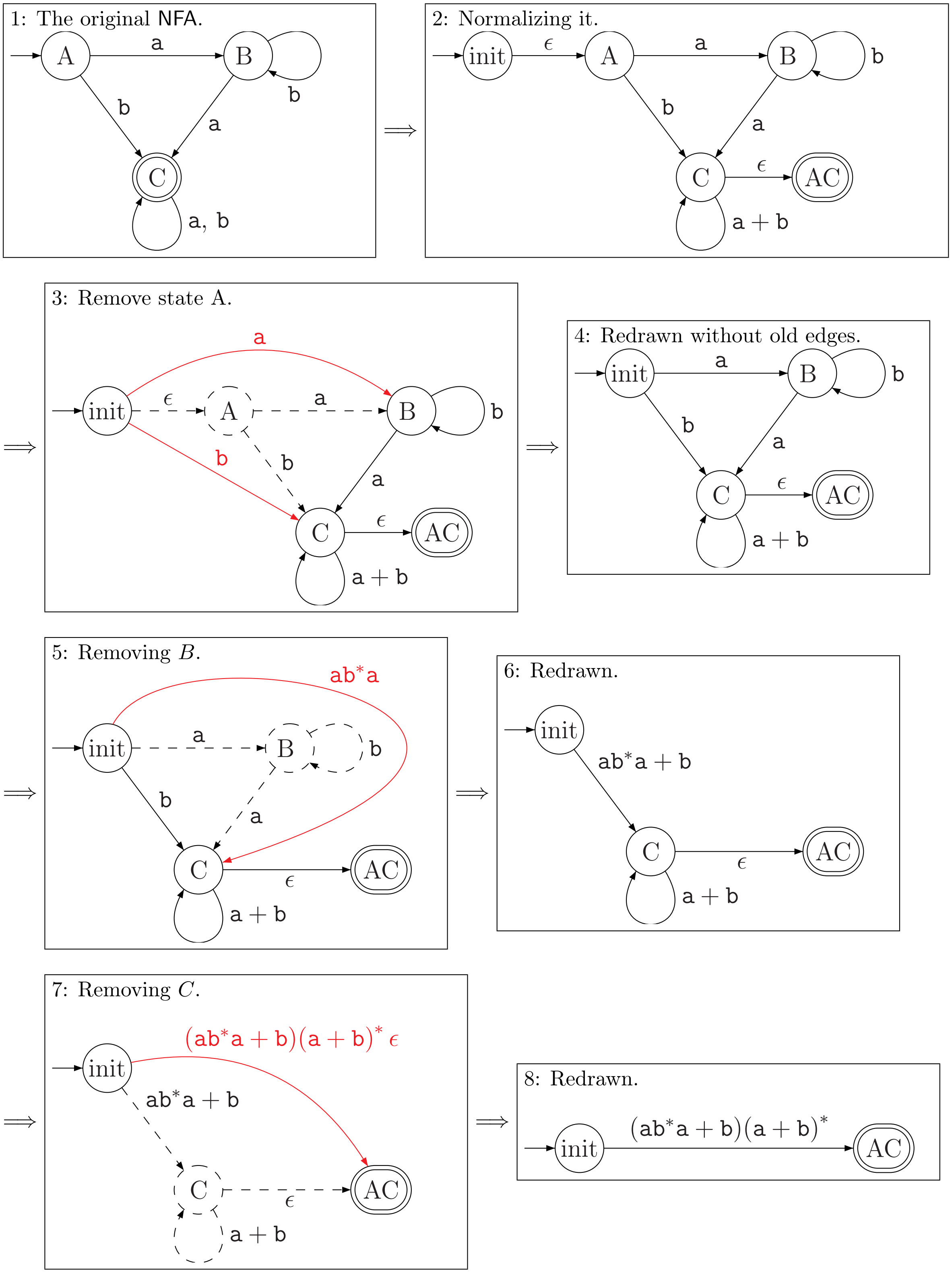
So, every time we rip a state in the GNFA, the length of the regular expressions associated with the edges of th GNFA get longer by a factor of four (at most). So, we repeat this n times, so the length of the final output regex is $O(4^n)$. And the actual running time of the algorithm is $O(n^3 4^n)$.

Typically output sizes and running times are not quite that bad. We really only need to consider triples of states that are connected by arcs with labels other than $\emptyset$. Many transitions are labelled with ϵ or $\emptyset$, so regular expression size often increases by less than a factor of 4. However, actual times are still unpleasant for anything but very small examples.

Interestingly, while this algorithm is not very efficient, it is not the algorithm "fault". Indeed, it is known that regular expressions for automata can be exponentially large. There is a lower bound of 2^n for regular expressions describing an automata of size n, see [EZ74] for details.

8.2 Examples

8.2.1 Example: From GNFA to regex in 8 easy figures



1: The original NFA.

2: Normalizing it.

3: Remove state A.

4: Redrawn without old edges.

5: Removing *B*.

6: Redrawn.

7: Removing *C*.

8: Redrawn.

Thus, this automata is equivalent to the regular expression $(\texttt{ab}^*\texttt{a} + \texttt{b})(\texttt{a} + \texttt{b})^*$.

8.3 Closure under homomorphism

Suppose that Σ and Γ are two alphabets (possibly the same, but maybe different). A ***homomorphism*** h is a function from Σ^* to Γ^* such that $h(xy) = h(x)h(y)$ for any strings x and y. Equivalently, if we divide w into a sequence of individual characters $w = c_1c_2\ldots c_k$, then $h(w) = h(c_1)h(c_2)\ldots h(c_k)$. (It's a nice exercise to prove that the two definitions are equivalent.)

Example 8.3.1 Let $\Sigma = \{a, b, c\}$ and $\Gamma = \{0, 1\}$, and let h be the mapping $h : \Sigma \to \Gamma$, such that $h(a) = 01$, $h(b) = 00$, $h(c) = \epsilon$. Clearly, h is a homomorphism.

So, suppose that we have a regular language L. If L is represented by a regular expression R, then it is easy to build a regular expression for $h(L)$. Just replace every character c in R by its image $h(c)$.

Example 8.3.2 The regular expression $R = (ac + b)^*$ over Σ becomes $h(R) = (01 + 00)^*$.

Lemma 8.3.3 *Let L Be a regular language over Σ, and let $h : \Sigma \to \Gamma$ be a homomorphism, then the language $h(L)$ is regular.*

Proof: (Informal.) Let R Be a regular expression for R. Replace any character $c \in \Sigma$ appearing in R by the string $h(c)$. Clearly, the resulting regular expression R' recognizes all the words in $h(L)$. ■

Proof:(More formal.) Let $\mathtt{D}$ be a NFA for L with a single accept state q_{final} and an initial state q_{init}, so that the only transitions from q_{init} is ϵ-transition out of it, and the is no outgoing transitions from q_{final} and only ϵ-transitions into it.

Now, replace every transition $q \xrightarrow{\mathtt{c}} q'$ in $\mathtt{D}$ by the transition $q \xrightarrow{\mathtt{h(c)}} q'$. Clearly, the resulting automata is a GNFA $\mathtt{C}$ that accepts the language $h(L)$. We showed in the previous lecture, that a GNFA can be converted into an equivalent regular expression R, such that $L(\mathtt{C}) = h(R)$. As such, we have that $h(L) = L(\mathtt{C}) = h(R)$. Namely, $h(L)$ is a regular language, as claimed. ■

Note, that in the above proof, instead of creating a GNFA, we can also create a NFA, by introducing temporary states. Thus, if we have the transition $q \xrightarrow{\mathtt{c}} q'$ in $\mathtt{D}$, and $h(c) = w_1w_2\ldots w_k$, then we will introduce new temporary states $s_1, \ldots s_{k-1}$, and replace the transition $q \xrightarrow{\mathtt{c}} q'$ by the transitions

$$q \xrightarrow{\mathtt{w_1}} s_1,\ s_1 \xrightarrow{\mathtt{w_2}} s_2,\ \ldots\ s_{k-2} \xrightarrow{\mathtt{w_{k-1}}} s_{k-1},\ s_{k-1} \xrightarrow{\mathtt{w_k}} q'.$$

Thus, we replace the transition $q \xrightarrow{\mathtt{c}} q'$ by a path between q and q' that accepts only the string $h(c)$. It is now pretty easy to argue that the language of the resulting NFA $\mathtt{C}$ is $h(L)$.

Note that when you have several equivalent representations, do your proofs in the one that makes the proof easiest. So we did set complement using DFAs, concatenation using NFAs, and homomorphism using regular expressions. Now we just have to finish the remaining bits of the proof that the three representations are equivalent.

An interesting point is that if a language L is note regular then $h(L)$ might be regular or not.

Example 8.3.4 Consider the language $L = \left\{ \mathtt{a}^n\mathtt{b}^n \,\middle|\, n \geq 0 \right\}$. The language L is not regular. Now, consider the homomorphism $h(\mathtt{a}) = \mathtt{a}$ and $h(\mathtt{b}) = \mathtt{a}$. Clearly, $h(L) = \left\{ \mathtt{a}^n\mathtt{a}^n = \mathtt{a}^{2n} \,\middle|\, n \geq 0 \right\}$, which is definitely regular. However, the identify homomorphism $I(\mathtt{a}) = \mathtt{a}$ and $I(\mathtt{b}) = \mathtt{b}$ maps L to itself $I(L) = L$, and as such $I(L)$ is not regular.

Intuitively, homomorphism can not make a language to be "harder" than it is (if it is regular, then it remains regular under homomorphism). However, if it is not regular, it might remain not regular.

Chapter 9

Lecture 9: Proving non-regularity

17 February 2009

Reminder: The first midterm Exam, takes place on Tuesday, 24 of February 7-9pm room in 151 Loomis. Be there. Please check for conflicts NOW. If you have one, send a note to Sariel or Madhu explaining the nature of the conflict and including your schedule.

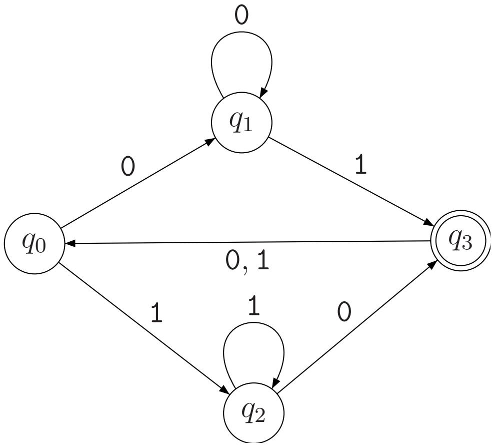
In this lecture, we will see how to prove that a language is **not** regular.

We will see two methods for showing that a language is not regular. The "pumping lemma" shows that certain key "seed" languages are not regular. From these seed languages, we can show that many similar languages are also not regular, using closure properties.

9.1 State and regularity

9.1.1 How to tell states apart

You are given the following DFA M, but we do not know what is its initial state (it was made in India, and the initial state indicator was broken during the shipment to the US). You want to figure out what is the initial state of this DFA.



You can do any of the following operations:

(i) Reset the DFA to its (unknown) initial position.

(ii) Feed input into the DFA.

The rule of the game is that when the DFA is in a final state, you would know it.

So, the question is how to decide in the above DFA what is the initial state?

Here is one possible solution.

1. Check if M is in already in a final state. If so q_3 is the initial state.

2. Otherwise, feed 0 to M. If M is now in a final state, then q_2 is the initial state.

3. Reset M, and feed it 1. If it accepts, then q_1 is the initial state.

4. Reset M, and feed it `01`. If it accepts, then q_0 is the initial state.

Definition 9.1.1 For a DFA $M = (Q, \Sigma, \delta, q_0, F)$, $p \in Q$ and $x \in \Sigma^*$, let $M(p, x)$ be true if setting the DFA to be in the state p, and then reading the input x causes M to arrive to an accepting state. Formally, $M(p, x)$ is true if and only if $\delta(p, x) \in F$, and false otherwise.

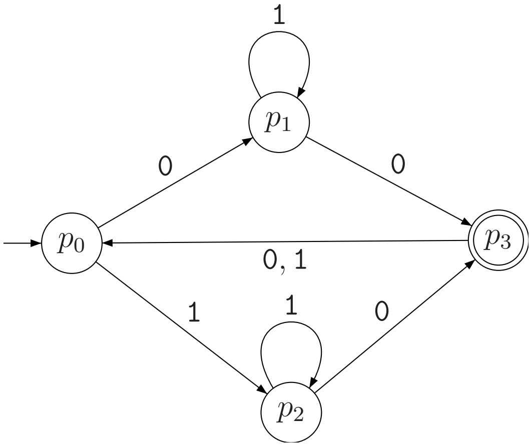
The moral of this story. So, we can differentiate between two states p and q of a DFA M, by finding strings x and y, such that $M(p, x)$ accepts, but $M(q.y)$ rejects, or vice versa. If x le.

Definition 9.1.2 Two states p and q of a DFA M ***disagree*** with each other, if there exists a string x, such that $M(p, x) \neq M(q, x)$ (that is, $M(p, x)$ accepts but $M(q, x)$ rejects, or vice versa).

Example 9.1.3 Note, that two states might be different, but yet the agree on all possible strings x. For example, consider the the DFA on the right.

Clearly, p_0 and p_1 disagree (for example on `0`). But notice that p_1 and p_2 agree on all possible strings.

Lemma 9.1.4 *Let M be a DFA and let $q_1, \ldots q_n$ be a list of states of M, such that any pair of them disagree. Then, M must have at least n states.*



Proof: For $i \neq j$, since q_i and q_j disagree with each other, they can not possibly be the same state of M, since if they were the same state then they would agree with each other on all possible strings. We conclude that $q_1, \ldots q_n$ are all different states of M; namely, M has at least n different states. ■

A Motivating Example

Consider the language $L = \left\{ \mathtt{a}^n \mathtt{b}^n \,\middle|\, n \geq 0 \right\}$. Intuitively, L can not be regular, because we have to remember how many `a`'s we have seen before reading the `b`'s, and this can not be done with a finite number of states.

Claim 9.1.5 *The language $L = \left\{ \mathtt{a}^n \mathtt{b}^n \,\middle|\, n \geq 0 \right\}$ is not regular.*

Proof: Suppose that L were regular. Then L is accepted by some DFA

$$M = (Q, \Sigma, \delta, q_0, F).$$

Let q_i denote the state M is in, after reading the string $\mathtt{a}^i$, for $i = 0, 1, 2, \ldots, \infty$. We claim that q_i disagrees with q_j if $i \neq j$. Indeed, observe that $M(q_i, \mathtt{b}^i)$ accepts but $M(q_j, \mathtt{b}^i)$ rejects, since $\mathtt{a}^i\mathtt{b}^i \in L$ and $\mathtt{a}^i\mathtt{b}^j \notin L$. As such, by Lemma 9.1.4, M has an infinite number of state, which is impossible. ■

9.2 Irregularity via differentiation

Definition 9.2.1 Two strings $x, y \in \Sigma^*$ are ***distinguishable*** by $L \subseteq \Sigma^*$, if there exists a word $w \in \Sigma^*$, such that *exactly one* of the strings xw and yw is in L.

Lemma 9.2.2 *Let $M = (Q, \Sigma, \delta, q_0, F)$ be a given DFA, and let $x \in \Sigma^*$ and $y \in \Sigma^*$ be two strings distinguishable by $L(M)$. Then $q_x \neq q_y$, where $q_x = \delta(q_0, x)$ (i.e., the state M is in after reading x) and $q_y = \delta(q_0, y)$ is the state that M is in after reading y.*

Proof: Indeed, let w be the string causing x and y to be distinguished by $L(M)$, and assume that $xw \in L(M)$ and $xy \notin L(M)$ (the other case is symmetric). Clearly, if $q_x = q_y$, then $M(q_0, xw) = M(q_x, w) = M(q_y, w) = M(q_0, yw)$, but it is given to us that $M(q_0, xw) \neq M(q_0, yw)$ since exactly one of the words xw and yw is in $L(M)$. ■

Lemma 9.2.3 *Let L be a language, and let $W = \{w_1, w_2, w_3, \ldots\}$ be an infinite set of strings, such that every pair of them is distinguishable by L. Then L is not a regular language.*

Proof: Assume for the sake of contradiction, that L is regular, and let M be a DFA for $M = (Q, \Sigma, \delta, q_0, F)$. Let us set $q_i = \delta(q_0, w_i)$. For $i \neq j$, w_i and w_j are distinguishable by L, and this implies by Lemma 9.2.2, that $q_i \neq q_j$. This implies that M has an infinite number of states, which is of course impossible. ■

9.2.1 Examples

Example

Lemma 9.2.4 *The language*

$$L = \left\{ \mathtt{1}^k y \;\middle|\; y \in \{\mathtt{0}, \mathtt{1}\}^*, \text{ and } y \text{ contains at most } k \text{ ones} \right\}$$

is not regular.

Proof: Let $w_i = \mathtt{1}^i$, for $i \geq 0$. Observe that for $j > i$ we have that $w_i\mathtt{0}\mathtt{1}^j = \mathtt{1}^i\mathtt{0}\mathtt{1}^j \notin L$ but $w_j\mathtt{0}\mathtt{1}^j = \mathtt{1}^j\mathtt{0}\mathtt{1}^j \in L$. As such, w_i and w_j are distinguishable by L, for any $i \neq j$. We conclude, by Lemma 9.2.3, that L is not regular. ■

Example: ww is not regular

Claim 9.2.5 *For $\Sigma = \{\mathtt{0}, \mathtt{1}\}$, the language $L = \left\{ ww \;\middle|\; w \in \Sigma^* \right\}$ is not regular.*

Proof: Set $w_i = \mathtt{0}^i$. And observe that, for $j > i$, we have that

$$\mathtt{0}^i \underbrace{\mathtt{1}\mathtt{0}^j\mathtt{1}}_{x_j} = w_i\mathtt{1}w_j\mathtt{1} \notin L \quad \text{but} \quad w_j\mathtt{1}w_j\mathtt{1} = \mathtt{0}^j \underbrace{\mathtt{1}\mathtt{0}^j\mathtt{1}}_{x_j} \in L$$

but this implies that w_i and w_j are distinguishable by L, using the string $x_j = \mathtt{1}\mathtt{0}^j\mathtt{1}$. As such, by Lemma 9.2.3, we have that L is not regular. ■

9.3 The Pumping Lemma

9.3.1 Proof by repetition of states

We next prove Claim 9.1.5 by a slightly different argument.

Claim. *The language $L = \left\{ \mathtt{a}^n\mathtt{b}^n \;\middle|\; n \geq 0 \right\}$ is not regular.*

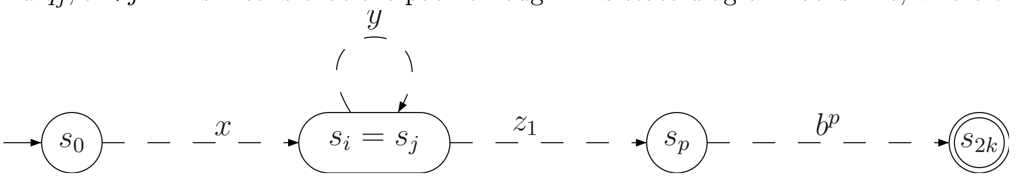
Proof: Suppose that L were regular. Then L is accepted by some DFA

$$M = (Q, \Sigma, \delta, q_0, F).$$

Suppose that M has p states.

Consider the string $\mathtt{a}^p\mathtt{b}^p$. It is accepted using a sequence of states $s_0s_1 \ldots s_{2p}$. Right after we read the last $\mathtt{a}$, the machine is in state s_p.

In the sub-sequence $s_0s_1 \ldots s_p$, there are $p+1$ states. Since L has only p distinct states, this means that two states in the sequence are the same (by the pigeonhole principle). Let us call the pair of repeated states q_i and q_j, $i < j$. This means that the path through M's state diagram looks like, where $\mathtt{a}^p = xyz_1$.



But this DFA will accept all strings of the form $xy^jz_1\mathtt{b}^p$, for $j \geq 0$. Indeed, for $j = 0$, this is just the string $xz_1\mathtt{b}^p$, which this DFA accepts, but it is not in the language, since it has less $\mathtt{a}$s than $\mathtt{b}$s. That is, if $|y| = m$, the DFA accepts all strings of the form $\mathtt{a}^{p-m+jm}\mathtt{b}^m$, for any $j \geq 0$. For any value of j other than 1, such strings are not in L.

So our DFA M accepts some strings that are not in L. This is a contradiction, because L was supposed to accept L. Therefore, we must have been wrong in our assumption that L was regular. ■

9.3.2 The pumping lemma

The pumping lemma generalizes the above argument into a standard template, which we can prove once and then quickly apply to many languages.

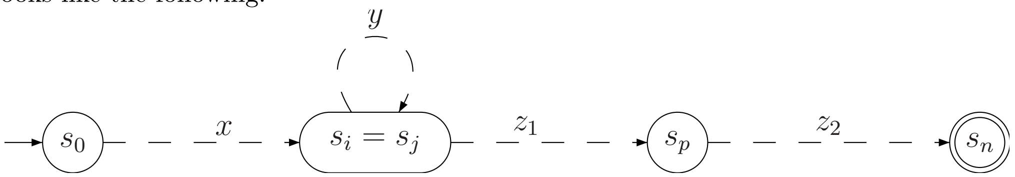
Theorem 9.3.1 (Pumping Lemma.) *Let L be a regular language. Then there exists an integer p (the "pumping length") such that for any string $w \in L$ with $|w| \geq p$, w can be written as xyz with the following properties:*

- $|xy| \leq p$.
- $|y| \geq 1$ *(i.e. y is not the empty string).*
- $xy^kz \in L$ *for every* $k \geq 0$.

Proof: The proof is written out in full detail in Sipser, here we just outline it.

Let M be a DFA accepting L, and let p be the number of states of M. Let $w = c_1c_2 \ldots c_n$ be a string of length $n \geq p$, and let the accepting state sequence (i.e., trace) for w be $s_0s_1 \ldots s_n$.

There must be a repeat within the sequence from s_0 to s_p, since M has only p states, and as such, the situation looks like the following.



So if we set $z = z_1z_2$, we now have x, y, and z satisfying the conditions of the lemma.

- $|xy| \leq p$ because repeat is within first $p+1$ states
- $|y| \geq 1$ because i and j are distinct
- $xy^kz \in L$ for every $k \geq 0$ because a loop in the state diagram can be repeated as many or as few times as you want.

  Formally, for any k, the word xy^iz goes through the following sequence of states:

$$s_0 \xrightarrow{\mathtt{x}} \overbrace{s_i \xrightarrow{\mathtt{y}} s_i \xrightarrow{\mathtt{y}} \cdots \xrightarrow{\mathtt{y}} s_i}^{k \text{ times}} = s_j \xrightarrow{\mathtt{z}} s_n,$$

  and s_n is an accepting state. Namely, M accepts xy^kz, and as such $xy^kz \in L$.

This completes the proof of the theorem. ■

Notice that we do not know exactly where the repeat occurs, so we have very little control over the length of x and z_1.

9.3.3 Using the PL to show non-regularity

If L is regular, then it satisfies the pumping lemma (PL). Therefore, intuitively, if L does not satisfy the pumping lemma, L cannot be regular.

Restating the Pumping Lemma via the contrapositive

We want to restate the pumping lemma in the contrapositive. Now, it is **not** true that if L satisfies the conditions of the PM, then L must be regular. Reminder from CS 173: contrapositive of if-then statement is equivalent, converse is not.

What does it mean to **not** satisfy the Pumping Lemma? Write out PL compactly:

$$\begin{array}{c}\text{L is}\\ \text{regular.}\end{array} \implies \left(\exists p\; \forall w \in L \quad |w| \geq p \Rightarrow \left(\exists x,y,z \text{ s.t. } \begin{array}{c} w = xyz, \\ |xy| \leq p, \\ |y| \geq 1, \end{array} \quad \text{and} \quad \forall i \quad xy^i z \in L. \right) \right).$$

Now, we know that if A implies B, then $\overline{B}$ implies $\overline{A}$ (contraposition), as such the Pumping Lemma, can be restated as

$$\overline{\left(\exists p\; \forall w \in L \quad |w| \geq p \Rightarrow \left(\exists x,y,z \quad \begin{array}{c} w = xyz, \\ |xy| \leq p, \\ |y| \geq 1, \end{array} \quad \text{and} \quad \forall i \quad xy^i z \in L. \right) \right)} \implies \overline{L \text{ is regular}}.$$

Now, the logical statement $A \Rightarrow B$ is equivalent to $\overline{A} \vee B = \overline{A \wedge \overline{B}}$. As such $\overline{A \Rightarrow B} = A \wedge \overline{B}$. In addition, negation flips quantifies, as such, the above is equivalent to

$$\left(\forall p\; \exists w \in L \quad |w| \geq p \text{ and } \overline{\left(\exists x,y,z \quad \begin{array}{c} w = xyz, \\ |xy| \leq p, \\ |y| \geq 1, \end{array} \quad \text{and} \quad \forall i \quad xy^i z \in L. \right)} \right) \implies \begin{array}{c}\text{L is}\\ \text{not regular.}\end{array}$$

Since, $\overline{A \wedge \overline{B}} = A \Rightarrow B$ we have that $\overline{A \wedge B} = \left(A \Rightarrow \overline{B}\right)$. Thus, we have

$$\left(\forall p\; \exists w \in L \quad |w| \geq p \text{ and } \left(\forall x,y,z \quad \begin{array}{c} w = xyz, \\ |xy| \leq p, \\ |y| \geq 1, \end{array} \quad \implies \quad \overline{\forall i \quad xy^i z \in L.} \right) \right) \implies \begin{array}{c}\text{L is}\\ \text{not regular.}\end{array}$$

Which is equivalent to

$$\left(\forall p\; \exists w \in L \quad |w| \geq p \text{ and } \left(\forall x,y,z \quad \begin{array}{c} w = xyz, \\ |xy| \leq p, \\ |y| \geq 1, \end{array} \quad \implies \quad \exists i \quad xy^i z \notin L. \right) \right) \implies \begin{array}{c}\text{L is}\\ \text{not regular.}\end{array}$$

The translation into words is the contrapositive of the Pumping Lemma (stated in Theorem 9.3.2 below).

The contrapositive of the Pumping Lemma

Theorem 9.3.2 (Pumping Lemma restated.) *Consider a language L. If for any integer $p \geq 0$ there exists a word $w \in L$, such that $|w| \geq p$, and for any breakup of w into three strings x, y, z, such that:*

- $w = xyz$,
- $|xy| \leq p$,
- $|y| \geq 1$,

implies that there exists an i such that $xy^i z \notin L$, then the language L is not regular.

Proving that a language is not regular

Let us assume that we want to show that a language L is *not* regular.

Such a proof is done by contradiction. To prove L is not regular, we assume it is regular. This gives us a specific (but unknown) pumping length p. We then show that L satisfies the rest of the contrapositive version of the pumping lemma, so it can not be regular.

So the proof outline looks like:

- Suppose L is regular. Let p be its pumping length.
- Consider $w =$ [formula for a specific class of strings]
- By the Pumping Lemma, we know there exist x, y, z such that $w = xyz$, $|xy| \leq p$, and $|y| \geq 1$.
- Consider $i =$ [some specific value, almost always 0 or 2]
- $xy^i z$ is not in L. [explain why it can't be]

Notice that our adversary picks p. We get to pick w whose length depends on p. But then our adversary gets to pick the specific division of w into x, y, and z.

9.3.4 Examples

The language $L = \mathtt{a}^n\mathtt{b}^n$ is not regular

Claim 9.3.3 *The language $L = \mathtt{a}^n\mathtt{b}^n$ is not regular.*

Proof: For any $p \geq 0$, consider the word $w = \mathtt{a}^p\mathtt{b}^p$, and consider any breakup of w into three parts, such that $w = xyz$ $|y| \geq 1$, and $|xy| \leq p$. Clearly, xy is a prefix of w made out of only as. As such, the word $xyyz$ has more as in it than bs, and as such, it is not in L.

But then, by the Pumping Lemma (Theorem 9.3.2), L is not regular. ■

The language $\{ww\}$ is not regular

Claim 9.3.4 *The language $L = \left\{ ww \;\middle|\; w \in \Sigma^* \right\}$ is not regular.*

Proof: For any $p \geq 0$, consider the word $w = \mathtt{0}^p\mathtt{1}\mathtt{0}^p\mathtt{1}$, and consider any breakup of w into three parts, such that $w = xyz$ $|y| \geq 1$, and $|xy| \leq p$. Clearly, xy is a prefix of w made out of only 0s. As such, the word $xyyz$ has more 0s in its first part than the second part. As such, $xyyz$ is not in L.

But then, by the Pumping Lemma (Theorem 9.3.2), L is not regular. ■

Consider the word w used in the above claim:

- It is concrete, made of specific characters, no variables left in it.
- These strings are a subset of L, chosen to exemplify what is not regular about L.
- Its length depends on p.
- The 1 in the middle serves as a barrier to separate the two groups of 0's. (Think about why the proof would fail if it was not there.)
- The 1 at the end of w does not matter to the proof, but we nee it so that $w \in L$.

9.3.5 A note on finite languages

A language L is ***finite*** if has a bounded number of words in it. Clearly, a finite language is regular (since you can always write a finite regular expression that matches all the words in the language).

It is natural to ask why we can not apply the pumping lemma Theorem 9.3.1 to L? The reason is because we can always choose the threshold p to be larger than the length of the longest word in L. Now, there is no word in L with length larger than p in L. As such, the claim of the Pumping Lemma holds trivially for a finite language, but no word can be pumped - and as such L stays finite. So the pumping lemma makes sense even for finite languages!

9.4 Irregularity via closure properties

If we know certain seed languages are not regular, then we can use closure properties to show other languages are not regular.

We remind the reader that *homomorphism* is a mapping $h : \Sigma_1 \to \Sigma_2^*$ (namely, every letter of Σ_1 is mapped to a string over Σ_2). We showed that if a language L over Σ_1 is regular, then the language $h(L)$ is regular. We referred to this property as *closure of regular languages under homomorphism.*

Claim 9.4.1 *The language $L' = \{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\}$ is not regular.*

Proof: Assume for the sake of contradiction that L' is regular. Let h be the homomorphism that maps $\mathtt{0}$ to $\mathtt{a}$ and $\mathtt{1}$ to $\mathtt{b}$. Then $h(L')$ must be regular (closure under homomorphism). But $h(L')$ is the language

$$L = \left\{\mathtt{a}^n\mathtt{b}^n \;\middle|\; n \geq 0\right\}, \tag{9.1}$$

which is not regular by Claim 9.1.5. A contradiction. As such, L' is not regular. ■

We remind the reader that regular languages are also closed under intersection.

Claim 9.4.2 *The language $L_2 = \left\{w \in \{\mathtt{a},\mathtt{b}\}^* \;\middle|\; w \text{ has an equal \# of } \mathtt{a}\text{'s and } \mathtt{b}\text{'s}\right\}$ is not regular.*

Proof: Suppose L_2 were regular. Consider $L_2 \cap \mathtt{a}^*\mathtt{b}^*$. This must be regular because L_2 and $\mathtt{a}^*\mathtt{b}^*$ are both regular and regular languages are closed under intersection. But $L_2 \cap \mathtt{a}^*\mathtt{b}^*$ is just the language L from Eq. (9.1), which is not regular (by Claim 9.1.5). ■

Claim 9.4.3 *The language $L_3 = \left\{\mathtt{a}^n\mathtt{b}^n \;\middle|\; n \geq 1\right\}$ is not regular.*

Proof: Assume for the sake of contradiction that L_3 is regular. Consider $L_3 \cup \{\epsilon\}$. This must be regular because L_3 and $\{\epsilon\}$ are both regular and regular languages are closed under union. But $L_3 \cup \{\epsilon\}$ is just L from Eq. (9.1), which is not regular (by Claim 9.1.5).

A contradiction. As such, L_3 is not regular. ■

9.4.1 Being careful in using closure arguments

Most closure properties must be applied in the correct direction: We show (or assume) that all inputs to the operation are regular, therefore the output of the operation must be regular.

For example, consider (again) the language $L_B = \{0^n1^n \mid n \geq 0\}$, which is not regular.

Since L_B is not regular, $\overline{L_B}$ is also not regular. If $\overline{L_B}$ were regular, then L_B would also have to be regular because regular languages are closed under set complement. However, many similar lines of reasoning do not work for other closure properties.

For example, L_B and $\overline{L_B}$ are both non-regular, but their union is regular. Similarly, suppose that L_k is the set of all strings of length $\leq k$. Then $L_B \cap L_k$ is regular, even though L_B is not regular.

If you are not absolutely sure of what you are doing, always use closure properties in the forward direction. That is, establish that L and L' are regular, then conclude that L OP L' must be regular.

Also, be sure to apply only closure properties that we know to be true. In particular, regular languages are **not** closed under the subset and superset relations. Indeed, consider $L_1 = \{001, 00\}$, which is regular. But L_1 is a subset of L_B, which is not regular. Similarly, $L_2 = (0+1)^*$ is regular. And it is a superset of L (from Eq. (9.1) in the proof of Claim 9.4.1)). But you can not deduce that L is therefore regular. We know it is not.

So regular languages can be subsets of non-regular ones and vice versa.

Chapter 10

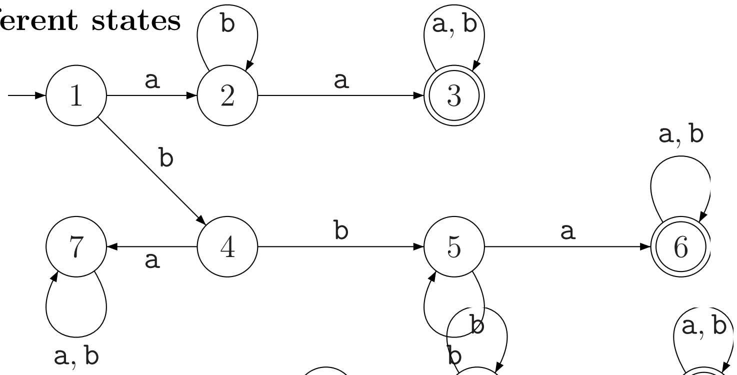
Lecture 10: DFA minimization

19 February 2009

In this lecture, we will see that every language has a unique minimal DFA. We will see this fact from two perspectives. First, we will see a practical algorithm for minimizing a DFA, and provide a theoretical analysis of the situation.

10.1 On the number of states of DFA

10.1.1 Starting a DFA from different states

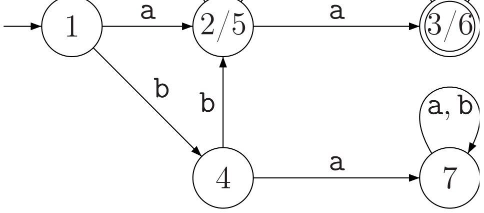


Consider the DFA on the right. It has a particular defined start state. However, we could start it from any of its states. If the original DFA was named M, define M_q to be the DFA with its start state changed to state q. Then the language L_q, is the one accepted if you start at q.

For example, in this picture, L_3 is $(\texttt{a}+\texttt{b})^*$, and L_6 is the same. Also, L_2 and L_5 are both $\texttt{b}^*\texttt{a}(\texttt{a}+\texttt{b})^*$. Finally, L_7 is $\emptyset$.

Suppose that $L_q = L_r$, for two states q and r. Then once we get to q or r, the DFA is going to do the same thing from then on (i.e., its going to accept or reject *exactly* the same strings).

So these two states can be merged. In particular, in the above automata, we can merge 2 and 5 and the states 3 and 6. We can the new automata, depicted on the right.



10.1.2 Suffix Languages

Let Σ be some alphabet.

Definition 10.1.1 Let $L \subseteq \Sigma^*$ be any language.

The ***suffix language*** of L with respect to a word $x \in \Sigma^*$ is defined as

$$\left\llbracket L/x \right\rrbracket = \left\{ y \;\middle|\; x\, y \in L \right\}.$$

In words, $\left\llbracket L/x \right\rrbracket$ is the language made out of all the words, such that if we append x to them as a prefix, we get a word in L.

The ***class of suffix languages*** of L is

$$\mathcal{C}(L) = \left\{ \left\llbracket L/x \right\rrbracket \;\middle|\; x \in \Sigma^* \right\}.$$

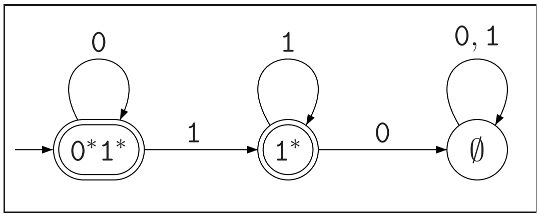
Example 10.1.2 For example, if $L = \texttt{0}^*\texttt{1}^*$, then:

- $\llbracket L/\epsilon \rrbracket = \texttt{0}^*\texttt{1}^* = L$
- $\llbracket L/\texttt{0} \rrbracket = \texttt{0}^*\texttt{1}^* = L$
- $\llbracket L/\texttt{0}^i \rrbracket = \texttt{0}^*\texttt{1}^* = L$, for any $i \in \mathbb{N}$
- $\llbracket L/\texttt{1} \rrbracket = \texttt{1}^*$
- $\llbracket L/\texttt{1}^i \rrbracket = \texttt{1}^*$, for any $i \geq 1$
- $\llbracket L/\texttt{10} \rrbracket = \left\{ y \,\middle|\, \texttt{10}y \in L \right\} = \emptyset$.

Hence there are only three suffix languages for L: $\texttt{0}^*\texttt{1}^*$, $\texttt{1}^*$, $\emptyset$. So $\mathcal{C}(L) = \left\{ \texttt{0}^*\texttt{1}^*,\ \texttt{1}^*,\ \emptyset \right\}$.

As the above example demonstrates, if there is a word x, such that any word w that have x as a prefix is not in L, then $\llbracket L/x \rrbracket = \emptyset$, which implies that $\emptyset$ is one of the suffix languages of L.

Example 10.1.3 The above suggests the following automata for the language of Example 10.1.2: $L = \texttt{0}^*\texttt{1}^*$.



And clearly, this is the automata with the smallest number of states that accepts this language.

Regular languages have few suffix languages

Now, consider a DFA $M = (Q, \Sigma, \delta, q_0, F)$ accepting some language L. Let $x \in \Sigma^*$, and let M reach the state q on reading x. The suffix language $\llbracket L/x \rrbracket$ is precisely the set of strings w, such that xw is in L. But this is exactly the same as L_q. That is, $\llbracket L/x \rrbracket = L_q$, where q is the state reached by M on reading x. Hence the suffix languages of a regular language accepted by a DFA are precisely those languages L_q, where $q \in Q$.

Notice that the definition of suffix languages is more general, because it can also be applied to non-regular languages.

Lemma 10.1.4 *For a regular language L, the number of different suffix languages it has is bounded; that is $\mathcal{C}(L)$ is bounded by a constant (that depends on L).*

Proof: Consider the DFA $M = (Q, \Sigma, \delta, q_0, F)$ that accepts L. For any string x, the suffix language $\llbracket L/x \rrbracket$ is just the languages associated with L_q, where q is the state M is in after reading x.

Indeed, the suffix language $\llbracket L/x \rrbracket$ is the set of strings w such that $xw \in L$. Since the DFA reaches q on x, it is clear that the suffix language of x is precisely the language accepted by M starting from the state q, which is L_q. Hence, for every $x \in \Sigma^*$, $\llbracket L/x \rrbracket = L_{\delta(q_0, x)}$, where q is the state the automaton reaches on x.

As such, any suffix language of L is realizable as the language of a state of M. Since the number of states of M is some constant k, it follows that the number of suffix languages of L is bounded by k. ■

An immediate implication of the above lemma is the following.

Lemma 10.1.5 *If a language L has infinite number of suffix languages, then L is not regular.*

The suffix languages of a non-regular language

Consider the language $L = \left\{ \mathtt{a}^n \mathtt{b}^n \,\middle|\, n \in \mathbb{N} \right\}$. The suffix language of L for $\mathtt{a}^i$ is

$$\left[\!\left[L/\mathtt{a}^i \right]\!\right] = \left\{ \mathtt{a}^{n-i} \mathtt{b}^n \,\middle|\, n \in \mathbb{N} \right\}.$$

Note, that $\mathtt{b}^i \in \left[\!\left[L/\mathtt{a}^i \right]\!\right]$, but this is the only string made out of only $\mathtt{b}$s that is in this language. As such, for any i, j, where i and j are different, the suffix language of L with respect to $\mathtt{a}^i$ is different from that of L with respect to $\mathtt{a}^j$ (i.e. $\left[\!\left[L/\mathtt{a}^i \right]\!\right] \neq \left[\!\left[L/\mathtt{a}^j \right]\!\right]$). Hence L has infinitely many suffix languages, and hence is not regular, by Lemma 10.1.5.

Let us summarize what we had seen so far:

- Any state of a DFA of a language L is associated with a suffix language of L.
- If two states are associated with the same suffix language, that we can merge them into a single state.
- At least one non-regular language $\left\{ \mathtt{a}^n \mathtt{b}^n \,\middle|\, n \in \mathbb{N} \right\}$ has an infinite number of suffix languages.

It is thus natural to conjecture that the number of suffix languages of a language, is a good indicator of how many states an automata for this language would require. And this is indeed true, as the following section testifies.

10.2 Regular Languages and Suffix Languages

10.2.1 A few easy observations

Lemma 10.2.1 *If $\epsilon \in \left[\!\left[L/x \right]\!\right]$ if and only if $x \in L$.*

Proof: By definition, if $\epsilon \in \left[\!\left[L/x \right]\!\right]$ then $x = x\epsilon \in L$. Similarly, if $x \in L$, then $x\epsilon \in L$, which implies that $\epsilon \in \left[\!\left[L/x \right]\!\right]$. ■

Lemma 10.2.2 *Let L be a language over alphabet Σ. For all $x, y \in \Sigma^*$ we have that if $\left[\!\left[L/x \right]\!\right] = \left[\!\left[L/y \right]\!\right]$ then for all $\mathtt{a} \in \Sigma$ we have $\left[\!\left[L/x\mathtt{a} \right]\!\right] = \left[\!\left[L/y\mathtt{a} \right]\!\right]$.*

Proof: If $w \in \left[\!\left[L/x\mathtt{a} \right]\!\right]$, then (by definition) $x\mathtt{a}w \in L$. But then, $\mathtt{a}w \in \left[\!\left[L/x \right]\!\right]$. Since $\left[\!\left[L/x \right]\!\right] = \left[\!\left[L/y \right]\!\right]$, this implies that $\mathtt{a}w \in \left[\!\left[L/y \right]\!\right]$, which implies that $y\mathtt{a}w \in L$, which implies that $w \in \left[\!\left[L/y\mathtt{a} \right]\!\right]$. This implies that $\left[\!\left[L/x\mathtt{a} \right]\!\right] \subseteq \left[\!\left[L/y\mathtt{a} \right]\!\right]$, a symmetric argument implies that $\left[\!\left[L/y\mathtt{a} \right]\!\right] \subseteq \left[\!\left[L/x\mathtt{a} \right]\!\right]$. We conclude that $\left[\!\left[L/x\mathtt{a} \right]\!\right] = \left[\!\left[L/y\mathtt{a} \right]\!\right]$. ■

10.2.2 Regular languages and suffix languages

We can now state a characterization of regular languages in term of suffix languages.

Theorem 10.2.3 (Myhill-Nerode theorem.) *A language $L \subseteq \Sigma^*$ is regular if and only if the number of suffix languages of L is finite (i.e. $\mathcal{C}(L)$ is finite).*

Moreover, if $\mathcal{C}(L)$ contains exactly k languages, we can build a DFA for L that has k states; also, any *DFA accepting L must have k states.*

Proof: If L is regular, then $\mathcal{C}(L)$ is a finite set by Lemma 10.1.4.

Second, let us show that if $\mathcal{C}(L)$ is finite, then L is regular. Let the suffix languages of L be

$$\mathcal{C}(L) = \left\{ \left[\!\!\left[L/x_1 \right]\!\!\right], \left[\!\!\left[L/x_2 \right]\!\!\right], \ldots, \left[\!\!\left[L/x_k \right]\!\!\right] \right\}. \tag{10.1}$$

Note that for any $y \in \Sigma^*$, $\left[\!\!\left[L/y \right]\!\!\right] = \left[\!\!\left[L/x_j \right]\!\!\right]$, for some $j \in \{1, \ldots, k\}$.

We will construct a DFA whose states are the various suffix languages of L; hence we will have k states in the DFA. Moreover, the DFA will be designed such that after reading y, the DFA will end up in the state $\left[\!\!\left[L/y \right]\!\!\right]$.

The DFA is $M = (Q, \Sigma, q_0, \delta, F)$ where

- $Q = \left\{ \left[\!\!\left[L/x_1 \right]\!\!\right], \left[\!\!\left[L/x_2 \right]\!\!\right], \ldots, \left[\!\!\left[L/x_k \right]\!\!\right] \right\}$
- $q_0 = \left[\!\!\left[L/\epsilon \right]\!\!\right]$,
- $F = \left\{ \left[\!\!\left[L/x \right]\!\!\right] \;\middle|\; \epsilon \in \left[\!\!\left[L/x \right]\!\!\right] \right\}$. Note, that by Lemma 10.2.1, if $\epsilon \in \left[\!\!\left[L/x \right]\!\!\right]$ then $x \in L$.
- $\delta\Big(\left[\!\!\left[L/x \right]\!\!\right], \mathtt{a} \Big) = \left[\!\!\left[L/x\mathtt{a} \right]\!\!\right]$ for every $\mathtt{a} \in \Sigma$.

The transition function δ is well-defined because of Lemma 10.2.2.

We can now prove, by induction on the length of x, that after reading x, the DFA reaches the state $\left[\!\!\left[L/x \right]\!\!\right]$. If $x \in L$, then $\epsilon \in \left[\!\!\left[L/x \right]\!\!\right]$, which implies that $\delta(q_0, x) = \left[\!\!\left[L/x \right]\!\!\right] \in F$. Thus, $x \in L(M)$. Similarly, if $x \in L(M)$, then $\left[\!\!\left[L/x \right]\!\!\right] \in F$, which implies that $\epsilon \in \left[\!\!\left[L/x \right]\!\!\right]$, and by Lemma 10.2.1 this implies that $x \in L$. As such, $L(M) = L$.

We had shown that the DFA M accepts L, which implies that L is regular, furthermore M has k states.

We next prove that *any* DFA for L must have at least k states. So, let $N = (Q', \Sigma, \delta_N q_{\mathrm{init}}, F)$ any DFA accepting L. The language L has k suffix languages, generated by the strings $x_1, x_2, \ldots, x_k$, see Eq. (10.1).

For any $i \neq j$, we have that $\left[\!\!\left[L/x_i \right]\!\!\right] \neq \left[\!\!\left[L/x_j \right]\!\!\right]$. As such, there must exist a word w such that $w \in \left[\!\!\left[L/x_j \right]\!\!\right]$ and $w \notin \left[\!\!\left[L/x_j \right]\!\!\right]$ (the symmetric case where $w \in \left[\!\!\left[L/x_j \right]\!\!\right] \setminus \left[\!\!\left[L/x_i \right]\!\!\right]$ is handled in a similar fashion. But then, $x_i w \in L$ and $x_j w \notin L$. Namely, $N(q_{\mathrm{init}}, x) \neq N(q_{\mathrm{init}}, y)$, and the two states that N reaches for x_i and x_j respectively, are distinguishable. Formally, let $q_i = \delta(q_{\mathrm{init}}, x_i)$, for $i = 1, \ldots, k$. All these states are pairwise distinguishable, which implies that N must have at least k states. ■

Remark 10.2.4 The full Myhill-Nerode theorem also shows that all minimal DFAs for L are isomorphic, i.e. have identical transitions as well as the same number of states, but we will not show that part.

This is done by arguing that any DFA for L that has k states must be *identical* to the DFA we created above. This is a bit more involved notationally, and is proved by showing a $1 - 1$ correspondence between the two DFAs and arguing they must be connected the same way. We omit this part of the theorem and proof.

10.2.3 Examples

Let us explain the theorem we just proved using examples.

The suffix languages of a non-regular language

Consider the language $L \subseteq \{\mathtt{a}, \mathtt{b}\}^*$:

$$L = \left\{ w \;\middle|\; w \text{ has an odd number of } \mathtt{a}\text{'s} \right\}.$$

The suffix language of $x \in \Sigma^*$, where x has an even number of $\mathtt{a}$'s is:

$$\left\llbracket L/x \right\rrbracket = \left\{ w \,\middle|\, w \text{ has an odd number of } \mathtt{a}\text{'s} \right\} = L.$$

The suffix language of $x \in \Sigma^*$, where x has an odd number of a's is:

$$\left\llbracket L/x \right\rrbracket = \left\{ w \,\middle|\, w \text{ has an even number of } \mathtt{a}\text{'s} \right\}.$$

Hence there are only two distinct suffix languages for L. By the theorem, we know L must be regular and the minimal DFA for L has two states. Going with the construction of the DFA mentioned in the proof of the theorem, we see that we have two states, $q_0 = \left\llbracket L/\epsilon \right\rrbracket$ and $q_1 = \left\llbracket L/\mathtt{a} \right\rrbracket$. The transitions are as follows:

- From $q_0 = \left\llbracket L/\epsilon \right\rrbracket$, on $\mathtt{a}$ we go to $\left\llbracket L/\mathtt{a} \right\rrbracket$, which is the state q_1.
- From $q_0 = \left\llbracket L/\epsilon \right\rrbracket$, on b we go to $\left\llbracket L/b \right\rrbracket$, which is same as $\left\llbracket L/\epsilon \right\rrbracket$, i.e. the state q_0.
- From $q_1 = \left\llbracket L/\mathtt{a} \right\rrbracket$, on $\mathtt{a}$ we go to $\left\llbracket L/aa \right\rrbracket$, which is same as $\left\llbracket L/\epsilon \right\rrbracket$, i.e. the state q_0.
- From $q_1 = \left\llbracket L/a \right\rrbracket$, on b we go to $\left\llbracket L/ab \right\rrbracket$, which is same as $\left\llbracket L/a \right\rrbracket$, i.e. the state q_1.

The initial state is $\left\llbracket L/\epsilon \right\rrbracket$ which is the state q_0, and the final states are those states $\left\llbracket L/x \right\rrbracket$ that have ϵ in them, which is the set $\{q_1\}$.

We hence have a DFA for L, and in fact this is the minimal automaton accepting L.

10.3 Minimization algorithm

The above discussion leaves us with a way to decide what is the minimum number of states of a DFA that accepts a language, but it is not clear how to turn this into an algorithm (in particular, we do not have an efficient way to compute suffix languages of a language).

The idea is to work directly on a given DFA and compute a minimum DFA from it. So consider the DFA of Figure 10.1. It is more complex than it needs to be.

The DFA minimization algorithm first removes any states which are not reachable from the start state, because they obviously aren't contributing anything to what the DFA accepts. (D in this example.) It then marks which of the remaining states are distinct. States not marked as distinct can then be merged, to create a simpler DFA.

10.3.1 Idea of algorithm

Suppose the given DFA is $M = (Q, \Sigma, \delta, q_0, F)$.

We know by the above discussion that two states p and q are distinct if their two languages are different. Namely, there is some word w that belongs to L_p but w is not in L_q (or vice versa). It is not clear however how to detect when two states have different suffix languages. The idea is to start with the "easy" case, and then propagate the information.

As such, p and q are ***distinct*** if there exists w, such that $\delta(p, w)$ is an accept state and $\delta(q, w)$ is not an accept state. In particular, for $w = \epsilon$, we have that p and q are distinct if $p \in F$ and $q \notin F$, or vice versa.

for $w = c_1 c_2 \dots c_m$, we have that p and q are ***distinct*** if

$$\delta\Big(\delta(p, c_1)\,, c_2 c_3 \dots c_m\Big) \in F \text{ and } \delta\Big(\delta(q, c_1)\,, c_2 c_3 \dots c_m\Big) \notin F,$$

or vice versa.

In particular, this implies that if p and q are distinct because of word w of length m, then $\delta(p, c_1)$ and $\delta(q, c_1)$ are distinct because of a word $w' = c_2 \dots c_m$ of length $m - 1$.

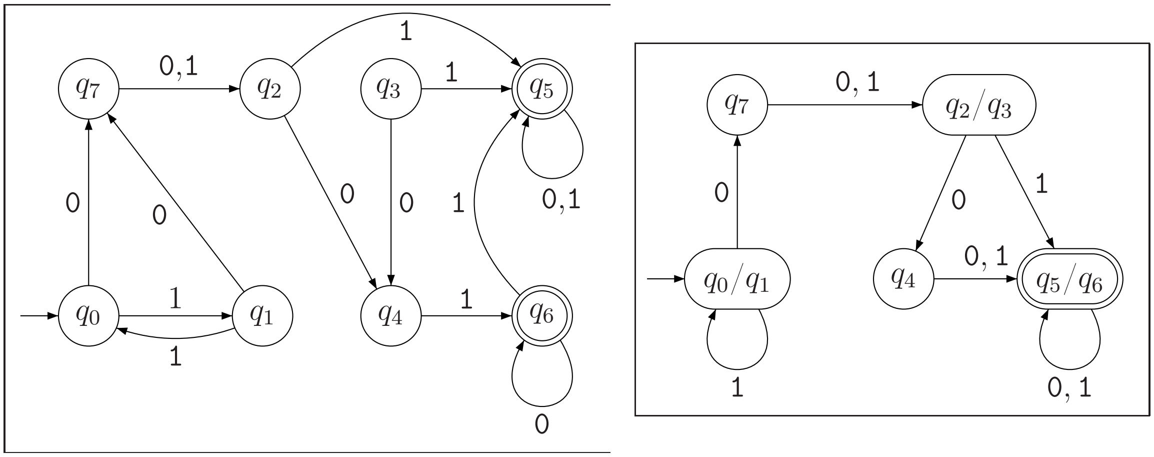
Thus, its easy to compute the pairs of states distinct because of empty words, and if we computed all the states distinct because of words of length $m - 1$, we can "propagate" this information for pairs of states distinct by states of length m.

10.3.2 The algorithm

The algorithm for marking distinct states follows the above (recursive) definition. Create a table Distinct with an entry for each pair of states. Table cells are initially blank.

```
(1) For every pair of states $(p, q)$
        If p is final and q is not, or vice versa,
            Set Distinct$(p, q)$ to be $\epsilon$.
(2) Loop until there is no change in the table contents
        For each pair of states $(p, q)$ and each character a in the alphabet:
            if Distinct$(p, q)$ is empty and Distinct$(\delta(p, a), \delta(q, a))$ is not empty
                Set Distinct$(p, q)$ to be a.
(3) Two states $p$ and $q$ are distinct iff Distinct$(p, q)$ is not empty.
```

Example of how the algorithm works



(a) (b)

Figure 10.1: (a) Original automata, (b) minimized autoamta.

.

The following is the execution of the algorithm on the DFA of Figure 10.1.
After step (1):

q_0								
q_1								
q_2								
q_3								
q_4								
q_5	ϵ	ϵ	ϵ	ϵ	ϵ			
q_6	ϵ	ϵ	ϵ	ϵ	ϵ			
q_7						ϵ	ϵ	
	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7

(Note, that for a pair of states (q_i, q_j) we need only a single entry since (q_j, q_i) is equivalent, and we do not need to consider pair on the diagonal of the form (q_i, q_i).)

q_0								
q_1								
q_2	1	1						
q_3	1	1						
q_4	0	0	0	0				
q_5	ϵ	ϵ	ϵ	ϵ	ϵ			
q_6	ϵ	ϵ	ϵ	ϵ	ϵ			
q_7			1	1	0	ϵ	ϵ	
	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7

After one iteration of step (2)

$\Rightarrow$

q_0								
q_1								
q_2	1	1						
q_3	1	1						
q_4	0	0	0	0				
q_5	ϵ	ϵ	ϵ	ϵ	ϵ			
q_6	ϵ	ϵ	ϵ	ϵ	ϵ			
q_7	1	1	1	1	0	ϵ	ϵ	
	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7

After the second iteration of step (2)

Third iteration of step (2) makes no changes to the table, so we halt. The cells (q_0, q_1), (q_2, q_3) and (q_5, q_6) are still empty, so these pairs of states are not distinct. Merging them produces the following simpler DFA recognizing the same language.

Chapter 11

Lecture 11: Context-free grammars

21 February 2008

This lecture introduces context-free grammars, covering section 2.1 from Sipser.

11.1 Context-free grammars

11.1.1 Introduction

Regular languages are efficient but very limited in power. For example, not powerful enough to represent the overall structure of a `C` program.

As another example, consider the following language

$$L = \{\text{all strings formed by properly nested parenthesis}\}.$$

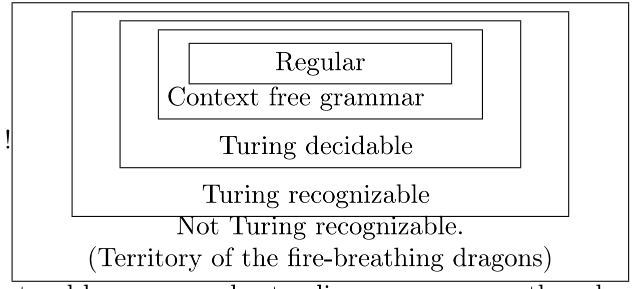
Here, the string (()()) is in L. ())(is not.

Lemma 11.1.1 *The language L is not regular.*

Proof: Assume for the sake of contradiction that L is regular. Then consider $L' = L \cap {}'('^*)^*$. Since L is regular and regular languages are closed under intersection, L' must be regular. But L is just $\left\{ (^n)^n \;\middle|\; n \geq 0 \right\}$. We can map this, with a homomorphism, to $\texttt{0}^n\texttt{1}^n$, which is not regular (as we seen before). A contradiction. ■

Our purpose is to come up with a way to describe the above language L in a compact way. It turns out that context-free grammars are one possible way to capture such languages.

Here is a diagram demonstrating the classes of languages we will encounter in this class. Currently, we only saw the weakest class – regular language. Next, we will see context free grammars.



A compiler or a natural language understanding program, use these languages as follows:

- It uses regular languages to convert character strings to tokens (e.g. words, variables names, function names).

- It uses context-free languages to parse token sequences into functions, programs, sentences.

Just as for regular languages, context-free languages have a procedural and a declarative representation, which we will show to be equivalent.

procedural	declarative
NFAs/DFAs	regular expressions
pushdown automata (PDAs)	context-free grammar

11.1.2 Deriving the context-free grammars by example

So, consider our old arch-nemesis, the language

$$L = \left\{ \mathtt{a}^n\mathtt{b}^n \;\middle|\; n \geq 0 \right\}.$$

we would like to come up with a recursive definition for a word in the language.

So, let S denote any word we can generate in the language, then a word w in this language can be generated as

$$w = \mathtt{a}^n\mathtt{b}^n = \mathtt{a}\ \underbrace{\boxed{\mathtt{a}^{n-1}\mathtt{b}^{n-1}}}_{=w'}\ \mathtt{b},$$

where $w' \in L$. Thus, we have a compact recursive way to generate L. It is the language containing the empty word, and one can generate a new word, by taking a word w' already in the language and padding it with a before, and b after it. Thus, generating the new word $\mathtt{a}w'\mathtt{b}$. This suggests a random procedure S to generate such a word. It either return without generating anything, or it prints a a, generates a word recursively by calling S, and then it outputs a b. Naturally, the procedure has to somehow guess which of the two options to perform. We demonstrate this idea in the C program on the right, where S uses randomization to decide which action to take. As such, running this program would generate a random word in this language.

```
#include  <stdlib.h>
#include  <stdio.h>
#include  <time.h>
int  guess()
{ return  random() % 16; }
void  S() {
    if  ( guess() == 0 ) return;
    else {
        printf( "a" );
        S();
        printf( "b" );
    }
}
int  main() {
    srand( time( 0 ) );
    S();
}
```

The way to write this recursive generation algorithm using context free grammar is by specifying

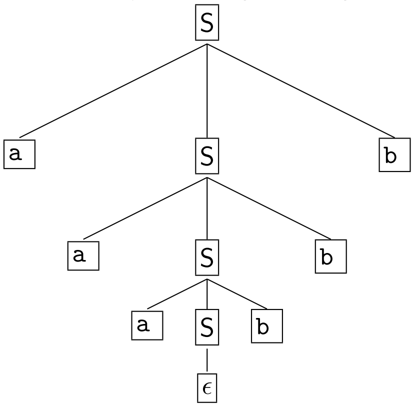
$$\mathsf{S} \rightarrow \epsilon \;\mid\; \mathtt{aSb}. \tag{11.1}$$

Thus, CFG can be taught of as a way to specify languages by a recursive means: We can build sole basic words, and then we can build up together more complicated words by recursively building fragments of words and concatenating them together.

For example, we can derive the word aaabbb from the grammar of Eq. (11.1), as follows:

$$\mathsf{S} \rightarrow \mathtt{aSb} \rightarrow \mathtt{aaSbb} \rightarrow \mathtt{aaaSbbb} \rightarrow \mathtt{aaa}\epsilon\mathtt{bbb} = \mathtt{aaabbb}.$$

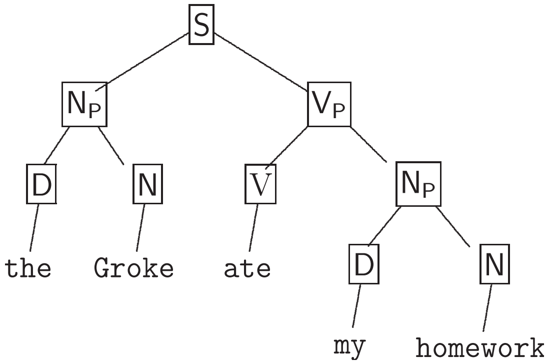
Alternatively, we can think about the recursion tree used by our program to generate this string.



This tree is known as the ***parse tree*** of the grammar of Eq. (11.1) for the word aaabbb.

Deriving the context-free grammars by constructing sentence structure

A context-free grammar defines the syntax of a program or sentence. The structure is easiest to see in a parse tree.



The interior nodes of the tree contain "variables", e.g. $\mathsf{N_P}$, D. The leaves of the tree contain "terminals". The "yield" of the tree is the string of terminals at the bottom. In this case, "the Groke at my homework." [1]

The grammar for this has several components:

(i) A start symbol: S.

(ii) A finite set of variables: $\{\mathsf{S}, \mathsf{N_P}, \mathsf{D}, \mathsf{N}, \mathsf{V}, \mathsf{V_P}\}$

(iii) A finite set of terminals $= \{\texttt{the}, \texttt{Groke}, \texttt{ate}, \texttt{my}, \ldots\}$

(iv) A finite set of rules.

Example of how rules look like

(i) $\mathsf{S} \rightarrow \mathsf{N_P}\ \mathsf{V_P}$

(ii) $\mathsf{N_P} \rightarrow \mathsf{D}\ \mathsf{N}$

(iii) $\mathsf{V_P} \rightarrow \mathsf{V}\ \mathsf{N_P}$

(iv) $\mathsf{N} \rightarrow \mid \texttt{Groke} \mid \text{homework} \mid \text{lunch}\ldots$

(v) $\mathsf{D} \rightarrow \texttt{the} \mid \texttt{my}\ldots$

(vi) $\mathsf{V} \rightarrow \texttt{ate} \mid \texttt{corrected} \mid \text{washed}\ldots$

If projection is working, show a sample computer-language grammar from the net. (See pointers on web page.)

Synthetic examples

In practical applications, the terminals are often whole words, as in the example above. In synthetic examples (and often in the homework problems), the terminals will be single letters.

Consider $L = \left\{ \mathtt{0}^n \mathtt{1}^n \,\middle|\, n \geq 0 \right\}$. We can capture this language with a grammar that has start symbol S and rule

$$\mathsf{S} \rightarrow \mathtt{0}\mathsf{S}\mathtt{1} \mid \epsilon\ .$$

For example, we can derive the string $\texttt{000111}$ as follows:

$$\mathsf{S} \rightarrow 0\mathsf{S}1 \rightarrow 00S11 \rightarrow 000S111 \rightarrow 000\epsilon 111 = 000111.$$

[1] Groke – Also known in Norway as Hufsa, in Estonia as Urr and in Mexico as La Coca is a fictional character in the Moomin world created by Tove Jansson.

Or, consider the language of palindromes $L = \left\{ w \in \{a,b\}^* \;\middle|\; w = w^R \right\}$. Here is a grammar with start symbol P. for this language

$$\mathsf{P} \rightarrow \mathtt{a}\mathsf{P}\mathtt{a} \;|\; \mathtt{b}\mathsf{P}\mathtt{b} \;|\; \epsilon \;|\; \mathtt{a} \;|\; \mathtt{b}.$$

A possible derivation of the string abbba is

$$\mathsf{P} \rightarrow \mathtt{a}\mathsf{P}\mathtt{a} \rightarrow \mathtt{ab}\mathsf{P}\mathtt{ba} \rightarrow \mathtt{abbba}.$$

11.2 Derivations

Consider our Groke example again. It has only one parse tree, but multiple derivations: After we apply the first rule, we have two variables in our string. So we have two choices about which to expand first:

$$S \rightarrow \mathsf{N_P}\ \mathsf{V_P} \rightarrow \ldots$$

If we expand the leftmost variable first, we get this derivation:

$$S \rightarrow \mathsf{N_P}\ \mathsf{V_P} \rightarrow \mathsf{D}\ N\ \mathsf{V_P} \rightarrow \textit{the } N\ \mathsf{V_P} \rightarrow \textit{the Groke } \mathsf{V_P} \rightarrow \textit{the Groke } V\ \mathsf{N_P} \rightarrow \ldots$$

If we expand the rightmost variable first, we get this derivation:

$$\begin{aligned} \mathsf{S} &\rightarrow \mathsf{N_P}\ \mathsf{V_P} \rightarrow \mathsf{N_P}\ V\ \mathsf{N_P} \rightarrow \mathsf{N_P}\ V\ \mathsf{D}\ N \rightarrow \mathsf{N_P}\ V\ \mathsf{D}\ \mathtt{homework} \\ &\rightarrow \mathsf{N_P}\ V\ \mathtt{my}\ \mathtt{homework}\ldots \end{aligned}$$

The first is called the ***leftmost derivation***. The second is called the ***rightmost derivation***. There are also many other possible derivations. Each parse tree has many derivations, but exactly one rightmost derivation, and exactly one leftmost derivation.

11.2.1 Formal definition of context-free grammar

Definition 11.2.1 (CFG) A ***context-free grammar*** (CFG) is a 4-tuple $\mathcal{G} = (V, \Sigma, R, \mathsf{S})$, where

(i) $\mathsf{S} \in V$ is the ***start variable***,

(ii) Σ is the alphabet (as such, we refer to $c \in \Sigma$ as a character or ***terminal***),

(iii) V is a finite set of ***variables***, and

(iv) R is a finite set of rules, each is of the form $\mathsf{B} \rightarrow w$ where $\mathsf{B} \in V$ and $w \in (V \cup \Sigma)^*$ is a word made out of variables and terminals..

Definition 11.2.2 (CFG yields.) Suppose x, y, and w are strings in $(V \cup \Sigma)^*$ and B is a variable. Then $x\mathsf{B}y$ ***yields*** xwy, written as

$$x\mathsf{B}y \Rightarrow xwy,$$

if there is a rule in R of the form $\mathsf{B} \rightarrow w$.

Notice that $x \Rightarrow x$, for any x and any set of rules.

Definition 11.2.3 (CFG derives.) If x and y in $(V \cup \Sigma)^*$, then w ***derives*** x, written as

$$w \overset{*}{\Rightarrow} x$$

if you can get from w to x in zero or more yields steps.

That is, there is a sequence of strings $y_1, y_2, \ldots y_k$ in $(V \cup \Sigma)^*$ such that

$$w = y_1 \Rightarrow y_2 \Rightarrow \ldots \Rightarrow y_k = x.$$

Definition 11.2.4 If $\mathcal{G} = (V, \Sigma, R, S)$ is a grammar, then $L(\mathcal{G})$ (the ***language*** of $\mathcal{G}$) is the set

$$L(\mathcal{G}) = \left\{ w \in \Sigma^* \;\middle|\; S \overset{*}{\Rightarrow} w \right\}..$$

That is, $L(G)$ is all the strings containing only terminals which can be derived from the start symbol of $\mathcal{G}$.

11.2.2 Ambiguity

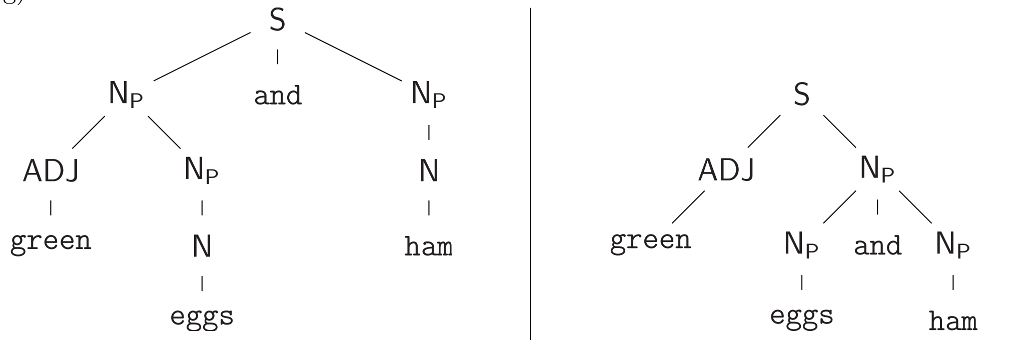
Consider the following grammar $\mathcal{G} = (V, \Sigma, R, \mathsf{S})$. Here

$$V = \{\mathsf{S}, N, \mathsf{N_P}, \mathsf{ADJ}\} \qquad \text{and} \qquad \Sigma = \{\mathtt{and}, \mathtt{eggs}, \mathtt{ham}, \mathtt{pencilgreen}, \mathtt{cold}, \mathtt{tasty}, \ldots\}.$$

The set R contains the following rules:

- $\mathsf{N_P} \rightarrow \mathsf{N_P}$ $\mathtt{and}$ $\mathsf{N_P}$
- $\mathsf{N_P} \rightarrow \mathsf{ADJ}\ \mathsf{N_P}$
- $\mathsf{N_P} \rightarrow \mathsf{N}$
- $\mathsf{N} \rightarrow \mathtt{eggs} \mid \mathtt{ham} \mid \mathtt{pencil} \mid \ldots$
- $\mathsf{ADJ} \rightarrow \mathtt{green} \mid \mathtt{cold} \mid \mathtt{tasty} \mid \ldots$
- ...

Here are two possible parse trees for the string `green eggs and ham` (ignore the spacing for the time being).



The two parse trees group the words differently, creating a different meaning. In the first case, only the eggs are green. In the second, both the eggs and the ham are green.

A string w is ***ambiguous*** with respect to a grammar $\mathcal{G}$ if w has more than one possible parse tree using the rules in $\mathcal{G}$.

Most grammars for practical applications are ambiguous. This is a source of real practical issues, because the end users of parsers (e.g. the compiler) need to be clear on which meaning is intended.

Removing ambiguity

There are several ways to remove ambiguity:

(A) Fix grammar so it is not ambiguous. (Not always possible or reasonable or possible.)

(B) Add grouping/precedence rules.

(C) Use semantics: choose parse that makes the most sense.

Grouping/precedence rules are the most common approach in programming language applications. E.g. "else" goes with the closest "if", * binds more tightly than +.

Invoking semantics is more common in natural language applications. For example, "The policeman killed the burgler with the knife." Did the burgler have the knife or the policeman? The previous context from the news story or the mystery novel may have made this clear. E.g. perhaps we have already been told that the burgler had a knife and the policeman had a gun.

Fixing the grammar is less often useful in practice, but neat when you can do it. Here's an ambiguous grammar with start symbol E. N stands for "number" and E stands for "expression".

$$\mathsf{E} \rightarrow \mathsf{E} \times \mathsf{E} \mid \mathsf{E}+\mathsf{E} \mid \mathsf{N}$$

$$\mathsf{N} \rightarrow \mathtt{0}\mathsf{N} \mid \mathtt{1}\mathsf{N} \mid \mathtt{0} \mid \mathtt{1}$$

An expression like $\mathtt{0110} \times \mathtt{110} + \mathtt{01111}$ has two parse trees and, therefore, we do not know which operation to do first when we evaluate it.

We can remove this ambiguity as follows, by rewriting the grammar as

$\mathsf{E} \rightarrow \mathsf{E} + \mathsf{T} \mid \mathsf{T}$

$\mathsf{T} \rightarrow \mathsf{N} \times \mathsf{T} \mid \mathsf{N}$

$\mathsf{N} \rightarrow \mathtt{0}\mathsf{N} \mid \mathtt{1}\mathsf{N} \mid \mathtt{0} \mid \mathtt{1}$

Now, the expression $\mathtt{0110} \times \mathtt{110} + \mathtt{01111}$ must be parsed with the + as the topmost operation.

Chapter 12

Lecture 12: Cleaning up CFGs and Chomsky Normal form

3 March 2009

In this lecture, we are interested in transforming a given grammar into a cleaner form. We start by describing how to clean up a grammar. Then, we show how to transform a cleaned up grammar into a grammar in Chomsky Normal Form.

12.1 Cleaning up a context-free grammar

The main problem with a general context-free grammar is that it might be complicated, contained parts that can not be used, and not necessarily effective.

It might be useful to think about CFG as a program, that we would like to manipulate (in a similar way that a compiler handles programs). If want to cleanup a program, we need to understand its structure. To this end, for CFGs, we will show a sequence of algorithms that analyze and cleanup a given CFG. Interestingly, these procedures uses similar techniques to the one used by compilers to manipulate programs being compiled.

Note, that some of the cleanup steps are not necessary if one just wants to transform a grammar into Chomsky Normal Form. In particular, Section 12.1.2 and Section 12.1.3 are not necessary for the CNF conversion. Note however, that the algorithm of Section 12.1.3 gives us am immediate way to decide if a grammar is empty or not, see Theorem 12.1.2.

12.1.1 Example of a messy grammar

For example, consider the following strange very strange grammar.

$$\text{(G1)} \qquad \boxed{\begin{array}{ll} \Rightarrow & \mathsf{S_0} \to \mathsf{S} \mid \mathsf{X} \mid \mathsf{Z} \\ & \mathsf{S} \to \mathsf{A} \\ & \mathsf{A} \to \mathsf{B} \\ & \mathsf{B} \to \mathsf{C} \\ & \mathsf{C} \to \mathsf{Aa} \\ & \mathsf{X} \to \mathsf{C} \\ & \mathsf{Y} \to \mathsf{aY} \mid \mathsf{a} \\ & \mathsf{Z} \to \epsilon. \end{array}}$$

This grammar is bad. How bad? Well, it has several problems:

(i) The variable Y can never be derived by the start symbol $\mathsf{S_0}$. It is a ***useless*** variable.

(ii) The rule $\mathsf{S} \to \mathsf{A}$ is redundant. We can replace any appearance of S by A, and reducing the number of variables by one. Rule of the form $\mathsf{S} \to \mathsf{A}$ is called a ***unit production*** (or ***unit rule***.

(iii) The variable A is also ***useless*** since we can note derive any word in Σ^* from A (because once we starting deriving from A we get into an infinite loop).

(iv) We also do not like Z, since one can generate ϵ from it (that is $\mathsf{Z} \overset{*}{\Longrightarrow} \epsilon$. Such a variable is called ***nullable***. We would like to have the property that only the start variable can be derived to ϵ.

We are going to present a sequence of algorithms that transform the grammar to not have these drawbacks.

12.1.2 Removing useless variables unreachable from the start symbol

Given a grammar $\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{R}, \mathsf{S})$, we would like to remove all variables that are not derivable from S. To this end, consider the following algorithm.

```
compReachableVars $\left(\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{R}, \mathsf{S})\right)$
        $V_{old} \leftarrow \emptyset$
        $V_{new} \leftarrow \{\mathsf{S}\}$
        while $V_{old} \neq V_{\mathrm{new}}$ do
            $V_{\mathrm{old}} \leftarrow V_{\mathrm{new}}$.
            for  $\mathsf{X} \in V_{old}$ do
                for  $(\mathsf{X} \rightarrow w) \in \mathcal{R}$ do
                    Add all variables appearing in $w$ to $V_{\mathrm{new}}$.
        return $V_{\mathrm{new}}$.
```

Clearly, this algorithm returns all the variables that are derivable form the start symbol S. As such, settings $\mathcal{V}' =$ **compReachableVars**$(\mathcal{G})$ we can set our new grammar to be $\mathcal{G}' = (\mathcal{V}', \Sigma, \mathcal{R}', \mathsf{S})$, where $\mathcal{R}'$ is the set of rules of $\mathcal{R}$ having only variables in $\mathcal{V}'$.

12.1.3 Removing useless variables that do not generate anything

In the next step we remove variables that do not generate any string.

Given a grammar $\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{R}, \mathsf{S})$, we would like to remove all variables that are not derivable from S. To this end, consider the following algorithm.

```
compGeneratingVars $\left(\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{R}, \mathsf{S})\right)$
        $V_{\mathrm{old}} \leftarrow \emptyset$
        $V_{\mathrm{new}} \leftarrow V_{\mathrm{old}}$
        do
            $V_{\mathrm{old}} \leftarrow V_{\mathrm{new}}$.
            for  $\mathsf{X} \in \mathcal{V}$ do
                for  $(\mathsf{X} \rightarrow w) \in \mathcal{R}$ do
                    if $w \in (\Sigma \cup V_{\mathrm{old}})^*$ then
                        $V_{\mathrm{new}} \leftarrow V_{\mathrm{new}} \cup \{\mathsf{X}\}$
        while $(V_{old} \neq V_{\mathrm{new}})$
        return $V_{\mathrm{new}}$.
```

As such, settings $\mathcal{V}' =$ **compReachableVars**$(\mathcal{G})$ we can set our new grammar to be $\mathcal{G}' = (\mathcal{V}', \Sigma, \mathcal{R}', \mathsf{S})$, where $\mathcal{R}'$ is the result of removing all rules that uses variables not in $\mathcal{V}'$. In the new grammar, every variable can derive some string.

Lemma 12.1.1 *Given a context-free grammar (CFG) $\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{R}, \mathsf{S})$ one can compute an equivalent CFG $\mathcal{G}'$ such that any variable of $\mathcal{G}'$ can derive some string in Σ^*.*

Note, that if a grammar $\mathcal{G}$ has an empty language, then the equivalent grammar generated by Lemma 12.1.1 will have no variables in it. Namely, given a grammar we have an algorithm to decide if the language it generates is empty or not.

Theorem 12.1.2 (CFG emptiness.) *Given a CFG $\mathcal{G}$, there is an algorithm that decides if the language of $\mathcal{G}$ is empty or not.*

Applying the algorithm of Section 12.1.2 together with the algorithm of Lemma 12.1.1 results in a CFG without any useless variables.

Lemma 12.1.3 *Given a CFG one can compute an equivalent CFG without any useless variables.*

12.2 Removing ϵ-productions and unit rules from a grammar

Next, we would like to remove ***ϵ-production*** (i.e., a rule of the form $\mathsf{X} \rightarrow \epsilon$) and ***unit-rules*** (i.e., a rule of the form $\mathsf{X} \rightarrow \mathsf{Y}$) from the language. This is somewhat subtle, and one needs to be careful in doing this removal process.

12.2.1 Discovering nullable variables

Given a grammar $\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{R}, \mathsf{S})$, we are interested in discovering all the nullable variables. A variable $\mathsf{X} \in \mathcal{V}$ is ***nullable***, if there is a way derive the empty string from X in $\mathcal{G}$. This can be done with the following algorithm.

```
compNullableVars $\left(\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{R}, \mathsf{S})\right)$
        $V_{\text{null}} \leftarrow \emptyset$
        do
            $V_{\text{old}} \leftarrow V_{\text{null}}$.
            for $\mathsf{X} \in \mathcal{V}$ do
                for $(\mathsf{X} \rightarrow w) \in \mathcal{R}$ do
                    if $w = \epsilon$ or $w \in (V_{\text{null}})^*$ then
                        $V_{\text{null}} \leftarrow V_{\text{null}} \cup \{\mathsf{X}\}$
        while $(V_{\text{null}} \neq V_{\text{old}})$
        return $V_{\text{null}}$.
```

12.2.2 Removing ϵ-productions

A rule is an ***ϵ-production*** if it is of the form $VX \rightarrow \epsilon$. We would like to remove all such rules from the grammar (or almost all of them).

To this end, we run **compNullableVars** on the given grammar $\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{R}, \mathsf{S})$, and get the set of all nullable variable V_{null}. If the start variable is nullable (i.e., $\mathsf{S} \in V_{\text{null}}$), then we create a new start state S', and add the rules to the grammar

$$\mathsf{S}' \rightarrow \mathsf{S} \mid \epsilon.$$

We also now remove all the other rules of the form $\mathsf{X} \rightarrow \epsilon$ from $\mathcal{R}$. Let $\mathcal{G}' = (\mathcal{V}', \Sigma, \mathcal{R}', \mathsf{S}')$ be the resulting grammar. The grammar $\mathcal{G}'$ is not equivalent to the original rules, since we missed some possible productions. For example, if we had the rule

$$\mathsf{X} \rightarrow \mathsf{ABC},$$

where B is nullable, then since B is no longer nullable (we removed all the ϵ-productions form the language), we missed the possibility that $\mathsf{B} \overset{*}{\Longrightarrow} \epsilon$. To compensate for that, we need to add back the rule

$$\mathsf{X} \rightarrow \mathsf{AC},$$

to the set of rules.

So, for every rule $\mathsf{A} \rightarrow X_1 X_2 \ldots X_m$ is in $\mathcal{R}'$, we add the rules of the form $\mathsf{A} \rightarrow \alpha_1 \ldots \alpha_m$ to the grammar, where

(i) If X_i is not nullable (its a character or a non-nullable variable), then $\alpha_i = X_i$.

(ii) If X_i is nullable, then α_i is either X_i or ϵ.

(iii) Not all α_is are ϵ.

Let $\mathcal{G}'' = (\mathcal{V}, \Sigma, \mathcal{R}', \mathsf{S}')$ be the resulting grammar. Clearly, no variable is nullable, except maybe the start variable, and there are no ϵ-production rules (except, again, for the special rule for the start variable).

Note, that we might need to feed $\mathcal{G}''$ into our procedures to remove useless variables. Since this process does not introduce new rules or variables, we have to do it only once.

12.3 Removing unit rules

A ***unit rule*** is a rule of the form $\mathsf{X} \rightarrow \mathsf{Z}$. We would like to remove all such rules from a given grammar.

12.3.1 Discovering all unit pairs

We have a grammar $\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{R}, \mathsf{S})$ that has no useless variables or ϵ-predictions. We would like to figure out all the unit pairs. A pair of variables Y and X is a ***unit pair*** if $\mathsf{X} \overset{*}{\Longrightarrow} \mathsf{Y}$ by $\mathcal{G}$. We will first compute all such pairs, and their we will remove all unit

Since there are no ϵ transitions in $\mathcal{G}$, the only way for $\mathcal{G}$ to derive Y from X, is to have a sequence of rules of the form

$$\mathsf{X} \rightarrow \mathsf{Z}_1, \mathsf{Z}_1 \rightarrow \mathsf{Z}_2, \ldots, \mathsf{Z}_{k-1} \rightarrow \mathsf{Z}_z = \mathsf{Y},$$

where all these rules are in $\mathcal{R}$. We will generate all possible such pairs, by generating explicitly the rules of the form $\mathsf{X} \rightarrow \mathsf{Y}$ they induce.

```
compUnitPairs $\left(\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{R}, \mathsf{S})\right)$
    $R_{\text{new}} \leftarrow \left\{ \mathsf{X} \rightarrow \mathsf{Y} \,\middle|\, (\mathsf{X} \rightarrow \mathsf{Y}) \in \mathcal{R} \right\}$
    do
        $R_{\text{old}} \leftarrow R_{\text{new}}$.
        for $(\mathsf{X} \rightarrow \mathsf{Y}) \in R_{\text{new}}$ do
            for $(\mathsf{Y} \rightarrow \mathsf{Z}) \in R_{\text{new}}$ do
                $R_{\text{new}} \leftarrow R_{\text{new}} \cup \{\mathsf{X} \rightarrow \mathsf{Z}\}$.
    while $(R_{\text{new}} \neq R_{\text{old}})$
    return $R_{\text{new}}$.
```

12.3.2 Removing unit rules

If we have a rule $\mathsf{X} \rightarrow \mathsf{Y}$, and $\mathsf{Y} \rightarrow w$, then if we want to remove the unit rule $\mathsf{X} \rightarrow \mathsf{Y}$, then we need to introduce the new rule $\mathsf{X} \rightarrow w$. We want to do that for all possible unit pairs.

```
removeUnitRules $\left(\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{R}, \mathsf{S})\right)$
    $U \leftarrow$ compUnitPairs$(\mathcal{G})$
    $\mathcal{R} \leftarrow \mathcal{R} \setminus U$
    for $(\mathsf{X} \rightarrow \mathsf{A}) \in U$ do
        for $(\mathsf{A} \rightarrow w) \in R_{\text{old}}$ do
            $\mathcal{R} \leftarrow \mathcal{R} \cup \{\mathsf{X} \rightarrow w\}$.
    return $(\mathcal{V}, \Sigma, \mathcal{R}, \mathsf{S})$.
```

We thus established the following result.

Theorem 12.3.1 *Given an arbitrary CFG, one can compute an equivalent grammar $\mathcal{G}'$, such that $\mathcal{G}'$ has no unit rules, no ϵ-productions (except maybe a single ϵ-production for the start variable), and no useless variables.*

12.4 Chomsky Normal Form

Chomsky Normal Form requires that each rule in the grammar is either

(C1) of the form $\mathsf{A} \rightarrow \mathsf{BC}$, where A, B, C are all variables and neither B nor C is the start variable.

(That is, a rule has exactly two variables on its right side.)

(C2) $\mathsf{A} \rightarrow \mathtt{a}$, where A is a variable and $\mathtt{a}$ is a terminal.

(A rule with terminals on its right side, has only a single character.)

(C3) $\mathsf{S} \rightarrow \epsilon$, where S is the start symbol.

(The start variable can derive ϵ, but this is the only variable that can do so.)

Note, that rules of the form $\mathsf{A} \rightarrow \mathsf{B}$, $\mathsf{A} \rightarrow \mathsf{BCD}$ or $\mathsf{A} \rightarrow \mathtt{a}\mathsf{C}$ are all illegal in a CNF.

Also a grammar in CNF never has the start variable on the right side of a rule.

Why should we care for CNF? Well, its an effective grammar, in the sense that every variable that being expanded (being a node in a parse tree), is guaranteed to generate a letter in the final string. As such, a word w of length n, must be generated by a parse tree that has $O(n)$ nodes. This is of course not necessarily true with general grammars that might have huge trees, with little strings generated by them.

12.4.1 Outline of conversion algorithm

All context-free grammars can be converted to CNF. We did most of the steps already. Here is an outline of the procedure:

(i) Create a new start symbol S_0, with new rule $\mathsf{S}_0 \rightarrow \mathsf{S}$ mapping it to old start symbol (i.e., S).

(ii) Remove nullable variables (i.e., variables that can generate the empty string).

(iii) Remove unit rules (i.e., variables that can generate each other).

(iv) Restructure rules with long righthand sides.

The only step we did not describe yet is the last one.

12.4.2 Final restructuring of a grammar into CNF

Assume that we already cleaned up a grammar by applying the algorithm of Theorem 12.3.1 to it. So, we now want to convert this grammar $\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{R}, \mathsf{S})$ into CNF.

Removing characters from right side of rules. As a first step, we introduce a variable V_c for every character $c \in \Sigma$ and it to $\mathcal{V}$. Next, we add the rules $\mathsf{V}_c \rightarrow c$ to the grammar, for every $c \in \Sigma$.

Now, for any string $w \in (\mathcal{V} \cup \Sigma)^*$, let $\widehat{w}$ denote the string, such that any appearance of a character c in w, is replaced by V_c.

Now, we replace every rule $\mathsf{X} \rightarrow w$, such that $|w| > 1$, by the rule $\mathsf{X} \rightarrow \widehat{w}$.

Clearly, (C2) and (C3) hold for the resulting grammar, and furthermore, any rule having variables on the right side, is made only of variables.

Making rules with only two variables on the right side. The only remaining problem, is that in the current grammar, we might have rules that are too long, since they have long string on the right side. For example, we might have a rule in the grammar of the form

$$\mathsf{X} \rightarrow \mathsf{B}_1\mathsf{B}_3 \ldots \mathsf{B}_k.$$

To make this into a binary rule (with only two variables on the right side, we remove this rule from the grammar, and replace it by the following set of rules

$$\begin{aligned}
&\mathsf{X} \rightarrow \mathsf{B}_1\mathsf{Z}_1 && \\
&\mathsf{Z}_1 \rightarrow \mathsf{B}_2\mathsf{Z}_2 && \mathsf{Z}_2 \rightarrow \mathsf{B}_3\mathsf{Z}_3 \\
\ldots& && \\
&\mathsf{Z}_{k-3} \rightarrow \mathsf{B}_{k-2}\mathsf{Z}_{k-2} && \\
&\mathsf{Z}_{k-2} \rightarrow \mathsf{B}_{k-1}\mathsf{B}_{k}, &&
\end{aligned}$$

where $\mathsf{Z}_1, \ldots, \mathsf{Z}_{k-2}$ are new variables.

We repeat this process, till all rules in the grammar is binary. This gramamr is now in CNF. We summarize our result.

Theorem 12.4.1 (CFG → CNF.) *Any context-free grammar can be converted into Chomsky normal form.*

12.4.3 An example of converting a CFG into CNF

Let us look at an example grammar with start symbol S.

$$(\text{G0}) \qquad \boxed{\begin{array}{ll} \Rightarrow & \mathsf{S} \rightarrow \mathsf{ASA} \mid \mathsf{aB} \\ & \mathsf{A} \rightarrow \mathsf{B} \mid \mathsf{S} \\ & \mathsf{B} \rightarrow \mathtt{b} \mid \epsilon \end{array}}$$

After adding the new start symbol S_0, we get the following grammar.

$$(\text{G1}) \qquad \boxed{\begin{array}{ll} \Rightarrow & \mathsf{S}_0 \rightarrow \mathsf{S} \\ & \mathsf{S} \rightarrow \mathsf{ASA} \mid \mathsf{aB} \\ & \mathsf{A} \rightarrow \mathsf{B} \mid \mathsf{S} \\ & \mathsf{B} \rightarrow \mathtt{b} \mid \epsilon \end{array}}$$

Removing nullable variables In the above grammar, both A and B are the nullable variables. We have the rule S → ASA. Since A is nullable, we need to add S → SA and S → AS and S → S (which is of course a silly rule, so we will not waste our time putting it in). We also have S → aB. Since B is nullable, we need to add S → a. The resulting grammar is the following.

$$(\text{G2}) \qquad \boxed{\begin{array}{ll} \Rightarrow & \mathsf{S}_0 \rightarrow \mathsf{S} \\ & \mathsf{S} \rightarrow \mathsf{ASA} \mid \mathsf{aB} \mid \mathsf{a} \mid \mathsf{SA} \mid \mathsf{AS} \\ & \mathsf{A} \rightarrow \mathsf{B} \mid \mathsf{S} \\ & \mathsf{B} \rightarrow \mathtt{b} \end{array}}$$

Removing unit rules. The unit pairs for this grammar are $\{\mathsf{A} \rightarrow \mathsf{B}, \mathsf{A} \rightarrow \mathsf{S}, \mathsf{S}_0 \rightarrow \mathsf{S}\}$. We need to copy the productions for S up to S_0, copying the productions for S down to A, and copying the production B → b to A → b.

$$(\text{G3}) \qquad \boxed{\begin{array}{ll} \Rightarrow & \mathsf{S}_0 \rightarrow \mathsf{ASA} \mid \mathsf{aB} \mid \mathsf{a} \mid \mathsf{SA} \mid \mathsf{AS} \\ & \mathsf{S} \rightarrow \mathsf{ASA} \mid \mathsf{aB} \mid \mathsf{a} \mid \mathsf{SA} \mid \mathsf{AS} \\ & \mathsf{A} \rightarrow \mathtt{b} \mid \mathsf{ASA} \mid \mathsf{aB} \mid \mathsf{a} \mid \mathsf{SA} \mid \mathsf{AS} \\ & \mathsf{B} \rightarrow \mathtt{b} \end{array}}$$

Final restructuring. Now, we can directly patch any places where our grammar rules have the wrong form for CNF. First, if the rule has at least two symbols on its righthand side but some of them are terminals, we introduce new variables which expand into these terminals. For our example, the offending rules are $\mathsf{S}_0 \rightarrow \mathtt{a}\mathsf{B}$, $\mathsf{S} \rightarrow \mathtt{a}\mathsf{B}$, and $\mathsf{A} \rightarrow \mathtt{a}\mathsf{B}$. We can fix these by replacing the $\mathtt{a}$'s with a new variable U, and adding a rule $\mathsf{U} \rightarrow \mathtt{a}$.

$$\text{(G4)} \qquad \boxed{\begin{array}{ll} \Rightarrow & \mathsf{S}_0 \rightarrow \mathsf{ASA} \mid \mathsf{UB} \mid \mathtt{a} \mid \mathsf{SA} \mid \mathsf{AS} \\ & \mathsf{S} \rightarrow \mathsf{ASA} \mid \mathsf{UB} \mid \mathtt{a} \mid \mathsf{SA} \mid \mathsf{AS} \\ & \mathsf{A} \rightarrow \mathtt{b} \mid \mathsf{ASA} \mid \mathsf{UB} \mid \mathtt{a} \mid \mathsf{SA} \mid \mathsf{AS} \\ & \mathsf{B} \rightarrow \mathtt{b} \\ & \mathsf{U} \rightarrow \mathtt{a} \end{array}}$$

Then, if any rules have more than two variables on their righthand side, we fix that with more new variables. For the grammar (G4), the offending rules are $\mathsf{S}_0 \rightarrow \mathsf{ASA}$, $\mathsf{S} \rightarrow \mathsf{ASA}$, and $\mathsf{A} \rightarrow \mathsf{ASA}$. We can rewrite these using a new variable Z and a rule $\mathsf{Z} \rightarrow \mathsf{SA}$. This gives us the CNF grammar shown on the right.

$$\boxed{\text{(G5)} \quad \begin{array}{ll} \Rightarrow & \mathsf{S}_0 \rightarrow \mathsf{AZ} \mid \mathsf{UB} \mid \mathtt{a} \mid \mathsf{SA} \mid \mathsf{AS} \\ & \mathsf{S} \rightarrow \mathsf{AZ} \mid \mathsf{UB} \mid \mathtt{a} \mid \mathsf{SA} \mid \mathsf{AS} \\ & \mathsf{A} \rightarrow \mathtt{b} \mid \mathsf{AZ} \mid \mathsf{UB} \mid \mathtt{a} \mid \mathsf{SA} \mid \mathsf{AS} \\ & \mathsf{B} \rightarrow \mathtt{b} \\ & \mathsf{U} \rightarrow \mathtt{a} \\ & \mathsf{Z} \rightarrow \mathsf{SA} \end{array}}$$

We are done!

Chapter 13

Leftover: Pushdown Automatas – PDAs

This lecture introduces pushdown automata, i.e. about the first half of section 2.2 from Sipser.

PDAs are the procedural counterpart to context-free grammars. A pushdown automaton (PDA) is simply an NFA with a stack. In a couple lectures, we'll prove that they generate the same languages.

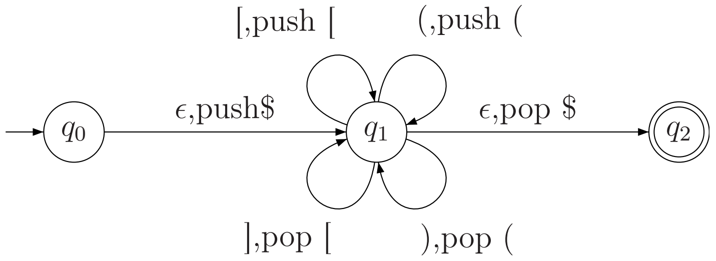
13.1 Bracket balancing example

Example: Let $\Sigma = \{(,),[,]\}$. $L = \{\text{properly nested strings from } \Sigma^*\}$. So ([]()) is in L, but not ([]) and not ([)].

So, what do we need to store on our stack to check whether an input is properly nested? Answer: ordered sequence of parentheses/brackets that are currently open.

Question: When should we accept? Answer: if the stack is empty when we have finished reading the input string. PDA's do not come with a built-in way to test if the stack is empty. They just reject input if we try to pop an empty stack. So we need to push a bottom-of-stack symbol onto the stack before we start to read the input.

Our machine looks like:



Do a short trace of the state sequence and sequence of stack contents as this machine recognizes the string [()()].

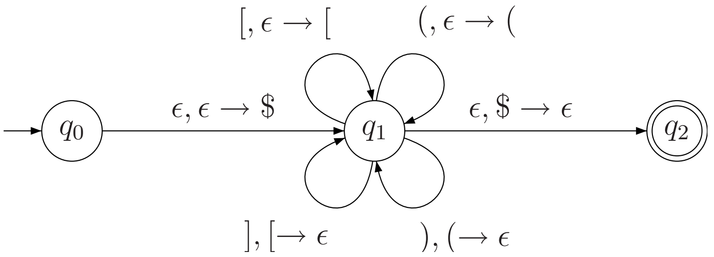
Formal notation for the labels on PDA transition arcs is

$$\boxed{c, s \rightarrow t},$$

where c is the character to be read from the input stream, s is the character to be popped from the top of the stack, and t is the character to be pushed back onto the stack. All of these can be ϵ.

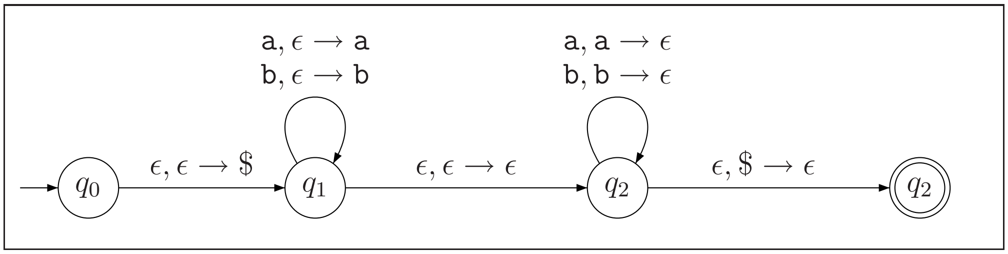
For example, to pop something from the stack, we use a label like $c, s \rightarrow \epsilon$. To push something onto the stack: $c, \epsilon \rightarrow t$. A transition like $c, s \rightarrow t$ pops s from the stack and substitutes t in its place.

So a properly drawn version of our state diagram would look like:


[, $\epsilon \rightarrow$ [
(, $\epsilon \rightarrow$ (
$\epsilon, \epsilon \rightarrow \$$
$\epsilon, \$ \rightarrow \epsilon$
q_0
q_1
q_2
], [$\rightarrow \epsilon$
), ($\rightarrow \epsilon$

13.2 The language ww^r

Another example: $L = \left\{ ww^R \;\middle|\; w \in \{\mathtt{a}, \mathtt{b}\}^* \right\}$.


$\mathtt{a}, \epsilon \rightarrow \mathtt{a}$
$\mathtt{b}, \epsilon \rightarrow \mathtt{b}$
$\mathtt{a}, \mathtt{a} \rightarrow \epsilon$
$\mathtt{b}, \mathtt{b} \rightarrow \epsilon$
$\epsilon, \epsilon \rightarrow \$$
$\epsilon, \epsilon \rightarrow \epsilon$
$\epsilon, \$ \rightarrow \epsilon$
q_0
q_1
q_2
q_2

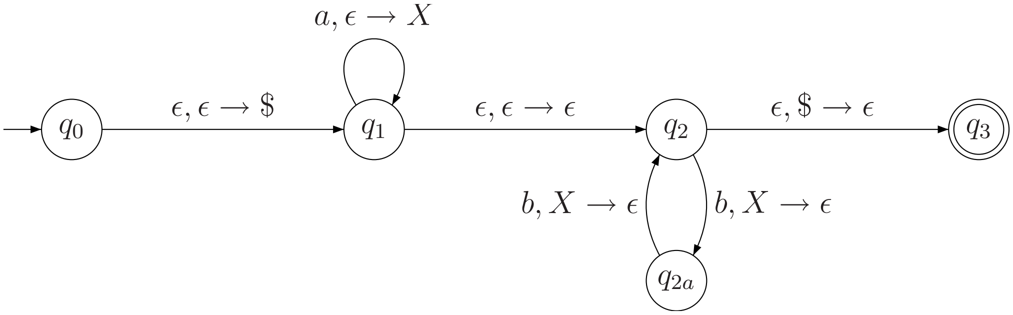
Notice a few things about this machine:

- It halts when we run out of input (like a DFA/NFA).
- You can take a transition if the input character **and** the top-of stack character both match what is on the transition or are ϵ.
- It is non-deterministic (like an NFA): more than one transition might match and you need to pick the "right" one. E.g. guessing when you have reached the midpoint of the string and need to take the transition from q_1 to q_2.

13.3 The language $\mathtt{a}^n\mathtt{b}^n$ with n being even

Another example: $L = \left\{ \mathtt{a}^n\mathtt{b}^n \;\middle|\; n \text{ is even} \right\}$.

Here's a PDAs that recognizes this language.

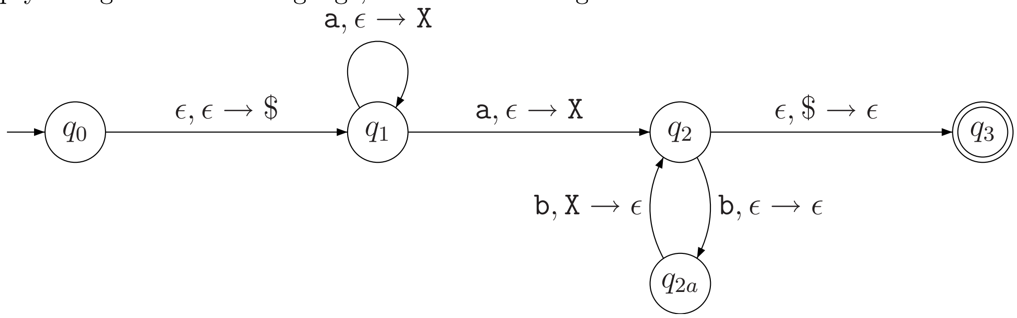

$a, \epsilon \rightarrow X$
$\epsilon, \epsilon \rightarrow \$$
$\epsilon, \epsilon \rightarrow \epsilon$
$\epsilon, \$ \rightarrow \epsilon$
q_0
q_1
q_2
q_3
$b, X \rightarrow \epsilon$
$b, X \rightarrow \epsilon$
q_{2a}

Things to notice:

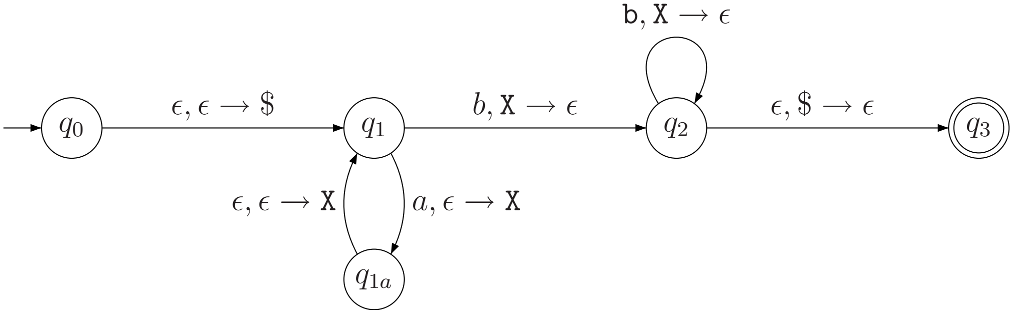
- The stack alphabet is different from the input alphabet. You can use some of the same characters but you don't have to.
- The 2-step loop involving q_2 and q_{2a} checks that the number of b's is even.

13.4 The language $\mathtt{a}^n\mathtt{b}^{2n}$

Another example: $L = \left\{ \mathtt{a}^n\mathtt{b}^{2n} \;\middle|\; n \geq 1 \right\}$. Notice that elements of L have twice as many b's as a's. Also, the empty string isn't in the language; the shortest string in L is abb.



Here's another one of the possible PDAs that recognizes this language.



Things to notice:

- The 2-step loop involving q_1 and q_{1a} reads one a off the input and pushes two X's onto the stack.
- The transition from $q_1$to q_2 explicitly reads an input character. So the empty string can not make it through to the final state.

13.5 Formal notation

Formally, a PDA is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$. The meanings of these are the same as for an NFA except:

- Γ is a finite stack alphabet
- $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \rightarrow \mathbb{P}(Q \times \Gamma_\epsilon)$

That is, the input to the transition function is a state, an input symbol, and a stack symbol. The input and stack symbols can be real characters from Σ and Γ or they can be ϵ. The output is a set of pairs (q, t), where q is a state and t is a stack symbol or ϵ.

A PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ accepts a string w if there is

(a) a sequence of states $s_0, s_1, \ldots s_k$,

(b) a sequence of characters and ϵ's $c_1, c_2, \ldots c_k$, and

(c) a sequence of strings $t_0, t_1, \ldots t_k$ (the stack snapshots),

such that

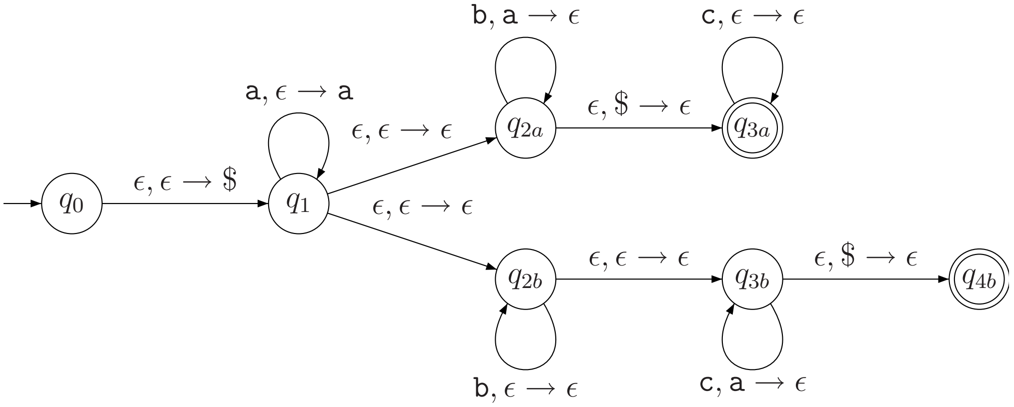
- $w = c_1 c_2 \ldots c_k$
- $s_0 = q_0$ and $t_0 = \epsilon$ (we start at the start state with an empty stack)
- $s_k \in F$ (we end at an accepting state)

- There are characters (or ϵ's) a and b and a string x such that $t_{k-1} = ax$, $t_k = bx$, and $(s_k, b) \in \delta(s_{k-1}, c_k, a)$. (Each change to the machine's state follows what is allowed by δ.)

The last condition will need some discussion and probably a picture or two.

13.6 A branching example

Let $L = \left\{ \mathtt{a}^i \mathtt{b}^j \mathtt{c}^k \,\middle|\, i = j \text{ or } i = k \right\}$.


q_0
$\epsilon, \epsilon \longrightarrow \$$
q_1
$\mathtt{a}, \epsilon \longrightarrow \mathtt{a}$
$\epsilon, \epsilon \longrightarrow \epsilon$
q_{2a}
$\mathtt{b}, \mathtt{a} \longrightarrow \epsilon$
$\epsilon, \$ \longrightarrow \epsilon$
q_{3a}
$\mathtt{c}, \epsilon \longrightarrow \epsilon$
$\epsilon, \epsilon \longrightarrow \epsilon$
q_{2b}
$\mathtt{b}, \epsilon \longrightarrow \epsilon$
$\epsilon, \epsilon \longrightarrow \epsilon$
q_{3b}
$\mathtt{c}, \mathtt{a} \longrightarrow \epsilon$
$\epsilon, \$ \longrightarrow \epsilon$
q_{4b}

It turns out that recognizing this language **requires** non-determinism. There is a deterministic version of a PDA, but it does not recognize as many languages as a normal (non-deterministic) PDA.

Chapter 14

Lecture 13: Even More on Context-Free Grammars

5 March 2009

14.1 Grammars in CNF form have compact parsing trees

In this section, we prove that CNF give very compact parsing trees for strings in the language of the grammar.

In the following, we will need the following easy observation.

Observation 14.1.1 *Consider a grammar G which is CNF, and a variable X of G which is not the start variable. Then, any string derived from X must be of length at least one.*

Claim 14.1.2 *Let $\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{R}, \mathsf{S})$ be a context-free grammar in Chomsky normal form, and w be a string of length one. Furthermore, assume that there is a $\mathsf{X} \in \mathcal{V}$, such that $\mathsf{X} \overset{*}{\Longrightarrow} w$ (i.e., w can be derived from X), and let T be the corresponding parse tree for w. Then the tree T has exactly $2\,|w| - 1$ internal nodes.*

Proof: A *full binary tree* is a tree were every node other than the leaves has two children. It is easy to verify by easy induction that such a tree with m leaves, has $m - 1$ internal nodes.

Now, the given tree T is not quite a full binary tree. Indeed, the kth leaf (from the left) of T, denoted by ℓ_k, is the k character of w, and its parent must be a node labeled by a variable, X_k, for $k = 1, \ldots, n$. Furthermore, we must have that the parent of ℓ_k has only a single child. As such, if we remove the n leafs of T, we remain with a full binary tree T' with n leafs (every parent of a leaf ℓ_k became a leaf). This tree is a full binary tree, because any internal node, must correspond to a non-terminal derivation of a CNF grammar, and any such derivation has the form $\mathsf{X} \rightarrow \mathsf{Y}VZ$; that is, the derivation corresponds to an internal node with two children in T. Now, the tree T' has $n - 1$ internal nodes, by the aforementioned fact about full binary trees. As such, T' has $n - 1$ internal nodes. Each leaf of T' is an internal node of T, and T' has n such leafs. We conclude that T has $2n - 1$ internal nodes. ■

Alternative proof for Claim 14.1.2.

Proof: The proof is by induction on the length of w.

If $|w| = 1$ then we claim that w must be derived by a single rule $\mathsf{X} \rightarrow c$, where c is a character. Otherwise, if the root corresponds to a rule of the form $\mathsf{X} \rightarrow \mathsf{YZ}$, then by the above observation, the generated string for Y and Z are each of length at least one, which implies that the overall length of the word is at least 2, a contradiction. (We are using here implicitly the property of a CNF that the start variable can never appear on the right side of a rule, and that the start symbol is the only symbol that can yield the empty string.)

As such, if $|w| = 1$, then the parsing tree has a single internal node, and the claim holds as $2\,|w| - 1 = 1$.

So, assume that we proved the claim for all words strictly shorter than w, and consider the parse tree T for w being derived from some variable $\mathsf{X} \in \mathcal{V}$. Since $|w| > 1$, it must be that the root of the parse tree T corresponds to a rule of the form $\mathsf{S} \rightarrow \mathsf{X}_1\mathsf{X}_2$. Let w_1 and w_2 be the portions of w generated by X_1 and X_2

respectively, and let T_1 and T_2 denote the corresponding subtrees of T. Clearly $w = w_1 w_2$, $|w_1| > 0$ and $|w_2| > 0$, by the above observation. Clearly, T_1 (resp. T_2) is a tree that derives w_1 (resp. w_2) from X_1 (resp. X_2). By induction, T_1 (resp. T_2) has $2|w_1| - 1$ (resp. $2|w_2| - 1$) internal nodes. As such, T has

$$\begin{aligned} N &= 1 + \begin{pmatrix} \text{\# internal} \\ \text{nodes of } T_1 \end{pmatrix} + \begin{pmatrix} \text{\# internal} \\ \text{nodes of } T_2 \end{pmatrix} \\ &= 1 + (2|w_1| - 1) + (2|w_2| - 1) = 2(|w_1| + |w_2|) - 1 \\ &= 2|w| - 1, \end{aligned}$$

■

The following is the same claim, restated as a claim on the number of derivation used.

Claim 14.1.3 *if G is a context-free grammar in Chomsky normal form, and w is a string of length $n \geq 1$, then any derivation of w from any variable X contains exactly $2n - 1$ steps.*

Theorem 14.1.4 *Given a context free grammar $\mathcal{G}$, and a word w, then one can decide if $w \in L(\mathcal{G})$ by an algorithm that always stop.*

Proof: Convert $\mathcal{G}$ into Chomsky normal form, and let $\mathcal{G}'$ be the resulting grammar tree. Let $n = |w|$. Observe that w has a parse tree using $\mathcal{G}'$ with $2n - 1$ internal nodes, by Claim 14.1.2. Enumerate all such possible parse trees (their number is large, but finite), and check if any of them is (i) a legal parse tree for $\mathcal{G}'$, and (ii) it derives the word w. If we found such a legal tree deriving w, then $w \in L(\mathcal{G})$. Otherwise, w can not be generated by $\mathcal{G}'$, which implies that $w \notin L(\mathcal{G})$. ■

14.2 Closure properties for context-free grammars

Context-free languages are closed under the following operations: union, concatenation, star, string reversal, homomorphism, and intersection with a regular language.

Notice that they are *not* closed under intersection (i.e. intersection of two context-free languages). Nor are they closed under set complement (i.e. the complement of a context-free language is not always context-free). We will show these non-closure facts later on, when we have established that some sample languages are definitely not context-free.

Example 14.2.1 Here is a quick argument why CFG languages are not closed under intersection. Consider the language $L = \left\{ \mathtt{a}^n \mathtt{b}^n \mathtt{c}^n \;\middle|\; n \geq 0 \right\}$, which (as we stated above is not CFG- a fact we will prove in the near future). However, it is easy to verify that any word $\mathtt{a}^n \mathtt{b}^n \mathtt{c}^n$, for $n \geq 0$, is in the intersection of the languages

$$L_1 = \left\{ \mathtt{a}^* \mathtt{b}^n \mathtt{c}^n \;\middle|\; n \geq 0 \right\} \quad \text{and} \quad L_2 = \left\{ \mathtt{a}^n \mathtt{b}^n \mathtt{c}^* \;\middle|\; n \geq 0 \right\}.$$

In fact, $L = L_1 \cap L_2$. Now, it is easy to verify that L_1 and L_2 are CFG, since

$$\begin{aligned} S &\rightarrow A_{\text{star}} X \\ A_{\text{star}} &\rightarrow \mathtt{a} A_{\text{star}} \mid \epsilon \\ X &\rightarrow \mathtt{b} X \mathtt{c} \mid \epsilon, \end{aligned}$$

with the start symbol being S is a CFG for L_1 (a similar grammar works for L_2). As such, both L_1 and L_2 are CFG, but their intersection $L = L_1 \cap L_2$ is not CFG. Thus, context-free languages are not closed under intersection.

14.2.1 Proving some CFG closure properties

Most of the closure properties are most easily proved using context-free grammars. These constructions are fairly easy, but they will help you become more familiar with the features of context-free grammars.

CFGs are closed under union

Suppose we have grammars for two languages, with start symbols S and T, respectively. Rename variables (in the two grammars) as needed to ensure that the two grammars do not share any variables. Then construct a grammar for the union of the languages, with start symbol Z, by taking all the rules from both grammars together and adding a new rule $Z \rightarrow S \mid T$.

Concatenation.

Suppose we have grammars for two languages, with start symbols S and T. Rename variables as needed to ensure that the two grammars do not share any variable. Then construct a grammar for the union of the languages, with start symbol Z, by taking all the rules from both grammars and adding a new rule $Z \rightarrow S\,T$.

Star operator

Suppose that we have a grammar for the language L, with start symbol S. The grammar for L^*, with start symbol T, contains all the rules from the original grammar plus the rule $T \rightarrow T\,S \mid \epsilon$.

String reversal

Reverse the character string on the righthand side of every rule in the grammar.

Homomorphism

Suppose that we have a grammar G for language L and a homomorphism h. To construct a grammar for $h(L)$, modify the righthand side of every rule in G to replace each terminal symbol t with its image $h(t)$ under the homomorphism.

14.2.2 CFG are closed under intersection with a regular language

Informal description

It is also true that the intersection of a context-free language with a regular language is always context-free. If we are manipulating a context-free language L in a proof, we can intersect it with a regular language to select a subset of L that has some particular form. For example, if L contains all strings with equal numbers of $\mathtt{a}$'s and $\mathtt{b}$'s, we can intersect it with $\mathtt{a}^*\mathtt{b}^*$ to get the language[1] $\mathtt{a}^n\mathtt{b}^n$.

So, assume we have a CFG $\mathcal{G} = (\mathcal{V}, \Sigma, R, \mathsf{S})$ accepting the context-free language L_{CFG} and a DFA $\mathtt{D} = (Q, \Sigma, \delta, q_{\mathrm{init}}, F)$ accepting a regular language L_{reg}. Furthermore, the CFG $\mathcal{G}$ is in Chomsky Normal Form (i.e., CNF).

The idea of building a grammar for the intersection language is to write a new CFG grammar, where a variable X of $\mathcal{G}$ would be replaced by the set of variables

$$\left\{ \mathsf{X}_{q \rightsquigarrow q'} \;\middle|\; q, q' \in Q \text{ and } \mathsf{X} \in \mathcal{V} \right\}.$$

Here, the variable $\mathsf{X}_{q \rightsquigarrow q'}$ represents all strings that can be derived by the variable X of $\mathcal{G}$, and furthermore if we feed such a string to $\mathtt{D}$ (starting at state q), then we would reach the state q'. So, consider a rule of the form

$$\mathsf{X} \rightarrow \mathsf{YZ}$$

that is in $\mathcal{G}$. For every possible starting state q, and ending state q', we want to generate a rule for the variable $\mathsf{X}_{q \rightsquigarrow q'}$. So we derive a substring w for Y. Feeding the $\mathtt{D}$ the string w, starting at q, would lead us to a state s. As such, the string generated from Yin this case, would move $\mathtt{D}$ from q to s, and the string generated by Zwould move $\mathtt{D}$ from s to q'. That is, this rule can be rewritten as

$$\forall q, q', s \in Q \quad \mathsf{X}_{q \rightsquigarrow q'} \rightarrow \mathsf{Y}_{q \rightsquigarrow s}\, \mathsf{Z}_{s \rightsquigarrow q'}.$$

[1] Here, and in a lot of other places, we abuse notations. When we write $\mathtt{a}^n\mathtt{b}^n$, what we really mean is the language $\left\{ \mathtt{a}^n\mathtt{b}^n \;\middle|\; n \geq 0 \right\}$.

If we have a rule of the from $\mathsf{X} \rightarrow c$ in $\mathcal{G}$, then we create the rule $\mathsf{X}_{q \rightsquigarrow q'} \rightarrow c$ if there is a transition in $\mathtt{D}$ from q to q' that reads the character c, where $c \in \Sigma_\epsilon$.

Finally, we create a new start variable S_0, and we introduce the rule $\mathsf{S}_0 \rightarrow \mathsf{S}_{q_{\mathrm{init}} \rightsquigarrow q'}$, where q_0 is the initial state of $\mathtt{D}$, and $q' \in F$ is an accept state of $\mathtt{D}$.

We claim that the resulting grammar accepts only words in the language $L_{\mathsf{CFG}} \cap L_{\mathrm{reg}}$.

Formal description

We have a CFG $\mathcal{G} = (\mathcal{V}, \Sigma, R, \mathsf{S})$ and a DFA $\mathtt{D} = (Q, \Sigma, \delta, q_{\mathrm{init}}, F)$. We now build a new grammar for the language $L(\mathcal{G}) \cap L(\mathtt{D})$. The set of new variables is

$$\mathcal{V}' = \{\mathsf{S}_0\} \cup \left\{ \mathsf{X}_{q \rightsquigarrow q'} \;\middle|\; \mathsf{X} \in \mathcal{R}, q, q' \in Q \right\}.$$

$$\mathcal{R}' = \left\{ \mathsf{X}_{q \rightsquigarrow q'} \rightarrow \mathsf{Y}_{q \rightsquigarrow s}\, \mathsf{Z}_{s \rightsquigarrow q'} \;\middle|\; \forall q, q', s \in Q \quad (\mathsf{X} \rightarrow \mathsf{YZ}) \in \mathcal{R} \right\} \tag{14.1}$$

$$\bigcup \left\{ \mathsf{S}_0 \rightarrow \mathsf{S}_{q_{\mathrm{init}} \rightsquigarrow q'} \;\middle|\; q' \in F \right\} \tag{14.2}$$

$$\bigcup \left\{ \mathsf{X}_{q \rightsquigarrow q'} \rightarrow c \;\middle|\; (\mathsf{X} \rightarrow c) \in \mathcal{R} \text{ and } \delta(q, c) = q' \right\}. \tag{14.3}$$

If $\mathsf{S} \rightarrow \epsilon \in \mathcal{R}$ and $q_{\mathrm{init}} \in F$ (i.e., ϵ is in the intersection language) then we add the rule $\{\mathsf{S}_0 \rightarrow \epsilon\}$ to $\mathcal{R}'$.

The new grammar is $\mathcal{G}_{\cap \mathtt{D}} = (\mathcal{V}', \Sigma, \mathcal{R}', \mathsf{S}_0)$.

Observation 14.2.2 *The new grammar $\mathcal{G}_{\cap \mathtt{D}}$ is "almost" a CNF. That is, if we ignore rules involving the start symbol S_0 of $\mathcal{G}_{\cap \mathtt{D}}$ then its a CNF.*

Correctness

Lemma 14.2.3 *Let $\mathcal{G}$ be a context-free grammar in Chomsky normal form, and let $\mathtt{D}$ be a DFA. Then one can construct a grammar is a grammar $\mathcal{G}_{\cap \mathtt{D}} = (\mathcal{V}', \Sigma, \mathcal{R}', \mathsf{S}_0)$, such that, for any word $w \in \Sigma^* \setminus \{\epsilon\}$, we have that $\mathsf{X} \overset{*}{\Longrightarrow} w$ and $\delta(q, w) = q'$ if and only if $\mathsf{X}_{q \rightsquigarrow q'} \overset{*}{\Longrightarrow} w$.*

Proof: The construction is described above, and proof is by induction of the length of w.

$\Big[|w| = 1\Big]$**:** If $|w| = 1$ then $w = c$, where $c \in \Sigma$.

Thus, if $\mathsf{X} \overset{*}{\Longrightarrow} w$ and $\delta(q, w) = q'$ then $\mathsf{X} \rightarrow c$ is in $\mathcal{R}$, which implies that we introduced the rule $\mathsf{X}_{q \rightsquigarrow q'} \rightarrow c$ into $\mathcal{R}$, which implies that $\mathsf{X}_{q \rightsquigarrow q'} \overset{*}{\Longrightarrow} w$.

Similarly, if $\mathsf{X}_{q \rightsquigarrow q'} \overset{*}{\Longrightarrow} w$ then since $\mathcal{G}_{\cap \mathtt{D}}$ is a CNF, and $|w| = 1$, this implies that there must be a derivation $\mathsf{X}_{q \rightsquigarrow q'} \rightarrow c$. But this implies, by construction, that $\mathsf{X} \rightarrow c$ is a rule of $\mathcal{G}$ and $\delta(q, c) = q'$, as required.

$\Big[|w| > 1\Big]$**:** Assume, that by induction, the claim holds for all words strictly shorter than w.

- $\mathsf{X} \overset{*}{\Longrightarrow} w$ and $\delta(q, w) = q' \implies \mathsf{X}_{q \rightsquigarrow q'} \overset{*}{\Longrightarrow} w$.

  IF $\mathsf{X} \overset{*}{\Longrightarrow} w$ and $\delta(q, w) = q'$, then consider the parse tree of $\mathcal{G}$ deriving X from w. Since $\mathcal{G}$ is a CNF, we have that the root of this parse tree T corresponds to a rule of the form $\mathsf{X} \rightarrow \mathsf{YZ}$. Let w_{Y} and w_{Z} be the two sub-words derived by these two subtrees of the T. Clearly, $w = w_{\mathsf{Y}} w_{\mathsf{Z}}$, and since $\mathcal{G}$ is a CNF, we have that $|w_{\mathsf{Y}}|, |w_{\mathsf{Z}}| > 0$ (since any symbol except the root in a CNF derives a word of length at least 1). As such, $|w_{\mathsf{Y}}|, |w_{\mathsf{Z}}| < |w|$. Now, let $q'' = \delta(q, w_{\mathsf{Y}})$. We have that

  $$\mathsf{Y} \overset{*}{\Longrightarrow} w_{\mathsf{Y}}, \;\; 0 < |w_{\mathsf{Y}}| < |w|, \quad \text{and} \quad q'' = \delta(q, w_{\mathsf{Y}}).$$

As such, by induction, it must be that $\mathsf{Y}_{q \rightsquigarrow q''} \overset{*}{\Longrightarrow} w_{\mathsf{Y}}$. Similarly, since $\delta(q'', w_{\mathsf{Z}}) = q'$, and by the same argument, we have that $\mathsf{Z}_{q'' \rightsquigarrow q'} \overset{*}{\Longrightarrow} w_{\mathsf{Z}}$. Now, by Eq. (14.1), we have the rule $\mathsf{X}_{q \rightsquigarrow q'} \rightarrow \mathsf{Y}_{q \rightsquigarrow q''} \mathsf{Y}_{q'' \rightsquigarrow q}$ in $\mathcal{R}'$. Namely,

$$\mathsf{X}_{q \rightsquigarrow q'} \rightarrow \mathsf{Y}_{q \rightsquigarrow q''} \mathsf{Y}_{q'' \rightsquigarrow q} \overset{*}{\Longrightarrow} w_{\mathsf{Y}} w_{\mathsf{Z}} = w,$$

implying the claim.

– $\mathsf{X}_{q \rightsquigarrow q'} \overset{*}{\Longrightarrow} w \quad \Longrightarrow \quad \mathsf{X} \overset{*}{\Longrightarrow} w$ and $\delta(q, w) = q'$.

If $\mathsf{X}_{q \rightsquigarrow q'} \overset{*}{\Longrightarrow} w$, and $|w| > 1$, then consider the parsing tree T' of w from $\mathsf{X}_{q \rightsquigarrow q'}$, and let

$$\mathsf{X}_{q \rightsquigarrow q'} \rightarrow \mathsf{Y}_{q \rightsquigarrow q''} \mathsf{Y}_{q'' \rightsquigarrow q}.$$

be the ruled used in the root of T', and let $w_{\mathsf{Y}}, w_{\mathsf{Z}}$ be the two substrings of w generated by these two subtrees. That is $w = w_{\mathsf{Y}} w_{\mathsf{Z}}$. By induction, we have that

$$\mathsf{Y} \overset{*}{\Longrightarrow} w_{\mathsf{Y}}, \delta(q, w_{\mathsf{Y}}) = q'', \qquad \text{and} \qquad \mathsf{Z} \overset{*}{\Longrightarrow} w_{\mathsf{Z}}, \delta(q'', w_{\mathsf{Z}}) = q'.$$

Now, by construction, the rule $\mathsf{X} \rightarrow \mathsf{YZ}$ must be in $\mathcal{R}$. As such $\mathsf{X} \rightarrow \mathsf{YZ} \overset{*}{\Longrightarrow} w_{\mathsf{Y}} w_{\mathsf{Z}} = w$, and

$$\delta(q, w) = \delta(q, w_{\mathsf{Y}} w_{\mathsf{Z}}) = \delta(\delta(q, w_{\mathsf{Y}}), w_{\mathsf{Z}}) = \delta(q'', w_{\mathsf{Z}}) = q'.$$

Thus, $\mathsf{X} \overset{*}{\Longrightarrow} w$ and $\delta(q, w) = q'$, thus implying the claim.

■

Theorem 14.2.4 *Let L be a context-free language and L' be a regular language. Then, $L \cap L'$ is a context free language.*

Proof: Let $\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{R}, \mathsf{S})$ be a CNF for L, and let $\mathtt{D} = (Q, \Sigma, \delta, q_{\text{init}}, F)$ be a DFA for L'. We apply the above construction to compute a grammar $\mathcal{G}_{\cap \mathtt{D}} = (\mathcal{V}', \Sigma, \mathcal{R}', \mathsf{S}_0)$ for the intersection.

- $w \in L \cap L' \implies w \in L(\mathcal{G}_{\cap \mathtt{D}})$.

  If $w = \epsilon$ then the rule $\mathsf{S}_0 \rightarrow \epsilon$ is in $\mathcal{G}_{\cap \mathtt{D}}$ and we have that $\epsilon \in L(\mathcal{G}_{\cap \mathtt{D}})$.

  For any other word, if $w \in L \cap L'$ then $\mathsf{S} \overset{*}{\Longrightarrow} w$ and $q' = \delta(q_{\text{init}}, w) \in F$ then, by Lemma 14.2.3, we have that

  $$\mathsf{S}_{q_{\text{init}} \rightsquigarrow q'} \overset{*}{\Longrightarrow} w.$$

  Furthermore, by construction, we have the rule

  $$\mathsf{S}_0 \rightarrow \mathsf{S}_{q_{\text{init}} \rightsquigarrow q'}.$$

  As such, $\mathsf{S}_0 \overset{*}{\Longrightarrow} w$, and $w \in L(\mathcal{G}_{\cap \mathtt{D}})$.

- $w \in L(\mathcal{G}_{\cap \mathtt{D}}) \implies w \in L \cap L'$.

  Similar to the above proof, and we omit it.

■

Chapter 15

Leftover: CFG to PDA, and Alternative proof of CNF effectiveness

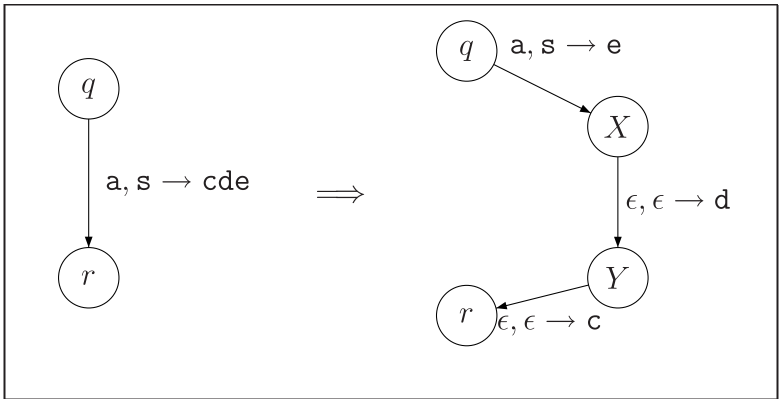
Here we start proving a set of results similar to those we proved for regular languages. That is, we will show that context-free grammars and pushdown automata generate the same languages. We will show that context-free languages are closed under a variety of operations, though not as many as regular languages. We will also prove a version of the pumping lemma, which we can use to show that certain languages (e.g. $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$) are not context-free.

15.1 PDA– Pushing multiple symbols

Our basic PDAs only allow a single symbol to be pushed onto the stack in each transition. And it takes two transitions to add something new to the stack if you also needed to examine the old character on its top. However, it's an easy extension to allow a single transition to push several symbols.

The notation for pushing several symbols is $a, s \rightarrow w$, where w is a string of stack characters. The last character in w is pushed first, so that the first character in w ends up on top of the stack. For example, a transition like $a, s \rightarrow cde$ (c, d and e are characters) pushes the e first, so the c ends up on top of the stack. This order is chosen because it makes notation simplest for the CFG to PDA conversion which we'll see next.

We can implement the multiple pushes with a series of extra states connected by regular PDA transitions. For example $a, s \rightarrow cde$ is implemented as follows:



15.2 CFG to PDA conversion

Claim 15.2.1 *Given any context-free grammar G for a language L, there is a PDA accepting L.*

Idea: The PDA needs to verify that there is a derivation for the input string w. Here's an algorithm that almost works:

- Push the start symbol onto the stack.
- Use the grammar rules from G to expand variables on the stack. The PDA guesses which rule to apply at each step.
- When the stack contains only terminals, compare this terminal string against the input w. Accept iff they are the same.

Consider the example grammar G with start symbol S: $\boxed{\begin{array}{l} S \rightarrow XB \\ X \rightarrow \mathtt{a}X\mathtt{b} \mid \epsilon \\ B \rightarrow \mathtt{c}B \mid \epsilon \end{array}}$

This generates the language $L = \left\{ \mathtt{a}^n\mathtt{b}^n\mathtt{c}^i \;\middle|\; i \geq 0, n \geq 0 \right\}$.
Here is a derivation for the string *aabbc*:

$$S \Rightarrow XB \Rightarrow \mathtt{a}X\mathtt{b}B \Rightarrow \mathtt{aa}X\mathtt{bb}B \Rightarrow \mathtt{aabb}B \Rightarrow \mathtt{aabbc}B \Rightarrow \mathtt{aabbc}.$$

If we could do this on the stack, we will get the following sequence of snapshots for the stack of the PDA.

top of stack →	S		X		a		a		a		a
	$		B		X		a		a		a
		⇒	$	⇒	b	⇒	b	⇒	b	⇒	b
					B		b		b		b
					$		B		c		c
							$		B		$
									$		

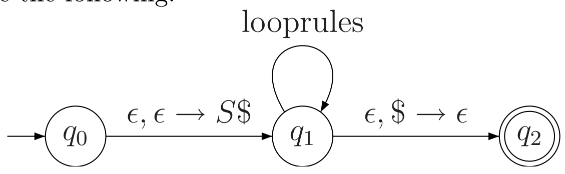
In the end of this process, the PDA would guess that it finished applying all the grammar rules, and it would each character of the input, verify that it is indeed equal to the top of the stack, pop the stack and go on to the next letter. Then, the PDA would verify that we had reached the bottom of the stack (by popping the character $ out of it), and move into the accept state.

Let us try to do this on the stack. We get two steps in, and the stack looks like the following. At this stage, we can not expand X because it is not on the top of the stack. There is a terminal (i.e., a) blocking our access to it.

a	⇐ top of the stack
X	
b	
B	

So, the "real" PDA has to interleave these two steps: expanding variables by guessing grammar rules and matching the input against the terminals at the top of the stack. We also need to mark the bottom of the stack, so we can make sure we have completely emptied it after we've finished reading the input string.

So the final PDA looks like the following.


looprules
q_0
$\epsilon, \epsilon \longrightarrow S\$$
q_1
$\epsilon, \$ \longrightarrow \epsilon$
q_2

The rules on the loop in this PDA are of two types:

- $\epsilon, X \rightarrow w$ for every rule $X \rightarrow w$ in G.
- $\mathtt{a}, \mathtt{a} \rightarrow \epsilon$ for every terminal $\mathtt{a} \in \Sigma$.

For our example grammar, the loop rules are:

- $S \to XB$,
- $X \to \mathtt{a}X\mathtt{b}$,
- $X \to \epsilon$,
- $B \to \mathtt{c}B$,
- $B \to \epsilon$,
- $\mathtt{a}, \mathtt{a} \to \epsilon$,
- $\mathtt{b}, \mathtt{b} \to \epsilon$,
- $\mathtt{c}, \mathtt{c} \to \epsilon$.

15.3 Alternative proof of CNF effectiveness

Claim 15.3.1 *if G is a context-free grammar in Chomsky normal form, and w is a string of length $n \geq 1$, then any derivation of w from any variable X contains exactly $2n-1$ steps.*

We include the proof for the sake of completeness. This claim is actually true for all derivations, not just leftmost ones. But the proof is easier to follow for leftmost derivations. We remind the reader that a ***leftmost derivation*** if at every step the leftmost remaining variable is the one replaced.

Proof: Proof by induction on the length of w. Suppose that the start symbol of G is S.

Base of induction: If $|w| = 1$, then w contains only a single character c. So any derivation of w from a variable X must contain exactly one step, using the rule $\mathsf{X} \to c$.

Induction: Suppose the claim is true for all strings of length $< k$, and let w be a string of length k and suppose we have a leftmost derivation for w starting with some variable X.

Consider the first step in this derivation. If the first step uses a rule of the form $\mathsf{X} \to c$, then w contains only a single character and we are back in the base case. So, let us consider the only other possibility: the first step uses a rule of the form $\mathsf{X} \to \mathsf{AB}$.

Then we can split up the derivation as follows: the first step expands X into AB. The next few steps expand A into some (non-empty) string x. The final few steps expand B into some string y. Suppose that $|x| = i$ and $|y| = j$. Since $w = xy$, we have that $i + j = k$.

Because the grammar is in Chomsky normal form, neither A nor B is the (glorious) start symbol. So neither A nor B can expand into the empty string. So x and y both contain some characters; that is, $i > 0$ and $j > 0$. This means that $0 < |x| = i < i + j = k$ and $0 < |y| = j < i + j = k$. Namely, $|x| < k$ and $|y| < k$. So we can apply our inductive hypothesis to x and y.

That is, since $|x| = i$, the derivation $\mathsf{A} \overset{*}{\Rightarrow} x$ must take exactly $2i-1$ steps. Similarly, the derivation $\mathsf{B} \overset{*}{\Rightarrow} y$ must take exactly $2j-1$ steps. But then the whole derivation of w from X takes $1 + (2i-1) + (2j-1) = 2(i+j) - 1 = 2k - 1$ steps. ■

Some things to notice about this proof:

- We remove the **first** step in the derivation. Not, for example, the final step. In general, results about grammars require removing the first step of a derivation or the top node in a parse tree.
- It almost never works right to start with a derivation or tree of size k and try to expand it into a derivation or parse tree of size $k+1$. Please avoid this common mistake.

  As an example of why inherent hopelessness of arguing in this <u>wrong</u> direction, consider the following grammar:

$$\text{(G6)} \qquad \boxed{\begin{array}{ll} \Rightarrow & \mathsf{S} \to \mathsf{A} \mid \mathsf{B} \\ & \mathsf{A} \to \mathtt{aa}\mathsf{A} \mid \epsilon \\ & \mathsf{B} \to \mathtt{bb}B \mid \mathtt{b} \end{array}}$$

  All the even-length strings are generated from A and contain all $\mathtt{a}$'s. We claim that all the odd-length strings are generated from B and contain all $\mathtt{b}$'s. As such, there is no way to prove this claim by induction, by taking the parse tree for an even-length string and graft on a couple nodes to make a parse tree for the next longer length string.

Chapter 16

Lecture 14: Repetition in context free languages

10 March 2009

16.1 Generating new words

We are interested in phenomena of repetition in context free languages. We had seen that regular languages repeat themselves if the strings are sufficiently long. We would like to make a similar statement about regular languages, but unfortunately, while the general statement is correct, the details are somewhat more involved.

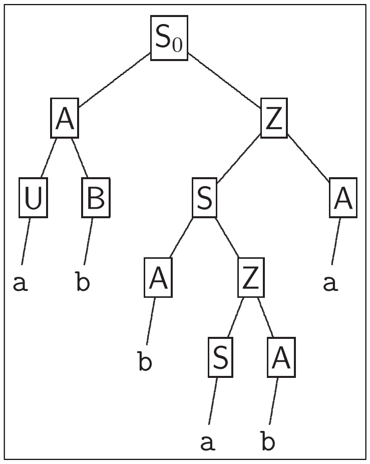
16.1.1 Example of repetition

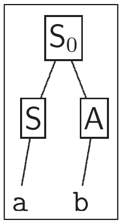
As a concrete example, consider the following context-free grammar which is in Chomsky normal form (CNF). (We remind the reader that any context free grammar can be converted into CNF, as such assuming that we have a grammar in CNF form does not restrict our discussion.)

As a concrete example, consider the following grammar from the previous lecture:

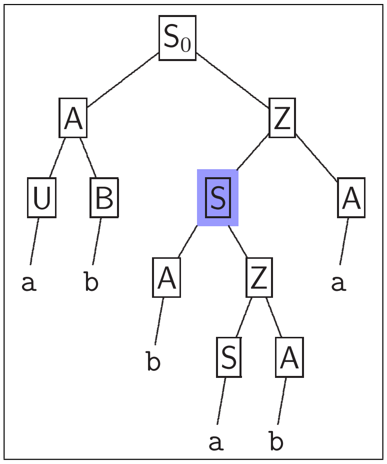
$$
\text{(G5)} \qquad
\begin{array}{ll}
\Rightarrow & \mathsf{S_0} \to \mathsf{AZ} \mid \mathsf{UB} \mid \mathtt{a} \mid \mathsf{SA} \mid \mathsf{AS} \\
 & \mathsf{S} \to \mathsf{AZ} \mid \mathsf{UB} \mid \mathtt{a} \mid \mathsf{SA} \mid \mathsf{AS} \\
 & \mathsf{A} \to \mathtt{b} \mid \mathsf{AZ} \mid \mathsf{UB} \mid \mathtt{a} \mid \mathsf{SA} \mid \mathsf{AS} \\
 & \mathsf{B} \to \mathtt{b} \\
 & \mathsf{U} \to \mathtt{a} \\
 & \mathsf{Z} \to \mathsf{SA}
\end{array}
$$

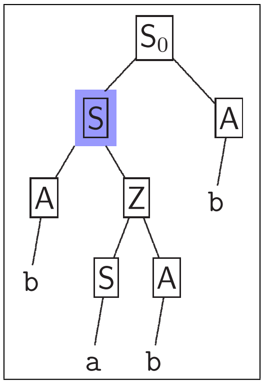
Next, consider the two words created from this grammar, as depicted on the right.



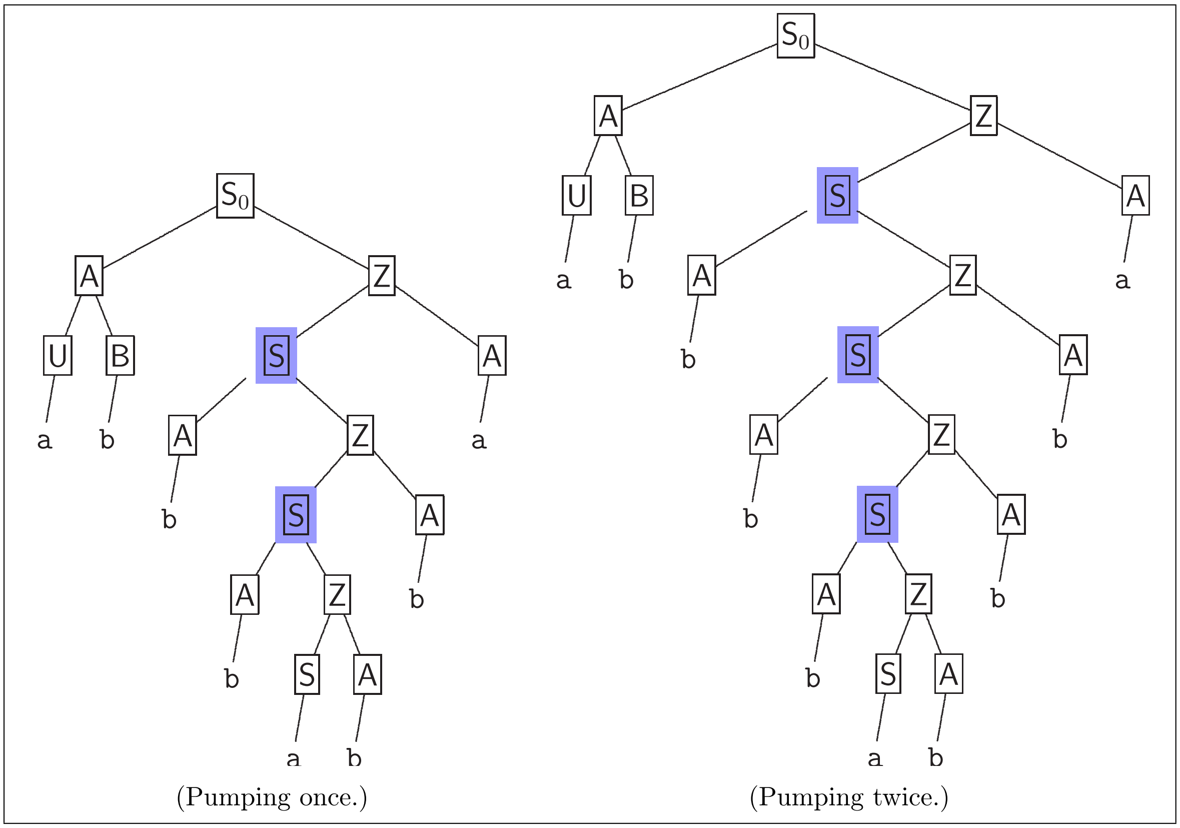


What if we wanted to make the second word longer without thinking too much? Well, both parse trees have subtrees with nodes generated by the variable S. As such, we could just cut the subtree of the first word using the variable S, and replace the subtree of S in the second word by this subtree, which would like the following:





Even more interestingly, we can do this cut and paste on the original tree:



(Pumping once.)

(Pumping twice.)

Naturally, we can repeat this pumping operation (cutting and pasting a subtree) as many times as want, see for example Figure 16.1. In particular, we get that the word $\texttt{abb}^i\texttt{ab}^i\texttt{a}$, for any i, is in the language of the grammar (G5). Notice that unlike the pumping lemma for regular languages, here the repetition happens in two places in the string. We claim that such a repetition (in two places) in the word must happen for any context free language, once we take a word which is sufficiently long.

16.2 The pumping lemma for CFG languages

So, assume we are given a context free grammar $\mathcal{G}$ which is in CNF, and it has m variables.

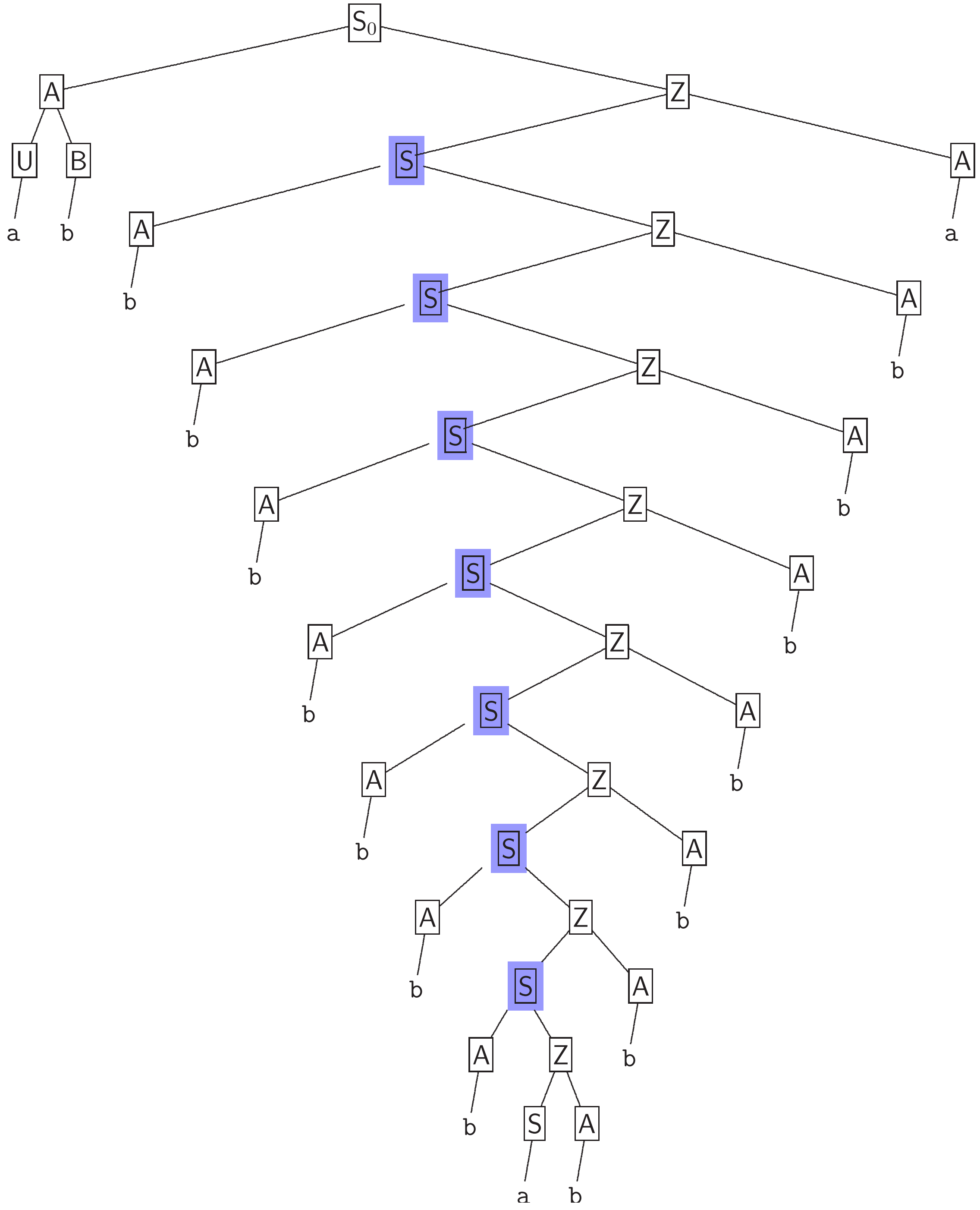
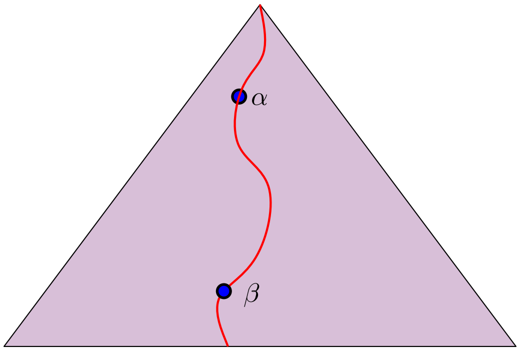


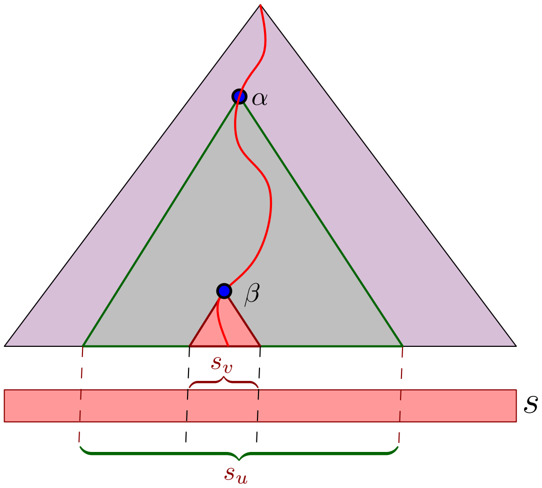
Figure 16.1: Naturally, one can pump the string as many times as one want, to get a longer and longer string.

16.2.1 If a variable repeats

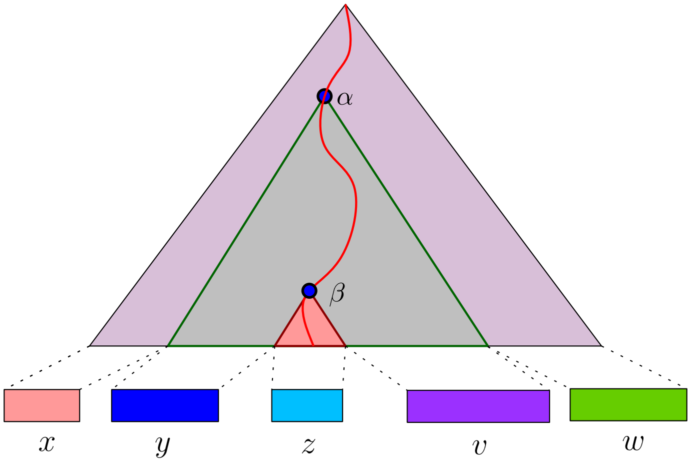
So, assume we have a parsing tree T for a word s (where the underlying grammar is in CNF), and there is a path in T from the root to a leaf, such that a variables repeats twice. So, say nodes α and β have the same variable (say S) stored in them:



The subtrees rooted at α and β corresponds to substrings of s:

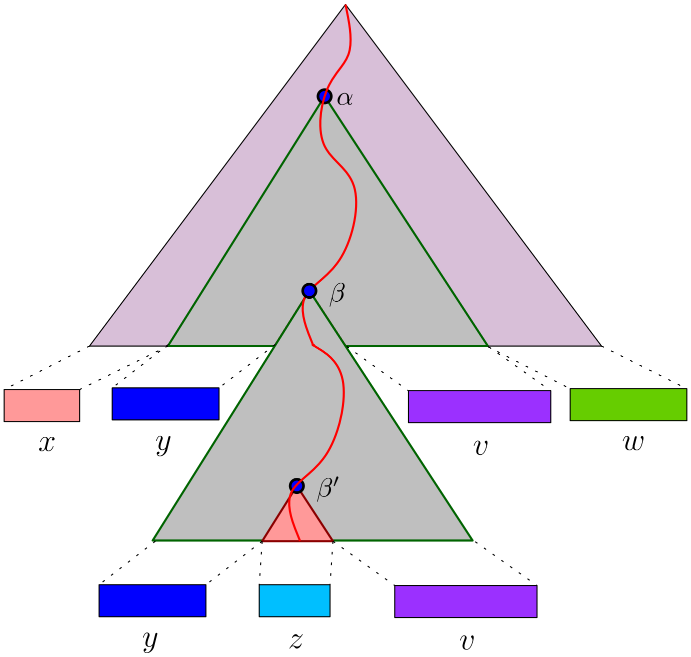


In particular, the substrings s_u and s_v break s into 5 parts:



Namely, s can be written as $s = xyzvw$.

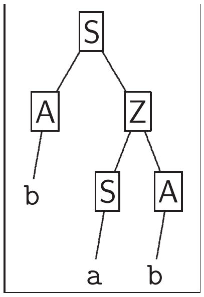
Now, if we copy the subtree rooted at α and copy it to β, we get a new parse tree:



The new tree is much bigger, and the new string it represents is $s = xyyzvvw$. In general, if we do this cut & paste operation $i - 1$ times, we get the string

$$xy^izv^iw.$$

16.2.2 How tall the parse tree have to be?



We will refer to a parse generated from a context free grammar in CNF form as a ***CNF tree***. A CNF tree has the special property that the parent of a leaf as a single child which is a terminal. The ***height*** of a tree is the maximum number of edges on a path from the root of the tree to a leaf. Thus, the tree depicted on the right has height 3.

The grammar $\mathcal{G}$ has m variables. As such, if the parse tree T has a path π from the root of length k, and $k > m$ (i.e., the path has k edges), then it must contain at least $m + 1$ variables (the last edge is between a variable and a terminal). As such, by the pigeon hole principle, there must be a repeated variable along π. In particular, a parse tree that does not have a repeated variable have height at most m.

Since $\mathcal{G}$ is in CNF, its a binary tree, and a variable either has two children, or a single child which is a leaf (and that leaf contains a single character of the input). As such, a tree of height at most m, contains at most 2^m leaves[1], and represents as such a string of length at most 2^m.

We restate the above observation formally for the record.

Observation 16.2.1 *If a CNF parse tree (or a subtree of such a tree) has height h, then the string it generates is of length at most 2^h.*

Lemma 16.2.2 *Let $\mathcal{G}$ be a grammar given in Chomsky Normal Form (CNF), and consider a word $s \in \mathsf{L}(\mathcal{G})$, such that $\ell = |s|$ is strictly larger than 2^m (i.e., $\ell > 2^m$). Then, any parse tree T for s (generated by $\mathcal{G}$) must have a path from the root to some leaf with a repeated variable on it.*

[1] In fact, a CNF tree of height m can have at most 2^{m-1} leaves (figure out why), but thats a subtlety we will ignore that anyway works in our favor.

Proof: Assume for the sake of contradiction that T has no repeated variable on any path from the root, then the height of T is at most m. But a parse tree of height m for a CNF can generate a string of length at most 2^m. A contradiction, since $\ell = |s| > 2^m$. ■

16.2.3 Pumping Lemma for CNF grammars

We need the following observation.

Lemma 16.2.3 (CNF is effective.) *In a CNF parse tree T, if u and v are two nodes, both storing variables in them, and u is an ancestor of v, then the string S_u generated by the subtree of u is strictly longer than the substring S_v generated by the subtree of u. Namely, $|S_u| > |S_v|$. (Of course, S_v is a substring of S_u.)*

Proof: Assume that the node u stores a variable X, and that we had used the rule $\mathsf{X} \rightarrow \mathsf{BC}$ to generate its two children u_L and u_R. Furthermore, assume that u_L and u_R generated the strings S_L and S_R, respectively. The string generated by v must be a substring of either S_L or S_R. However, CNF has the property that no variable[2] can generate the empty word ϵ. As such $|S_R| > 0$ and $|S_L| > 0$.

In particular, assume without loss of generality, that v is in the left subtree of u, and as such S_v is a substring of S_L. We have that

$$|S_v| \leq |S_L| < |S_L| + |S_R| = |S_u| .$$

■

Lemma 16.2.4 *Let T be a tree, and π be the longest path in the tree realizing the height h of T. Fix $k \geq 0$, and let u be the kth node from the end of π (i.e., u is in distance $h - k$ from the root of T). Then the tree rooted at u has height at most k.*

Proof: Let r be the root of T, and assume, for the sake of contradiction, that T_u (i.e., the subtree rooted at u) has height larger than k, and let σ be the path from u to the leaf γ of T_u realizing this height (i.e., the length of σ is $> k$). Next, consider the path formed by concatenating the path in T from r to u with the path σ. Clearly, this is a new path of length $h - k + |\sigma| > h$ that leads from the root of T into a leaf of T. As such, the height of T is larger than h, which is a contradiction. ■

Lemma 16.2.5 (Pumping lemma for Chomsky Normal Form (CNF).) *Let $\mathcal{G}$ be a CNF context-free grammar with m variables in it. Then, given any word $\mathcal{S}$ in $\mathsf{L}(\mathcal{G})$ of length $> 2^m$, one can break $\mathcal{S}$ into 5 substrings $\mathcal{S} = xyzvw$, such that for any $i \geq 0$, we have that xy^izv^iw is a word in $\mathsf{L}(\mathcal{G})$. In addition, the following holds:*

1. *The strings y and v are not both empty (i.e., the pumping is getting us new words).*

2. *$|yzv| \leq 2^m$.*

Proof: Let T be a CNF parse tree for $\mathcal{S}$ (generated by $\mathcal{G}$). Since $\ell = |s| > 2^m$, by Lemma 16.2.2, there is a path in T from its root to a leaf which has a repeated variable (and its length is longer than m). In fact, let π be the longest path in T from the root to a leaf (i.e., π is the path realizing the height of the tree T). We know that T has more than $m + 1$ variables on it and as such it has a repetition.

We need to be a bit careful in picking the two nodes α and β on π to apply the pumping to. In particular, let α be the last node on π such that there is a repeated appearance of the symbol stored in u later in the path. Clearly, the length of the subpath τ of π starting at α till the end of π has at most m symbols on it (because otherwise, there would be another repetition on π). Let β be the node of $\tau \subseteq \pi$ which has repetition of the symbol stored in α.

By Lemma 16.2.4 the subtree T_α (i.e., the subtree of T rooted at α) has height at most m. As above, T_α and T_β generate two strings $\mathcal{S}_\alpha$ and $\mathcal{S}_\beta$, respectively. By Observation 16.2.1, we have that $|\mathcal{S}_\alpha| \leq 2^m$. By

[2] Except the start variable, but this not relevant here.

Lemma 16.2.3, we have that $|\mathcal{S}_\alpha| > |\mathcal{S}_\beta|$. As such, the two substrings $\mathcal{S}_\alpha$ and $\mathcal{S}_\beta$ breaks $\mathcal{S}$ into 5 substrings $\mathcal{S} = xyzvw$. Here, we have

$$\mathcal{S} = x \quad \overbrace{y \quad \underbrace{z}_{=\mathcal{S}_\beta} \quad v}^{\mathcal{S}_\alpha=} \quad w.$$

As such, we know that $|yv| = |\mathcal{S}_\alpha| - |\mathcal{S}_\beta| > 0$. Namely, the strings y and v are not both empty. Furthermore, $|yzv| = |\mathcal{S}_\alpha| \leq 2^m$.

The remaining task is to show the pumping. Indeed, if we replace T_β by the tree T_α we get a parse tree generating the string xy^2zv^2w. If we repeat this process $i-1$ times, we get the word

$$xy^izv^iw \in \mathsf{L}(\mathcal{G}),$$

for any i, establishing the lemma. ■

Lemma 16.2.6 (Pumping lemma for context-free languages.) *If L is a context-free language, then there is a number p (the pumping length) where, if $\mathcal{S}$ is any string in L of length at least p, then $\mathcal{S}$ may be divided into five pieces $\mathcal{S} = xyzvw$ satisfying the conditions:*

1. *for any $i \geq 0$, we have $xy^izv^iw \in L$,*
2. *$|yv| > 0$,*
3. *and $|yzv| \leq p$.*

Proof: Since L is context free it has a CNF grammar $\mathcal{G}$ that generates it. Now, if m is the number of variables in $\mathcal{G}$, then for $p = 2^m + 1$, the lemma follows by Lemma 16.2.5. ■

16.3 Languages which are not context-free

16.3.1 The language $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ is not context-free

Lemma 16.3.1 *The language $L = \left\{ \mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \;\middle|\; n \geq 0 \right\}$ is not context-free.*

Proof: Assume, for the sake of contradiction, that L is context-free, and apply the Pumping Lemma to it (Lemma 16.2.6). As such, there exists $p > 0$ such that any word in L longer than p can be pumped. So, consider the word $\mathcal{S} = \mathtt{a}^{p+1}\mathtt{b}^{p+1}\mathtt{c}^{p+1}$. By the pumping lemma, it can be written as $\mathtt{a}^{p+1}\mathtt{b}^{p+1}\mathtt{c}^{p+1} = xyzvw$, where $|yzv| \leq p$.

We claim, that yzv can made out of only two characters. Indeed, if yzv contained all three characters, it would have to contain the string $\mathtt{b}^{p+1}$ as a substring (as $\mathtt{b}^{p+1}$ separates all the appearances of $\mathtt{a}$ from all the appearances of $\mathtt{c}$ in $\mathcal{S}$). This would require that $|yzv| > p$ but we know that $|yzv| \leq p$.

In particular, let $i_\mathtt{a}, i_\mathtt{b}$ and $i_\mathtt{c}$ be the number of $\mathtt{a}$s, $\mathtt{b}$s and $\mathtt{c}$s in the string yv, respectively. All we know is that $i_\mathtt{a} + i_\mathtt{b} + i_\mathtt{c} = |yv| > 0$ and that $i_\mathtt{a} = 0$ or $i_\mathtt{c} = 0$. Namely, $i_\mathtt{a} \neq i_\mathtt{b}$ or $i_\mathtt{b} \neq i_\mathtt{c}$ (the case $i_\mathtt{a} \neq i_\mathtt{c}$ implies one of these two cases). In particular, by the pumping lemma, the word

$$\mathcal{S}_2 = xy^2zv^2w \quad \in \quad L.$$

We have the following:

character	how many times it appears in $\mathcal{S}_2$
$\mathtt{a}$	$p + 1 + i_\mathtt{a}$
$\mathtt{b}$	$p + 1 + i_\mathtt{b}$
$\mathtt{c}$	$p + 1 + i_\mathtt{c}$

If $i_\mathtt{a} \neq i_\mathtt{b}$ then $\mathcal{S}_2$, by the above table, does not have the same number of $\mathtt{a}$s and $\mathtt{b}$s and as such it is not in L.

If $i_\mathtt{b} \neq i_\mathtt{c}$ then $\mathcal{S}_2$, by the above table, does not have the same number of $\mathtt{b}$s and $\mathtt{c}$s and as such it is not in L.

In either case, we get that $\mathcal{S}_2 \notin L$, which is a contradiction. Namely, our assumption that L is context-free is false. ■

16.4 Closure properties

16.4.1 Context-free languages are not closed under intersection

We know that the languages

$$L_1 = \left\{ \mathtt{a}^* \mathtt{b}^n \mathtt{c}^n \;\middle|\; n \geq 0 \right\} \text{ and } L_2 = \left\{ \mathtt{a}^n \mathtt{b}^n \mathtt{c}^* \;\middle|\; n \geq 0 \right\}$$

are context-free (prove this). But

$$L = \left\{ \mathtt{a}^n \mathtt{b}^n \mathtt{c}^n \;\middle|\; n \geq 0 \right\} = L_1 \cap L_2$$

is not context-free by Lemma 16.3.1. We conclude that the intersection of two context-free languages is not necessarily context-free.

Lemma 16.4.1 *Context-free languages are not closed under intersection.*

16.4.2 Context-free languages are not closed under complement

Lemma 16.4.2 *Context-free languages are not closed under complement.*

Proof: Consider the language

$$L_3 = \left\{ \mathtt{a}^i \mathtt{b}^j \mathtt{c}^k \;\middle|\; i \neq j \text{ or } j \neq k \right\}.$$

The language L_3 is clearly context-free (why?). Consider its complement language $\overline{L_3}$. If L_3 is context-free, and context-free languages are closed under complement (which we are assuming here for the sake of contradiction), then $\overline{L_3}$ is context-free.

Now, $\overline{L_3}$ contains many strings we are not interested in (for example, `cccccba`. So, let us intersect it with the regular language $\mathtt{a}^*\mathtt{b}^*\mathtt{c}^*$. We proved (in previous lecture), that the intersection of a context-free language and a regular language is still context-free. As such,

$$\widehat{L} = \overline{L_3} \cap \{\mathtt{a}^*\mathtt{b}^*\mathtt{c}^*\}$$

is context-free. But this language contains all the strings $\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k$, where $i = j$ and $j = k$. That is, its the language of Lemma 16.3.1, which we know is not context-free. A contradiction. ■

Proof: (Alternative proof.) The language $L = \mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ can be written, by De Morgan laws, as

$$L = \mathtt{a}^n\mathtt{b}^n\mathtt{c}^* \cap \mathtt{a}^*\mathtt{b}^n\mathtt{c}^n = \overline{\overline{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^*} \cup \overline{\mathtt{a}^*\mathtt{b}^n\mathtt{c}^n}}. \tag{16.1}$$

We assume for the sake of contradiction, that context-free languages are closed under complement, and we already know that they are closed under union. However, the languages

$$\mathtt{a}^n\mathtt{b}^n\mathtt{c}^* \text{ and } \mathtt{a}^*\mathtt{b}^n\mathtt{c}^n$$

are context-free. As such, by closure properties and Eq. (16.1), the language L is context-free, which is a contradiction to Lemma 16.3.1. ■

Chapter 17

Leftover: PDA to CFG conversion

This lecture covers the construction for converting a PDA to an equivalent CFG (Section 2.2 of Sipser). We also cover the Chomsky Normal Form for context-free grammars and an example of grammar-based induction.

If it looks like this lecture is too long, we can push the grammar-based induction part into lecture 15.

17.1 NFA to CFG conversion

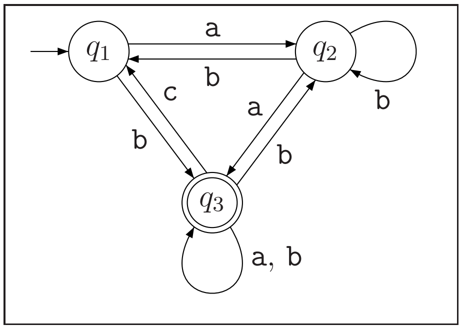
Before converting a PDA to a context-free grammar, let's first see how to convert an NFA to a context-free grammar.

The idea is quite simple: We introduce a symbol in our grammar for each state in the given NFA $N = (Q, \Sigma, \delta, q_0, F)$. We introduce a symbol for every state, and a rule for every transition. In particular, a state q_j would correspond to a symbol L_j (naturally, the language generated by L_j is the suffix language of the state q_j). A transition $q_j \in \delta(q_i, x)$, for $x \in \Sigma_\epsilon$, would be translated into the rule

$$L_i \rightarrow xL_j$$

We also add for any accepting state q_i the rule $L_i \rightarrow \epsilon$. (Note, that x can be ϵ. Then the ϵ-transition of moving from q_i to q_j, is translated into the rule $L_i \rightarrow L_j$.

As a concrete example, consider the NFA on the right. We introduce a CFG with symbols L_1, L_2, L_3, where L_1 is the initial symbol. We have the following rules:



$$\begin{aligned}
L_1 &\rightarrow \mathtt{a}L_2 \mid \mathtt{b}L_3 \\
L_2 &\rightarrow \mathtt{b}L_1 \mid \mathtt{b}L_2 \mid \mathtt{a}L_3 \\
L_3 &\rightarrow \mathtt{b}L_2 \mid \mathtt{c}L_1 \mid \mathtt{a}L_3 \mid \mathtt{b}L_3 \mid \epsilon.
\end{aligned}$$

Interestingly, the state q_3 is an accept state, and as such we add the transition $L_3 \rightarrow \epsilon$ to the rules.

17.2 PDA to CFG conversion

In this section, we will show how to convert a PDA into an equivalent CFG. We will first convert the PDA into a "normalized" form, and then we will generate a context free grammar (CFG) that explicitly writes down all the input strings that get the PDA from one state to another. Next, we will convert this normalized PDA into a CFG.

17.2.1 From PDA to a normalized PDA

Given a PDA N' we would like to convert into a PDA N that has the following three properties:

(A) There is a single accept state q_{acc}.

(B) It empties the stack before accepting.

(C) Each transition either pushes a symbol to the stack or pop it, but not both.

Transforming a given PDA into an equivalent PDA with these properties might seem like a tool order initially, but in fact can be easily done with some care.

(A) There is a single accept state q_{acc}.

This can be easily enforced by creating a new state q_{acc}, and creating ϵ-transitions from the old accept states to this new accept state.

(B) It empties the stack before accepting.

This can be easily enforced by pushing a special character \$ into the stack in the start state (introducing a new start state in the process). Next, we introduce a new temporary state q_{temp} which replaces q_{acc}, which has transitions popping all characters from the stack (excepting \$), and finally, we introduce the transition

$$q_{\text{temp}} \xrightarrow{\$} q_{\text{acc}}.$$

And of course, q_{acc} is the only accept state in this new automata.

(C) Each transition either pushes a symbol to the stack or pop it, but not both.

One bad case for us is a transition that both pushes and pops from the stack. For example, we have a transition

$$q_i \xrightarrow{\mathtt{x},\mathtt{b}\rightarrow\mathtt{c}} q_j.$$

(Read the character x from the input, pop $\mathtt{b}$ from the stack, and push $\mathtt{c}$ instead of it.) To remove such transitions, we will introduce a special state q'_{temp}, and introduce two transitions – one doing the pop, the other doing the push. Formally, for the above transition, we will introduce the transitions

$$q_i \xrightarrow{\mathtt{x},\mathtt{b}\rightarrow\epsilon} q'_{\text{temp}} \quad \text{and} \quad q'_{\text{temp}} \xrightarrow{\epsilon,\epsilon\rightarrow\mathtt{c}} q_j.$$

Similarly, if we have a transition that neither pushes nor pops anything, we replace it with a sequence of two transitions that push and then immediately pop some newly-created dummy stack symbol.

In the end of this normalization process, we end up with an equivalent PDA N that complies with our requirements.

17.2.2 From a normalized PDA to CFG

intuition

Consider a run of a normalized PDA $N = (Q, \Sigma, \delta, q_{\text{init}}, F)$ that accepts a word w. It starts at a state q_{init} (with an empty stack), and ends up at q_{acc}, again, with an empty stack. As such, it is natural to define for any two states $p, q \in Q$, the language $L_{p,q}$ of all the strings that starts at p with an empty stack, and end up in q with an empty stack.

For each pair of states p and q, we will have a symbol $S_{p,q}$ in our CFG for the language $L_{p,q}$. $S_{p,q}$ will generate all the strings in $L_{p,q}$. The language of N is then $L_{q_{\text{init}},q_{\text{acc}}}$.

So, consider a word $w \in L_{p,q}$ and how the PDA N works for this word. In particular, consider the stack as the PDA starting at p (with an empty stack) handles w.

Stack is empty in the middle. If during this execution, the stack ever becomes empty at some intermediate state r, then a word of $L_{p,q}$ can be formed by concatenating a word of $L_{p,r}$ (that got N from state p into state r with an empty stack), and a word of $L_{r,q}$ (that got N from r to q).

Stack never empty in the middle. The other possibility is that the stack is never empty in the middle of the execution as N transits from p to q, for the input $w \in L_{p,q}$. But then, it must be that the first transition (from p into say p_1) must have been a push, and the last transition into q (from say q_1) was a pop. Furthermore, the this pop transition, popped exactly the character pushed into the stack by the first transition (from p to p_1). Thus, if the PDA read the character x (from the input) as it moved from p to p_1 and read the letter y (from the input) as it moved from q_1 to q, then

$$w = xw'y,$$

where w' is an input that causes the PDA N to start from p_1 with an empty stack, and end up in q_1 with an empty stack. Namely, $w' \in L_{p_1 q_1}$.

Formally, if there is a push transition (pushing z into the stack) from p to p_1 (reading x) and pop transition from q_1 to q (popping the same z from the stack and reading y), then a word in $L_{p,q}$ can be constructed from the expression

$$xL_{p_1,q_1}y.$$

Notice that x and/or y could be ϵ, if one of the two transitions didn't read anything from the input.

The construction

We now explicitly state the construction. First, for every state p, we introduce the rule

$$S_{p,p} \rightarrow \epsilon.$$

The case that the stack is empty in the middle of transitioning from p to q is captured by introducing, for any states p, q, r of N, we define the following rule in our CFG:

$$S_{p,q} \rightarrow S_{p,r}S_{r,q}.$$

As for the other case, that the stack is never empty, we specify for any given states p, p_1, q_1, r of N, such that there is a push transition from p to p_1 and a pop transition from q_1 to r (that push and pop the same letter), we introduce an appropriate rule. Formally, for any p, p_1, q_1, r, if there are transitions in N of the form

$$\underbrace{p \xrightarrow{\text{x},\epsilon\rightarrow\text{z}} p_1}_{\text{push } z} \quad \text{and} \quad \underbrace{q_1 \xrightarrow{\text{y},\text{z}\rightarrow\epsilon} q}_{\text{pop } z}.$$

The introduce the rule

$$S_{p,q} \rightarrow xS_{p_1,q_1}y$$

into the CFG.

We create such rules for all possible choices of states of N. Let C be the resulting grammar. This completes the description of how we constructed the CFG equivalent to the given PDA N.

We claim that $S_{q_{\text{init}},q_{\text{acc}}}$ in the grammar C generates all the words that the PDA N accepts.

Remark 17.2.1 At the start of our construction, we got rid of all the transitions that don't touch the stack at all. Another option would have been to handle them with a variation of our second type of context-free rule. That is, we have a transition from a state p to p_1 that does not touch the stack (and reads the character x from the input). A small extension of the above construction would give us the transition:

$$S_{p,q} \rightarrow xS_{p_1,q}.$$

17.2.3 Proof of correctness

Here, prove that the language generated by $S_{q_{\mathrm{init}},q_{\mathrm{acc}}}$ is the language recognized by the PDA N.

Claim 17.2.2 *If the string w can be generated by $S_{p,q}$ then there is an execution of N stating at p (with an empty stack) and ending at q (with an empty stack).*

Proof: The proof is by induction on the number n of steps used in the derivation generating w from $S_{p,q}$

For the base of the induction, consider $n = 1$. The only rules in C that have no symbols in them are of the form

$$S_{p,p} \rightarrow \epsilon.$$

Which implies the claim trivially.

Thus, consider the case where $n > 1$, and assume that we proved that any word generated by at most n derivation steps (in the CFG grammar C) can be realized by an execution of the PDA N. We would like to prove the inductive step for $n + 1$. So, assume that w is generated from $S_{p,q}$ using $n + 1$ derivation steps. There are two possibilities for what is the first derivation rule used. The first possibility is that we used the rule

$$\underbrace{S_{p,q}}_{w} \underbrace{\rightarrow}_{=} \underbrace{S_{p,r}}_{w_1} \underbrace{S_{r,q}}_{w_2}.$$

As such, w_1 is generated from $S_{p,r}$ in at most $(n + 1) - 1 = n$ steps, and w_2 is generated from $S_{r,q}$ in at most $(n + 1) - 1 = n$ steps. As such, by induction, there is an execution of N starting at p and ending in r (with empty stack in the beginning and the end) and, similarly, there is an execution of N starting at r and ending q (with empty stack in the beginning and the end). By performing these two execution one of after the other, we end up with an execution starting at p and ending at q, with an empty stack on both ends, such that the PDA N reads the input w during this execution. Thus, this establishes the claim in this case.

The other possibility, is that w was derived by first applying a rule of the form

$$\underbrace{S_{p,q}}_{w} \underbrace{\rightarrow}_{=} \underbrace{x}_{x} \underbrace{S_{p_1,q_1}}_{w'} \underbrace{y}_{y}.$$

But then, by construction, the PDA N must have two transitions

$$p \xrightarrow{\texttt{x},\epsilon\rightarrow\texttt{z}} p_1 \quad \text{and} \quad q_1 \xrightarrow{\texttt{y},\texttt{z}\rightarrow\epsilon} q \tag{17.1}$$

that generated this rule. Furthermore, by induction, the word w' was generated from S_{p_1,q_1} using n derivation steps. As such, there exists a compliant execution X from p_1 to q_1 generating w'. Thus, if we start at p, apply the first transition of Eq. (17.1), then the execution X and then the second transition of Eq. (17.1), then we end up with a complaint execution of N that starts at p, ends at q (with empty stack on both ends), and reads the string w, which establishes the claim in this case, since we showed an execution that reads N. ■

Claim 17.2.3 *If there is an execution of N (with empty stack in both ends) starting at a state p and ending at a state q, that reads the string w, then w can be generated by $S_{p,q}$.*

Proof: The proof is somewhat similar to the previous proof. Consider the execution for w, and assume that it takes n steps. We will prove the claim by induction on n.

For $n = 0$, the execution is empty, and starts at p and ends at $q = p$. But then, w is ϵ, and it can be derived by $S_{p,p}$ since the CFG C has the rule $S_{p,p} \rightarrow \epsilon$.

Otherwise, for $n > 0$, assume by induction that we had proved the claim for all executions of length n, and we now consider an execution of length $n + 1$.

If the first transition in the execution is a push to the stack of a character z, and z is being popped by the last transition in the execution, then the first and last transitions are of the form

$$\underbrace{p \xrightarrow{\texttt{x},\epsilon\rightarrow\texttt{z}} p_1} \quad \text{and} \quad \underbrace{q_1 \xrightarrow{\texttt{y},\texttt{z}\rightarrow\epsilon} q},$$

respectively, and furthermore $w = xw'y$. As such, we have an execution of length $(n+1) - 1 \leq n$ that reads w', and by induction, w' can be generated by the symbol S_{p_1,q_1}. But then, by construction, the rule

$$S_{p,q} \rightarrow x S_{p_1,q_1} y$$

is in the CFG C, and as such w can be generated by $S_{p,q}$, as claimed.

The other possibility is that z is being popped out at some earlier stages, as the PDA N enters a state r (after reading a prefix of w, denoted by w_1). But then, arguing as above, we can break w into two strings w_1 and w_2, such that $w = w_1 w_2$, and by induction, w_1 can be generated by the rule $S_{p,r}$ and w_2 can be generated by the rule $S_{r,q}$. But then, the CFG C contains the rule

$$S_{p,q} \rightarrow S_{p,r} S_{r,q}.$$

Which implies that $S_{p,q}$ can generate the string w, as claimed. ■

As such, we conclude the following:

Lemma 17.2.4 *If the language L is accepted by a PDA N then L is a context-free language.*

Together with the results from earlier lectures, we can conclude the following.

Theorem 17.2.5 *A language L is context-free if and only if there is a PDA that recognizes it.*

Chapter 18

Lecture 15: CYK Parsing Algorithm

3 March 2009

18.1 CYK parsing

18.1.1 Discussion

We already saw that one can decide if a word is in the language defined by a context-free grammar, but the construction made no attempt to be practical. There were two problems with this algorithm. First, we converted the grammar to be in Chomsky Normal Form (CNF). Most practical applications need the parse to show the structure using the original input grammar. Second, our method blindly generated all parse trees of the right size, without regard to what was in the string to be parsed. This is inefficient since there may be a large number of parse trees (i.e., exponential in the length of the word) of this size.

The Cocke-Younger-Kasami (CYK) algorithm solves the second of these problems, using a table data-structure called the ***chart***. This basic technique can be extended (e.g. Earley's algorithm) to handle grammars that are not in Chomsky Normal Form and to linear-time parsing for special types of CFGs.

In general, the number of parses for a string w is exponential in the length of w. For example, consider the sentence "`Mr Plum kill Ms Marple at midnight in the bedroom with a sword`." There are three prepositional phrases: "`at midnight`", "`in the bedroom`," and "`with a sword`." Each of these can either be describing the main action (e.g. the killing was done with a sword) or the noun right before it (e.g. there are several bedrooms and we mean the one with a sword hanging over the dresser). Since each decision is independent, a sentence with k prepositional phrases like this has 2^k possible parses.[1]

So it's really bad to organize parsing by considering all possible parse trees. Instead, consider all substrings in our input. If w has length n, then it has $\Sigma_{k=1}^{n}(n-k+1) = \frac{n(n+1)}{2} = \binom{n+1}{2}$ substrings.[2] CYK computes a table summarizing the possible parses for each substring. From the table, we can quickly tell whether an input has a parse and extract one representative parse tree.[3]

18.1.2 CYK by example

Suppose the input sentence w is "`Jeff trains geometry students`" and the grammar has start symbol Sand the following rules:

$$\begin{array}{ll}
\Longrightarrow & \mathsf{S} \to \mathsf{N}\ \mathsf{V_P} \\
 & \mathsf{VN} \to \mathsf{N}\ \mathsf{N} \\
 & \mathsf{V_P} \to \mathsf{V}\ \mathsf{N} \\
 & \mathsf{N} \to \texttt{students} \mid \texttt{Jeff} \mid \texttt{geometry} \mid \texttt{trains} \\
 & \mathsf{V} \to \texttt{trains}
\end{array}$$

[1] In real life, long sentences in news reports often exhibit versions of this problem.

[2] Draw an n by $n+1$ rectangle and fill in the lower half.

[3] It still takes exponential time to extract all parse trees from the table, but we usually interested only in one of these trees.

Given a string w of length n, we build a triangular table with n rows and n columns. Conceptually, we write w below the bottom row of the table. The ith column correspond to the ith word. The cell at the ith column and the jth row (from the bottom) of the table corresponds to the substring starting the ith character of length j. The following is the table, and the substrings each entry corresponds to.

len				
4	`Jeff trains geometry students`			
3	`Jeff trains geometry`	`trains geometry students`		
2	`Jeff trains`	`trains geometry`	`geometry students`	
1	`Jeff`	`trains`	`geometry`	`students`
	`Jeff`	`trains`	`geometry`	`students`

first word in substring

CYK builds a table containing a cell for each substring. The cell for a substring x contains a list of variables V from which we can derive x (in one or more steps).

	4				
	3				
length	2				
	1				
		`Jeff`	`trains`	`geometry`	`students`

first word in substring

The bottom row contains the variables that can derive each substring of length 1. This is easy to fill in:

	4				
	3				
length	2				
	1	N	N,V	N	N
		Jeff	trains	geometry	students

first word in substring

Now we fill the table row-by-row, moving upwards. To fill in the cell for a 2-word substring x, we look at the labels in the cells for its two constituent words and see what rules could derive this pair of labels. In this case, we use the rules $\mathsf{N} \rightarrow \mathsf{N}\ \mathsf{N}$ and $\mathsf{V_P} \rightarrow \mathsf{V}\ \mathsf{N}$ to produce:

	4				
	3				
length	2	N	N,$\mathsf{V_P}$	N	
	1	N	N,V	N	N
		Jeff	trains	geometry	students

first word in substring

For each longer substring x, we have to consider all the ways to divide x into two shorter substrings. For example, suppose x is the substring of length 3 starting with "trains". This can be divided into divided into (a) "trains geometry" plus "students" or (b) "trains" plus "geometry students."

Consider option (a). Looking at the lower rows of the table, "students" has label N. One label for "trains geometry" is $\mathsf{V_P}$, but we don't have any rule whose righthand side contains $\mathsf{V_P}$ followed by N. The other label for "trains geometry" is N. In this case, we find the rule $\mathsf{N} \rightarrow \mathsf{N}\ \mathsf{N}$. So one label for x is N. (That is, x is one big long compound noun.)

Now consider option (b). Again, we have the possibility that both parts have label N. But we also find that "trains" could have the label V. We can then apply the rule $\mathsf{V_P} \rightarrow \mathsf{V}\ \mathsf{N}$ to add the label $\mathsf{V_P}$ to the cell for x.

```
CYK ( $\mathcal{G}, w$ )
        $\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{R}, \mathsf{S})$, $\Sigma \cup \mathcal{V} = \{\mathsf{X}_1, \ldots, \mathsf{X}_r\}$, $w = w_1 w_2 \ldots w_n$.
begin
    Initialize the 3d array $\mathsf{B}[1 \ldots n, 1 \ldots n, 1 \ldots r]$ to FALSE
    for $i = 1$ to $n$ do
        for $(\mathsf{X}_j \to x) \in \mathcal{R}$ do
            if $x = w_i$ then $\mathsf{B}[i, i, j] \leftarrow$ TRUE.
    for $i = 2$ to $n$ do      /* Length of span */
        for $L = 1$ to $n - i + 1$ do      /* Start of span */
            $R = L + i - 1$       /* Current span $s = w_L w_{L+1} \ldots w_R$ */
            for $M = L + 1$ to $R$ do      /* Partition of span */
                /* $x = w_L w_{L+1} \ldots w_{M-1}$, $y = w_M w_{M+1} \ldots w_R$, and $s = xy$ */
                for $(\mathsf{X}_\alpha \to \mathsf{X}_\beta \mathsf{X}_\gamma) \in \mathcal{R}$ do
                    /* Can we match $\mathsf{X}_\beta$ to $x$ and $\mathsf{X}_\gamma$ to $y$? */
                    if $\mathsf{B}[L, M-1, \beta]$ and $\mathsf{B}[M, R, \gamma]$ then
                        $\mathsf{B}[L, R, \alpha] \leftarrow$ TRUE      /* If so, then can generate $s$ by $\mathsf{X}_\alpha$! */
    for $i = 1$ to $r$ do
        if $\mathsf{B}[1, n, i]$ then return TRUE
    return FALSE
```

Figure 18.1: The CYK algorithm.

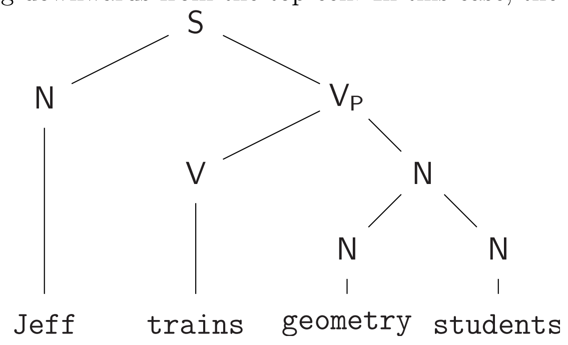
	4				
	3		N,$\mathsf{V_P}$		
length	2	N	N,$\mathsf{V_P}$	N	
	1	N	N,V	N	N
		Jeff	trains	geometry	students
		first word in substring			

Repeating this procedure for the remaining two cells, we get:

	4	N,S			
	3	N,S	N,$\mathsf{V_P}$		
length	2	N	N,$\mathsf{V_P}$	N	
	1	N	N,V	N	N
		Jeff	trains	geometry	students
		first word in substring			

Remember that a string is in the language if it can be derived from the start symbol S. The top cell in the table contains the variables from which we can derive the entire input string. Since S is in that top cell, we know that our string is in the language.

By adding some simple annotations to these tables as we fill them in, we can make it easy to read out an entire parse tree by tracing downwards from the top cell. In this case, the tree:



We have $O\big(n^2\big)$ cells in the table. For each cell, we have to consider n ways to divide its substring into two smaller substrings. So the table-filling procedure takes only $O\big(n^3\big)$ time.

18.2 The CYK algorithm

In general, we get the following result.

Theorem 18.2.1 *Let $\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{R}, \mathsf{S})$ be a grammar in CNF with $r = |\Sigma| + |\mathcal{V}|$ variables and terminals, and $t = |\mathcal{R}|$ rules. Let $w \in \Sigma^*$ be a word of length n. Then, one can compute a parse tree for w using $\mathcal{G}$, if $w \in \mathsf{L}(\mathcal{G})$. The running time of the algorithm is $O(n^3 t)$.*

The result just follow from the CYK algorithm depicted in Figure 18.1. Note, that our pseudo-code just decides if a word can be generated by a grammar. With slight modifications, one can even generate the parse tree.

Chapter 19

Lecture 16: Recursive automatas

17 March 2009

19.1 Recursive automata

A finite automaton can be seen as a program with only a finite amount of memory. A recursive automaton is like a program which can use *recursion* (calling procedures recursively), but again over a finite amount of memory in its variable space. Note that the recursion, which is typically handled by using a stack, gives a limited form of *infinite* memory to the machine, which it can use to accept certain non-regular languages. It turns out that the recursive definition of a language defined using a context-free grammar precisely corresponds to recursion in a finite-state recursive automaton.

19.1.1 Formal definition of RAs

A recursive automaton (RA) over Σ is made up of a finite set of NFAs that can call each other (like in a programming language), perhaps recursively, in order to check if a word belongs to a language.

Definition 19.1.1 A ***recursive automaton*** (RA) over Σ is a tuple

$$\left(M, \textbf{main}, \left\{ \mathtt{D}_M \;\middle|\; m \in M \right\} \right),$$

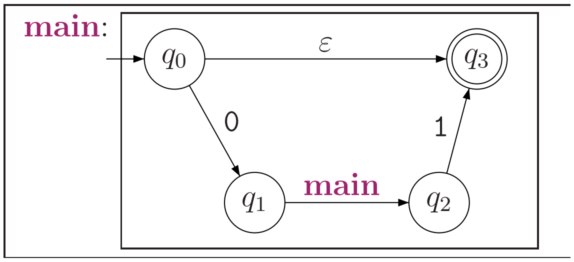
where

- M is a finite set of module names,
- $\textbf{main} \in M$ is the initial module,
- For each $m \in M$, there is an associated automaton $\mathtt{D}_m = (Q_m, \Sigma \cup M, \delta_m, q_0^m, F_m)$ which is an NFA over the alphabet $\Sigma \cup M$. In other words, Q_m is a finite set of states, $q_0^m \in Q_m$ is the initial state (of the module m), $F_m \subseteq Q_m$ is the set of final states of the module m (from where the module can return), and $\delta_m : Q_m \times (\Sigma \cup M \cup \{\epsilon\}) \to 2^{Q_m}$ is the (non-deterministic) transition function.
- For any $m, m' \in M$, $m \neq m'$ we have $Q_m \cap Q_{m'} = \emptyset$ (the set of states of different modules are disjoint).

Intuitively, we view a recursive automaton as a set of procedures/modules, where the execution starts with the **main**-module, and the automaton processes the word by calling modules recursively.

Example of a recursive automata

Let $\Sigma = \{\mathtt{0}, \mathtt{1}\}$ and let $L = \left\{ 0^n 1^n \;\middle|\; n \in \mathbb{N} \right\}$. The language L is accepted by the following recursive automaton.



Why? The recursive automaton consists of single module, which is also the **main** module. The module either accepts ϵ, or reads `0`, calls itself, and after returning from the call, reads `1` and reaches a final state (at which point it can return if it was called). In order to accept, we require the run to return from all calls and reach the final state of the module **main**.

For example, the recursive automaton accepts `01` because of the following execution

$$q_0 \xrightarrow{\texttt{0}} q_1 \xrightarrow{\text{call } \textbf{main}} q_0 \xrightarrow{\epsilon} q_3 \xrightarrow{\text{return}} q_2 \xrightarrow{\texttt{1}} q_3.$$

Note that using a transition of the form $q \xrightarrow{\texttt{m}} q'$ calls the module m; the module m starts with its initial state, will process letters (perhaps recursively calling more modules), and when it reaches a final state will return to the state q' in the calling module.

Formal definition of acceptance

Stack. We first need the concept of a ***stack***. A stack s is a list of elements. The ***top of the stack*** (TOS) is the first element in the list, denoted by topOfStack(() s). Pushing an element x into a stack (i.e., ***push*** operation) s, is equivalent to creating a new list, with the first element being x, and the rest of the list being s. We denote the resulting stack by $\text{push}(s, x)$. Similarly, popping the stack s (i.e., ***pop*** operation) is the list created from removing the first element of s. We will denote the resulting stack by $\text{pop}(s)$.

We denote the empty stack by $\langle\rangle$. A stack containing the elements x, y, z (in this order) is written as $s = \langle x, y, z\rangle$. Here $\text{topOfStack}(s) = x$, $\text{pop}(s) = \langle y, z\rangle$ and $\text{push}(s, b) = \langle b, x, y, z\rangle$.

Acceptance. Formally, let $\mathtt{C} = \left(M, \textbf{main}, \left\{(Q_m, \Sigma \cup M, \delta_m, q_0^m, F_m) \;\middle|\; m \in M\right\}\right)$ be a recursive automaton.

We define a run of $\mathtt{C}$ on a word w. Since the modules can call each other recursively, we define the run using a stack. When $\mathtt{C}$ is in state q and calls a module m using the transition $q \overset{*}{\Longrightarrow} mq'$, we push q' onto the stack so that we know where to go to when we return from the call. When we return from a call, we pop the stack and go to the state stored on the top of the stack.

Formally, let $Q = \bigcup_{m\in M} Q_m$ be the set of all states in the automaton $\mathtt{C}$. A ***configuration*** of $\mathtt{C}$ is a pair (q, s) where $q \in Q$ and s is a stack.

We say that a word w is ***accepted*** by $\mathtt{C}$ provided we can write $w = y_1 \dots y_k$, such that each $y_i \in \Sigma \cup \{\epsilon\}$, and there is a sequence of $k+1$ configurations $(q_0, s_0), \dots (q_k, s_k)$, such that

- $q_0 = q_0^{\textbf{main}}$ and $s_0 = \langle\rangle$.

  We start with the initial state of the main module with the stack being empty.

- $q_k \in F_{\textbf{main}}$ and $s_k = \langle\rangle$.

  We end with a final state of the main module with the stack being empty (i.e. we expect all calls to have returned).

- For every $i < k$, one of the following must hold:

  Internal: $q_i \in Q_m$, $q_{i+1} \in \delta_m(q_i, y_{i+1})$, and $s_{i+1} = s_i$.

  Call: $q_i \in Q_m$, $y_{i+1} = \epsilon$, $q' \in \delta_m(q_i, m')$, $q_{i+1} = q_0^{m'}$ and $s_{i+1} = \text{push}(s_i, q')$.

  Return: $q_i \in F_m$, $y_{i+1} = \epsilon$, $q_{i+1} = \text{topOfStack}(s_i)$ and $s_{i+1} = \text{pop}(s_i)$.

19.2 CFGs and recursive automata

We will now show that context-free grammars and recursive automata accept precisely the same class of languages.

19.2.1 Converting a CFG into a recursive automata

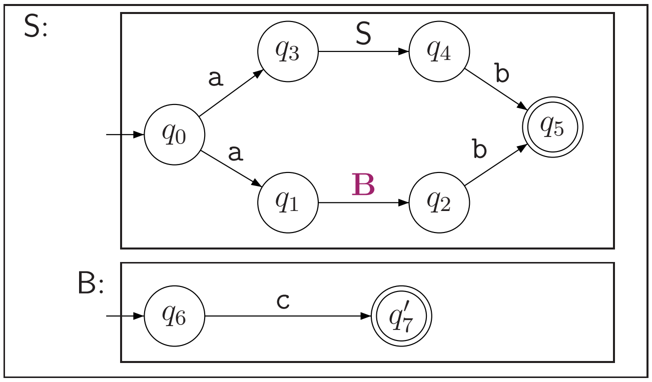
Given a CFG, we want to construct a recursive automaton for the language generated by the CFG. Let us first do this for an example.

Consider the grammar (where S is the start variable) which generates $\left\{ \mathtt{a}^n \mathtt{c} \mathtt{b}^n \,\middle|\, n \in \mathbb{N} \right\}$:

$$\Longrightarrow \begin{array}{rcl} \mathsf{S} & \rightarrow & \mathtt{aSb} \mid \mathtt{aBb} \\ \mathsf{B} & \rightarrow & \mathtt{c}. \end{array}$$

Each variable in the CFG corresponds to a language; this language is recursively defined using other variables. We hence look upon each variable as a module; and define modules that accept words by calling other modules recursively.

For example, the recursive automaton for the above grammar is:



(Here S is the main modules of the recursive automaton.)

Formal construction. Let $\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{R}, \mathsf{S})$ be the given context free grammar.

Let $\mathtt{D}_{\mathcal{G}} = \left(M, \mathsf{S}, \left\{ (Q_m, \Sigma \cup M, \delta_m, q_0^m, F_m) \,\middle|\, m \in M \right\} \right)$ where $M = \mathcal{V}$, and the main module is S. Furthermore, for each $\mathsf{X} \in M$, let $\mathtt{D}_{\mathsf{X}} = \left(Q_{\mathsf{X}}, \Sigma \cup M, \delta_{\mathsf{X}}, q_0^{\mathsf{X}}, F_{\mathsf{X}})\right)$ be an NFA that accepts the (finite, and hence) regular language $L_{\mathsf{X}} = \left\{ w \,\middle|\, (\mathsf{X} \rightarrow w) \in \mathcal{R} \right\}$.

Let us elaborate on the construction of $\mathtt{D}_{\mathsf{X}}$. We create two special states $q_{\text{init}}^{\mathsf{X}}$ and $q_{\text{final}}^{\mathsf{X}}$. Here $q_{\text{init}}^{\mathsf{X}}$ is the initial state of $\mathtt{D}_{\mathsf{X}}$ and $q_{\text{final}}^{\mathsf{X}}$ is the accepting state of $\mathtt{D}_{\mathsf{X}}$. Now, consider a rule $(\mathsf{X} \rightarrow w) \in \mathcal{R}$. We will introduce a path of length $|w|$ in $\mathtt{D}_{\mathsf{X}}$ (corresponding to w) leading from $q_{\text{init}}^{\mathsf{X}}$ to $q_{\text{final}}^{\mathsf{X}}$. Creating this path requires introducing new "dummy" states in the middle of the path, if $|w| > 1$. The ith transition along this path reads the ith character of w. Naturally, if this ith character is a variable, then this edge would correspond to a recursive call to the corresponding module. As such, if the variable X has k rules in the grammar $\mathcal{G}$, then $\mathtt{D}_{\mathsf{X}}$ would contain k disjoint paths from $q_{\text{init}}^{\mathsf{X}}$ to $q_{\text{final}}^{\mathsf{X}}$, corresponding to each such rule. For example, if we have the derivation $(\mathsf{X} \rightarrow \epsilon) \in \mathcal{R}$, then we have an ϵ-transition from $q_{\text{init}}^{\mathsf{X}}$ to $q_{\text{final}}^{\mathsf{X}}$.

19.2.2 Converting a recursive automata into a CFG

Let $\mathtt{C} = (M, \textbf{main}, \{(Q_m, \Sigma \cup M, \delta_m, q_{\text{init}}^m, F_m)\}_{m \in M})$ be a recursive automaton. We construct a CFG $\mathcal{G}_{\mathtt{C}} = (\mathcal{V}, \Sigma, \mathcal{R}, \mathsf{S})$ with $\mathcal{V} = \left\{ \mathsf{X}_q \mid q \in \bigcup_{m \in M} Q_m \right\}$.

Intuitively, the variable X_q will represent the set of all words accepted by starting in state q and ending in a final state of the module q is in (however, on recursive calls to this module, we still enter at the original initial state of the module).

The set of rules R is generated as follows.

- **Regular transitions.** For any $m \in M$, $q, q' \in Q_m$, $c \in \Sigma \cup \{\epsilon\}$, if $q' \in \delta_m(q, c)$, then the rule $\mathsf{X}_q \rightarrow c\mathsf{X}_{q'}$ is added to $\mathcal{R}$.

  Intuitively, a transition within a module is simulated by generating the letter on the transition and generating a variable that stands for the language generated from the next state.

- **Recursive call transitions.** for all $m, m' \in M$ and $q, q' \in Q_m$, if $q' \in \delta_m(q, m')$, then the rule $\mathsf{X}_q \rightarrow \mathsf{X}_{q^{m'}_{\mathrm{init}}} \mathsf{X}_{q'}$ is in R,

  Intuitively, if $q' \in \delta_m(q, m')$, then X_q can generate a word of the form xy where x is accepted using a call to module m and y is accepted from the state q'.

- **Acceptance/return rules.**

  For any $q \in \bigcup_{m \in M} F_m$, we add $\mathsf{X}_q \rightarrow \epsilon$ to $\mathcal{R}$.

  When arriving at a final state, we can stop generating letters and return from the recursive call.

The initial variable S is $\mathsf{X}_{q^{\mathbf{main}}_{\mathrm{init}}}$; that is, the variable corresponding to the initial state of the main module.

We have a CFG and it is not too hard to see intuitively that the language generated by this grammar is equal to the RA C language. We will not prove it formally here, but we state the result for the sake of completeness.

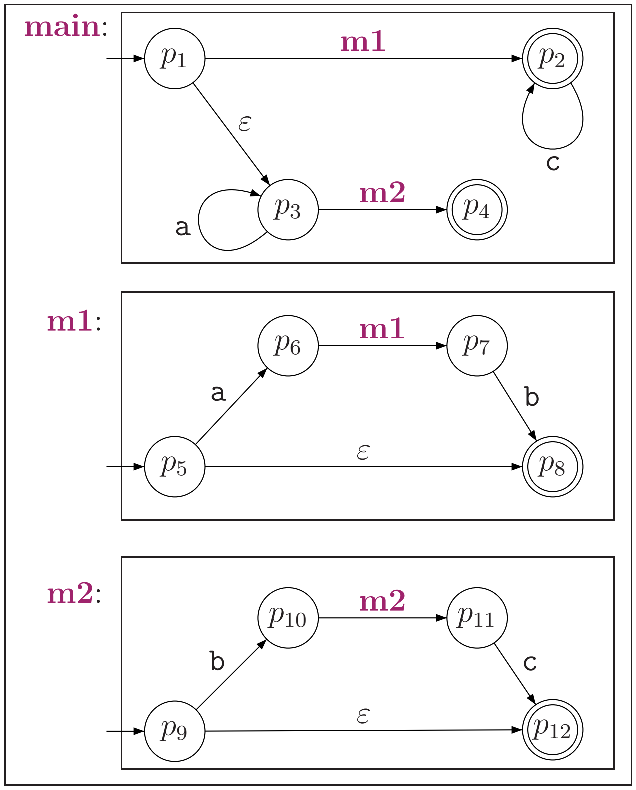
Lemma 19.2.1 $\mathsf{L}(\mathcal{G}_{\mathsf{C}}) = \mathsf{L}(\mathsf{C})$.

An example of conversion of a RA into a CFG

Consider the following recursive automaton, which accepts the language

$$\left\{ \mathtt{a}^i \mathtt{b}^j \mathtt{c}^k \;\middle|\; i = j \textit{ or } j = k \right\},$$

and the grammar generating it.



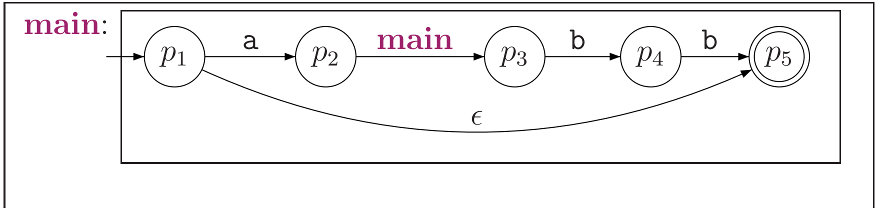
$$
\begin{array}{lcl}
\mathsf{X}_{p1} & \rightarrow & \mathsf{X}_{p5}\mathsf{X}_{p2} \mid \mathsf{X}_{p3} \\
\mathsf{X}_{p2} & \rightarrow & \mathtt{c}\mathsf{X}_{p2} \mid \epsilon \\
\mathsf{X}_{p3} & \rightarrow & \mathtt{a}X_{p3} \mid \mathsf{X}_{p9}X_{p4} \\
\mathsf{X}_{p4} & \rightarrow & \epsilon \\
\mathsf{X}_{p5} & \rightarrow & \mathtt{a}\mathsf{X}_{p6} \mid \mathsf{X}_{p8} \\
\mathsf{X}_{p6} & \rightarrow & \mathsf{X}_{p5}\mathsf{X}_{p7} \\
\mathsf{X}_{p7} & \rightarrow & \mathtt{b}X_{p8} \\
\mathsf{X}_{p8} & \rightarrow & \epsilon \\
\mathsf{X}_{p9} & \rightarrow & \mathtt{b}\mathsf{X}_{p10} \mid \mathsf{X}_{p12} \\
\mathsf{X}_{p10} & \rightarrow & \mathsf{X}_{p9}\mathsf{X}_{p11} \\
\mathsf{X}_{p11} & \rightarrow & \mathtt{c}X_{p12} \\
\mathsf{X}_{p12} & \rightarrow & \epsilon
\end{array}
$$

The start variable is X_{p1}.

19.3 More examples

19.3.1 Example 1: RA for the language $\mathtt{a}^n\mathtt{b}^{2n}$

Let us design a recursive automaton for the language $L = \left\{ \mathtt{a}^n\mathtt{b}^{2n} \,\middle|\, n \in \mathbb{N} \right\}$. We would like to generate this recursively. How do we generate $\mathtt{a}^{n+1}\mathtt{b}^{2n+2}$ using a procedure to generate $\mathtt{a}^n\mathtt{b}^{2n}$? We read $\mathtt{a}$ followed by a call to generate $\mathtt{a}^n\mathtt{b}^{2n}$, and follow that by generating two $\mathtt{b}$'s. The "base-case" of this recursion is when $n = 0$, when we must accept ϵ. This leads us to the following automaton:

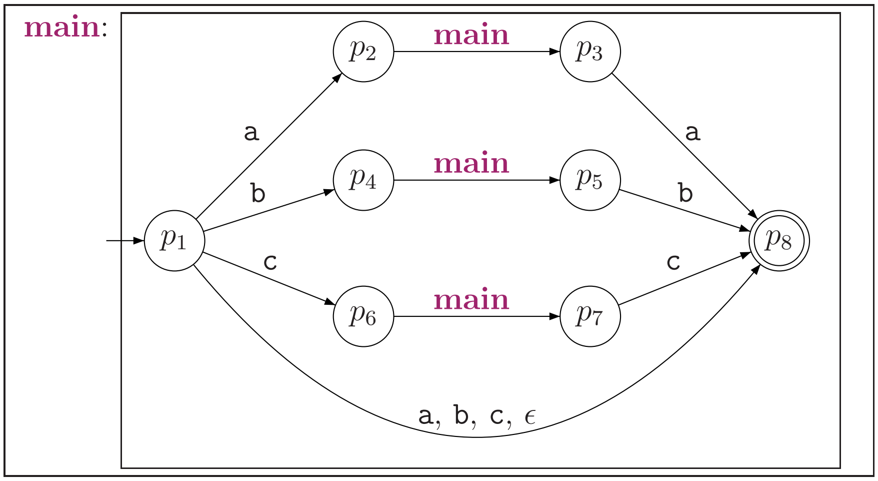


19.3.2 Example 2: Palindrome

Let us design a recursive automaton for the language

$$L = \left\{ w \in \{\mathtt{a},\mathtt{b},\mathtt{c}\}^* \,\middle|\, w \text{ is a palindrome} \right\}.$$

Thinking recursively, the smallest palindromes are ϵ, a, b, c, and we can construct a longer palindrome by generating awa, bwb, cwc, where w is a smaller palindrome. This give us the following recursive automaton:



19.3.3 Example 3: $\#_a = \#_b$

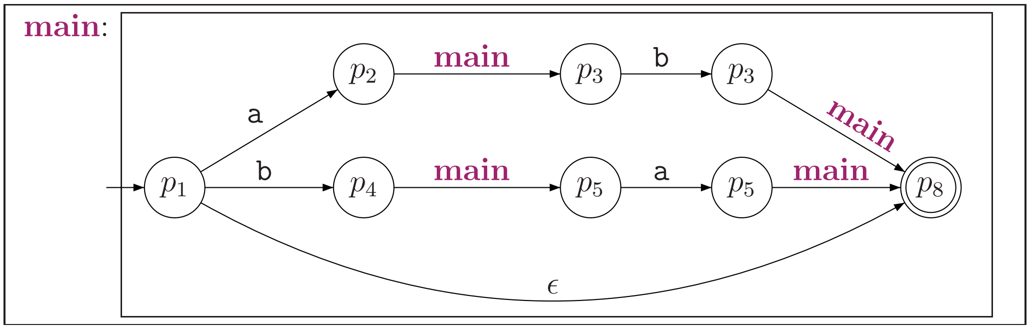
Let us design a recursive automaton for the language L containing all strings $w \in \{\mathtt{a},\mathtt{b}\}^*$ that has an equal number of $\mathtt{a}$'s and $\mathtt{b}$'s.

Let w be a string, of length at least one, with equal number of $\mathtt{a}$'s and $\mathtt{b}$'s.

Case 1: w starts with $\mathtt{a}$. As we read longer and longer prefixes of w, we have the number of $\mathtt{a}$'s seen is more than the number of $\mathtt{b}$'s seen. This situation can continue, but we must reach a place when the number of $\mathtt{a}$'s seen is precisely the number of $\mathtt{b}$'s seen (at worst at the end of the word). Let us consider some prefix longer than $\mathtt{a}$ where this happens. Then we have that $w = \mathtt{a}w_1\mathtt{b}w_2$, where the number of $\mathtt{a}$'s and $\mathtt{b}$'s in $\mathtt{a}w_1\mathtt{b}$ is the same, i.e. the number of $\mathtt{a}$'s and $\mathtt{b}$'s in w_1 are the same. Hence the number of $\mathtt{a}$'s and $\mathtt{b}$'s in w_2 are also the same.

Case 2: If w starts with $\mathtt{b}$, then by a similar argument as above, $w = \mathtt{b}w_1\mathtt{a}w_2$ for some (smaller) words w_1 and w_2 in L.

Hence any word w in L of length at least one is of the form $\mathtt{a}w_1\mathtt{b}w_2$ or $\mathtt{b}w_1\mathtt{a}w_2$, where $w_1, w_2 \in L$, and they are strictly shorter than w. Also, note ϵ is in L. So this gives us the following recursive automaton.



19.4 Recursive automata and pushdown automata

The definition of acceptance of a word by a recursive automaton employs a ***stack***, where the target state gets pushed on a call-transition, and gets popped when the called module returns. An alternate way (and classical) way of defining automata models for context-free languages directly uses a stack. A ***pushdown automaton*** (PDA) is a non-deterministic automaton with a finite set of control states, and where transitions are allowed to push and pop letters from a finite alphabet Γ (Γ is fixed, of course) onto the stack. It should be clear that a recursive automaton can be easily simulated by a pushdown automaton (we simply take the union of all states of the recursive automaton, and replace call transitions $q \xrightarrow{\mathtt{m}} q'$ with an explicit push-transition that pushes q' onto the stack and explicit pop transitions from the final states in F_m to q' on popping q'.

It turns out that pushdown automata can be converted to recursive automata (and hence to CFGs) as well. This is a fact worth knowing! But we will not define pushdown automata formally, nor show this direction of the proof.

Chapter 20

Instructor notes: Recursive automatas vs. Pushdown automatas

20.1 Instructor Notes

Stacks vs recursion: The class of context-free languages (CFLs) is interesting only because of context-free grammars. Pushdown automata are not interesting as a first-class computable machine. We think moving from NFA/DFAs to a model with a stack is highly artificial (why a stack? why not a queue? why not two stacks?). If we didn't have context-free grammars or parsing in mind, we would skip CFLs entirely, and cover only finite automata and Turing machines. Hence, it's a valid question to ask whether PDAs are the right machine model for CFLs. We claim not!

The primary objection to PDAs is that it makes you think of a finite-state program with a *stack as a data structure*. When we think of designing a PDA we think of how we can use the stack wisely to achieve our goal. A context-free grammar is more tied to recursion than to using a stack data-structure. The variables in a CFG are recursively defined using other variables. When we design a CFG for a language, we think recursively.

The idea behind *recursive automata* (RA) introduced here is that they bring about recursive thinking as opposed to thinking using a stack. A recursive automaton is simply a program with finite memory and recursive procedure calls.

Having an automaton model that corresponds to recursive programs with finite memory is very intuitive, and helpful for students. They can think of solving things recursively; this is always useful, and may have benefits for other courses (like 473) too. We doubt dealing with a stack explicitly brings anything conceptually interesting.

Example 3 above (L containing all words with an equal number of a's and b's) is a good example. The PDA one may come up for it is perhaps quite different from the recursive automaton. The recursive automaton is more the way you'd come up with a grammar:

$$S \rightarrow aSbS \mid bSaS \mid \epsilon.$$

The PDA we come up with for this language explicitly thinks of using the stack to do counting: the excess number of letters of one kind as opposed to the other is stored in the stack. In other words, we are using the stack as a "counter". This example shows clearly the difference between PDA-thinking and RA-thinking... we believe that thinking recursively of the language is a lot more useful and typical in computer science. Anyway, if we use recursive automata, we probably are not teaching the idea behind using the stack to count— but the point is "who cares?"

In summary, going away from using an explicit pushdown stack, and relying instead on a recursion idiom is natural, beautiful, and pleasing!

Finally, I think this model keeps things easy and natural to remember: regular languages are accepted by programs with finite memory; CFLs are accepted by recursive programs with finite memory; and decidable languages are accepted by Turing machines with infinite memory.

CFGs to RA and back: The conversions from CFGs to RAs is very simple (and visual!). In contrast, converting CFGs to PDAs is mildly harder and way less visual (we usually define an extension of PDAs that can push words in reverse to a stack, and have the guess a left-most derivation).

The conversion from RAs to CFGs is a lot easier than from PDAs to CFGs. To convert PDAs to CFGs, one must convert the PDA to a PDA that accepts with empty stack, and then argue that any run of the PDA can be "summarized" using what happens between a push and a pop of the same symbol, and build a CFG that captures this idea. When going from a recursive automaton to a CFG, it is more natural to divide a run into the part that is read by a module and the suffix read after returning from the module.

Closure properties: Turning to closure properties: given two programs, clearly one can create the automaton for the union by building a new initial module that calls either of them nondeterministically. Closure under concatenation and Kleene-* are also easy. We can quickly sketch these closure properties in about 10 minutes (we will do them for CFGs anyway). The intuition behind writing programs will make this very accessible to students.

Closure under intersection does not work as we cannot take the product of the two recursive automata—for if one wants to call a module and the other does not, then there is no way to simulate both of them.

An aside: in *visibly pushdown languages*, the recursive automata synchronize on calls and returns, and hence the class is closed under intersection. In fact, it's closed under complement as well, and determinizable.

Turning to closure on intersection with a regular language, this is easy. We just do the product construction... and at a call, guessing the return values of the finite automaton. We will do this using CFGs as well.

Non-context-freeness: When we have later shown $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ is not context-free (using pumping lemmas for CFLs), we think it is nice to go back and say that this means there is no finite memory recursive program to generate $a^n b^n c^n$, which I think is interesting and perhaps more appreciable.

CFLs are decidable: The negative aspect that recursive automata are not clearly simulated by (deterministic) Turing machines exists, as it used to exist with pushdown automata. The way out is still by showing membership in CFGs is decidable using a conversion to CNF or by showing the CYK algorithm. This has little to do with either automaton model.

Applications of CFLs: There are three main applications of CFLs we see: the first is parsing languages (like programming languages), the second is parsing natural languages, and the third is static program analysis.

Parsing programming languages and the like is done using an abstraction of pushdown automata, with a million variants and lookahead distances and what not. The recursive automaton definition does yield a pushdown automaton definition which we will talk about, and hence the students will be able to understand these algorithms. However, they will be a tad unfamiliar with them.

In parsing natural languages, Julia provided us with papers where they build automata for grammars almost exactly the way we do. In particular, there are many models with NFAs calling each other. And they argue this is a more natural representation of a machine for a grammar (and often use it to characterize languages that admit a faster membership problem).

Turning to program analysis, statically analyzing a program (for data-flows, control-flows, etc.) often involves abstracting the variables into a finite set of data-facts and looking at the recursive program accurately. The natural model one obtains (for flow-sensitive context-sensitive analysis) is a recursive automaton! In fact, the recursive automata we have here are motivated by the class of "recursive state machines" studied in the verification literature and used to model programs with finite memory.

Foreseeable objections:

- PDAs are part of tradition
  We agree PDAs are something we expect students to know when they graduate. But if we think it's the wrong model, then it's time to change the tradition. Students will know of the pushdown automaton

model anyway. They wouldn't have seen a formal conversion from PDAs to CFGs. Which, arguably, they wouldn't remember even if they have seen them.

Instead, now, they would know that CFGs are simply languages accepted by recursive programs with finite memory. Which is a natural characterization worthy of knowledge.

- PDAs being far away from CFGs is a good thing
  We agree; showing a model further away from CFGs is same as CFGs is a more interesting result. However, the argument to teaching PDAs holds only if it is a natural or useful model in its own right. There's very little evidence to claim PDAs are better than recursive automata.

- Reasoning with an explicit stack data structure is mental weightlifting.

  We don't think the course is well-served by introducing weight-lifting exercises. The students, frankly, lift too much weight already, given their level of maturity. Let's teach them beautiful and natural things; not geeky, useless, hard things.

- Don't the upper level classes needs PDAs?

  We have asked most people (Elsa, Sam, Margaret, etc.). The unanimous opinion we hear is that while seeing *some* machine model for CFLs is good, pushdown automata are not crucial.

- We must carefully consider making changes to such a fundamental topic:

  We think we have paid careful attention, and by seeking comments, we would have covered most opinions. The problem is that this is not a small change. It's a big change that can happen only if we do it now... if we postpone it, it will never happen!

- How come no one else thinks this way?

  PDAs and much of what we teach comes from the Hopcroft-Ullman textbook. People haven't seriously thought of changing the contents. Moreover, there are very few automata researchers in the US (Sipser isn't an automata-theory person).

- Why do you care so much?
  Good question! We don't really know. Maybe because CHANGE is in the air. Maybe because recursion seems better than stacks.

Chapter 21

Lecture 17: Computability and Turing Machines

13 March 2008

> The Electric Monk was a labor-saving device, like a dishwasher or a video recorder. Dishwashers washed tedious dishes for you, thus saving you the bother of washing them yourself, video recorders watched tedious television for you, thus saving you the bother of looking at it yourself; Electric Monks believed things for you, thus saving you what was becoming an increasingly onerous task, that of believing all the things the world expected you to believe.
>
> – Dirk Gently's Holistic Detective Agency, Douglas Adams.

This lecture covers the beginning of section 3.1 from Sipser.

21.1 Computability

For the alphabet

$$\Sigma = \{\mathtt{0}, \mathtt{1}, \ldots, \mathtt{9}, +, -\},$$

consider the language

$$L = \left\{ a_n a_{n-1} \ldots a_0 + b_m b_{m-1} \ldots b_0 = c_r c_{r-1} \ldots c_0 \;\middle|\; \begin{array}{l} a_i, b_j, c_k \in [0, 9] \text{ and} \\ \langle a_n a_{n-1} \ldots a_0 \rangle \\ \quad + \quad \langle b_m b_{m-1} \ldots b_0 \rangle \\ \qquad\qquad = \langle c_r c_{r-1} \ldots c_0 \rangle \end{array} \right\},$$

where $\langle a_n a_{n-1} \ldots a_0 \rangle = \sum_{i=0}^{n} a_i \cdot 10^i$ is the number represented in base ten by the string $a_n a_{n-1} \ldots a_0$. We are interested in the question of whether or not a given string belongs to this language. This is an example of a decision problem (where the output is either `yes` or `no`), which is easy in this specific case, but clearly too hard for a PDA to solve it[1].

Usually, we are interested in algorithms that compute something for their input and output the results. For example, given the strings $a_n a_{n-1} \ldots a_0$ and $b_m b_{m-1} \ldots b_0$ (i.e., two numbers) we want to compute the string representing their sum.

Here is another example for such a decision algorithm: Given a quadratic equation $ax^2 + bx + c = 0$, we would like to find the ***roots*** of this equation. Namely, two numbers r_1, r_2 such that $ax^2 + bx + c = a(x - r_1)(x - r_2) = 0$. Thus, given numbers a, b and c, the algorithm should output the numbers r_1 and r_2.

To see how subtle this innocent question can be, consider the question of computing the roots of a polynomial of degree 5. That is, given an equation

$$ax^5 + bx^4 + cx^3 + dx^2 + ex + f = 0,$$

[1] We use the world clearly here to indicate that the fact that this language is not context-free can be formally proven, but it is tedious and not the point of the discussion. The interested reader can try and prove this using the pumping lemma for CFGs.

can we compute the values of x for which is equation holds? Interestingly, if we limit our algorithm to use only the standard operators on numbers $+, -, *, /, \sqrt{\ }, \sqrt[k]{\ }$ then no such algorithm exists.[2]

In the final part of this course, we will look at the question of what (formally) is a computation? Or, in other words, what is (what we consider to be) a computer or an algorithm? A precise model for computation will allow us to prove that computers can solve certain problems but not others.

21.1.1 History

Early in this century, mathematicians (e.g. David Hilbert) thought that it might be possible to build formal algorithms that could decide whether any mathematical statement was true or false. For obvious reasons, there was great interest in whether this could really be done. In particular, he took upon himself the project of trying to formalize the known mathematics at the time. Gödel showed in 1929 that the project (of explicitly describing all of mathematics) is hopeless and there is no finite description of mathematical models.

In 1936, Alonzo Church and Alan Turing independently showed that this goal was impossible. In his paper, Alan Turing introduced the Turing machine (described below). Alonzo Church introduced the ***λ-calculus***, which formed the starting point for the development of a number of functional programming languages and also formal models of meaning in natural languages. Since then, these two models and some others (e.g. ***recursion*** theory) have been shown to be equivalent.

This has led to the Church-Turing Hypothesis.

> ***Church-Turing Hypothesis***: All reasonable models of (general-purpose) computers are equivalent. In particular, they are equivalent to a Turing machine.

This is not something you could actually prove is true (what is reasonable in the above statement, for example?). It could be proved false if someone found another model of computation that could solve more problems than a Turing machine, but no one has done this yet. Notice that we are ignoring how fast the computation can be done: it is certainly possible to improve on the speed of a Turing machine (in fact, every Turing machine can be speeded up by making it more complicated). We are only interested in what problems the machines can or can not solve.

21.2 Turing machines

21.2.1 Turing machines at a high level

So far, we have seen two simple models of computation:

- DFA/NFA: finite control, no extra memory, and
- Recursive automatas/PDA: finite control, unbounded stack.

Both types of machines read their input left-to-right. They halt exactly when the input is exhausted. Turing machines are like a RA/PDA, in that they have a finite control and an unbounded one dimensional memory tape (i.e., stack). However, a Turing machine is different in the following ways.

(A) The input is delivered on the memory tape (not in a separate stream).

(B) The machine head can move freely back and forth, reading and writing on the tape in any pattern.

(C) The machine halts immediately when it enters an ***accept*** or ***reject*** state.

Notice condition (C) in particular. A Turing machine can read through its input several times, or it might halt without reading the whole input (e.g. the language of all strings that start with `ab` can be recognized by just reading two letters).

[2] This is the main result of Evariste Galois that died at the age of 20(!) in a duel. Niels Henrik Abel (which also died relatively young) proved this slightly before Galois, but Galois work lead to a more general theory.

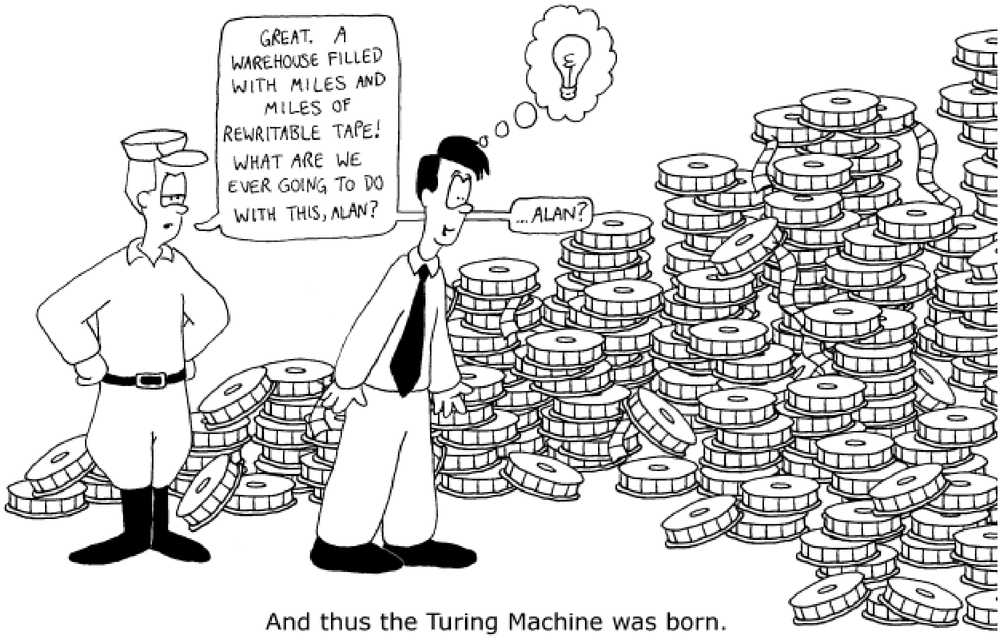
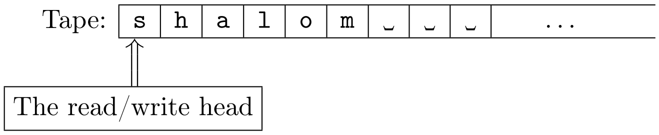

GREAT. A WAREHOUSE FILLED WITH MILES AND MILES OF REWRITABLE TAPE! WHAT ARE WE EVER GOING TO DO WITH THIS, ALAN?
...ALAN?
And thus the Turing Machine was born.

Figure 21.1: Comic by Geoff Draper.

Moving back and forth along the tape allows a Turing machine to (somewhat slowly) simulate random access to memory. Surprisingly, this very simple machine can simulate all the features of "regular" computers. Here equivalent is meant only in the sense that whatever a regular computer can compute, so can a Turing machine compute. Of course, Turing machines do not have graphics/sound cards, internet connection and they are generally considered to be an inferior platform for computer games. Nevertheless, computationally, TMs can compute whatever a "regular" computer can compute.

21.2.2 Turing Machine in detail

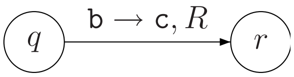
Specifically, a ***Turing machine*** (TM) has a finite control and an infinite tape. In this class, our basic model will have a tape that is infinite only in one direction. A Turing machine starts up with the input string written at the start of the tape. The rest of the tape is filled with a special blank character (i.e., '␣'). Initially, the head is located at the first tape position. Thus, the initial configuration of a Turing machine for the input `shalom` is as follows.



Each step of the Turing machine first reads the symbol on the cell of the tape under the head. Depending on the symbol and the current state of the controller, it then

- (optionally) writes a new symbol at the current tape position,
- moves either left or right, and
- (optionally) changes to a new state.

For example, the following transition is taken if the controller is in state q and the symbol under the read head is `b`. It replaces the `b` with the character `c` and then moves right, switching the controller to the state r.



Note, that Turing machines are deterministic. That is, once you know the state of the controller and which symbol is under the read/write head, there is exactly one choice for what the machine can (and must) do.

The controller has two special states q_{acc} and q_{rej}. When the machine enters one of these states, it halts. It either ***accepts*** or ***rejects***, depending on which of the two it entered.

Note 21.2.1 If the Turing machine is at the start of the tape and tries to move left, it simply stays put on the start position. This is not the only reasonable way to handle this case.

Note 21.2.2 Nothing guarantees that a Turing machine will eventually halt (i.e., stop). Like your favorite Java program, it can get stuck in an infinite loop[3]. This will have important consequences later, when we show that deciding if a program halts or not is in fact a task that computers can not solve.

Remark 21.2.3 Some authors define Turing machines to have a doubly-infinite tape. This does not change what the Turing machine can compute. There are many small variations on Turing machines which do not change the power of the machine. Later, we will see a few sample variations and how to prove they are equivalent to our basic model. The robustness of this model to minor changes in features is yet another reason computer scientists believe the Church-Turing hypothesis.

21.2.3 Turing machine examples

The language $w\$w$

For $\Sigma = \{\mathtt{a}, \mathtt{b}, \$\}$, consider the language

$$L = \left\{ w\$w \;\middle|\; w \in \Sigma^* \right\},$$

which is not context-free. So, let describe a TM that accepts this language.

One algorithm for recognizing L works as follows. It first

1. Cross off the first character a or b in the input (i.e. replace it with x, where x is some special character)) and remember what it was (by encoding the character in the current state). Let u denote this character.
2. Move right until we see a \$.
3. Read across any x's.
4. Read the character (not x) on the tape. If this character is different from u, then it immediately rejects.
5. Cross off this character, and replace it by x.
6. Move left past the \$ and then keep going until we see an x on the tape.
7. Move one position right and go back to the first step.

We repeat this until the first step can not find any more a's and b's to cross off.

Figure 21.2 depicts the resulting TM. Observe, that for the sake of simplicity of exposition, we did not include the state q_{rej} in the diagram. In particular, all missing transitions in the diagram are transitions that go into the reject state.

Notice that we did not include the reject state in the diagram, because it is already too messy. If there is no transition shown, we will assume that one goes into the reject state.

Note 21.2.4 For most algorithms, the Turing machine code is complicated and tedious to write out explicitly. In particular, it is not reasonable to write it out as a state diagram or a transition function. This only works for the relatively simple examples, like the ones shown here. In particular, its important to be able to describe a TM in high level in pseudo-code, but yet be able to translate it into the nitty-gritty details if necessary.

[3] Or just get stuck inside of Mobile with the Memphis blues again...

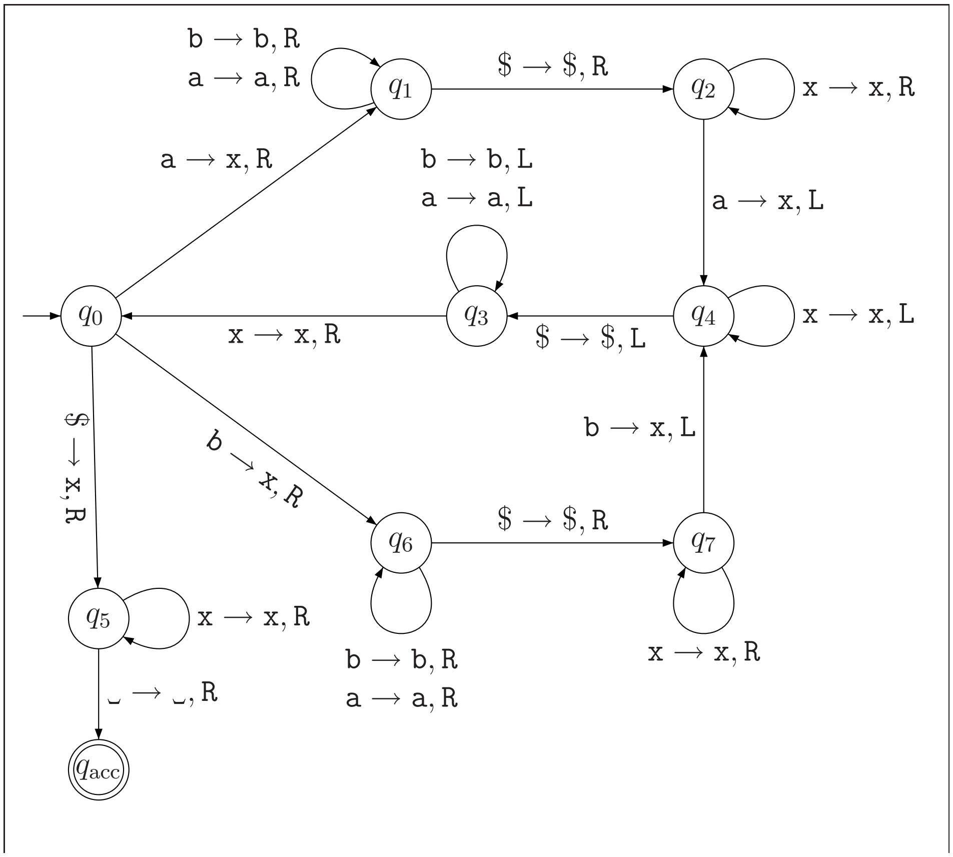


Figure 21.2: A TM for the language $w\$w$.

Mark start position by shifting

Let $\Sigma = \{a, b\}$. Write a Turing machine that puts a special character x at the start of the tape, shifting the input over one position, then accepting the input.

Accepting or rejecting is not the point of this machine. Rather, marking the start of the input is a useful component for creating more complex algorithms. So you had normally see this machine used as part of a larger machine, and the larger machine would do the accepting or rejecting.

21.2.4 Formal definition of a Turing machine

A ***Turing machine*** is a 7-tuple

$$(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}),$$

where

- Q: finite set of states.
- Σ: finite input alphabet.
- Γ: finite tape alphabet.
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\mathtt{L}, \mathtt{R}\}$: Transition function.

  As a concrete example, if $\delta(q, c) = (q', c', \mathtt{L})$ means that, that if the TM is at state q, and the head on the tape reads the character c, then it should move to state q', replace c on the tape by c', and move the head on the tape to the left.
- $q_0 \in Q$ is the initial state.

- $q_{\text{acc}} \in Q$ is the ***accepting/final*** state.
- $q_{\text{rej}} \in Q$ is the ***rejecting*** state.

This definition assumes that we've already defined a special blank character. In Sipser, the blank is written $\sqcup$ or ␣. A popular alternative is B. (If you use any other symbol for blank, you should write a note explaining what it is.)

The special blank character (i.e., ␣) is in the tape alphabet but it is not in the input alphabet.

Example

For the TM of Figure 21.2, we have the following $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$, where

(i) $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_{\text{acc}}, q_{\text{rej}}\}$.

(ii) $\Sigma = \{\texttt{a}, \texttt{b}, \$\}$.

(iii) $\Gamma = \{\texttt{a}, \texttt{b}, \$, ␣, \texttt{x}\}$.

(iv) $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\texttt{L}, \texttt{R}\}$.

	a	b	$	␣	x
q_0	$(q_1, \texttt{x}, \texttt{R})$	$(q_6, \texttt{x}, \texttt{R})$	$(q_5, \texttt{x}, \texttt{R})$	`reject`	`reject`
q_1	$(q_1, \texttt{a}, \texttt{R})$	$(q_1, \texttt{b}, \texttt{R})$	$(q_2, \$, \texttt{R})$	`reject`	`reject`
q_2	$(q_4, \texttt{x}, \texttt{L})$	`reject`	`reject`	`reject`	$(q_2, \texttt{x}, \texttt{R})$
q_3	$(q_3, \texttt{a}, \texttt{L})$	$(q_3, \texttt{b}, \texttt{L})$	`reject`	`reject`	$(q_0, \texttt{x}, \texttt{R})$
q_4	`reject`	`reject`	$(q_3, \$, \texttt{L})$	`reject`	$(q_4, x, \texttt{L})$
q_5	`reject`	`reject`	`reject`	$(q_{\text{acc}}, ␣, \texttt{R})$	$(q_5, \texttt{x}, \texttt{R})$
q_6	$(q_6, \texttt{a}, \texttt{R})$	$(q_6, \texttt{b}, \texttt{R})$	$(q_7, \$, \texttt{R})$	`reject`	`reject`
q_7	`reject`	$(q_4, \texttt{x}, \texttt{L})$	`reject`	`reject`	$(q_7, \texttt{x}, \texttt{R})$
q_{acc}	No need to define				
q_{rej}	No need to define				

Here, `reject` stands for $(q_{\text{rej}}, \texttt{x}, \texttt{R})$.

(Filling this table was fun, fun, fun!)

Chapter 22

Lecture 18: More on Turing Machines

31 March 2009

This lecture covers the formal definition of a Turing machine and related concepts such as configuration and Turing decidable. It surveys a range of variant forms of Turing machines and shows for one of them (multi-tape) why it is equivalent to the basic model.

22.1 A Turing machine

A ***Turing machine*** is a 7-tuple

$$(Q, \Sigma, \Gamma, \delta, q_0, q_{\mathrm{acc}}, q_{\mathrm{rej}}),$$

where

- Q: finite set of states.
- Σ: finite input alphabet.
- Γ: finite tape alphabet.
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\mathtt{L}, \mathtt{R}\}$.
- $q_0 \in Q$ is the initial state.
- $q_{\mathrm{acc}} \in Q$ is the ***accepting***/***final*** state.
- $q_{\mathrm{rej}} \in Q$ is the ***rejecting*** state.

TM has a working space (i.e., tape) and its deterministic. It has a reading/writing head that can travel back and forth along the tape and rewrite the content on the tape. TM halts immediately when it enters the accept state (i.e., q_{acc}) and then it accepts the input, or when the TM enters the reject state (i.e., q_{rej}), and then it rejects the input.

Example 22.1.1 Here we describe a TM that takes it input on the tape, shifts it to the right by one character, and put a \$ on the leftmost position on the tape.

So, let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ (but the machine we describe would work for any alphabet). Let

$$Q = \{q_0, q_{\mathrm{acc}}, q_{\mathrm{rej}}\} \cup \left\{ q_c \;\middle|\; c \in \Sigma \right\}.$$

Now, the transitions function is

$$\begin{aligned}
\forall s \in \Sigma \qquad & \delta(q_0, s) = (q_s, \$, \mathtt{R}) \\
\forall s, t \in \Sigma \qquad & \delta(q_s, t) = (q_t, s, \mathtt{R}) \\
\forall s \in \Sigma \qquad & \delta(q_s, \text{␣}) = (q_{\mathrm{acc}}, \$, \mathtt{R}). \\
& \delta(q_0, \text{␣}) = (q_{\mathrm{acc}}, \$, \mathtt{R})
\end{aligned}$$

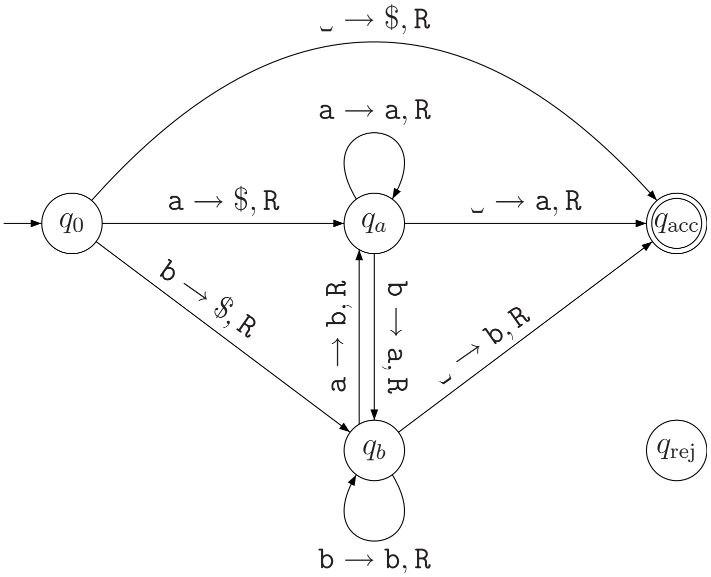


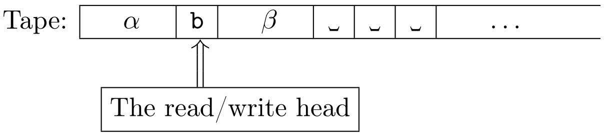
Figure 22.1: A TM that shifts its input right by one position, and inserts $ in the beginning of the tape.

The resulting machine is depicted in Figure 22.1, and here its pseudo-code:

```
Shift_Tape_Right
    At first tape position,
            remember character and write $
    At later positions,
            remember character on tape,
            and write previously remembered character.
    On blank, write remembered character and halt accepting.
```

22.2 Turing machine configurations

Consider a TM where the tape looks as follows,

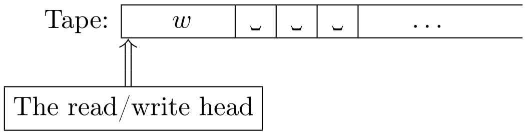


and the current control state of the TM is q_i. In this case, it would be convenient to write the TM ***configuration*** as

$$\alpha q_i \mathtt{b} \beta.$$

Namely, imagine that the head is just to the left of the cell its reading/writing, and $\mathtt{b}\beta$ is the string to the right of the head.

As such, the start ***configuration***, with a word w is



And this configuration is just $q_0 w$.

An ***accepting*** configuration for a TM is any configuration of the form $\alpha q_{\text{acc}} \beta$.

We can now describe a transition of the TM using this configuration notation. Indeed, imagine the given TM is in a configuration $\alpha q_i \mathtt{a} \beta$ and its transition is

$$\delta(q_i, \mathtt{a}) = (q_j, \mathtt{c}, \mathtt{R}),$$

then the resulting configuration is $\alpha \mathtt{c} q_j \beta$. We will write the resulting transition as

$$\alpha q_i \mathtt{a} \beta \quad \Rightarrow \quad \alpha \mathtt{c} q_j \beta.$$

Similarly, if the given TM is in a configuration

$$\gamma \ \mathtt{d} \ q_k \ \mathtt{e} \ \tau,$$

where γ and τ are two strings, and $\mathtt{d}, \mathtt{e} \in \Sigma$. Assume the TM transition in this case is

$$\delta(q_k, \mathtt{e}) = (q_m, \mathtt{f}, \mathtt{L}),$$

then the resulting configuration is $\gamma \ q_m \ \mathtt{d} \ \mathtt{f} \ \tau$. We will write this transition as

$$\underbrace{\gamma \ \mathtt{d} \ q_k \ \mathtt{e} \ \tau}_{c} \quad \Rightarrow \quad \underbrace{\gamma \ q_m \ \mathtt{d} \ \mathtt{f} \ \tau}_{c'}.$$

In this case, we will say that c ***yields*** c', we will use the notation $c \mapsto c'$.

As we seen before, the ends of tape are special, as follows:

- You can not move off the tape from the left side. If the head is instructed to move to the left, it just stays where it is.
- The tape is padded on the right side with spaces (i.e., ␣). Namely, you can think about the tape as initially as being full with spaces (spaced out?), except for the input that is written on the beginning of the tape.

22.3 The languages recognized by Turing machines

Definition 22.3.1 For a TM M and a string w, the Turing machine M ***accepts*** w if there is a sequence of configurations

$$C_1, C_2, \ldots, C_k,$$

such that

(i) $C_1 = q_0 w$, where q_0 is the start state of M,

(ii) for all i, we have C_i yields C_{i+1} (using M transition function, naturally), and

(iii) C_k is an accepting configuration.

Definition 22.3.2 The ***language*** of a TM M (i.e., Turing machine M) is

$$L(M) = \left\{ w \;\middle|\; M \text{ accepts } w \right\}.$$

The language L is called ***Turing recognizable***.

Note, that if $w \in L(M)$ then M halts on w and accepts it. On the other hand, if $w \notin L(M)$ then either M halts and rejects w, or M loops forever on the input w. Specifically, for an input w a TM can either:

(a) accept (and then it halts),

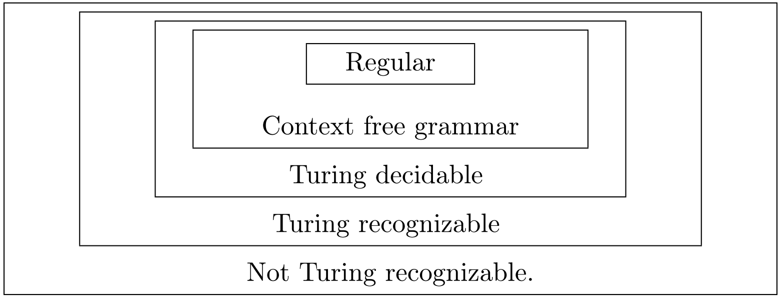
(b) reject (and then it halts),

(c) or be in an infinite loop.

Definition 22.3.3 A TM that halts on all inputs is called a ***decider***.

As such, a language L is ***Turing decidable*** if there is a decider TM M, such that $L(M) = L$.

The hierarchy of languages looks as follows:



22.4 Variations on Turing Machines

There are many variations on the definition of a Turing machine which do not change the languages that can be recognized. Well-known variations include doubly-infinite tapes, a stay-put option, non-determinism, and multiple tapes. Turing machines can also be built with very small alphabets by encoding symbol names in unary or binary.

22.4.1 Doubly infinite tape

What if we allow the Turing machine to have an infinite tape on both sides? It turns out the resulting machine is not stronger than the original machine. To see that, we will show that a doubly infinite tape TM can be simulated on the standard TM.

So, consider a TM M that uses a doubly infinite tape. We will simulate this machine by a standard TM. Indeed, fold the tape of M over itself, such that location $i \in [-\infty, \infty]$ is mapped to location

$$h(i) = \begin{cases} 2|i| & i \leq 0 \\ 2i - 1 & i > 0. \end{cases}$$

on the usual tape. Clearly, now the doubly infinite tape becomes the usual one-sided infinite tape, and we can easily simulate the original machine on this new machine. Indeed, as long as we are far from the folding point on the tape, all we need to do is to just move in jumps of two (i.e., move `L` is mapped into move `LL`). Now, if we reach the beginning of the tape, we need to change between odd location and even location, but that's also easy to do with a bit of care. We omit the easy but tedious details.

Another approach would be to keep the working part of the doubly-infinite tape in its original order. When the machine tries to move off the lefthand end, push everything to the right to make more space.

22.4.2 Allow the head to stay in the same place

Allowing the read/write head to stay in the same place is clearly not a significant extension, since we can easily simulate this ability by moving the head to the right, and then moving it back to the left. Formally, we allow transitions to be of the form

$$\delta(q, \mathtt{c}) = (q', \mathtt{d}, \mathtt{S}),$$

where `S` denotes the command for the read/write head to stay where it is (rewriting the character on the tape from `c` to `d`).

22.4.3 Non-determinism

This does not buy you anything, but the details are not trivial, and we will delay the discussion of this issue to later.

22.4.4 Multi-tape

Consider a TM that has k tapes, where $k > 1$ is a some finite integer constant. Here each tape has its own read/write head, but there is only one finite control. The transition function of this machine, is a function

$$\delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{\mathtt{L}, \mathtt{R}, \mathtt{S}\}^k,$$

and the initial input is placed on the first tape.

22.5 Multiple tapes do not add any power

We next prove that one of these variations (multi-tape) is equivalent to a standard Turing machine. Proofs for most other variations are similar.

Claim 22.5.1 *A multi-tape TM N can be simulated by a standard TM.*

Proof: We will build a standard (single tape) TM simulating N.

Initially, the input $w$ is written on the (only) tape of M. We rewrite the tape so that it contains k strings, each string matches the content of one of the tapes of N. Thus, the rewriting of the input, would result in a tape that looks like the following:

$$\$w \underbrace{\$ \text{␣} \$ \text{␣} \ldots \$ \text{␣}}_{k-1 \text{times}} \$.$$

The string between the ith and $(i+1)$th \$ in this string, is going to be the content of the ith tape. We need to keep track on each of these tapes where the head is supposed to be. To this end, we create for each character $\mathtt{a} \in \Gamma$, we create a dotted version, for example $\boxed{\overset{\bullet}{\mathtt{a}}}$. Thus, if the initial input $w = xw'$, where x is a character, the new rewritten tape, would look like:

$$\$\overset{\bullet}{x}w' \underbrace{\$ \overset{\bullet}{\text{␣}} \$ \overset{\bullet}{\text{␣}} \ldots \$ \overset{\bullet}{\text{␣}}}_{k-1 \text{times}} \$.$$

This way, we can keep track of the head location in each one of the tapes.

For each move of N, we go back on M to the beginning of the tape and scan the tape from left to right, reading all the dotted characters and store them (encoding them in the current state), once we did that, we know which transition of N needs to be executed:

$$q_{\langle c_1,\ldots,c_k\rangle} \quad \rightarrow \quad q'_{\langle d_1,D_1,d_2,D_2,\ldots,d_k,D_k\rangle},$$

where $D_i \in \{\mathtt{L}, \mathtt{R}, \mathtt{S}\}$ is the instruction where the ith head must move. To implement this transition, we scan the tape from left to right (first moving the head to the start of the tape), and when we encounter the ith dotted character c_i, we replace it by (the undotted) d_i, and we move the head as instructed by D_i, by rewriting the relevant character (immidiately near the head location) by its dotted version. After doing that, we continue the scan to the right, to perform the operation for the remaining $i+1, \ldots, k$ tapes.

After completing this process, we might have $\overset{\bullet}{\$}$ on the tape (i.e., the relevant head is located on the end of the space allocated to its tape). We use the **Shift_Tape_Right** algorithm we describe above, to create space to the left of such a dotted dollar, and write in the newly created spot a dotted space. Thus, if the tape locally looked like

$$\ldots \mathtt{ab}\, \overset{\bullet}{\$}\, \mathtt{c} \ldots$$

then after the shifting right and dotting the space, the new tape would look like

$$\ldots \mathtt{ab}\, \overset{\bullet}{\text{␣}}\, \$\mathtt{c} \ldots$$

By doing this shift-right operation to all the dotted \$'s, we end up with a new tape that is guaranteed to have enough space if we decide to write new characters to any of the k tapes of N.

Its easy to now verify that we can now simulate N on this Turing machine M, which uses a single tape. In particular, any language that N recognizes is also recognized by M, which is a standard TM, establishing the claim. ■

Chapter 23

Lecture 19: Encoding problems and decidability

7 April 2009

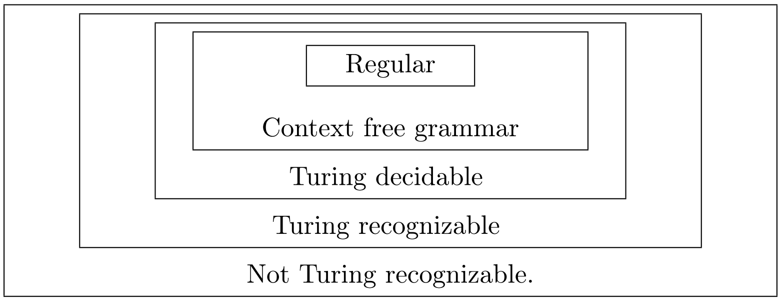
This lecture presents examples of languages that are ***Turing decidable***.

23.1 Review and context

Remember that a Turing machine D can do three sorts of things on an input w. The TM D might halt and accept. It might halt and reject. Or it might never halt. A TM is a ***decider*** if it always halts on all inputs.

A TM ***recognizable*** language is a language L for which there is a TM D, such that $\mathsf{L}(\mathtt{D}) = L$. A TM decidable language is a language L for which there is a decider TM D, such that $\mathsf{L}(\mathtt{D}) = L$.

Here is a figure showing the hierarchy of languages.



Conceptually, when we think about algorithms in computer science, we are normally interested in code which is guaranteed to halt on all inputs. So, for questions about languages, our primary interest is in Turing decidable (not just recognizable) languages.

Any algorithmic task can be converted into decision problem about languages. Some tasks are naturally in this form, e.g. "Is the length of this string prime?". In other cases, we have to restructure the question in one of two ways:

- A complex input object (e.g. a graph) may need to be encoded as a string.
- A construction task may have to be rephrased as a yes/no question.

23.2 TM example: Adding two numbers

23.2.1 A simple decision problem

For example, consider the task of adding two decimal numbers. The obvious algorithm might take two numbers a and b as input, and produce a number c as output. We can rephrase this as a question about languages by asking "Given inputs a, b, and c, is $c = a + b$".

For the alphabet

$$\Sigma = \{\mathtt{0}, \mathtt{1}, \ldots, \mathtt{9}, +, -\},$$

consider the language

$$L = \left\{ a_n a_{n-1} \ldots a_0 + b_m b_{m-1} \ldots b_0 = c_r c_{r-1} \ldots c_0 \;\middle|\; \begin{array}{l} a_i, b_j, c_k \in [0,9] \text{ and} \\ \langle a_n a_{n-1} \ldots a_0 \rangle \\ \quad + \quad \langle b_m b_{m-1} \ldots b_0 \rangle \\ \qquad\qquad = \langle c_r c_{r-1} \ldots c_0 \rangle \end{array} \right\},$$

where $\langle a_n a_{n-1} \ldots a_0 \rangle = \sum_{i=0}^{n} a_i \cdot 10^i$ is the number represented in base ten by the string $a_n a_{n-1} \ldots a_0$.

We then ask whether we can build a TM which decides the language L.

23.2.2 A decider for addition

To build a decider for this addition problem, we will use a multi-tape TM. We showed (last class) that a multi-tape TM is equivalent to a single tape TM. First, let us build a useful helper function, which reverses the contents of one tape.

Reversing a tape

Given the content of tape ✇$_1$, we can reverse it easily in two steps using a temporary tape. First, we put a marker onto the temporary tape. Moving the heads on both tapes to the right, we copy the contents of ✇$_1$ onto the temporary tape.

Next, we put the ✇$_1$ head at the start of its tape, but the temporary tape head remains at the end of this tape. We copy the material back onto ✇$_1$, but in reverse order, moving the ✇$_1$ head rightwards and the temporary tape head leftwards.

Let **ReverseTape**(t) denote the TM mechanism (i.e. procedure) that reverses the tth tape. We are going to buildup TM by putting together such procedures.

Adding two numbers

Now, let us assemble the addition algorithm. We will use five tapes: the input (✇$_1$), three tapes to hold numbers (✇$_2$, ✇$_3$, and ✇$_4$), and a scratch tape used for the reversal operation.

The TM will first scan the input tape (i.e., ✇$_1$), and copy the first number to ✇$_2$, and the second number to ✇$_3$. Next, we do **ReverseTape**(2) and **ReverseTape**(3). Now, we move the head of ✇$_2$ and ✇$_3$ to the beginning of the tapes, and we start moving them together computing the sum of the digits under the two heads, writing the output to ✇$_4$, and moving the three heads to the right. Naturally, we have a carry over digit, which we encode in the current state of the TM controller (the carry over digit is either $0, 1$ or 2).

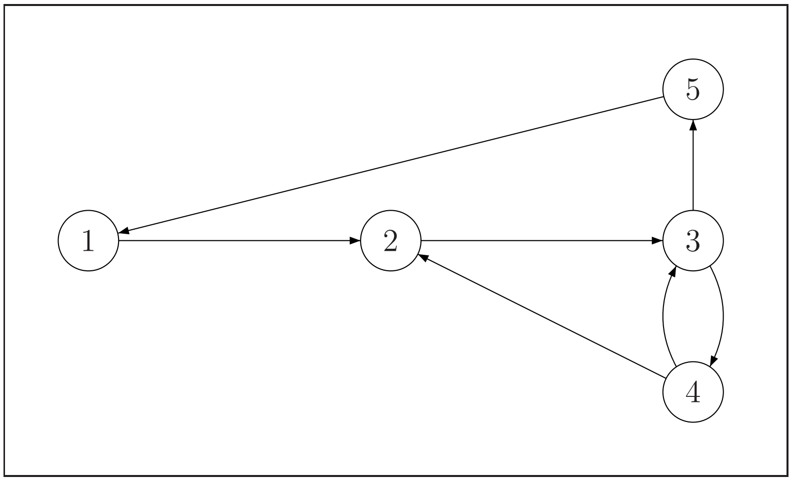
If one of the heads of ✇$_2$ or ✇$_3$ reaches the end of the tape, then we continue moving it, interpreting ␣ as a 0. We halt when the heads on both tapes see ␣.

Next, we move the head of ✇$_4$ back to the beginning of the tape, and do **ReverseTape**(4). Finally, we compare the content of ✇$_4$ with the number written on ✇$_1$ after the = character. If they are equal, the TM accepts, otherwise it rejects.

23.3 Encoding a graph problem

As the above example demonstrates, the ***coding scheme*** used for the input has big impact on the complexity of our algorithm. The addition algorithm would have been easier if the numbers were written in reverse order, or if they had been in binary. Such details may affect the running time of the algorithm, but they do not change whether the problem is Turing decidable or not.

When algorithms operate on objects that are not strings, these objects need to be ***encoded*** into strings before we can make the algorithm into a decision problem. For example, consider the following situation.



Graph encoding

```
5
7
(1,2)
(2,3)
(3,5)
(5,1)
(3,4)
(4,3)
(4,2)
```

Figure 23.1: A graph encoded as text. The string encoding the graph is in fact “`5`⟨`NL`⟩`7`⟨`NL`⟩`(1,2)`⟨`NL`⟩`(2,3)`⟨`NL`⟩`(3,5)`⟨`NL`⟩`(5,1)`⟨`NL`⟩`(3,4)`⟨`NL`⟩`(4,3)`⟨`NL`⟩`(4,2)`”. Here ⟨`NL`⟩ denotes the spacial new-line character.

We are given a ***directed graph*** $G = (V, E)$, and two vertices $s, t \in V$, and we would like to decide if there is a way to reach t from s.

All sorts of encodings are possible. But it is easiest to understand if we use encodings that look like standard `ASCII` file, of the sort you might use as input to your `Java` or `C++` program. `ASCII` files look like they are two-dimensional. But remember that they are actually one-dimensional strings inside the computer. Line breaks display in a special way, but they are underlyingly just a special separator character (<`NL`> on a unix system), very similar to the \$ or # that we've used to subdivide items in our string examples.

To make things easy, we will number the vertices of V from 1 to $n = |V|$. To specify that there is an edge between two vertices u and v, we then specify the two indices of u and v. We will use the notation (u, v). Thus, to specify a graph as a text file, we could use the following format, where n is the number of vertices and m is the number of edges in the graph.

$$\begin{array}{l} n \\ m \\ (n_1, n'_1) \\ (n_2, n'_2) \\ \vdots \\ (n_m, n'_m) \end{array}$$

Namely, the first line of the file, will contain the number (written explicitly using `ASCII`), next the second line is the number of edges of G (i.e., m). Then, every line specify one edge of the graph, by specifying the two numbers that are the vertices of the edge. As a concrete example, consider the following graph.

The number of edges is a bit redundant, because we could just stop reading at the end of the file. But it is convenient for algorithm design.

See Figure 23.1, for an example of a graph its encoding using these scheme.

23.4 Algorithm for graph reachability

To encode an instance of the ***s, t-reachability problem***, our `ASCII` file will need to contain not only the graph but also the vertices s and t. The input tape for our `TM` would contain all this information, laid out in 1D (i.e. imagine the line break displayed as an ordinary separator character).

To solve this problem, we will need to search the graph, starting with node s. The TM accepts iff this search finds the node t. We will store information on four TM tapes, in addition to the input tape. The TM would have the following tapes:

✇$_1$: Input tape

✇$_2$: Target node t.

✇$_3$: Edge list.

✇$_4$: *Done list*: list of nodes that we've finished processing

✇$_5$: *To-do list*: list of nodes whose outgoing edges have not been followed

Given the graph, the TM reads the graph (checking that the input is in the right format). It puts the list of edges onto tape ✇$_3$, puts t onto its own tape (i.e., ✇$_2$), and puts the node s onto the to-do list tape (i.e., ✇$_5$).

Next, the TM loops. In each iteration, it removes the first node x from the to-do list. If $x = t$, the TM halts and accepts. Otherwise, x is added to the done list (i.e., ✇$_4$). Then the TM searches the Edge list for all edges going outwards from x. Suppose an outgoing edge goes from x to y. Then if y is not already on the finished list or the to-do list, then y is added to the to-do list.

If there is nothing left on on the to-do list, the TM halts and rejects.

This algorithm is a ***graph search*** algorithm. It is breadth-first search if the new nodes are added to the end of the to-do list and depth-first search if they are added in the start of the list. (Or, said another way, the to-do list operates as either a queue or a stack.)

The separate visited list is necessary to prevent the algorithm from going into an infinite loop if the graph contains cycles.

23.5 Some decidable DFA problems

If D is a DFA, the string encoding of D is written as $\langle \mathtt{D} \rangle$.

The string encoding of a DFA is similar to the encoding of a directed graph except that our encoding has to have a label for each edge, specify the start state, and list the final states.

Emptiness of DFA. Consider the language

$$\mathsf{E}_{\mathsf{DFA}} = \left\{ \langle \mathtt{D} \rangle \;\middle|\; \mathtt{D} \text{ is a } \mathsf{DFA}\text{, and } \mathsf{L}(\mathtt{D}) = \emptyset \right\}.$$

This language is decidable. Namely, given an instance $\langle \mathtt{D} \rangle$, there is a TM that reads $\langle \mathtt{D} \rangle$, this TM always stops, and accepts if and only if $\mathsf{L}(\mathtt{D})$ is empty. Indeed, do a graph search on the DFA (as above) starting at the start state of D, and check whether any of the final states is reachable. If so, the $\mathsf{L}(\mathtt{D}) \neq \emptyset$.

Lemma 23.5.1 *The language $\mathsf{E}_{\mathsf{DFA}}$ is decidable.*

Emptiness of NFA. Consider the following language

$$\mathsf{E}_{\mathsf{NFA}} = \left\{ \langle \mathtt{D} \rangle \;\middle|\; \mathtt{D} \text{ is a } \mathsf{NFA}\text{, and } \mathsf{L}(\mathtt{D}) = \emptyset \right\}.$$

This language is decidable. Indeed, convert the given NFA into a DFA (as done in class, long time ago) and then call the code for E_{DFA} on the encoded DFA. Notice that the first step in this algorithm takes the encoded version of D and writes the encoding for the corresponding DFA. You can imagine this as taking a state diagram as input and producing a new state diagram as output.

Equal languages for DFAs. Consider the language

$$\mathsf{EQ}_{\mathsf{DFA}} = \left\{ \langle \mathtt{D}, \mathtt{C} \rangle \;\middle|\; \mathtt{D} \text{ and } \mathtt{C} \text{ are } \mathsf{NFA}\text{s, and } \mathsf{L}(\mathtt{D}) = \mathsf{L}(\mathtt{C}) \right\}.$$

This language is also decidable. Remember that the ***symmetric difference*** of two sets X and Y is $X \oplus Y = (X \cap \overline{Y}) \cup (Y \cap \overline{X})$. The set $X \oplus Y$ is empty if and only if the two sets are equal. But, given a DFA, we know how to make a DFA recognizing the complement of its language. And we also know how to take two DFA's and make a DFA recognizing the union or intersection of their languages.

So, given the encodings for D and C, our TM will construct the encoding of a DFA $\langle \mathtt{B} \rangle$ recognizing the symmetric difference of their languages. Then it would call the code for deciding if $\langle \mathtt{B} \rangle \in \mathsf{E}_{\mathsf{DFA}}$.

Informally, problems involving regular languages are always decidable, because they are so easy to manipulate. Problems involving context-free languages are sometimes decidable. And only the simplest problems involving Turing machines are decidable.

23.6 The acceptance problem for DFA's

The following language is also decidable:

$$\mathsf{A}_{\mathsf{DFA}} = \left\{ \langle \mathtt{D}, w \rangle \;\middle|\; \mathtt{D} \text{ is a } \mathsf{DFA},\ w \text{ is a word, and } \mathtt{D} \text{ accepts } w. \right\}.$$

As before, the notation $\langle \mathtt{D}, w \rangle$ is the encoding of the DFA D and the word w; that is, it is the pair $\langle \mathtt{D} \rangle$ and $\langle w \rangle$. For example, if $\langle w \rangle$ is just w (it's already a string), then $\langle \mathtt{D}, w \rangle$ might be $\langle \mathtt{D} \rangle \# w$ where $\#$ is some separator character. Or it might be $(\langle \mathtt{D} \rangle, w)$. Or anything similar that encodes the input well. We will just assume that it is in some such reasonable encoding of a pair and that the low-level code for our TM (which we will not spell out in detail) knows what it is.

A Turing machine deciding $\mathsf{A}_{\mathsf{DFA}}$ needs to be able to take the code for some arbitrary DFA, plus some arbitrary string, and decide if that DFA accepts that string. So it will need to contain a general-purpose DFA simulator. This is called the ***acceptance problem*** for DFA's.

It's useful to contrast this with a similar-sounding claim. If D is any DFA, then $\mathsf{L}(\mathtt{D})$ is Turing-decidable. Indeed, to build a TM that accepts $\mathsf{L}(\mathtt{D})$, we simply move the TM head to the right over the input, using the TM's controller to simulate the controller of the DFA directly.

In this case, we are given a specific fixed DFA D and we only need to cook up a TM that recognizes strings from this one particular language. This is much easier than $\mathsf{A}_{\mathsf{DFA}}$.

To decide $\mathsf{A}_{\mathsf{DFA}}$, our TM will use five tapes:

✇$_1$: input: $\langle \mathtt{D}, w \rangle$,

✇$_2$: state,

✇$_3$: final states

✇$_4$: transition triples

✇$_5$: input string.

The simulator then runs as follows:

(1) Check the format of the input. Copy the start state to tape ✇$_2$. Copy the input string to tape ✇$_5$. Copy the transition triples and final states of the input machine $\langle \mathtt{D} \rangle$ to tapes ✇$_3$ and ✇$_4$.

(2) Put the tape ✇$_5$ head at the beginning of the tape.

(3) Find a transition triple $p \xrightarrow{\mathsf{c}} q$ (written on tape ✇$_4$) whose input state and character match the state written on tape ✇$_1$ (i.e., p) and the character (i.e., c) under the head on tape ✇$_5$.

(4) Change the current state of the simulated DFA from p to q.

Specifically, copy the state q (written on the triple we just found on ✇$_4$), to tape ✇$_2$.

(5) Move the tape ✇$_5$ head to the right (i.e., the simulation handled this input character).

(6) Goto step (3).

(7) Halt the loop when the tape ✇$_5$ head sees a blank. Accept if and only if the state on tape ✇$_2$ is one of the states on list of final states, stored on tape ✇$_3$.

Chapter 24

Lecture 20: More decidable problems, and simulating TM and “real” computers

9 April 2009

This lecture presents more example of languages that are *Turing decidable*, from Sipser section 4.1.

24.1 Review: decidability facts for regular languages

A language is ***decidable*** if there is a TM that is a decider (i.e., a TM that always stops) that accepts this language. If $\mathtt{D}$ is a DFA, the string encoding of $\mathtt{D}$ is written as $\langle \mathtt{D} \rangle$. The encoding of a pair $\mathtt{D}$ and w is written as $\langle \mathtt{D}, w \rangle$.

Decidable DFA problems. The following languages are all decidable.

(A) $\mathsf{E}_{\mathsf{DFA}} = \left\{ \langle \mathtt{D} \rangle \,\middle|\, \mathtt{D} \text{ is a DFA and } \mathsf{L}(\mathtt{D}) = \emptyset \right\}$.

This is the language of all DFAs with an empty language.

(B) $\mathsf{EQ}_{\mathsf{DFA}}$: the language of all pairs of DFAs that have the same language.

(C) $\mathsf{A}_{\mathsf{DFA}} = \left\{ \langle \mathtt{D}, w \rangle \,\middle|\, \mathtt{D} \text{ is a DFA, } w \text{ is a word, and } \mathtt{D} \text{ accepts } w \right\}$.

Here $\langle \mathtt{D}, w \rangle$ is in the language if and only if the DFA $\mathtt{D}$ accepts w.

(D) $\mathrm{A}_{\mathsf{NFA}} = \left\{ \langle \mathtt{D}, w \rangle \,\middle|\, \mathtt{D} \text{ is a NFA generating } w \right\}$.

(E) $\mathsf{EQ}_{\mathsf{DFA}} = \left\{ \langle \mathtt{D}, \mathtt{C} \rangle \,\middle|\, \mathtt{D}, \mathtt{C} \text{ are DFA's and } \mathsf{L}(\mathtt{D}) = \mathsf{L}(\mathtt{C}) \right\}$.

(F) $\mathrm{A}_{\mathrm{regex}} = \left\{ \langle R, w \rangle \,\middle|\, R \text{ is a regular expression generating } w \right\}$.

To decide this language, the TM can convert R into a DFA $\mathtt{D}$, and then check if $\langle \mathtt{D}, w \rangle \in \mathsf{A}_{\mathsf{DFA}}$.

24.2 Problems involving context-free languages

The situation with context-free languages is more complicated, because some problems are Turing decidable and some are not.

24.2.1 Context-free languages are TM decidable

Given a RA P, we are interested in the question of whether we can build a TM decider that accepts L(D). Observe, that we can turn P into an equivalent CFG, and this CFG can be turned into an equivalent CNF grammar $\mathcal{G}$. With $\mathcal{G}$ it is now easy to decide if an input word w is in $\mathsf{L}(\mathcal{G})$. Indeed, we can either using the CYK algorithm to decide if a word is in the grammar, or alternatively, enumerate all possible parse trees for the given CNF that generates the given word w. That is, if $n = |w|$, then we need to generate all possible parse trees with $2n - 1$ internal nodes (since this is the size of a parse tree deriving such a word in CNF), and see if any of them generates w. In either case, we have the following.

Lemma 24.2.1 *Given a RA* P*, there is a TM* T *which is a decider, and* $\mathsf{L}(\mathtt{P}) = \mathsf{L}(\mathtt{T})$*. Namely, for every RA there exists an equivalent TM.*

24.2.2 Is a word in a CFG?

The following construction of a TM is somewhat similar to the one in Section 24.2.1.

Lemma 24.2.2 *The language* $\mathrm{A}_{\mathsf{CFG}} = \left\{ \langle \mathcal{G}, w \rangle \;\middle|\; \mathcal{G} \text{ is a CFG and } \mathcal{G} \text{ generates } w \right\}$*. is decidable.*

Proof: We build a TM $\mathtt{T}_{\mathsf{CFG}}$ for $\mathrm{A}_{\mathsf{CFG}}$. The input for it is the pair $\langle \mathcal{G}, w \rangle$. AS a first step, we convert $\mathcal{G}$ to be in CNF (we saw the algorithm of how to do this in detail in class). Let $\mathcal{G}'$ denote the resulting grammar. Next, we use CYK to decide if $w \in \mathsf{L}(\mathcal{G}')$. If it is, the TM $\mathtt{T}_{\mathsf{CFG}}$ accepts, otherwise it rejects. ■

Given a TM decider $\mathtt{T}_{\mathsf{CFG}}$ for $\mathrm{A}_{\mathsf{CFG}}$, building a TM decider that has language equal to a specific given $\mathcal{G}$ is easy. Specifically, given $\mathcal{G}$, we would like to build a TM decider $\mathtt{T}'$ such that $\mathsf{L}(\mathtt{T}') = \mathsf{L}(\mathcal{G})$.

So, modify the given TM to encode $\mathcal{G}$. As a first step, the new TM $\mathtt{T}'$ would write $\mathcal{G}$ on the input tape (next to the input word w). Next, it would run the TM $\mathtt{T}_{\mathsf{CFG}}$ on this given input to decide if $\langle \mathcal{G}, w \rangle \in \mathrm{A}_{\mathsf{CFG}}$.

Remark 24.2.3 (Encoding instances inside a TM.) The above demonstrates that given a more general algorithm we can use it to solve the problem for specific instances. This is done by encoding the given specific instance in the constructed TM.

If you have trouble imaging this encoding the whole CFG grammar into the TM, as done above, think about storing a short string like UIUC in the TM state diagram, to be written out on (say) tape 2. The first state transition in the TM would write U onto tape 2, the next three transitions would write I, then U, then C. Finally, it would move the tape 2 head back to the beginning and transition into the first state that does the actual computation.

Note, that doing this encoding of a specific instance inside the TM, does not necessarily yields the most efficient TM for the problem. For example, in the above, we could first convert the given instance into CNF before encoding it into the TM.

We could also hard-code the string w into our TM but leave the grammar as a variable input. We omit the proof of the following easy lemma.

Lemma 24.2.4 *Let w be a specific string. The language* $A_{\mathsf{CFG},w} = \left\{ \langle \mathcal{G} \rangle \;\middle|\; \mathcal{G} \text{ is a CFG and } \mathcal{G} \text{ generates } w \right\}$ *is decidable.*

24.2.3 Is a CFG empty?

Lemma 24.2.5 *The language* $\mathsf{E}_{\mathsf{CFG}} = \left\{ \langle \mathcal{G} \rangle \;\middle|\; \mathcal{G} \text{ is a CFG and } \mathsf{L}(\mathcal{G}) = \emptyset \right\}$ *is decidable.*

Proof: We already saw that in the conversion algorithm of a CFG into CNF (this was one of the initial steps of this conversion). We shortly re-sketch the algorithm.

To this end, the TM mark all the variables in $\mathcal{G}$ that can generate (in one step) a string of terminals (or ϵ of course). We will refer to such a variable as being useful. Now, the TM iterates repeatedly over the rules of $\mathcal{G}$. For a rule $\mathsf{X} \rightarrow w$, where w is a string of terminals and variables, the variable X is useful, if all the variables of w are useful, and in such a case we will mark X as useful. The loop halts when the TM has made a full pass through the rules of $\mathcal{G}$ without marking anything new as useful.

This TM accepts the input grammar if the initial variable of $\mathcal{G}$ is useful, and otherwise it rejects.

At every iteration over all the rules of $\mathcal{G}$ the TM must designate at least one new variable as new to repeat this process again. So it follows that the number of outer iterations performed by this algorithm is bounded by the number of variables in the grammar $\mathcal{G}$‘, implying that this algorithm always terminates. ■

24.2.4 Undecidable problems for CFGs

We quickly mention a few problems that are not TM decidable. We will prove this fact later in the course. The following languages are *not* TM decidable.

(i) $\mathsf{EQ}_{\mathsf{CFG}} = \left\{ \langle \mathcal{G}, \mathcal{G}' \rangle \,\middle|\, \mathcal{G}, \mathcal{G}' \text{ are } \mathsf{CFG} \text{ and } \mathsf{L}(\mathcal{G}) = \mathsf{L}(\mathcal{G}') \right\}$.

To see why this is surprising, we remind the reader that this language was solvable for DFAs.

(ii) $\mathsf{ALL}_{\mathsf{CFG}} = \left\{ \langle \mathcal{G} \rangle \,\middle|\, \mathcal{G} \text{ is a } \mathsf{CFG} \text{ and } \mathsf{L}(\mathcal{G}) = \Sigma^* \right\}$.

24.3 Simulating a real computer with a Turing machine

We would like to argue that we can simulate a "real" world computer on a Turing machine. Here are some key program features that we would like to simulate on a TM.

- **Numbers & arithmetic**: We already saw in previous lecture how some basic integer operations can be handled. It is not too hard to extend these to negative integers and perform all required numerical operations if we allow a TM with multiple tapes. As such, we can assume that we can implement any standard numerical operation.

  Of course, can also do floating point operations on a TM. The details are overwhelming but they are quite doable. In fact, until 20 years[1] ago, many computers implemented floating point operations using integer arithmetic. Hardware implementation of floating point-operations became mainstream, when Intel introduced the i486 in 1989 that had FPU (floating-point unit). You would probably will see/seen how floating point arithmetic works in computer architecture courses.

- **Stored constant strings**: The program we are trying to translate into a TM might have strings and constants in it. For example, it might check if the input contains the (all important) string `UIUC`. As we saw above, we can encode such strings in the states. Initially, on power-up, the TM starts by writing out such strings, onto a special tape that we use for this purpose.

- **Random-access memory**: We will use an associative memory. Here, consider the memory as having a unique label to identify it (i.e., its address), and content. Thus, if cell 17 contains the value `abc`, we will consider it as storing the pair $(17, \texttt{abc})$. We can store the memory on a tape as a list of such pairs. Thus, the tape might look like:

  $$(17, \texttt{abc})\$(1, \texttt{samuel})\$(85, \texttt{noclue})\$ \ldots (11, \texttt{stamp})\$\textvisiblespace\textvisiblespace\textvisiblespace\textvisiblespace\textvisiblespace \ldots$$

  Here, address 17 stores the string `abc`, address 1 stores the string `samuel`, and so on.

  Reading the value of address x from the tape is easy. Suppose x is written on ✇$_i$, and we would like to find the value associated with x on the memory tape and write it onto ✇$_j$. To do this, the TM scans ✇$_{mem}$ the memory tape (i.e., the tape we use to simulate the associative memory) from the beginning, till the TM encounter a pair in ✇$_{mem}$ having x as its first argument. It then copies the second part of the pair to the output tape ✇$_j$.

  Storing new value (x, y) in memory is almost as easy. If a pair having x as first element exists you delete it out (by writing a special cross-out character over it), and then you write the new pair (x, y) in the end of the tape ✇$_{mem}$.

[1] This number keep changing. Very irritating.

If you wanted to use memory more efficiently, the new value could be written into the original location, whenever the original location had enough room. You could also write new pairs into crossed-out regions, if they have enough room. Implementations of C `malloc`/`free` and Java garbage collection use slightly more sophisticated versions of these ideas. However, TM designers rarely care about efficiency.

- **Subroutine calls**: To simulate a real program, we need to be able to do calls (and recursive calls). The standard way to implement such things is by having a stack. It is clear how to implement a stack on its own TM tape.

  We need to store three pieces of information for each procedure call:

  (i) private working space,

  (ii) the return value,

  (iii) and the name of the state to return to after the call is done.

  The private working space needs to be implemented with a stack, because a set of nested procedure calls might be active all at once, including several recursive calls to the same procedure.

  The return value can be handled by just putting it onto a designated register tape, say $☢_{24}$.

  Right before we give control over to a procedure, we need to store the name of the state it should return to when it is done. This allows us to call a single fixed piece of code from several different places in our TM. Again, these return points need to be put on a stack, to handle nested procedure calls.

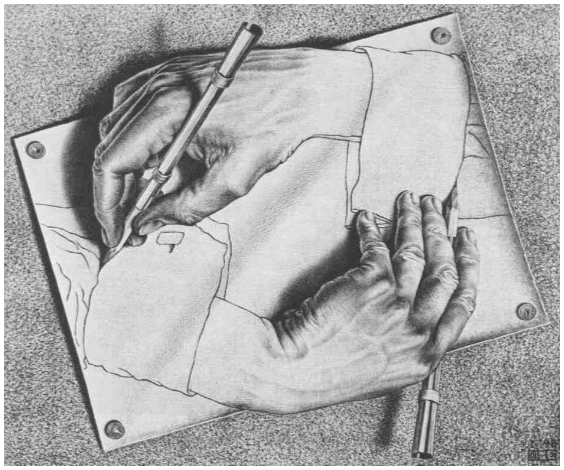
  After it returns from a procedure, the TM reads the state name to return to. A special set of TM states handle reading a state name and transitioning to the corresponding TM state.

These are just the most essential features for a very simple general-purpose computer. In some computer architecture class, you will see how to implement fancier program features (e.g. garbage collection, objects) on top of this simple model.

24.4 Turing machine simulating a Turing machine

24.4.1 Motivation

We already seen that a TM can simulate a DFA. We think about TMs as being just regular computer programs. So think about an interpreter. What is it? It is a program that reads in another program (for example, think about Java virtual machine) and runs it.[2]



So, what would be the equivalent of an interpreter in the language of Turing machines? Well, its a TM that reads in a description of a TM M, and an input w for it, and simulates running M on w.

Initially this construct looks very weird - inherently circular in nature. But it is useful for the same reason interpreters are useful: It enable us to manipulate TMs (i.e., programs) directly and modify them without knowing in advance what they are. In particular, we can start talking computationally about ways of manipulating TMs (i.e., programs).

For example, in a perfect world (which we are not living in, naturally), we would like to give a formal specification of a program (say, a TM that decides if a prime number is prime), and have another program

[2] Things of course are way more complicated in practice, since Java virtual machines nowadays usually compile portions of the code being run frequently to achieve faster performance (i.e., just in time compilation [JIT]), but still, you can safely think about a JVM as an interpreter.

that would swallow this description and spits out the program performing this computation (i.e., have a computer that writes our programs for us).

A more realistic example is a compiler which translates (say) Java code into assembly code. It takes code as input and procedures code in a different language as output. We could also build an optimizer that reads Java code and produces new code, also in Java but more efficient. Or a cheating helper program that reads Java code and writes out a new version with different variable names and modified comments.

24.4.2 The universal Turing machine

We would like to build the ***universal Turing machine*** U_{TM} that recognizes the language

$$\mathsf{A}_{\mathsf{TM}} = \left\{ \langle \mathtt{T}, w \rangle \;\middle|\; \mathtt{T} \text{ is a } \mathsf{TM} \text{ and } \mathtt{T} \text{ accepts } w \right\}.$$

We emphasize that U_{TM} is **<u>not</u>** a decider. Namely, its stops only if $\mathtt{T}$ accepts w, but it might run forever if $\mathtt{T}$ does not accept w.

To simplify our discussion, we assume that $\mathtt{T}$ is a single tape machine with some fixed alphabet (say $\Sigma_{\mathtt{T}} = \{\mathtt{0}, \mathtt{1}\}$ and the tape alphabet is $\Gamma_{\mathtt{T}} = \{\mathtt{0}, \mathtt{1}, \llcorner\!\lrcorner\}$. To simplify the discussion, the TM for A_{TM} is going to be a multi-tape machine. Naturally, one can convert this TM into a single tape TM.

So, the input for U_{TM} is an encoding $\langle \mathtt{T}, w \rangle$. As a first step, the U_{TM} would verify that the input is in the right format (such a reasonable encoding for a TM was given as an exercise in the homework). The U_{TM} would copy different components of the input into different tapes:

☢_1 : Transition function δ of $\mathtt{T}$.

It is going to be a sequence (separated by \$) of transitions. A transition $(q, c) \rightarrow (q', t, \mathtt{L})$ would be encouded as a string of the form:

$$(\#q, c) - (\#q', t, \mathtt{L})$$

where $\#q$ is the index which is the index of the state q (in $\mathtt{T}$) and $\#q'$ is the index of q'.

More specificially, you an think about the states of $\mathtt{T}$ being numbered between 1 and m, and $\#q$ is just the binary representation of the index of the state q.

☢_2 : $\#q_0$ – the initial state of $\mathtt{T}$.

☢_3 : $\#q_{\mathrm{acc}}$ – the accept state of $\mathtt{T}$.

☢_4 : $\#q_{\mathrm{rej}}$ – the reject state of $\mathtt{T}$.

☢_5 : $\$w$ – the input tape to be handled.

Once done copying the input, the U_{TM} would move the head of ☢_5 to the beginning of the tape. It then performs the following loop:

(I) Loop:

(i) Scan ☢_1 to find transition matching state on ☢_2 and the character under the head of ☢_5.

(ii) Update state on ☢_2.

(iii) Update character and head position on ☢_5.

We repeat this till the state in ☢_2 is equal to the state written on either ☢_3 (q_{acc}) or ☢_4 (q_{rej}).

Naturally, U_{TM} accepts if $\text{☢}_2 = \text{☢}_3$ and rejects if $\text{☢}_2 = \text{☢}_4$ at any point during the simulation.

Chapter 25

Lecture 21: Undecidability, halting and diagonalization

14 April 2009

> 'There must be some mistake,' he said, 'are you not a greater computer than the Milliard Gargantubrain at Maximegalon which can count all the atoms in a star in a millisecond?'
>
> 'The Milliard Gargantubrain?' said Deep Thought with unconcealed contempt. 'A mere abacus - mention it not.'
>
> – The Hitch Hiker's Guide to the Galaxy, by Douglas Adams.

In this lecture we will discuss the ***halting problem*** and ***diagonalization***. This covers most of Sipser section 4.2.

25.1 Liar's Paradox

There's a widespread fascination with logical paradoxes. For example, in the Deltora Quest novel "The Lake of Tears" (author Emily Rodda), the hero Lief has just incorrectly answered the trick question posed by the giant guardian of a bridge.

> "We will play a game to decide which way you will die," said the man. "You may say one thing, and one thing only. If what you say is true, I will strangle you with my bare hands. If what you say is false, I will cut off your head."

After some soul-searching, Lief replies "My head will be cut off." At this point, there's no way for the giant to make good on his threat, so the spell he's under melts away, he changes back to his original bird form, and Lief gets to cross the bridge.

The key problem for the giant is that, if he strangles Lief, then Lief's statement will have been false. But he said he would strangle him only if his statement was true. So that does not work. And cutting off his head does not work any better. So the giant's algorithm sounded good, but it turned out not to work properly for certain inputs.

A key property of this paradox is that the input (Lief's reply) duplicates material used in the algorithm. We've fed part of the algorithm back into itself.

25.2 The halting problem

Consider the following language

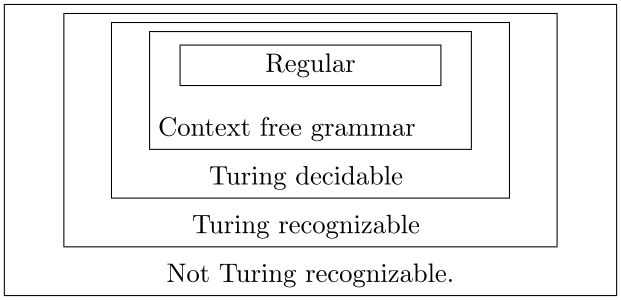
$$A_{\mathsf{TM}} = \left\{ \langle M, w \rangle \;\middle|\; M \text{ is a } \mathsf{TM} \text{ and } M \text{ accepts } w \right\}.$$

We saw in the previous lecture, that one can build a universal Turing machine U_{TM} that can simulate any Turing machine on any input. As such, using U_{TM}, we have the following TM recognizing A_{TM}:

```
Recognize-$A_{TM}$( $\langle M, w \rangle$)
  Simulate $M$ using $U_{TM}$ till it halts
  if $M$ halts and accepts then
          accept
  else
          reject
```

Note, that if M goes into an infinite loop on the input w, then the TM **Recognize-**A_{TM} would run forever. This means that this TM is only a recognizer, not a decider. A decider for this problem would call a halt to simulations that will loop forever. So the question of whether A_{TM} is TM decidable is equivalent to asking whether we can tell if a TM M will halt on input w. Because of this, both versions of this question are typically called the ***halting*** problem.

We remind the reader that the language hierarchy looks as depicted on the right.



25.2.1 Implications

So, let us suppose that the Halting problem (i.e., deciding if a word in is in A_{TM}) were decidable. Namely, there is an algorithm that can solves it (for any input). this seems somewhat hard to believe since even humans can not solve this problem (and we still live under the delusion that we are smarter than computers).

If we could decide the Halting problem, then we could build compilers that would automatically prevent programs from going into infinite loops and other very useful debugging tools. We could also solve a variety of hard mathematical problems. For example, consider the following program.

```
Percolate ( $n$)
        for $p < q < n$ do
                if $p$ is prime and $q$ is prime, and $p + q = n$ then
                    return

        If program reach this point then Stop!!!



Main:
        $n \leftarrow 4$
        while true do
                Percolate (n)
                $n \leftarrow n + 2$
```

Does this program stops? We do not know. If it does stop, then the ***Strong Goldbach conjecture*** is false.

Conjecture 25.2.1 (Strong Goldbach conjecture.) *Every even integer greater than 2 can be written as a sum of two primes.*

This conjecture is still open and its considered to be one of the major open problems in mathematics. It was stated in a letter on 7 of June 1742, and it is still open. Its seems unlikely that a computer program would be able to solve this, and a larger number of other mathematical conjectures. If A_{TM} is decidable, then we can write a program that would try to generate all possible proofs of a conjecture and verify each proof. Now, if we can decide if a programs stop, then we can discover whether or not a mathematical conjecture is true or not, and this seems extremely unlikely (that a computer would be able to solve all problems in mathematics).

I hope that this informal argument convinces you that its seems extremely unlikely that A_{TM} is TM decidable. Fortunately, we can prove this fact formally.

25.3 Not all languages are recognizable

Let us show a non-constructive proof that not all languages are Turing recognizable. This is true because there are fewer Turing machines than languages.

Fix an alphabet Σ and define the lexicographic order on Σ^* to be: first order strings by length, within each length put them in dictionary order.

Lexicographic order gives us a mapping from the integers to all strings, e.g. s_1 is the first string in our ordered list, and s_i is the ith string.

The encoding of each Turing machine is a finite-length string. So we can put all Turing machines into an ordered list by sorting their encodings in lexicographic order. Let us call the Turing machines in our list M_1, M_2, and so forth.

We can make an (infinite) table of how each Turing machine behaves on each input string. This table in depicted on the right. Here, the ith row represents the ith TM M_i, where the jth entry in the row is acc if M_i accepts the jth word s_j.

The idea is now to define a language from the table. Consider the language L_{diag} which is the language formed by taking the diagonal of this table.

	s_1	s_2	s_3	s_4	$\ldots$
M_1	**acc**	acc	rej	rej	$\ldots$
M_2	rej	**acc**	rej	acc	$\ldots$
M_3	acc	rej	**acc**	acc	$\ldots$
M_4	rej	acc	rej	**rej**	$\ldots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Formally, the word $s_i \in L_{\mathrm{diag}}$ if and only if M_i accepts the string s_i. Now, consider the complement language $L = \overline{L_{\mathrm{diag}}}$.

This language can not be recognized by an of the Turing machines on the list $M_1, M_2, \ldots$. Indeed, if M_k recognized the language L, then consider s_k. There are two possibilities.

- If M_k accepts s_k then the kth entry in the kth row of this infinite table is acc. Which implies in turn that $s_k \notin L$ (since L is the complement language), but then M_k (which recognizes L) must not accept s_k. A contradiction.
- If M_k does not accept s_k then the kth entry in the kth row of this infinite table is rej. Which implies in turn that $s_k \in L$ (since L is the complement language), but then M_k (which recognizes L) must accept s_k. A contradiction.

Thus, our assumption that all languages have a TM that recognizes them is false. Let us summarize this very surprising result.

Theorem 25.3.1 *Not all languages have a TM that recognize them.*

Intuitively, the above claim is a statement above infinities: There are way more languages (essentially, any real number defines a language) than TMs, as the number of TMs is countable (i.e., as numerous as integer numbers). Since the cardinality of real numbers (i.e., $\aleph$) is strictly larger than the cardinality of integer numbers (i.e., $\aleph_0$), it follows that there must be an orphan language without a machine recognizing it.

A limitation of the preceding proof is that it does not identify any particular tasks that are not TM recognizable or decidable. Perhaps the problem tasks are only really obscure problems of interest only to mathematicians. Sadly, that is not true.

25.4 The Halting theorem

We will now show that a particular concrete problem is not TM decidable. This will let us construct particular concrete problems that are not even TM recognizable.

Theorem 25.4.1 (The halting theorem.) *The language A_{TM} is not TM decidable,* *Proof:* Assume A_{TM} is TM decidable, and let **Halt** be this TM deciding A_{TM}. That is, **Halt** is a TM that always halts, and works as follows

$$\textbf{Halt}\Big(\langle M, w\rangle\Big) = \begin{cases} \text{accept} & M \text{ accepts } w \\ \text{reject} & M \text{ does not accept } w. \end{cases}$$

We will now build a new TM **Flipper**, such that on the input $\langle M\rangle$, it runs **Halt** on the input $\langle M, M\rangle$. If **Halt**$\Big(\langle M, M\rangle\Big)$ accepts than **Flipper** rejects, and if **Halt**$\Big(\langle M, M\rangle\Big)$ rejects than **Flipper** accepts. Formally

```
Flipper ( ⟨M⟩)
    res ← Halt(⟨M, M⟩)
    if res is accept then
              reject
    else
              accept
```

The key observation is that **Flipper** *always stops*. Indeed, it uses **Halt** as a subroutine and **Halt** by our assumptions always halts. In particular, we have the following

$$\textbf{Flipper}\Big(\langle M\rangle\Big) = \begin{cases} \text{reject} & M \text{ accepts } \langle M\rangle \\ \text{accept} & M \text{ does not accept } \langle M\rangle\,. \end{cases}$$

Flipper is a TM (duh!), and as such it has an encoding $\langle$**Flipper**$\rangle$. Now, consider running **Flipper** on itself. We get the following

$$\textbf{Flipper}\Big(\langle \textbf{Flipper}\rangle\Big) = \begin{cases} \text{reject} & \textbf{Flipper} \text{ accepts } \langle \textbf{Flipper}\rangle \\ \text{accept} & \textbf{Flipper} \text{ does not accept } \langle \textbf{Flipper}\rangle\,. \end{cases}$$

This is absurd. Ridiculous even! Indeed, if **Flipper** accepts $\langle$**Flipper**$\rangle$, then it rejects it (by the above definition), which is impossible. Indeed, if **Flipper** must reject (note, that **Flipper** always stops!) $\langle$**Flipper**$\rangle$, but then by the above definition it must accept $\langle$**Flipper**$\rangle$, which is also impossible.

Thus, it must be that our assumption that **Halt** exists is false. We conclude that A_{TM} is not TM decidable. ■

Corollary 25.4.2 *The language A_{TM} is TM recognizable but not TM decidable,*

25.4.1 Diagonalization view of this proof

Let us redraw the diagonalization table from Section 25.3. This time, we will include only input strings that happen to be encodings of Turing machines. The table on the right shows the behavior of **Halt** on inputs of the form $\langle M_i, M_j\rangle$. Our constructed TM **Flipper** takes two inputs that are identical $\langle M, M\rangle$ and its output is the opposite of **Halt**$\Big(\langle M, M\rangle\Big)$.

	$\langle M_1\rangle$	$\langle M_2\rangle$	$\langle M_3\rangle$	$\langle M_4\rangle$	$\dots$
M_1	**rej**	acc	rej	rej	$\dots$
M_2	rej	**acc**	rej	acc	$\dots$
M_3	acc	acc	**acc**	rej	$\dots$
M_4	rej	acc	acc	**rej**	$\dots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

So it corresponds to the negation of the entries down the diagonal of this table. Again, we essentially argued that there is no row in this infinite table that its entries are the negation of the diagonal. As such, our assumption that **Halt** is a decider, was false.

25.5 More Implications

From this basic result, we can derive a huge variety of problems that can not be solved. Spinning out these consequences will occupy us for most of the rest of the term.

Theorem 25.5.1 *There is no `C` program that reads a `C` program P and input w, and decides if P "accepts" w.*

The proof of the above theorem is identical to the halting theorem - we just perform our rewriting on the `C` program.

Also, notice that being able to recognize a language and its complement implies that the language is decidable, as the following theorem testifies.

Theorem 25.5.2 *A language is TM decidable iff it is TM recognizable and its complement is also TM recognizable.*

Proof: It is obvious that decidability implies that the language and its complement are recognizable. To prove the other direction, assume that L and $\overline{L}$ are both recognizable. Let M and N be Turing machines recognizing them, respectively. Then we can build a decider for L by running M and N in parallel.

Specifically, suppose that w is the string input to M. Simulate both M and N using U_{TM}, but single-step the simulations. Advance each simulation by one step, alternating between the two simulations. Halt when either of the simulations halts, returning the appropriate answer.

If w is in L, then the simulation of M must eventually halt. If w is not in L, then the simulation of N must eventually halt. So our combined simulation must eventually halt and, therefore, it is a decider for L. ■

A quick consequence of this theorem is that:

Theorem 25.5.3 *The set complement of A_{TM} is not TM recognizable.*

If it were recognizable, then we could build a decider for A_{TM} by Theorem 25.5.2.

Chapter 26

Lecture 22: Reductions

16 April 2009

26.1 What is a reduction?

Last lecture we proved that A_{TM} is undecidable. Now that we have one example of an undecidable language, we can use it to prove other problems to be undecidable.

Meta definition: Problem A ***reduces*** to problem B, if given a solution to B, then it implies a solution for A. Namely, we can solve B then we can solve A. We will done this by A $\implies$ B.

An ***oracle*** ORAC for a language L is a function that receives as a word w, and it returns true if and only if $w \in L$. An oracle can be thought of as a black box that can solve membership in a language without requiring us to consider the question of whether L is computable or not. Alternatively, you can think about an oracle as a provided library function that computes whatever it requires to do, and it always return (i.e., it never goes into an infinite loop).

Intuitively, a TM decider for a language L is the ultimate oracle. Not only it can decide if a word is in L, but furthermore, it can be implemented as a TM that always stops.

In the context of showing languages are undecidable, the following more specific definition would be useful.

Definition 26.1.1 A language X ***reduces*** to a language Y, if one can construct a TM decider for X using a given oracle ORAC_Y for Y.

We will denote this fact by $X \implies Y$.

In particular, if X reduces to Y then given a decider for the language Y (i.e., an oracle for Y), then there is a program that can decide X. So Y must be at least as "hard" as X. In particular, if X is undecidable, then it must be that Y is also undecidable.

Warning. It is easy to get confused about which of the two problems "reduces" to the other. Do not get hung up on this. Instead, concentrate on getting the right outline for your proofs (proving them in the right direction, of course).

Reduction proof technique. Formally, consider a problem B that we would like to prove is undecidable. We will prove this via reduction, that is a proof by contradiction, similar in outline to the ones we have seen for regular and context-free languages. You assume that your new language L (i.e., the language of B) is decided by some TM M. Then you use M as a component to create a decider for some language known to be undecidable (typically A_{TM}). This is would imply that we have a decider for A (i.e., A_{TM}). But this is a contradiction since A (i.e., A_{TM}) is not decidable. As such, we must have been wrong in assuming that L was decidable.

We will concentrate on using reductions to show that problems are undecidable. However, the technique is actually very general. Similar methods can be used to show problems to be not TM recognizable. We have used similar proofs to show languages to be not regular or not context-free. And reductions will be used in CS 473 to show that certain problems are "NP complete", i.e. these problems (probably) require exponential time to solve.

26.1.1 Formal argument

Lemma 26.1.2 *Let X and Y be two languages, and assume that $X \implies Y$. If Y is TM decidable then X is TM decidable.*

Proof: Let $\mathtt{T}$ be the TM decider for Y. Since X reduces to Y, it follows that there is a procedure $\mathtt{T}_{X|Y}$ (i.e., TM decider) for X that uses an oracle for Y as a subroutine. We replace the calls to this oracle in $\mathtt{T}_{X|Y}$ by calls to $\mathtt{T}$. The resulting TM $\mathtt{T}_X$ is a TM decider and its language is X. Thus X is TM decidable. ■

The counter-positive of this lemma, is what we will use.

Lemma 26.1.3 *Let X and Y be two languages, and assume that $X \implies Y$. If X is TM undecidable then Y is TM undecidable.*

26.2 Halting

We remind the reader that A_{TM} is the language

$$\mathrm{A}_{\mathsf{TM}} = \left\{ \langle M, w \rangle \;\middle|\; M \text{ is a } \mathsf{TM} \text{ and } M \text{ accepts } w \right\}.$$

This is the problem that we showed (last class) to be undecidable (via diagonalization). Right now, it is the only problem we officially know to be undecidable.

Consider the following slight modification, which is all the pairs $\langle M, w \rangle$ such that M ***halts*** on w. Formally,

$$A_{\mathrm{Halt}} = \left\{ \langle M, w \rangle \;\middle|\; M \text{ is a } \mathsf{TM} \text{ and } M \text{ stops on } w \right\}.$$

Intuitively, this is very similar to A_{TM}. The big obstacle to building a decider for A_{TM} was deciding whether a simulation would ever halt or not.

To show formally that A_{Halt} is undecidable, we show that we can use a oracle for A_{Halt} to build a decider for A_{TM}. This construction looks like the following.

Lemma 26.2.1 *The language A_{TM} reduces to A_{Halt}. Namely, given an oracle for A_{Halt} one can build a decider (that uses this oracle) for A_{TM}.*

Proof: Let ORAC_{Halt} be the given oracle for A_{Halt}. We build the following decider for A_{TM}.

```
Decider-$\mathrm{A}_{\mathsf{TM}}\big(\langle M, w \rangle\big)$
        $res \leftarrow \mathsf{ORAC}_{Halt}\big(\langle M, w \rangle\big)$
        // if $M$ does not halt on $w$ then reject.
        if $res =$ reject then
                halt and reject.

        // $M$ halts on $w$ since $res =$accept.
        // Thus, simulating $M$ on $w$ would terminate in finite time.
        $res_2 \leftarrow$Simulate $M$ on $w$ (using $U_{\mathsf{TM}}$).

        return $res_2$.
```

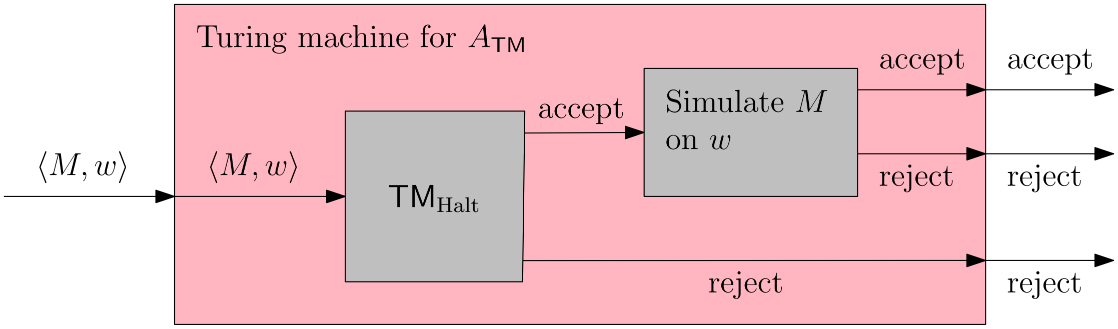
Clearly, this procedure always return and as such its a decider for A_{TM}. ■

Theorem 26.2.2 *The language A_{Halt} is not decidable.*

Proof: Assume, for the sake of contradiction, that A_{Halt} is decidable. As such, there is a TM, denoted by $\mathsf{TM}_{\text{Halt}}$, that is a decider for A_{Halt}. We can use $\mathsf{TM}_{\text{Halt}}$ as an implementation of an oracle for A_{Halt}, which would imply by Lemma 26.2.1 that one can build a decider for A_{TM}. However, A_{TM} is undecidable. A contradiction. It must be that A_{Halt} is undecidable. ■

We will be usually less formal in our presentation. We will just show that given a TM decider for A_{Halt} implies that we can build a decider for A_{TM}. This would imply that A_{TM} is undecidable.

Thus, given a black box (i.e., decider) $\mathsf{TM}_{\text{Halt}}$ that can decide membership in A_{Halt}, we build a decider for A_{TM} is follows.



This would imply that if A_{Halt} is decidable, then we can decide A_{TM}, which is of course impossible.

26.3 Emptiness

Now, consider the language

$$E_{\mathsf{TM}} = \left\{ \langle M \rangle \;\middle|\; M \text{ is a } \mathsf{TM} \text{ and } L(M) = \emptyset \right\}.$$

Again, we assume that we have a decider for E_{TM}. Let us call it $\mathsf{TM}_{E_{\mathsf{TM}}}$. We need to use the component $\mathsf{TM}_{E_{\mathsf{TM}}}$ to build a decider for A_{TM}.

A decider for A_{TM} is given M and w and must decide whether M accepts w. We need to restructure this question into a question about some Turing machine having an empty language. Notice that the decider for E_{TM} takes only one input: a Turing machine. So we have to somehow make the second input (w) disappear.

The key trick here is to hard-code w into M, creating a TM M_w which runs M on the fixed string w. Specifically the code for M_w might look like:

TM M_w:

1. Input $= x$ (which will be ignored)
2. Simulate M on w.
3. If the simulation accepts, accept. If the simulation rejects, reject.

Its important to understand what is going on. The input is $\langle M \rangle$ and w. Namely, a string encoding M and a the string w. The above shows that we can write a procedure (i.e., TM) that accepts this two strings as input, and outputs the string $\langle M_w \rangle$ which encodes M_w. We will refer to this procedure as **EmbedString**. The algorithm **EmbedString**$(\langle M, w\rangle)$ as such, is a procedure reading its input, which is just two strings, and outputting a string that encodes the TM $\langle M_w \rangle$.

It is natural to ask, what is the language of the machine encoded by the string $\langle M_w \rangle$; that is, what is $L(M_w)$?

Because we are ignoring the input x, the language of M_w is either Σ^* or $\emptyset$. It is Σ^* if M accepts w, and it is $\emptyset$ if M does not accept w.

We are now ready to prove the following theorem.

Theorem 26.3.1 *The language E_{TM} is undecidable.*

Proof: We assume, for the sake of contradiction, that E_{TM} is decidable, and let $\mathsf{TM}_{E_{\mathsf{TM}}}$ be its decider. Next, we build our decider **AnotherDecider-A**$_{\mathsf{TM}}$ for A_{TM}, using the **EmbedString** procedure described above.

```
AnotherDecider-A_TM(⟨M, w⟩)
        $\langle M_w \rangle \leftarrow$ EmbedString $(\langle M, w \rangle)$
        $r \leftarrow \mathsf{TM}_{E_{\mathsf{TM}}}(\langle M_w \rangle)$.
        if $r =$ accept then
                reject.

        // $\mathsf{TM}_{E_{\mathsf{TM}}}(\langle M_w \rangle)$ rejected its input

        return accept
```

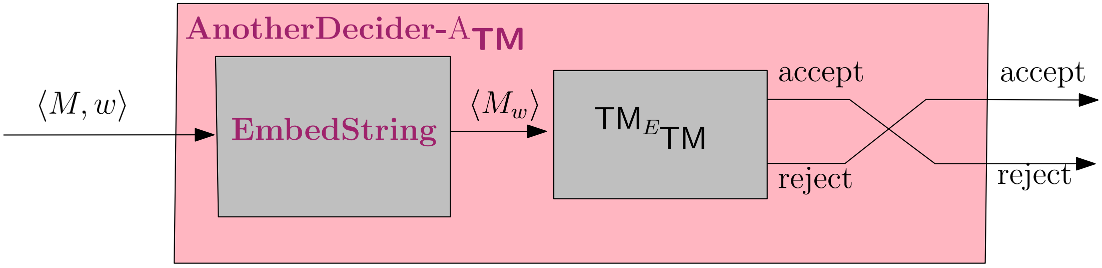
Consider the possible behavior of **AnotherDecider-A**$_{\mathsf{TM}}$ on the input $\langle M, w \rangle$.

- If $\mathsf{TM}_{E_{\mathsf{TM}}}$ accepts $\langle M_w \rangle$, then $L(M_w)$ is empty. This implies that M does not accept w. As such, **AnotherDecider-A**$_{\mathsf{TM}}$ rejects its input $\langle M, w \rangle$.
- If $\mathsf{TM}_{E_{\mathsf{TM}}}$ accepts $\langle M_w \rangle$, then $L(M_w)$ is not empty. This implies that M accepts w. So **AnotherDecider-A**$_{\mathsf{TM}}$ accepts $\langle M, w \rangle$.

Namely, **AnotherDecider-A**$_{\mathsf{TM}}$ is indeed a decider for A_{TM}, (its a decider since it always stops on its input). But we know that A_{TM} is undecidable, and as such it must be that our assumption that E_{TM} is decidable is false. ■

In the above proof, note that **AnotherDecider-A**$_{\mathsf{TM}}$ is indeed a decider, so it always halts, either accepting or rejecting. By contrast, M_w might not always halt. So, when we do our analysis, we need to think about what happens if M_w never halts. In this example, if M never halts on w, then w will be treated just like the explicit rejection cases and this is what we want.

Here is the code for **AnotherDecider-A**$_{\mathsf{TM}}$ in flow diagram form.



Observe, that **AnotherDecider-A**$_{\mathsf{TM}}$ never actually runs the code for M_w. It hands the code to a function $\mathsf{TM}_{E_{\mathsf{TM}}}$ which analyzes what the code would do if we ever did choose to run it. But we never run it. So it does not matter that M_w might go into an infinite loop.

Also notice that we have two input strings floating around our code: w (one input to the decider for A_{TM}) and x (input to M_w). Be careful to keep track of which strings are input to which functions. Also be careful about how many inputs, and what types of inputs, each function expects.

26.4 Equality

An easy corollary of the undecidability of E_{TM} is the undecidability of the language

$$EQ_{\mathsf{TM}} = \left\{ \langle M, N \rangle \;\middle|\; M \text{ and } N \text{ are } \mathsf{TM}\text{'s and } L(M) = L(N) \right\}.$$

Lemma 26.4.1 *The language EQ_{TM} is undecidable.*

Proof: Suppose that we had a decider **DeciderEqual** for EQ_{TM}. Then we can build a decider for E_{TM} as follows:

```
TM $R$:
  1. Input = $\langle M \rangle$
  2. Include the (constant) code for a TM $T$ that rejects all its input. We denote the string encoding $T$ by $\langle T \rangle$.
  3. Run DeciderEqual on $\langle M, T \rangle$.
  4. If DeciderEqual accepts, then accept.
  5. If DeciderEqual rejects, then reject.
```

■

Since the decider for E_{TM} (i.e., $\mathsf{TM}_{E_{\mathsf{TM}}}$) takes one input but the decider for EQ_{TM} (i.e. **DeciderEqual**) requires two inputs, we are tying one of **DeciderEqual**'s input to a constant value (i.e., T).

There are many Turing machines that reject all their input and could be used as T. Building code for R just requires writing code for one such TM.

26.5 Regularity

It turns out that almost any property defining a TM language induces a language which is undecidable, and the proofs all have the same basic pattern. Let us do a slightly more complex example and study the outline in more detail.

Let

$$\text{Regular}_{\mathsf{TM}} = \left\{ \langle M \rangle \;\middle|\; M \text{ is a } \mathsf{TM} \text{ and } L(M) \text{ is regular} \right\}.$$

Suppose that we have a TM **DeciderRegL** that decides $\text{Regular}_{\mathsf{TM}}$. In this case, doing the reduction from halting, would require to turn a problem about deciding whether a TM M accepts w (i.e., is $w \in \mathrm{A}_{\mathsf{TM}}$) into a problem about whether some TM accepts a regular set of strings.

Given M and w, consider the following TM M'_w:

```
TM $M'_w$:
  (i) Input = $x$
  (ii) If $x$ has the form $\mathtt{a}^n\mathtt{b}^n$, halt and accept.
  (iii) Otherwise, simulate $M$ on $w$.
  (iv) If the simulation accepts, then accept.
  (v) If the simulation rejects, then reject.
```

Again, we are ***not*** going to execute M'_w directly ourself. Rather, we will feed its description $\langle M'_w \rangle$ (which is just a string) into **DeciderRegL**. Let **EmbedRegularString**denote this algorithm, which accepts as input $\langle M \rangle$ and w, and outputs $\langle M'_w \rangle$, which is the encoding of the machine M'_w.

If M accepts w, then every input x will eventually be accepted by the machine M'_w. Some are accepted right away and some are accepted in step (i). So if M accepts w then the language of M'_w is Σ^*.

If M does not accept w, then some strings x (that are of the form $\mathtt{a}^n\mathtt{b}^n$) will be accepted in step (ii) of M'_w. However, after that, either step (iii) will never halt or step (iv) will reject. So the rest of the strings (that are in the set $\Sigma^* \setminus \left\{ \mathtt{a}^n\mathtt{b}^n \;\middle|\; n \geq 0 \right\}$) will not be accepted. So the language of M'_w is $\mathtt{a}^n\mathtt{b}^n$ in this case.

Since $\mathtt{a}^n\mathtt{b}^n$ is not regular, we can use our decider **DeciderRegL** on M'_w to distinguish these two cases.

Notice that the test in step (ii) was cooked up specifically to match the capabilities of our given decider **DeciderRegL**. If **DeciderRegL** had been testing whether our language contained the string "uiuc", step (ii) would be comparing x to see if it was equal to "uiuc". This test can be anything that a TM can compute without the danger of going into an infinite loop.

Specifically, we can build a decider for A_{TM} as follows.

```
YetAnotherDecider-$A_{\mathsf{TM}}$($\langle M, w\rangle$)
    $\langle M'_w\rangle \leftarrow$ EmbedRegularString ($\langle M, w\rangle$)
    $r \leftarrow$ DeciderRegL($\langle M'_w\rangle$).
    return $r$
```

The reason why **YetAnotherDecider-**A_{TM} does the right thing is that:

— If **DeciderRegL** accepts, then $L(M'_w)$ is regular. So it must be Σ^*. This implies that M accepts w. So **YetAnotherDecider-**A_{TM} should accept $\langle M, w\rangle$.

— If **DeciderRegL** rejects, then $L(M'_w)$ is not regular. So it must be $\mathtt{a}^n\mathtt{b}^n$. This implies that M does not accept w. So **YetAnotherDecider-**A_{TM} should reject $\langle M, w\rangle$.

26.6 Windup

Notice that the code in Section 26.5 is almost exactly the same as the code for the E_{TM} example in Section 26.3. The details of M_w and M'_w were different. And one example passed on the return values from **YetAnotherDecider-**A_{TM} directly, whereas the other example negated them. This similarity is not accidental, as many examples can be done with very similar proofs.

Next class, we will see Rice's Theorem, which uses this common proof template to show a very general result. Namely, almost any nontrivial property of a TM's language is undecidable.

Chapter 27

Lecture 23: Rice Theorem and Turing machine behavior properties

21 April 2009

This lecture covers Rice's theorem, as well as decidability of TM behavior properties.

27.1 Outline & Previous lecture

27.1.1 Forward outline of lectures

This week and next, we'll see three major techniques for proving undecidability:

- Rice's Theorem (today): generalize a lot of simple reductions with common outline.
- Linear Bounded automata (Thursday): Allow us to show that ALL_{CFG}, EQ_{CFG} are undecidable. Also, LBAs illustrate a useful compromise in machine power: much of the flexibility of a TM but enough resource limits to be more analyzable.
- Post's Correspondence problem (a week from Thursday): allows us to show that $AMBIG_{\mathsf{CFG}}$ is undecidable.

27.1.2 Recap of previous class

In the previous class, we proved that the following language is undecidable.

$$\text{Regular}_{\mathsf{TM}} = \left\{ \langle M \rangle \;\middle|\; M \text{ is a } \mathsf{TM} \text{ and } L(M) \text{ is regular} \right\}.$$

To do this, we assume that $\text{Regular}_{\mathsf{TM}}$ was decided by some TM S. We then used this to build a decider for A_{TM} (which can not exist)

Decider for A_{TM}

(i) Input $= \langle M, w \rangle$

(ii) Construct $\langle M_w \rangle$ (see below).

(iii) Feed $\langle M_w \rangle$ to S and return the result.

Our auxiliary TM M_w looked like:

TM M_w:

(i) Input $= x$

(ii) If x has the form $\mathtt{a}^n\mathtt{b}^n$, halt and accept.

(iii) Otherwise, simulate M on w.

(iv) If the simulation accepts, then accept.

(v) If the simulation rejects, then reject.

The language of M_w was either Σ^* or $\texttt{a}^n\texttt{b}^n$, depending on whether M accepts w.

27.2 Rice's Theorem

27.2.1 Another Example - The language L_3

Let us consider another reduction with a very similar outline. Suppose we have the following language

$$\mathsf{L}_3 = \left\{ \langle M \rangle \;\middle|\; |L(M)| = 3 \right\}.$$

That is L_3 contains all Turing machines whose languages contain exactly three strings.

Lemma 27.2.1 *The language* L_3 *is undecidable.*

Proof: Proof by reduction from A_{TM}. Assume, for the sake of contradiction, that L_3 was decidable and let $\texttt{decider}_{\mathsf{L}_3}$ be a TM deciding it. We use $\texttt{decider}_{\mathsf{L}_3}$ to construct a Turing machine $\texttt{decider}_9\texttt{-}\mathrm{A}_{\mathsf{TM}}$ deciding A_{TM}. The decider $\mathsf{TM}\texttt{decider}_9\texttt{-}\mathrm{A}_{\mathsf{TM}}$ is constructed as follows:

```
decider$_9$-A$_{\mathsf{TM}}$ ( $\langle M, w \rangle$ )
        Construct a new Turing machine $M_w$:

            $M_w$( $x$ ): // $x$:  input
                $res \leftarrow$ Run $M$ on $w$
                if  ($res$ = reject) then
                    reject
                if $x$ = UIUC or $x$ = Iowa or $x$ = Michigan then
                    accept

                reject

        return decider$_{\mathsf{L}_3}$ ($\langle M_w \rangle$).
```

(We emphasize here again, that constructing M_w involve taking the encoding of $\langle M \rangle$ and w, and generating the encoding of $\langle M_w \rangle$.)

Notice that the language of M_w has only two possible values. If M loops or rejects w, then $L(M_w) = \emptyset$. If M accepts w, then th the language of M_w contains exactly three strings: "`UIUC`", "`Iowa`", and "`Michigan`".

So $\texttt{decider}_9\texttt{-}\mathrm{A}_{\mathsf{TM}}\Big(\langle M_w \rangle\Big)$ accepts exactly when M accepts w. Thus, $\texttt{decider}_9\texttt{-}\mathrm{A}_{\mathsf{TM}}$ is a decider for A_{TM} But we know that A_{TM} is undecidable. A contradiction. As such, our assumption that L_3 is decidable is false. ■

27.2.2 Rice's theorem

Notice that these two reductions have very similar outlines. Our hypothetical decider **decider** looks for some property P. The auxiliary TM's tests x for membership in an example set with property P. The big difference is whether we simulate M on w before or after testing x and, consequently, whether the second possibility for $L(M_w)$ is $\emptyset$ or Σ^*.

It's easy to cook up many examples of reductions similar to this one, all involving sets of TM's whose *languages* share some property (e.g. they are regular, they have size three). Rice's Theorem generalizes all these reductions into a common result.

Theorem 27.2.2 (Rice's Theorem.) *Suppose that* L *is a language of Turing machines; that is, each word in* L *encodes a* TM. *Furthermore, assume that the following two properties hold.*

(a) Membership in L *depends only on the Turing machine's language, i.e. if* $L(M) = L(N)$ *then* $\langle M \rangle \in \mathsf{L} \Leftrightarrow \langle N \rangle \in \mathsf{L}$.

(b) The set L *is "non-trivial," i.e.* $\mathsf{L} \neq \emptyset$ *and* L *does not contain all Turing machines.*

Then L *is a undecidable.*

Proof: Assume, for the sake of contradiction, that L is decided by TM**deciderFor**L. We will construct a TM`Decider`$_4$-A_{TM} that decides A_{TM}. Since `Decider`$_4$-A_{TM} does not exist, we will have a contradiction, implying that **deciderFor**L does not exist.

Remember from last class that $\mathsf{TM}_\emptyset$ is a TM (pick your favorite) which rejects all input strings. Assume, for the time being, that $\mathsf{TM}_\emptyset \notin \mathsf{L}$. This assumption will be removed shortly.

Since L is non-trivial, also choose some other TM $Z \in \mathsf{L}$. Now, given $\langle M, w \rangle$ `Decider`$_4$-A_{TM} will construct the encoding of the following TM M_w.

TM M_w:

(1) Input $= x$.

(2) Simulate M on w.

(3) If the simulation rejects, halt and reject.

(4) If the simulation accepts, simulate Z on x and accept if and only if T halts and accepts.

If M loops or rejects w, then M_w will get stuck on line (2) or stop at line (3). So $L(M_w)$ is $\emptyset$. Because membership in L depends only on a Turing machine's language and $\langle \mathsf{TM}_\emptyset \rangle$ is not in L, this means that M_w is not in L. So M_w will be rejected by N.

If M accepts w, then M_w will proceed to line (4), where it simulates the behavior of Z. So $L(M_w)$ will be $L(Z)$. Because membership in L depends only on a Turing machine's language and T is L, this means that M_w is in L. So M_w will be accepted by N.

As usual, our decider for A_{TM} looks like:

```
Decider$_4$-$\mathrm{A}_{\mathsf{TM}}$ ($\langle M, w \rangle$)
        Construct $\langle M_w \rangle$ from $\langle M, w \rangle$
        return deciderFor$\mathsf{L}$ ($\langle M_w \rangle$)
```

So `Decider`$_4$-A_{TM} ($\langle M, w \rangle$) will accept $\langle M, w \rangle$ iff **deciderFor**L accepts M_w. But we saw above that **deciderFor**L accepts M_w iff M accepts w. So `Decider`$_4$-A_{TM} is a decider for A_{TM}. Since such a decider cannot exist, we must have been wrong in our assumption that there was a decider for L.

Now, let us remove the assumption that $\mathsf{TM}_\emptyset \notin \mathsf{L}$. The above proof showed that L is undecidable, assuming that $\langle \mathsf{TM}_\emptyset \rangle$ was not in L. If $\mathsf{TM}_\emptyset \in \mathsf{L}$, then we run the above proof using $\overline{\mathsf{L}}$ in place of L. At the end, we note that $\overline{\mathsf{L}}$ is decidable iff L is decidable. ■

27.3 TM decidability by behavior

27.3.1 TM behavior properties

One thinking about TMs there are three kind of properties one might consider:

(1) The language accepted by the TM's, e.g. the TM accepts the string "`UIUC`". In this case, such a property is very likely undecidable by Rice's theorem.

(2) The TM's structure, e.g. the TM has 13 states. In this case, the property can probably be checked directly on the given description of the TM, and as such this is (probably) decidable.

(3) The TM's behavior, e.g. the TM never moves left on input "UIUC". This kind properties can be either decidable or not depending on the behavior under consideration, and this classification might be non-trivial.

27.3.2 A decidable behavior property

For example, consider the following set of Turing machines:

$$L_R = \left\{ \langle M \rangle \;\middle|\; M \text{never moves left for the input } x \text{, where } x \text{ is the empty word} \right\}.$$

Surprising, the language L_R is decidable because never moving left (equivalently: always moving right) destroys the Turing machine's ability to do random access into its tape. It is effectively made into a DFA.

Specifically, if a Turing machine M never moves left, it reads through the whole input, then starts looking at blank tape cells. Once it is on the blank part of the tape, it can cycle through its set of states. But after $|Q|$ moves, it has run out of distinct states and must be in a loop. So, if you watch M for four moves (the length of the string "UIUC") plus $|Q| + 1$ moves, it has either halted or its in an infinite loop.

Therefore, to decide L_R, you simulate the input Turing machine for $|Q| + 5$ moves. After that many moves, it has either

- moved left (in which case you reject), or
- has halted or gone into an infinite loop without ever moving left (in which case you accept).

This algorithm is a decider (not just a recognizer) for L, because it definitely halts on any input Turing machine M.

27.3.3 An undecidable behavior property

By contract, consider the following language:

$$L_x = \left\{ \langle M \rangle \;\middle|\; M \text{ writes an } x \text{ at some point, when started on blank input} \right\}.$$

This language L_x is undecidable. The reason is that a Turing machine with this restriction (no writing x's) can simulate a Turing machine without the restriction.

Proof: Suppose that L_x were decidable. Let R be a Turing machine deciding L_x. We will now construct a Turing machine S that decides A_{TM}.

S is constructed as follows:

- Input is $\langle M, w \rangle$, where M is the code for a Turing Machine and w is a string.
- Construct the code for a new Turing machine M_w as follows
  (a) On input y (which will be ignored).
  (b) Substitute X for x every where in $< M >$ and w, creating new versions $< M' >$ and w'.
  (c) Simulate M' on w'
  (d) If M' rejects w', reject.
  (e) If M' accepts w', print x on the tape and then accept.
- Run R on $\langle M_w \rangle$. If R accepts, then accept. If R rejects, then reject.

If M accepts w, then M_w will print x on any input (and thus on a blank input). If M rejects w or loops on w, then M_w is guaranteed never to print x accidently. So R will accept $\langle M_w \rangle$ exactly when M accepts w. Therefore, S decides A_{TM}.

But we know that A_{TM} is undecidable. So S can not exist. Therefore we have a contradiction. So L_x must have been undecidable. ■

Appendix - more examples of undecidable languages

27.4 More examples

The following examples weren't presented in lecture, but may be helpful to students.

27.4.1 The language L_{UIUC}

Here's another example of a reduction that fits the Rice's Theorem outline.

Let

$$L_{\text{UIUC}} = \left\{ \langle M \rangle \;\middle|\; L(M) \text{ contains the string “UIUC”} \right\}.$$

Lemma 27.4.1 L_{UIUC} *is undecidable.*

Proof: Proof by reduction from A_{TM}. Suppose that L_{UIUC} were decidable and let R be a Turing machine deciding it. We use R to construct a Turing machine deciding A_{TM}. S is constructed as follows:

- Input is $\langle M, w \rangle$, where M is the code for a Turing Machine and w is a string.
- Construct code for a new Turing machine M_w as follows:
  - Input is a string x.
  - Erase the input x and replace it with the constant string w.
  - Simulate M on w.
- Feed $\langle M_w \rangle$ to R. If R accepts, accept. If R rejects, reject.

If M accepts w, the language of M_w contains all strings and, thus, the string “UIUC”. If M does not accept w, the language of M_w is the empty set and, thus, does not contain the string “UIUC”. So R($\langle M_w \rangle$) accepts exactly when M accepts w. Thus, S decides A_{TM}

But we know that A_{TM} is undecidable. So S does not exist. Therefore we have a contradiction. So L_{UIUC} must have been undecidable. ■

27.4.2 The language Halt_Empty_TM

Here's another example which isn't technically an instance of Rice's Theorem, but has a very similar structure.

Let

$$\mathsf{Halt\_Empty\_TM} = \left\{ \langle M \rangle \;\middle|\; M \text{ halts on blank input} \right\}.$$

Lemma 27.4.2 Halt_Empty_ *TM is undecidable.*

Proof: By reduction from A_{TM}. Suppose that Halt_Empty_TM were decidable and let R be a Turing machine deciding it. We use R to construct a Turing machine deciding A_{TM}. S is constructed as follows:

- Input is $\langle M, w\rangle$, where M is the code for a Turing Machine and w is a string.
- Construct code for a new Turing machine M_w as follows:
  - Input is a string x.
  - Ignore the value of x.
  - Simulate M on w.
- Feed $\langle M_w\rangle$ to R. If R accepts, then accept. If R rejects, then reject.

If M accepts w, the language of M_w contains all strings and, thus, in particular the empty string. If M does not accept w, the language of M_w is the empty set and, thus, does not contain the empty string. So $R\Big(\langle M_w\rangle\Big)$ accepts exactly when M accepts w. Thus, S decides A_{TM}

But we know that A_{TM} is undecidable. So S can not exist. Therefore we have a contradiction. So Halt_Empty_TM must have been undecidable. ■

27.4.3 The language L_{111}

Here is another example of an undecidable language defined by a Turing machine's behavior, to which Rice's Theorem does not apply.

Let

$$L_{111} = \Big\{\langle M\rangle \;\Big|\; M \text{ prints three one's in a row on blank input}\Big\}.$$

Lemma 27.4.3 *The language L_{111} is undecidable.*

Proof: Suppose that L_{111} were decidable. Let R be a Turing machine deciding L_{111}. We will now construct a Turing machine S that decides A_{TM}.

The decider S for A_{TM} is constructed as follows:

- Input is $\langle M, w\rangle$, where M is the code for a Turing Machine and w is a string.
- Construct the code for a new Turing machine M', which is just like M except that
  - every use of the character `1` is replaced by a new character $\mathtt{1}'$ which M does not use.
  - when M would accept, M' first prints `111` and then accepts
- Similarly, create a string w' in which every character `1` has been replaced by $\mathtt{1}'$.
- Create a second new Turing machine M'_w which simulates M' on the hard-coded string w'.
- Run R on $\langle M'_w\rangle$. If R accepts, accept. If R rejects, then reject.

If M accepts w, then M'_w will print `111` on any input (and thus on a blank input). If M does not accept w, then M'_w is guaranteed never to print `111` accidently. So R will accept $\langle M'_w\rangle$ exactly when M accepts w. Therefore, S decides A_{TM}.

But we know that A_{TM} is undecidable. So S can not exist. Therefore we have a contradiction. So L_{111} must have been undecidable. ■

Chapter 28

Lecture 24: Dovetailing and non-deterministic Turing machines

23 April 2009

This lecture covers dovetailing, a method for running a gradually expanding set of simulations in parallel. We use it to demonstrate that non-deterministic TMs can be simulated by deterministic TMs.

28.1 Dovetailing

28.1.1 Interleaving

We have seen that you can run two Turing machines in parallel, to compute some function of their outputs, e.g. recognize the union of their languages.

Suppose that we had Turing machines $M_1, \ldots, M_k$ recognizing languages $L_1, \ldots, L_k$, respectively. Then, we can build a TM M which recognizes $\bigcup_{i=1}^{k} L_i$. To do this, we assume that M has k simulation tapes, plus input and working tapes. The TM M cycles through the k simulations in turn, advancing each one by a single step. If any of the simulations halts and accepts, then M halts and accepts.

We could use this same method to run a single TM M on a set of k input strings $w_1, \ldots, w_k$; that is, accept the input list if M accepts any of the strings $w_1, \ldots, w_k$.

The limitation of this approach is that the number of tapes is finite and fixed for any particular Turing machine.

28.1.2 Interleaving on one tape

Suppose that TM M recognizes language L and consider the language

$$\widehat{L} = \left\{ w_1 \# w_2 \# \ldots \# w_k \;\middle|\; M \text{ accepts } w_k \text{ for some } k \right\}.$$

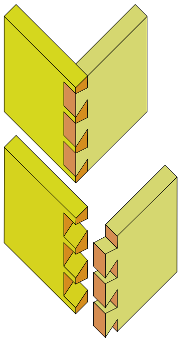
The language $\widehat{L}$ is recognizable, but we have to be careful how we construct its recognizer $\widehat{M}$. Because M is not necessarily a decider, we can not process the input strings one after another, because one of them might get stuck in an infinite loop. Instead, we need to run all k simulations in parallel. But k is different for different inputs to $\widehat{M}$, so we can not just give k tapes to $\widehat{M}$.

Instead, we can store all the simulations on a single tape ✇$_T$. Divide up

✇$_T$

into k sections, one for each simulation. If a simulation runs out of space in its section, push everything over to the right to make more room.

28.1.3 Dovetailing



Dovetailing (in carpentry) is a way of connecting two pieces of wood by interleaving them, see picture on the right.

Dovetailing is an interleaving technique for simulating many (in fact, infinite number of) TM together. Here, we would like to interleave an infinite number of simulations, so that if any of them stops, our simulation of all of them would also stop.

Consider the language:

$$J = \left\{ \langle M \rangle \;\middle|\; M \text{ accepts at least one string in } \Sigma^* \right\}.$$

It is tempting to design our recognizer for J as follows.

```
algBuggyRecog ($\langle M \rangle$)
    $x = \epsilon$
    while True do
        simulated $M$ on $x$ (using $U_{\mathsf{TM}}$)
        if $M$ accepts then
            halt and accept
        $x \leftarrow$ next string in lexicographic order
```

Unfortunately, if M never halts on one of the strings, this process will get stuck before it even reaches the string that M does accept. So we need to run our simulations in parallel. Since we can not start up an infinite number of simulations all at once, we use the following idea.

Dovetailing is the idea of running k simulations in parallel, but keep dynamically increasing k. So, for our example, suppose that we store all our simulations on tape ✇$_T$ and x lives on some other tape. Then our code might look like:

```
algDovetailingRecog ($\langle M \rangle$)
    $x = \epsilon$
    while True do
        On ✇$_T$, start up the simulation of $M$ on $x$
        Advance all simulations on ✇$_T$ by one step.
        if any simulation on ✇$_T$ accepted then
            halt and accept
        $x \leftarrow$ Next($x$)
```

Here **Next**(x) yields the next string in the lexicographic ordering.

In each iteration through the loop, we only start up one new simulation. So, at any time, we are only running a finite number of simulations. However, the set of simulations keeps expanding. So, for any string $w \in \Sigma^*$, we will eventually start up a simulation on w.

Increasing resource bounds

The effect of dovetailing can also be achieved by running simulations with a resource bound and gradually increasing it. For example, the following code can also recognize J.

```
Increasing resource bound

for $i = 0, 1, \ldots$

  (1) Generate the first $i$ strings (in lexicographic order) in $\Sigma^*$
  (2) On tape $T$, start up simulations of $M$ on these $i$ input strings
```

```
(3) Run the set of simulations for $i$ steps.
(4) If any simulation has accepted, halt and accept
(5) Otherwise increment $i$ and repeat the loop
```

Each iteration of the loop does only a finite amount of work: i steps for each of i simulations. However, because i increases without bound, the loop will eventually consider every string in Σ^* and will run each simulation for more and more steps. So if there is some string w which is accepted by M, our procedure will eventually simulate M on w for enough steps to see it halt.

28.2 Nondeterministic Turing machines

A ***non-deterministic Turing machine*** (NTM) is just like a normal TM, except that the transition function generates a set of possible moves (not just a single one) for a given state and character being read. That is, the output of the transition function is a set of triples (r, c, D) where r is the new state, c is the character to write onto the tape, and D is either L or R. That is

$$\delta(q, \mathtt{c}) = \left\{ (r, \mathtt{d}, D) \;\middle|\; \text{for some } r \in Q, \mathtt{d} \in \Gamma \text{ and } D \in \{\mathtt{L}, \mathtt{R}\} \right\}.$$

An NTM M accepts an input w if there is some possible run of M on w which reaches the accept state. Otherwise, M does not accept w.

This works just like non-determinism for the simpler automata. That is, you can either imagine searching through all possible runs, or you can imagine that the NTM magically makes the right guess for what option to take in each transition.

For regular languages, the deterministic and non-deteterministic machines do the same thing. For context-free languages, they do different things. We claim that non-deterministic Turing machines can recognize the same languages as ordinary Turing machines.

28.2.1 NTMs recognize the same languages

Theorem 28.2.1 *NTMs recognize the same languages as normal TMs.*

Suppose that a language L is recognized by an NTM M. We need to construct a deterministic TM that recognizes L. We can do this using dovetailing to search all possible choices that the NTM could make for its moves.

Our simulation will use two simulation tapes S and T in a similar way. In this case, flicker isn't a big problem. But the double buffering makes the algorithm slightly easier to understand.

```
simulate NTM $M$ on input $w$
  (1) put the start configuration $q_0 w$ onto tape $S$
  (2) for each configuration $C$ on $S$
        – generate all options $D_1, D_2, \ldots D_k$ for the next configuration
        – if any of $D_1, D_2, \ldots D_k$ is an accept configuration, halt and accept
        – otherwise, erase all the $D_i$ that have halted and rejected
        – copy the rest onto the end of $T$
  (3) if tape $T$ is empty, halt and reject
  (4) copy the contents of $T$ to $S$, overwriting what was there
  (5) erase tape $T$.
  (6) goto step 2
```

You can think about the set of possible configurations as a tree. The root is the start configuration. Its children are the configurations that could be reached in one step. Its grandchildren are the configurations that could be reached in two steps. And so forth. Our simulator is then doing a breadth-first search of this tree of possibilities, looking for a branch that ends in acceptance.

28.2.2 Halting and deciders

Like a regular TM, an NTM can cleanly reject an input string w or it can implicitly reject it by never halting. An NTM halts if all possible runs eventually halt. Once it halts, the NTM accepts the input if some run ended in the accept state, and rejects the input if all runs ended in the reject state.

A NTM is a ***decider*** if it halts on all possible inputs on all branches. Formally, if you think about all possible configurations that a TM might generate for a specific input as a tree (i.e., a branch represents a non-deterministic choice) then an NTM is a decider if and only if this tree is finite, for all inputs.

The simulation we used above has the property that the simulation halts exactly if the NTM would have halted. So we have also shown that

Theorem 28.2.2 *NTMs decide the same languages as normal TMs.*

28.2.3 Enumerators

A language can be ***enumerated***, if there exists a TM with an output tape (in addition to its working tape), that the TM prints out on this tape all the words in the language (assume that between two printed words we place a special separator character). Note, that the output tape is a write only tape.

Definition 28.2.3 (Lexicographical ordering.) For two strings s_1 and s_2, we have that $s_1 < s_2$ in lexicographical ordering if $|s_1| < |s_2|$ or $|s_1| = |s_2|$ then s_1 appears before s_2 in the dictionary ordering.

(That is, lexicographical ordering is a dictionary ordering for strings of the same length, and shortest strings appear before longer strings.)

Claim 28.2.4 *A language* L *is TM recognizable iff it can be enumerated.*

Proof: Let T the TM recognizer for L, and we need to build an enumerator for this language. Using dovetailing, we "run" T on all the strings in $\Sigma^* = \{w_1, w_2, \ldots\}$ (say in lexicographical ordering). Whenever one of this executions stops and accepts on a string w_i, we print this string w_i to the enumerator output string. Clearly, all the words of L would be sooner or later printed by this enumerated. As such, this language can be enumerated.

As for the other direction, assume that we are given an enumerate $\mathtt{T}_{\text{enum}}$ for L. Given a word $x \in \Sigma^*$, we can recognize if it is in L, by running the enumerator and reading the strings it prints out one by one. If one of these strings is x, then we stop and accept. Otherwise, this TM would continue running. Clearly, if $x \in$ L sooner or later the enumerator would output x and our TM would stop and accept it. ■

Claim 28.2.5 *A language* L *is decidable iff it can be enumerated in lexicographic order.*

Proof: If L is finite the claim trivially hold, so we assume that L is infinite.

If L is decidable, then there is a decider **deciderForL** for it. Generates the words of Σ^* in lexicographic ordering, as $w_1, w_2, \ldots$. In the ith stage, check if $w_i \in$ L by calling **deciderForL** on w_i. If **deciderForL** accepts w_i then we print it to the output tape. Clearly, this procedure never get stuck since **deciderForL** always stop. More importantly, the output is sorted in lexicographical ordering. Note, that if $x \in$ L, then there exists an i such that $w_i = x$. Thus, in the ith stage, the program would output x. Thus, **deciderForL** indeed prints all the words in L.

Similarly, assume we are given a lexicographical enumerator $\mathtt{T}_{\text{enum}}$ for L. Consider a word $x \in \Sigma^*$. Clearly, the number of words in Σ^* that appear before x in the lexicographical ordering is finite. As such, if x is the ith word in Σ^* in the lexicographical ordering, then if $\mathtt{T}_{\text{enum}}$ outputs i words and none of them is x, then we know it would never x, and as such x is not in the language. As such, we stop and reject. Similarly, if x is output in the first i words of the output of $\mathtt{T}_{\text{enum}}$ then we stop and accept. Clearly, since $\mathtt{T}_{\text{enum}}$ enumerates an infinite language, it continuously spits out new strings. As such, sooner or later it would output i words of L, and at this point our procedure would stop. Namely, this procedures accepts the language L, and it always stop; namely, it is a decider for L. ■

Chapter 29

Lecture 25: Linear Bounded Automata and Undecidability for CFGs

28 April 2009

> "It is a damn poor mind indeed which can't think of at least two ways to spell any word."
> – Andrew Jackson

This lecture covers Linear Bounded Automata, an interesting compromise in power between Turing machines and the simpler automatas (DFAs, NFAs, PDAs). We will use LBAs to show two CFG grammar problems (equality and generating all strings) are undecidable.

In some of the descriptions we uses PDAs. However, the last part of this notes show how these PDAs can be avoided, resulting in arguably a simpler and slightly more elegant argument.

29.1 Linear bounded automatas

A ***linear bounded automata*** (***LBA***) is a TM whose head never moves off the portion of the tape occupied by the initial input string.

That is, an LBA is a TM that uses only the tape space occupied by the input.

An equivalent definition of an LBA is that it uses only k times the amount of space occupied by the input string, where k is a constant fixed for the particular machine. To simulate k tape cells with a single tape cell, increase the size of the tape alphabet Γ. E.g. the new tape alphabet has symbols that are k-tuples of the symbols from the old alphabet.

A lot of interesting algorithms are LBAs, because they use only space proportional to the length of the input. (Naturally, you need to pick the variants of the algorithms that use space efficiently.) Examples include A_{DFA}, A_{CFG}, $\mathsf{E}_{\mathsf{DFA}}$, $\mathsf{E}_{\mathsf{CFG}}$, $s-t$ graph reachability, and many others.

When an LBA runs, a transition off the righthand edge of the input area cause the input to be rejected. Or maybe the read head simply sticks on the rightmost input position. You can define them either way and it will not matter for what we are doing here.

29.1.1 LBA halting is decidable

Suppose that a given LBA `T` (which is a TM, naturally) has

- q states,
- k characters in the tape alphabet Γ (we remind the reader that the input alphabet $\Sigma \subseteq \Gamma$ and also the special blank character $\sqcup$),
- and the input length is n.

Then $\mathtt{T}$ can be in at most

$$\alpha(n) = \overbrace{k^n}^{\substack{\text{tape}\\ \text{content}}} \quad * \quad \overbrace{n}^{\substack{\text{head}\\ \text{position}}} \quad * \quad \overbrace{q}^{\substack{\text{controller}\\ \text{state}}} \quad = k^n n q \qquad (29.1)$$

configurations.

Here is the shocker: If an LBA runs more than $\alpha(n)$ steps, then it must be looping. As such, given an LBA $\mathtt{T}$ and a word w (with n characters), we can simulate it for α steps. If it does not terminate by then, then it must be looping, and as such it can never stop on its input. Thus, an LBA that stops on input w must stop in at most $\alpha(|w|)$ steps.

This implies that

$$\mathrm{A}_{\mathsf{LBA}} = \left\{ \langle \mathtt{T}, w \rangle \;\middle|\; \mathtt{T} \text{ is a } \mathsf{LBA} \text{ and } \mathtt{T} \text{ accepts } w \right\}$$

is decidable. Similarly, the language

$$\mathrm{Halt}_{\mathsf{LBA}} = \left\{ \langle \mathtt{T}, w \rangle \;\middle|\; \mathtt{T} \text{ is a } \mathsf{LBA} \text{ and } \mathtt{T} \text{ stops on } w \right\}.$$

is decidable.

We formally prove one of the above claims. The other one follows by a similar argumentation.

Claim 29.1.1 *The language* $\mathrm{A}_{\mathsf{LBA}}$ *is decidable.*

Proof: Indeed, our decider would receive as input $\langle \mathtt{T}, w \rangle$, where $\mathtt{T}$ is an LBA. Let $n = |w|$. By the argumentation above, if $\mathtt{T}$ accepts w, then it does it in at most $\alpha(n)$ steps (see Eq. (29.1)). As such, simulate $\mathtt{T}$ on the input w for $\alpha(n)$ steps, using the universal TM simulator U_{TM}. If the simulation accepts, then accept. If the simulation rejects, then reject. Now, if the simulation did not accept after simulating $\mathtt{T}$ for $\alpha(n)$ steps, then $\mathtt{T}$ is looping forever on w, and so we reject.

Of course, if during the simulation $\mathtt{T}$ decides to move past the end of the input, it's not an LBA, and as such we reject the input.[1] ■

29.1.2 LBAs with empty language are undecidable

In light of the above claim, one might presume that all "natural" languages on LBAs are decidable, but surprisingly, this is not the situation. Indeed, consider the language of all empty LBAs; that is,

$$\mathsf{E}_{\mathsf{LBA}} = \left\{ \langle \mathtt{T} \rangle \;\middle|\; \mathtt{T} \text{ is a } \mathsf{LBA} \text{ and } \mathsf{L}(\mathtt{T}) = \emptyset \right\}.$$

The language $\mathsf{E}_{\mathsf{LBA}}$ is actually undecidable.

A proof via verifying accepting traces

We remind the reader that configuration x of a TM $\mathtt{T}$ ***yields*** the configuration y of tm, if running $\mathtt{T}$ on x for one step results in the configuration y of $\mathtt{T}$. We denote this fact by $x \mapsto y$.

The idea. Assume we are given a general TM $\mathtt{T}$ (which we emphasize is not an LBA) and a word w. We would like to decide if $\mathtt{T}$ accepts w (which is of course undecidable).

If $\mathtt{T}$ does accept w, we can demonstrate that it does by providing a trace of the execution of $\mathtt{T}$ on w. This trace (defined formally below) is just a string. We can easily build a TM that verifies that a supposed trace is legal (e.g. uses the correct transitions for $\mathtt{T}$), and indeed shows that $\mathtt{T}$ accepts w.

Crucially, this trace verification can be done by a TM $\mathtt{Vrf}_{\mathtt{T},w}$ that just uses the space provided by the string itself. That is, the verifier $\mathtt{Vrf}_{\mathtt{T},w}$ is an LBA. The language of $\mathtt{Vrf}_{\mathtt{T},w}$ is empty if $\mathtt{T}$ does not accept w

[1] For the very careful reader, Sipser handles this case slightly differently. His encoding of $\mathtt{T}$ would specify that the machine is supposed to be an LBA. Attempts to move off the input region would cause the read head to stay put.

(because then there is no accepting trace). If T does accept w then the language of $\mathtt{Vrf}_{\mathtt{T},w}$ contains a single word: the trace showing that T accepts w. So, if we have a decider than decides if $\langle \mathtt{Vrf}_{\mathtt{T},w} \rangle \in \mathsf{E}_{\mathsf{LBA}}$, then we can decide if T accepts w.

Observe that we assumed nothing about T or w. The only required property is that $\mathtt{Vrf}_{\mathtt{T},w}$ is a LBA.

The verifier. A ***computation history*** (i.e., trace) for a TM T on input w is a string

$$\#C_1\#C_2\#\ldots\#C_k\#\#,$$

where $C_1, \ldots, C_k$ are configurations of T, such that

(i) $C_1 = q_0 w$ is the initial configuration of T when executed on w, and

(ii) C_i yields C_{i+1} (according to the transitions of T), for $i = 1, \ldots, k-1$.

The pair of sharp signs marks the end of the trace, so the algorithm knows when the trace ends.

Such a trace is an ***accepting trace*** if the configuration C_k is an accepting configuration (i.e., the accept state q_{acc} of T is the state of T encoded in C_k).[2]

Initial checks. So, we are given $\langle \mathtt{T} \rangle$ and w, and we want to build a verifier $\mathtt{Vrf}_{\mathtt{T},w}$ that checks, given a trace t as input, that this trace is indeed an accepting trace for T on w. As a first step, $\mathtt{Vrf}_{\mathtt{T},w}$ will verify that C_1 (the first configuration written in t) is indeed $q_0 w$. Next it needs to verify that C_k (the last configuration in t) is an accepting configuration which is also easy (i.e., just verify that q_{acc} is the state written in it). Finally, the verifier needs to make sure that the ith configuration implies the $(i+1)$th configuration in the trace t, for all i.

Verifying two consecutive configurations. So, consider the ith configuration in t, that is

$$C_i = \alpha \mathtt{a} q \mathtt{b} \beta,$$

where α and β are two strings. Naturally, C_{i+1} is the next configuration in the input trace t. Since $\mathtt{Vrf}_{\mathtt{T},w}$ has the code of T inside it (as a built-in constant), it knows what $\delta_{\mathtt{T}}(q, \mathtt{b})$, the transition function of T, is. Say it knows that $\delta_{\mathtt{T}}(q, \mathtt{b}) = (q', \mathtt{c}, \mathtt{R})$. If our input is a valid trace, then C_{i+1} is supposed to be

$$C_{i+1} = \alpha \mathtt{a} \mathtt{c} q' \beta.$$

To verify that C_i and C_{i+1} do match up in this way, the TM $\mathtt{Vrf}_{\mathtt{T},w}$ goes back and forth on the tape erasing the parts of C_i and C_{i+1} that must be identical. We can not erase these symbols: we will need to keep C_{i+1} around so we can check it against C_{i+2}. So instead we translate each letter a into a special version of this character $\widehat{\mathtt{a}}$.[3]

After we have marked all the identical characters, we've verified this pair of configurations except for the middle two to three letters (depending on whether this was a left or right move). So the tape in this stage looks like

$$\ldots \# \overbrace{\alpha \mathtt{a} q \mathtt{b} \beta}^{C_i} \# \overbrace{\widehat{\alpha} \mathtt{a} \mathtt{c} q' \widehat{\beta}}^{C_{i+1}} \# \ldots$$

We have verified that the prefix of C_i (i.e., α) is equal to $\widehat{\alpha}$, and the suffix of C_i (i.e., β) is equal to the suffix of C_{i+1} (i.e., $\widehat{\beta}$). So only thing that remains to be verified is the middle part, which can be easily done since we know T's transition function.

After that, the verifier removes the hats from the characters in C_{i+1} and moves right to match C_{i+1} against C_{i+2}. If it gets to the end of the trace and all these checks were successful, the verifier $\mathtt{Vrf}_{\mathtt{T},w}$ accepts the input trace t.

[2] It should also be the case that no previous configuration in this trace is either accepting or rejecting. This is implied by the fact that TM's don't have transitions out of the accept and reject states.

[3] We have omitted some details about how to handle moves near the right and left ends of the non-blank tape area. There details are tedious but easy to fill in, and the reader should verify that they know how to fill in the missing details.

Lemma 29.1.2 *Given a (general) TM* $\mathtt{T}$ *and a string* w*, one can build a verifier* $\mathtt{Vrf}_{\mathtt{T},w}$*, such that given an accepting trace* t*, the verifier accepts* t*, and no other string. Note, that* $\mathtt{Vrf}_{\mathtt{T},w}$ *is a decider that always stops.*

Moreover, $\mathtt{Vrf}_{\mathtt{T},w}$ *is a LBA.*

Proof: The details of $\mathtt{Vrf}_{\mathtt{T},w}$ are described above.

It is easy to see that $\mathtt{Vrf}_{\mathtt{T},w}$ never goes on the portion of the tape which are not parts of its original input t. As such, $\mathtt{Vrf}_{\mathtt{T},w}$ is a LBA. ■

Theorem 29.1.3 *The language* $\mathsf{E}_{\mathsf{LBA}}$ *is undecidable.*

Proof: Proof by reduction from A_{TM}. Assume for the sake of contradiction that $\mathsf{E}_{\mathsf{LBA}}$ is decidable, and let $\mathsf{E}_{\mathsf{LBA}}$**−Decider** be the TM that decides it. We will build a decider $\mathtt{decider}_5$-A_{TM} for A_{TM}.

```
decider$_5$-$\mathrm{A}_{\mathsf{TM}}$ $\big(\langle \mathtt{T}, w\rangle\big)$
        Check that $\langle\mathtt{T}\rangle$ is syntactically correct TM code
        Compute $\langle\mathtt{Vrf}_{\mathtt{T},w}\rangle$ from $\langle\mathtt{T},w\rangle$.
        $res \leftarrow$ $\mathsf{E}_{\mathsf{LBA}}$−Decider$\big(\langle\mathtt{Vrf}_{\mathtt{T},w}\rangle\big)$.
        if $res$ == accept then
                reject
        else
                accept
```

Since we can compute $\langle\mathtt{Vrf}_{\mathtt{T},w}\rangle$ from $\langle\mathtt{T},w\rangle$, it follows that this algorithm is a decider. Furthermore, given $\langle\mathtt{T},w\rangle$ such that $\mathtt{T}$ accepts w, then there exists an accepting trace t for $\mathtt{T}$ accepting w, and as such, $L(\mathtt{Vrf}_{\mathtt{T},w}) \neq \emptyset$. As such $\mathsf{E}_{\mathsf{LBA}}$**−Decider**$\big(\langle\mathtt{Vrf}_{\mathtt{T},w}\rangle\big)$ rejects its input, which imply that $\mathtt{decider}_5$-A_{TM} accepts $\langle\mathtt{T},w\rangle$.

Similarly, if $\mathtt{T}$ does not accept w, then $L(\mathtt{Vrf}_{\mathtt{T},w}) = \emptyset$. As such, $\mathsf{E}_{\mathsf{LBA}}$**−Decider**$\big(\langle\mathtt{Vrf}_{\mathtt{T},w}\rangle\big)$ accepts its input, which imply that $\mathtt{decider}_5$-A_{TM} rejects $\langle\mathtt{T},w\rangle$.

Thus $\mathtt{decider}_5$-A_{TM} is indeed a decider for A_{TM}, but this is impossible, and we thus conclude that our assumption, that $\mathsf{E}_{\mathsf{LBA}}$ is decidable, was false, implying the claim. ■

A direct proof that $\mathsf{E}_{\mathsf{LBA}}$ is undecidable

We provide a direct proof of Theorem 29.1.3 because it is shorter and simpler. The benefit of the previous proof is that it introduces the idea of verifying accepting traces, which we would revisit shortly.

Alternative direct proof of Theorem 29.1.3: We are given $\langle\mathtt{T},w\rangle$, were $\mathtt{T}$ is a TM and w is an input for it. We will assume that the tape alphabet of $\mathtt{T}$ is Γ, its input alphabet is Σ, and assume that $\mathtt{z}$ and \$ are not in Γ. We build a new machine $\mathtt{Z}_w$ from $\mathtt{T}$ and w that gets as input a word of the form $\mathtt{z}^k\$$. The machine $\mathtt{Z}_w$ first writes w on the input tape, move the head to te beginning of the tape, and then just runs $\mathtt{T}$ on the input, with the modification that the new machine treats $\mathtt{z}$ as a space. However, if the new machine ever reaches the $T\$$ character on the input (in any state), it immediately stops and rejects.

Clearly, $\mathtt{Z}_w$ is an LBA (by definition). Furthermore, if $\mathtt{T}$ accepts w after k steps, then $\mathtt{Z}_w$ would accept the word $w\mathtt{z}^{k+1}\$$. Similarly, if $w\mathtt{z}^j\$$ is accepted by $\mathtt{Z}_w$ then $\mathtt{T}$ would accept w. We thus conclude that $\mathsf{L}(\mathtt{Z}_w)$ is not empty if and only if $w \in \mathsf{L}(\mathtt{T})$.

Going back to the proof, given $\langle\mathtt{T}\rangle$ and w the construction of $\langle\mathtt{Z}_w\rangle$ is easy. As such, assume for the sake of contradiction, that $\mathsf{E}_{\mathsf{LBA}}$ is decidable, and we are given a decider for membership of $\mathsf{E}_{\mathsf{LBA}}$, we can feed it $\langle\mathtt{Z}_w\rangle$, and if this decider accepts (i.e., $\mathsf{L}(\mathtt{Z}_w) = \emptyset$, then we know that $\mathtt{T}$ does not accept w. Similarly, if $\mathtt{Z}_w$ is being rejected by the decider, then $\mathsf{L}(\mathtt{Z}_w) \neq \emptyset$, which implies that $\mathtt{T}$ accepts w. Namely, we just constructed a decider for A_{TM}, which is undecidable. A contradiction. ■

29.2 On undecidable problems for context free grammars

We would like to prove that some languages involved with context-free grammars are undecidable. To this end, to reduce A_{TM} to a question involving CFGs, we somehow need to map properties of TMs to CFGs.

29.2.1 TM consecutive configuration pairs is a CFG

Lemma 29.2.1 *Given a TM* $\mathtt{T}$*, the language*

$$\mathsf{L}_{\mathtt{T}:x\mapsto y} = \left\{ x\#y^R \;\middle|\; x, y \textit{ are valid configurations of } \mathtt{T} \textit{ and } x \textit{ yields } y \right\}$$

is a CFG.

Proof: Let Γ be the tape alphabet of $\mathtt{T}$, and Q be the set of states of $\mathtt{T}$. Let δ be the transition function of $\mathtt{T}$. We have the following rewriting rules depending on δ:

$$\begin{array}{l}
\forall \alpha, \beta \in \Gamma^* \;\; \forall \mathtt{b}, \mathtt{c}, \mathtt{d} \in \Gamma \;\; \forall q \in Q \\
\qquad \text{if } \; \delta(q, \mathtt{c}) = (q', \mathtt{d}, \mathtt{R}) \;\; \text{ then } \;\; \alpha q \mathtt{c} \beta \mapsto \alpha \mathtt{d} q' \beta \;\; \equiv \;\; \alpha q \mathtt{c} \beta \mapsto \left(\beta^R q' \mathtt{d} \alpha^R\right)^R \\
\qquad \text{if } \; \delta(q, \mathtt{c}) = (q', \mathtt{d}, \mathtt{L}) \;\; \text{ then } \;\; \alpha \mathtt{b} q \mathtt{c} \beta \mapsto \alpha q' \mathtt{b} \mathtt{d} \beta. \;\; \equiv \;\; \alpha \mathtt{b} q \mathtt{c} \beta \mapsto \left(\beta^R \mathtt{d} \mathtt{b} q' \alpha^R\right)^R.
\end{array}$$

Intuitively, $x \mapsto y$ is equivalent to saying that the string x can be very locally edited and generate y. In the above, we need to copy the α and β portions, and then do the rewriting which only involves at most 3 letters. As such, the grammar

$$\begin{array}{rcll}
\Longrightarrow \mathsf{S}_1 & \rightarrow & \mathsf{C} & \\
\mathsf{C} & \rightarrow & \mathtt{x}\mathsf{C}\mathtt{x} \;\; \forall \mathtt{x} \in \Gamma & \\
\mathsf{C} & \rightarrow & \mathsf{T} & \\
\mathsf{T} & \rightarrow & q\mathtt{c}\mathsf{Z}q'\mathtt{d} & \forall \mathtt{b}, \mathtt{c}, \mathtt{d} \in \Gamma \; \forall q \in Q \;\; \text{such that } \delta(q, \mathtt{c}) = (q', \mathtt{d}, \mathtt{R}) \\
\mathsf{T} & \rightarrow & \mathtt{b}q\mathtt{c}\mathsf{Z}\mathtt{b}\mathtt{d}q' & \forall \mathtt{b}, \mathtt{c}, \mathtt{d} \in \Gamma \; \forall q \in Q \;\; \text{such that } \delta(q, \mathtt{c}) = (q', \mathtt{d}, \mathtt{L}) \\
\mathsf{Z} & \rightarrow & \mathtt{x}\mathsf{Z}\mathtt{x} \;\; \forall \mathtt{x} \in \Gamma & \\
\mathsf{C} & \rightarrow & \#. &
\end{array}$$

generates $\mathsf{L}_{\mathtt{T}:x\mapsto y}$ as can now be easily verified. ■

Lemma 29.2.2 *Given a TM* $\mathtt{T}$ *and an input string* w*, the language*

$$\mathsf{L}_{\mathtt{T},w,trace} = \left\{ C_1\#C_2^R\#C_3\#C_4^R\#\ldots C_k \;\middle|\; \begin{array}{c} C_1\#C_2\#C_3\#C_4\#\ldots\#C_k \\ \textit{is an accepting trace of } \mathtt{T} \textit{ on } w \end{array} \right\}$$

can be written as the intersection of two context free languages.

Proof: Let L_1 be the regular language $q_0 w \# \Gamma_{\#}^*$ – these are all traces that start with the initial state of $\mathtt{T}$ on w, where q_0 is the initial state of $\mathtt{T}$, and $\Gamma_{\#} = \Gamma \cup \{\#\}$.

Let L_2 be the language of all traces, such that the configuration C_{2i}^R written in the even position $2i$ is implied by the configuration C_{2i-1} written in position $2i-1$, for all $i \geq 1$. Clearly, this is a context free grammar, by just extending the grammar of Lemma 29.2.1.

Using similar argument, L_3 be the language of all traces, such that the configuration C_{2i+1} written in the odd position $2i+1$ are implied by the configuration C_{2i}^R written in position $2i$. Clearly, this is a context free grammar, by just modifying and extending the grammar of Lemma 29.2.1.

Finally, let L_4 be the regular language of all traces, such that the last trace written on them is accepting. That is $\Gamma_{\#}^* \# \Gamma^* q_{\mathrm{acc}} \Gamma^*$, where q_{acc} is the accepting state of $\mathtt{T}$.

Now, L_1 and L_4 are regular, and L_2 and L_3 are context free. Since context free language are closed under intersection with regular languages, it follows that the language $\mathsf{L}' = L_1 \cap L_2 \cap L_4$ is CFL. Now, the required language is clearly $L_1 \cap L_2 \cap L_3 \cap L_4 = \mathsf{L}' \cap L_3$, which is the intersection of two context free languages. ■

Theorem 29.2.3 *The language* $\left\{ \langle \mathcal{G}, \mathcal{G}' \rangle \;\middle|\; \mathsf{L}(\mathcal{G}) \cap \mathsf{L}(\mathcal{G}') \neq \emptyset \right\}$ *is undecidable. Namely, given two context free grammars, there is no decider that can decide if there is a word that they both generates.*

Proof: If this was decidable, then given $\langle \mathtt{T}, w \rangle$, we can decide if the language $\mathsf{L}_{\mathtt{T},w,trace}$ of Lemma 29.2.2 is empty or not, since it is the intersection of two context free grammars that can be computed from $\langle \mathtt{T}, w \rangle$. But if this language is not empty, then $\mathtt{T}$ accepts w. Namely, we got a decider for $\mathsf{A_{TM}}$, which is a contradiction. ■

29.2.2 The language of a context-free grammar generates all strings is undecidable

Consider the language

$$\mathsf{ALL_{CFG}} = \left\{ \langle G \rangle \;\middle|\; G \text{ is a } \mathsf{CFG}\text{, and } L(G) = \Sigma^* \right\}.$$

This language seems like it should be decidable, since $\mathsf{E_{CFG}}$ was decidable. But it is not. It is a fairly delicate matter whether questions about CFGs are decidable or not. The proof technique is similar to what we used for $\mathsf{E_{LBA}}$.

The idea

The idea is that given $\mathtt{T}$ and w to build a verifier to an accepting traces for $\mathtt{T}$ and w. Here the verifier is going to be a CFG. The problem is, if you think about it, is that there is no way that a CFG can verify a trace, as the checks needed to be performed are too complicated to be performed by a CFG.

Luckily, we can generate a CFG $\mathtt{Vrf}\mathcal{G}_{\mathtt{T},w}$ that would accept all the traces that are ***not*** accepting traces for $\mathtt{T}$ on w. Indeed, we will build several CFGs, each one "checking" one condition, and their union would be the required grammar. As such, $L(\mathtt{Vrf}\mathcal{G}_{\mathtt{T},w})$ is the set of all strings, if and only if, $\mathtt{T}$ does not have an accepting trace for w.

The alphabet of our grammar is going to be

$$\Sigma = \Gamma \cup Q \cup \{\#\},$$

where Γ is the tape alphabet of $\mathtt{T}$, Q is the set of states of $\mathtt{T}$, and $\#$ is the special separator character.

(Or, almost. There is a small issue that needs to be fixed, but we will get to that in a second.)

The details of the PDA trace checker

It is easier to understand checking the trace if we build a PDA. We can then transform our PDA into an equivalent grammar.

Checking that a trace $t = \#C_1\#C_2\#\ldots\#C_k\#\#$ is valid, requires checking the following:

(i) t looks syntactically like a trace

(ii) Initial check: $C_1 = q_0 w$.

(iii) Middle check: C_i implies C_{i+1}, for all i.

(iv) Final check: C_k contains q_{acc}.

It is not hard for a PDA to check that syntax and the first and last configurations are OK.

However, we can not check that the middle configurations match, because a PDA can only compare strings that are in reverse order. Furthermore, it is not clear how a PDA can perform this check for more than one pair $C_i\#C_{i+1}$. So, we need to modify the format of our traces, so that every odd-numbered configuration is written backwards. Thus, the trace would be given as

$$t = \#C_1\#C_2^R\#C_3\#\ldots\#C_{k-1}^R\#C_k\#,$$

or, if there are an even number of configurations in the trace, the trace would be written as

$$t = \#C_1\#C_2^R\#C_3\#\ldots\#C_{k-1}\#C_k^R\#.$$

Our basic plan is still valid. Indeed, there will be an accepting trace in this modified format if and only if T accepts w.

Verifying two consecutive configurations. Let us build a pushdown automata that reads two configurations $X\#Y^R\#$ and decides if the configuration X does not imply Y. To make things easier, let us first build a PDA that checks that the configuration X does imply the configuration Y.

The PDA P would scan X and push it as it to the stack. As it reads X and read the state written in X, it can push on the stack how the output configuration should look like (there is a small issue with having to write the state on the stack. This can be easily be done by some a few pushes and pops, but this is tedious but manageable). Thus, by the time we are done reading X (when P encounters $\#$), the stack already contains the implied (reversed) configuration of X, let use denote it by Z^R. Now, P just read the input (Y^R) and matches it to the stack content. It accepts if and only if the configuration X implies the configuration Y.

Interestingly, the PDA P is deterministic, and as such, we can complement it (this is not true of a general PDA because of the nondeterminism). Alternatively, just observe that P has a reject state that is arrived to after the comparison fails. In the complement PDA, we just make this "hell" state into an accept state. Thus, we have a PDA $\overline{P}$ that accepts $X\#Y^R\#$ iff the configuration X does not imply the configuration Y. Now, its easy to modify this PDA so that it accepts the language

$$L_1 = \left\{ \Sigma^*\#X\#Y^R\#\Sigma^* \;\middle|\; \text{Configuration } X \text{ does not imply configuration } Y \right\},$$

which clearly contains only invalid traces. Similarly, we can build a PDA that accepts the language

$$L_2 = \left\{ \Sigma^*\#X^R\#Y\#\Sigma^* \;\middle|\; \text{Configuration } X \text{ does not imply configuration } Y \right\},$$

Putting these two PDAs together, yield a PDA that accepts all strings containing two consecutive configurations, such that the first one does not imply the second one.

Now, since we a PDA for this language, we clearly can build a CFG G_M that accepts all such strings.

Strings with invalid initial configurations. Consider all traces having invalid initial configurations. Clearly, they are generated by strings of the form

$$(\Sigma \setminus \{\#, q_0\})^* \, \#\Sigma^*.$$

Clearly, one can generate a grammar G_I that accepts these strings.

Strings with invalid final configurations. Consider all traces having invalid initial configurations. Clearly, they are generated by strings of the form

$$\Sigma^*\#(\Sigma \setminus \{\#, q_{\mathrm{acc}}\})^* .$$

Clearly, one can generate a grammar G_F that accepts these strings.

Putting things together. Clearly, all invalid (i.e., non-accepting) traces of T on w are generated by the grammars G_I, G_M, G_F. Thus, consider the context free grammar $G_{M,w}$ formed by the union of G_I, G_M, G_F.

When T does not accept w, there is no accepting trace for T on w, so $L(G_{M,w})$ (the strings that are not accepting traces) is Σ^*. When T accepts w, there is an accepting trace for T on w, so $L(G_{M,w})$ (the strings that are not accepting traces) is not equal to Σ^*.

The reduction proof

Theorem 29.2.4 *The language*

$$\mathsf{ALL}_{\mathsf{CFG}} = \left\{ \langle G \rangle \;\middle|\; G \text{ is a CFG, and } L(G) = \Sigma^* \right\}$$

is undecidable.

Proof: Let us assume, for the sake of contradiction, that the language $\mathsf{ALL}_{\mathsf{CFG}}$ is decidable, and let **deciderAll**$_{\mathsf{CFG}}$ be its decider. We will now reduce A_{TM} to it, by building a decider for it as follows.

```
decider_6-A_TM $\Big(\langle M, w\rangle\Big)$
    Check that $\langle M\rangle$ is syntactically correct TM code
    Compute $\langle G_{M,w}\rangle$ from $\langle M, w\rangle$, as described above.
    $res \leftarrow$ deciderAll_CFG$\Big(\langle G_{M,w}\rangle\Big)$.
    if $res ==$ accept then
        reject
    else
        accept
```

Clearly, this is a decider, and indeed if `T` accepts w, then there exists an accepting trace t showing it. As such, $L(G_{M,w}) = \Sigma^* \setminus \{t\} \neq \Sigma^*$. Thus, **deciderAll**$_{\mathsf{CFG}}$ rejects $\langle G_{M,w}\rangle$, and thus `decider`$_6$-A_{TM} accepts $\langle M, w\rangle$.

Similarly, if `T` rejects w then $L(G_{M,w}) = \Sigma^*$, and as such **deciderAll**$_{\mathsf{CFG}}$ accepts $\langle G_{M,w}\rangle$. Implying that `decider`$_6$-A_{TM} rejects $\langle M, w\rangle$.

Thus, `decider`$_6$-A_{TM} is a decider for A_{TM}, which is impossible. We conclude that our assumption, that $\mathsf{ALL}_{\mathsf{CFG}}$ is decidable, is false, implying the claim. ■

Now, suppose that $\mathsf{ALL}_{\mathsf{CFG}}$ is decided by R. We construct a decider for A_{TM} as follows:

29.2.3 CFG equivalence is undecidable

From the undecidability of $\mathsf{ALL}_{\mathsf{CFG}}$, we can quickly deduce that

$$EQ_{\mathsf{CFG}} = \left\{ \langle G, H\rangle \;\middle|\; G \text{ and } H \text{ are } \mathsf{CFG}\text{s and } L(G) = L(H) \right\}$$

is undecidable. This proof is almost identical to the reduction of E_{TM} to EQ_{TM} that we saw in lecture 21.

Theorem 29.2.5 *The language EQ_{CFG} is undecidable.*

Proof: Proof by contradiction. Suppose that EQ_{CFG} is decidable and let **deciderEq**$_{\mathsf{CFG}}$ be a TM that decides it.

Given an alphabet Σ, it is not hard to construct a grammar F_Σ that generates all strings in Σ^*. E.g. if $\Sigma = \{c_1, c_2, \ldots c_k\}$, then we could use rules:

$$S \rightarrow XS \mid \epsilon$$

$$X \rightarrow c_1 \mid c_2 \mid \ldots \mid c_k$$

As such, here is a TM **deciderAll**$_{\mathsf{CFG}}$ that decides $\mathsf{ALL}_{\mathsf{CFG}}$.

```
deciderAll_CFG $\Big(\langle G\rangle\Big)$
    $\Sigma \leftarrow$ alphabet used by $\langle G\rangle$
    Compute $F_\Sigma$ from $\Sigma$.
    $res \leftarrow$ deciderEq_CFG$(\langle G, F_\Sigma\rangle)$
    return $res$
```

It is easy to verify that **deciderAll**$_{\mathsf{CFG}}$ is indeed a decider. However, we have already shown that $\mathsf{ALL}_{\mathsf{CFG}}$ is undecidable. So this decider **deciderAll**$_{\mathsf{CFG}}$ can not exist. As such, our assumption that EQ_{CFG} is decidable is false. As such, EQ_{CFG} is undecidable decidable. ■

29.3 Avoiding PDAs

The proofs we used above are simpler when one uses PDAs. The same argumentation can be done without PDAs by slightly changing the rules. The basic idea is to interleave two configurations together. This is best imagined by thinking about each character as being a tile of two characters. Thus, the following

b	d	x	a	b	q	c	e	b	d	x
b	d	x	a	q'	b	d	e	b	d	x

describes the two configurations $x = \mathtt{bdxab}q\mathtt{cebdx}$ and $y = \mathtt{bdxa}q'\mathtt{bdebdx}$. If we are given a TM $\mathtt{T}$ with tape alphabet Γ and set of states Q, then the alphabet of the tiles is

$$\widehat{\Sigma} = \left\{ \begin{array}{|c|}\hline \mathtt{x} \\ \hline \mathtt{y} \\ \hline \end{array} \,\middle|\, x, y \in \Gamma \cup Q \right\}.$$

Note, that in the above example x yields y, which implies that except for a region of three columns the two strings are identical, see

b	d	x	a	b	q	c	e	b	d	x
b	d	x	a	q'	b	d	e	b	d	x

.

Thus, a single step of a TM is no more than a local rewrite of the configuration string.

Given two configurations x, y of $\mathtt{T}$, we will refer to the string resulting from writing them together interleaved over $\widehat{\Sigma}$ as described above as ***pairing***, denoted by $\begin{array}{c}\boxed{\mathtt{x}} \\ \boxed{\mathtt{y}}\end{array}$. Note, that if one of the configurations is shorter than the other, we will pad the other configuration by introducing blanks characters (i.e., ␣) so that they are of the same length.

Lemma 29.3.1 *Given a TM* $\mathtt{T}$*, one can construct an NFA* $\mathtt{D}$*, such that* $\mathtt{D}$ *accepts a pairing* $\begin{array}{c}\boxed{\mathtt{x}} \\ \boxed{\mathtt{y}}\end{array}$ *if and only if x and y are two valid configurations of* $\mathtt{T}$*, and $x \mapsto y$.*

Proof: First making sure x and y are valid configurations when reading the string $s = \begin{array}{c}\boxed{\mathtt{x}} \\ \boxed{\mathtt{y}}\end{array}$ is easy using a DFA (you verify that x contains only a single state in it, and the rest of the characters of x are from the tape alphabet of x, one also has to do the same check for y). Let refer to the DFAs verifying the x and y parts of s as $\mathtt{D}_x$ and $\mathtt{D}_y$, respectively. Note that $\mathtt{D}_x$ (resp. $\mathtt{D}_y$) reads the string s but ignores the bottom (resp. top) part of each character of s.

As such, we just need to verify that x yields y. To this end, observe that x yields y if and only if they are identical except for three positions where the transitions happens. We build a NFA that verify that the top and bottom parts are equal, till it guess that it needs to rewrite this 3 tile region. It then guesses what is the tile that needs to be written (note, that the transition function of $\mathtt{T}$ specify all valid such tiles), it verifies that indeed thats what in the next three characters of the input, and then it compares the rest of the input. Let this NFA be $\mathtt{D}_=$.

Now, we construct a DFA that accepts the language of $\mathsf{L}(\mathtt{D}_x) \cap \mathsf{L}(\mathtt{D}_y) \cap \mathsf{L}(\mathtt{D}_=)$. Clearly, this DFA accepts the required language. ■

Similarly, it is easy to build a DFA that verifies that the pairing $\begin{array}{c}\boxed{\mathtt{x}^{\mathtt{R}}} \\ \boxed{\mathtt{y}^{\mathtt{R}}}\end{array}$ is valid and x yields y (according to $\mathtt{T}$). Now, consider an extended execution trace

$$\begin{array}{c}\boxed{\mathtt{C}_1} \\ \boxed{\mathtt{C}_2}\end{array} \begin{array}{c}\boxed{\$} \\ \boxed{\$}\end{array} \begin{array}{c}\boxed{\mathtt{C}_2^{\mathtt{R}}} \\ \boxed{\mathtt{C}_3^{\mathtt{R}}}\end{array} \begin{array}{c}\boxed{\$} \\ \boxed{\$}\end{array} \begin{array}{c}\boxed{\mathtt{C}_3} \\ \boxed{\mathtt{C}_4}\end{array} \begin{array}{c}\boxed{\$} \\ \boxed{\$}\end{array} \cdots \begin{array}{c}\boxed{\$} \\ \boxed{\$}\end{array} \begin{array}{c}\boxed{\mathtt{C}_{\mathtt{k}-2}^{\mathtt{R}}} \\ \boxed{\mathtt{C}_{\mathtt{k}-1}^{\mathtt{R}}}\end{array} \begin{array}{c}\boxed{\$} \\ \boxed{\$}\end{array} \begin{array}{c}\boxed{\mathtt{C}_{\mathtt{k}-1}} \\ \boxed{\mathtt{C}_{\mathtt{k}}}\end{array}.$$

We would like to verify that this encodes a valid accepting trace for $\mathtt{T}$ on the input string w. This would require verifying that following conditions are met.

(i) The trace has the right format of pairings separated with dollar tiles. Can be easily be done by a DFA.

Let L_1 be the language that this DFA accepts.

(ii) Check that $C_1 = q_0 w$ - can be done with a DFA.

Let L_2 be the language that this DFA accepts.

(iii) The last configuration C_k is an accepting configuration. Easily can be done by a DFA.

Let L_3 be the language that this DFA accepts.

(iv) The pairs $\left\|\begin{array}{|c|}\hline \mathtt{C_{2i-1}}\\\hline \mathtt{C_{2i}}\\\hline\end{array}\right\|$ and $\left\|\begin{array}{|c|}\hline \mathtt{C^R_{2i}}\\\hline \mathtt{C^R_{2i+1}}\\\hline\end{array}\right\|$ are valid pairing such that $C_{2i} \mapsto C_{2i+1}$ and $C_{2i-1} \mapsto C_{2i}$, for all i (again, according to TM. This can be done by a DFA, by Lemma 29.3.1.

Let L_4 be the language that this DFA accepts.

(v) Finally, we need to verify that the configurations are copied correctly from the bottom of one tile to the top of the next tile.

Let L_5 be the language of all string that their copying is valid.

Clearly, the set of all valid traces of $\mathtt{T}$ on w is the set $\mathsf{L} = \mathsf{L}_1 \cap \mathsf{L}_2 \cap \mathsf{L}_3 \cap \mathsf{L}_4 \cap \mathsf{L}_5$.

We are interested in building a CFG that recognized the complement language $\overline{\mathsf{L}}$, which is the language

$$\overline{\mathsf{L}} = \overline{\mathsf{L}_1} \cup \overline{\mathsf{L}_2} \cup \overline{\mathsf{L}_3} \cup \overline{\mathsf{L}_4} \cup \overline{\mathsf{L}_5}.$$

Now, since $\mathsf{L}_1, \ldots, \mathsf{L}_4$ it is easy to build a CFG that accepts $\overline{\mathsf{L}_1}$, $\overline{\mathsf{L}_2}$, $\overline{\mathsf{L}_3}$ and $\overline{\mathsf{L}_4}$, respectively.

The only problematic language is $\overline{\mathsf{L}_5}$ which is just all strings where there is a consecutive pair of configurations such that the copying failed. That is

$$\cdots \begin{array}{|c|}\hline \$\\\hline \$\\\hline\end{array} \left\|\begin{array}{|c|}\hline \mathtt{x^R}\\\hline \mathtt{y^R}\\\hline\end{array}\right\| \begin{array}{|c|}\hline \$\\\hline \$\\\hline\end{array} \left\|\begin{array}{|c|}\hline \mathtt{y'}\\\hline \mathtt{z}\\\hline\end{array}\right\| \cdots$$

where $\left(y^R\right)^R \neq y'$. But if we ignore the rest of the string and top and bottom portions of these two pairings, this is just recognized the language "not palindrome", which we know is CFG. Indeed, the grammar of not-palindrome over an alphabet Γ is

$$\begin{array}{rrcll}
\Longrightarrow & \mathsf{S}_2 & \rightarrow & \mathtt{x}\mathsf{S}_2\mathtt{x} & \forall \mathtt{x} \in \Gamma \\
& \mathsf{S}_2 & \rightarrow & \mathtt{x}\mathsf{C}\mathtt{y} & \forall \mathtt{x}, \mathtt{y} \in \Gamma \text{ and } \mathtt{x} \neq \mathtt{y} \\
& \mathsf{C} & \rightarrow & \mathsf{C}\mathtt{x} \mid \mathtt{x}\mathsf{C} & \forall \mathtt{x} \in \Gamma \\
& \mathsf{C} & \rightarrow & \$. &
\end{array}$$

We now extend this grammar for the extended alphabet $\widehat{\Sigma}$ as follows

$$\begin{array}{rrcll}
\Longrightarrow & \mathsf{S}_3 & \rightarrow & \begin{array}{|c|}\hline \mathtt{u}\\\hline \mathtt{x}\\\hline\end{array} \mathsf{S}_3 \begin{array}{|c|}\hline \mathtt{x}\\\hline \mathtt{v}\\\hline\end{array} & \forall u, v, x \in \Gamma_{\mathtt{T}} \\
& \mathsf{S}_2 & \rightarrow & \begin{array}{|c|}\hline \mathtt{u}\\\hline \mathtt{x}\\\hline\end{array} \mathsf{C} \begin{array}{|c|}\hline \mathtt{y}\\\hline \mathtt{v}\\\hline\end{array} & \forall x, y, u, v \in \Gamma_{\mathtt{T}} \text{ and } x \neq y \\
& \mathsf{C} & \rightarrow & \mathsf{C}\begin{array}{|c|}\hline \mathtt{x}\\\hline \mathtt{y}\\\hline\end{array} \;\Big|\; \begin{array}{|c|}\hline \mathtt{x}\\\hline \mathtt{y}\\\hline\end{array}\mathsf{C} & \forall x, y \in \Gamma_{\mathtt{T}} \\
& \mathsf{C} & \rightarrow & \begin{array}{|c|}\hline \$\\\hline \$\\\hline\end{array}, &
\end{array}$$

where $\Gamma_{\mathtt{T}}$ is the tape alphabet of $\mathtt{T}$. Thus, the context-free language

$$\widehat{\Sigma}^* \begin{array}{|c|}\hline \$\\\hline \$\\\hline\end{array} \mathsf{L}(\mathsf{S}_3) \begin{array}{|c|}\hline \$\\\hline \$\\\hline\end{array} \widehat{\Sigma}^*$$

is exactly $\overline{\mathsf{L}_5}$. We conclude that $\overline{\mathsf{L}}$ is a context-free language (being the union of 5 context-free/regular languages). Furthermore, $\overline{\mathsf{L}} = \widehat{\Sigma}^*$ if and only if $\mathtt{T}$ rejects w. We conclude the following.

Theorem 29.3.2 (Restatement of Theorem 29.2.4.) *The language*

$$\mathsf{ALL}_{\mathsf{CFG}} = \left\{ \langle G \rangle \;\middle|\; G \textit{ is a } \mathsf{CFG}\textit{, and } L(G) = \Sigma^* \right\}$$

is undecidable.

Chapter 30

Lecture 26: NP Completeness I

30 April 2008

> "Then you must begin a reading program immediately so that you man understand the crises of our age," Ignatius said solemnly. "Begin with the late Romans, including Boethius, of course. Then you should dip rather extensively into early Medieval. You may skip the Renaissance and the Enlightenment. That is mostly dangerous propaganda. Now, that I think about of it, you had better skip the Romantics and the Victorians, too. For the contemporary period, you should study some selected comic books."
>
> "You're fantastic."
>
> "I recommend Batman especially, for he tends to transcend the abysmal society in which he's found himself. His morality is rather rigid, also. I rather respect Batman."
>
> – A confederacy of Dunces, John Kennedy Toole

30.1 Introduction

The question governing this course, would be the development of efficient algorithms. Hopefully, what is an algorithm is a well understood concept. But what is an efficient algorithm? A natural answer (but not the only one!) is an algorithm that runs quickly.

What do we mean by quickly? Well, we would like our algorithm to:

1. Scale with input size. That is, it should be able to handle large and hopefully huge inputs.

2. Low level implementation details should not matter, since they correspond to small improvements in performance. Since faster CPUs keep appearing it follows that such improvements would (usually) be taken care of by hardware.

3. What we will really care about are asymptotic running time. Explicitly, polynomial time.

In our discussion, we will consider the input size to be n, and we would like to bound the overall running time by a function of n which is asymptotically as small as possible. An algorithm with better asymptotic running time would be considered to be *better*.

Example 30.1.1 It is illuminating to consider a concrete example. So assume we have an algorithm for a problem that needs to perform $c2^n$ operations to handle an input of size n, where c is a small constant (say 10). Let assume that we have a CPU that can do 10^9 operations a second. (A somewhat conservative assumption, as currently [Jan 2006][1], the blue-gene supercomputer can do about $3 \cdot 10^{14}$ floating-point operations a second. Since this super computer has about $131,072$ CPUs, it is not something you would have on your desktop any time soon.) Since $2^{10} \approx 10^3$, you have that our (cheap) computer can solve in (roughly) 10 seconds a problem of size $n = 27$.

[1] But the recently announced Super Computer that would be completed in 2011 in Urbana, is naturally way faster. It supposedly would do 10^{15} operations a second (i.e., petaflop). Blue-gene probably can not sustain its theoretical speed stated above, which is only slightly slower.

But what if we increase the problem size to $n = 54$? This would take our computer about 3 million years to solve. (In fact, it is better to just wait for faster computers to show up, and then try to solve the problem. Although there are good reasons to believe that the exponential growth in computer performance we saw in the last 40 years is about to end. Thus, unless a substantial breakthrough in computing happens, it might be that solving problems of size, say, $n = 100$ for this problem would forever be outside our reach.)

The situation dramatically change if we consider an algorithm with running time $10n^2$. Then, in one second our computer can handle input of size $n = 10^4$. Problem of size $n = 10^8$ can be solved in $10n^2/10^9 = 10^{17-9} = 10^8$ which is about 3 years of computing (but blue-gene might be able to solve it in less than 20 minutes!).

Thus, algorithms that have asymptotically a polynomial running time (i.e., the algorithms running time is bounded by $O(n^c)$ where c is a constant) are able to solve large instances of the input and can solve the problem even if the problem size increases dramatically.

Can we solve all problems in polynomial time? The answer to this question is unfortunately no. There are several synthetic examples of this, but in fact it is believed that a large class of important problems can not be solved in polynomial time.

Problem: Satisfiability

Instance: A boolean formula F with m variables
Question: Is there an assignment of values to variables, such that F evaluates to true?

This problem is usually referred to as SAT.

Currently, all solutions known to SAT require checking all possibilities, requiring (roughly) 2^m time. Which is exponential time and too slow to be useful in solving large instances of the problem.

This leads us to the most important open question in theoretical computer science:

Question 30.1.2 *Can one solve SAT in polynomial time?*

The common belief is that SAT can **NOT** be solved in polynomial time in the size of the formula.

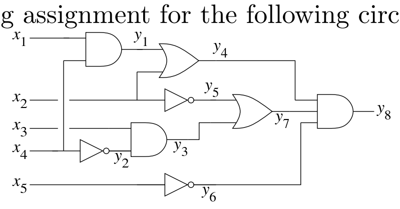
SAT has two interesting properties.

1. Given a supposed positive solution, with a detailed assignment (i.e., proof): $x_1 \leftarrow 0$, $x_2 \leftarrow 1, ..., x_m \leftarrow 1$ one can verify in polynomial time if this assignment really satisfies F. This is done by computing F on the given input.

   Intuitively, this is the difference in hardness between coming up with a proof (hard), and checking that a proof is correct (easy).

2. It is a ***decision problem***. For a specific input an algorithm that solves this problem has to output either TRUE or FALSE.

A teaser. Can one find a satisfying assignment for the following circuit in polynomial time?



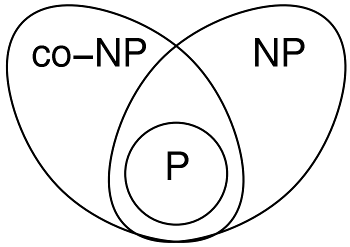
30.2 Complexity classes

Definition 30.2.1 (P: Polynomial time) Let P denote is the class of all decision problems that can be solved in polynomial time in the size of the input.

Definition 30.2.2 (NP: Nondeterministic Polynomial time) Let NP be the class of all decision problems that can be verified in polynomial time. Namely, for an input of size n, if the solution to the given instance is true, one (i.e., an oracle) can provide you with a proof (of polynomial length!) that the answer is indeed TRUE for this instance. Furthermore, you can verify this proof in polynomial time in the length of the proof.

Clearly, if a decision problem can be solved in polynomial time, then it can be verified in polynomial time. Thus, P ⊆ NP.

Remark 30.2.3 The notation NP stands for Non-deterministic Polynomial. The name come from a formal definition of this class using Turing machines where the machines first guesses (i.e., the non-deterministic stage) the proof that the instance is TRUE, and then the algorithm verifies the proof.



The relation between the different complexity classes P, NP, $co\!-\!NP$.

Definition 30.2.4 (co-NP) The class co-NP is the opposite of NP – if the answer is FALSE, then there exists a short proof for this negative answer, and this proof can be verified in polynomial time.

See Figure 30.2 for the currently *believed* relationship between these classes (of course, as mentioned above, P ⊆ NP and P ⊆ co-NP is easy to verify). Note, that it is quite possible that P = NP = co-NP, although this would be extremely surprising.

Definition 30.2.5 A problem Π is NP-Hard, if being able to solve Π in polynomial time implies that P = NP.

Question 30.2.6 *Are there any problems which are* NP-Hard*?*

Intuitively, being NP-Hard implies that a problem is ridiculously hard. Conceptually, it would imply that proving and verifying are equally hard - which nobody that did 473g believes is true.

In particular, a problem which is NP-Hard is at least as hard as ALL the problems in NP, as such it is safe to assume, based on overwhelming evidence that it can not be solved in polynomial time.

Theorem 30.2.7 (Cook-Levin Theorem) *SAT is* NP-Hard*.*

Definition 30.2.8 A problem Π is NP-Complete (NPC in short) if it is both NP-Hard and in NP.

Problem: Circuit Satisfiability

Instance: A circuit C with m inputs
Question: Is there an input for C such that C returns true for it.

Clearly, Circuit Satisfiability is NP-Complete, since we can verify a positive solution in polynomial time in the size of the circuit,

By now, thousands of problems have been shown to be NP-Complete. It is extremely unlikely that any of them can be solved in polynomial time.

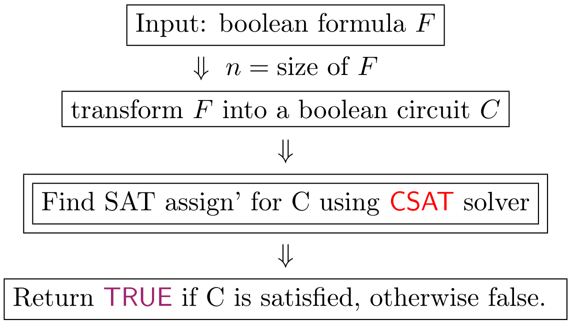

Input: boolean formula F
$\Downarrow$ $n =$ size of F
transform F into a boolean circuit C
$\Downarrow$
Find SAT assign' for C using CSAT solver
$\Downarrow$
Return TRUE if C is satisfied, otherwise false.

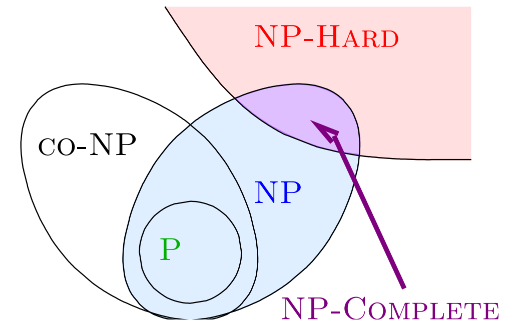
Figure 30.1: An algorithm for solving SAT using an algorithm that solves the CSAT problem

Definition 30.2.9 In the formula satisfiability problem, (a.k.a. SAT) we are given a formula, for example:

$$\left(a \vee b \vee c \vee \overline{d}\right) \Leftrightarrow \left(\left(b \wedge \overline{c}\right) \vee \overline{\left(\overline{a} \Rightarrow d\right)} \vee \left(c \neq a \wedge b\right)\right)$$

and the question is whether we can find an assignment to the variables $a, b, c, \ldots$ such that the formula evaluates to TRUE.

It seems that SAT and Circuit Satisfiability are "similar" and as such both should be NP-Hard.



The relation between the complexity classes.

30.2.1 Reductions

Let A and B be two decision problems.

Given an input I for problem A, a *reduction* is a transformation of the input I into a new input I', such that

$$A(I) \;\; is \;\; \text{TRUE} \qquad \Leftrightarrow \qquad B(I') \;\; is \;\; \text{TRUE}.$$

Thus, one can solve A by first transforming and input I into an input I' of B, and solving $B(I')$.

This idea of using reductions is omnipresent, and used almost in any program you write.

Let $T : I \rightarrow I'$ be the input transformation that maps A into B. How fast is T? Well, for our nafarious purposes we need ***polynomial reductions***; that is, reductions that take polynomial time.

Problem: Circuit Satisfiability

> *Instance:* A circuit C with m inputs
> *Question:* Is there an input for C such that C returns true for it.

For example, given an instance of SAT, we would like to generate an equivalent circuit C. We will explicitly write down what the circuit computes in a formula form. To see how to do this, consider the following example.

The resulting reduction is depicted in Figure 30.1.

Namely, given a solver for CSAT that runs in $T_{\text{CSAT}}(n)$, we can solve the SAT problem in time

$$T_{SAT}(n) \leq O(n) + T_{CSAT}(O(n)),$$

where n is the size of the boolean formula. Namely, if we have polynomial time algorithm that solves CSAT then we can solve SAT in polynomial time.

Another way of looking at it, is that we believe that solving SAT requires exponential time; namely, $T_{\text{SAT}}(n) \geq 2^n$. Which implies by the above reduction that

$$2^n \leq T_{SAT}(n) \leq O(n) + T_{CSAT}(O(n)).$$

Namely, $T_{\mathsf{CSAT}}(n) \geq 2^{n/c} - O(n)$, where c is some positive constant. Namely, if we believe that we need exponential time to solve CSAT then we need exponential time to solve SAT.

This implies that if CSAT $\in$ P then SAT $\in$ P.

We just proved that CSAT is as hard as SAT. Clearly, CSAT $\in$ NP which implies the following theorem.

Theorem 30.2.10 *CSAT (formula satisfiability) is* NP-Complete.

30.3 Other problems that are known to be NP-Complete

We next list (without proof) a bunch of well known NP-Complete problems. Namely, for each one of these problems if you solve them in polynomial time then we can solve all the problems in NP in polynomial time.

Problem: Clique

Instance: A graph G, integer k
Question: Is there a clique in G of size k?

Problem: Independent Set

Instance: A graph G, integer k
Question: Is there an independent set in G of size k?

Problem: Vertex Cover

Instance: A graph G, integer k
Question: Is there a vertex cover in G of size k?

Problem: 3Colorable

Instance: A graph G.
Question: Is there a coloring of G using three colors?

Problem: Hamiltonian Cycle

Instance: A graph G.
Question: Is there a Hamiltonian cycle in G?

Problem: TSP

Instance: $G = (V, E)$ a complete graph - n vertices, $c(e)$: Integer cost function over the edges of G, and k an integer.
Question: Is there a traveling-salesman tour with cost at most k?

Problem: Subset Sum

Instance: S - set of positive integers,t: - an integer number (Target)
Question: Is there a subset $X \subseteq S$ such that $\sum_{x \in X} x = t$?

Problem: Vec Subset Sum

Instance: S - set of n vectors of dimension k, each vector has non-negative numbers for its coordinates, and a target vector $\overrightarrow{t}$.
Question: Is there a subset $X \subseteq S$ such that $\sum_{\overrightarrow{x} \in X} \overrightarrow{x} = \overrightarrow{t}$?

Problem: 3DM

Instance: X, Y, Z sets of n elements, and T a set of triples, such that $(a, b, c) \in T \subseteq X \times Y \times Z$.
Question: Is there a subset $S \subseteq T$ of n disjoint triples, s.t. every element of $X \cup Y \cup Z$ is covered exactly once.?

Problem: Partition

Instance: A set S of n numbers.
Question: Is there a subset $T \subseteq S$ s.t. $\sum_{t \in T} t = \sum_{s \in S \setminus T} s$.?

30.4 Proof of Cook-Levin theorem

FILL IN

Chapter 31

Lecture 27: Post's Correspondence Problem and Tilings

?

This lecture covers Post's Correspondence Problem (section 5.2 in Sipser). Undecidability of this problem implies the undecidability of CFG ambiguity. We will also see how to simulate a TM with 2D tiling patterns and, as a consequence, show how undecidability implies the existence of aperiodic tilings.

31.1 Post's Correspondence Problem

Suppose that we have a set of ***domino tiles***. Each domino piece has a string at the top and a string at the bottom., for example $\begin{array}{|c|}\hline \texttt{abb} \\ \hline \texttt{bc} \\ \hline \end{array}$. So a set S of dominos might look like:

$$S = \left\{ \begin{array}{|c|}\hline \texttt{b} \\ \hline \texttt{ca} \\ \hline \end{array}, \begin{array}{|c|}\hline \texttt{a} \\ \hline \texttt{ab} \\ \hline \end{array}, \begin{array}{|c|}\hline \texttt{ca} \\ \hline \texttt{a} \\ \hline \end{array}, \begin{array}{|c|}\hline \texttt{abc} \\ \hline \texttt{c} \\ \hline \end{array} \right\}.$$

A ***match*** for S is an ordered list of, one or more, dominos from S, such that if you read the symbols on the tops, this makes the same string as reading the symbols on the bottom. You can use the same domino more than once in a match, and you do not have to use all the elements of S. For example, here is a match for our example set

$$\begin{array}{|c|}\hline \texttt{a} \\ \hline \texttt{ab} \\ \hline \end{array} \begin{array}{|c|}\hline \texttt{b} \\ \hline \texttt{ca} \\ \hline \end{array} \begin{array}{|c|}\hline \texttt{ca} \\ \hline \texttt{a} \\ \hline \end{array} \begin{array}{|c|}\hline \texttt{a} \\ \hline \texttt{ab} \\ \hline \end{array} \begin{array}{|c|}\hline \texttt{abc} \\ \hline \texttt{c} \\ \hline \end{array}.$$

The tops and bottoms of the dominos both form the string `abcaaabc`.

Not all sets have a match. For example, T does not have a match because the tops are all longer than the bottoms.

$$T = \left\{ \begin{array}{|c|}\hline \texttt{abc} \\ \hline \texttt{c} \\ \hline \end{array}, \begin{array}{|c|}\hline \texttt{ba} \\ \hline \texttt{a} \\ \hline \end{array}, \begin{array}{|c|}\hline \texttt{bb} \\ \hline \texttt{b} \\ \hline \end{array} \right\}.$$

The set R does not have a match because there are no `d`'s or `f`'s in the top strings.

$$R = \left\{ \begin{array}{|c|}\hline \texttt{ab} \\ \hline \texttt{df} \\ \hline \end{array}, \begin{array}{|c|}\hline \texttt{cb} \\ \hline \texttt{bd} \\ \hline \end{array}, \begin{array}{|c|}\hline \texttt{aa} \\ \hline \texttt{fa} \\ \hline \end{array} \right\}.$$

It seems like it should be fairly easy to figure out whether a set of dominos has a match, but this problem is actually undecidable.

Post's Correspondence Problem (PCP) is the problem of deciding whether a set of dominos has a match or not.

The ***modified Post's Correspondence Problem*** (MPCP) is just like PCP except that we specify both the set of tiles and also a special tile. Matches for MPCP have to start with the special tile.

We will show that PCP undecidable in two steps. First, we will reduce A_{TM} to MPCP. Then we will reduce MPCP to PCP.

31.1.1 Reduction of A_{TM} to MPCP

An informal description of the set of tiles generated and why it works

Given a string $\langle M, w\rangle$, we will generate a set of tiles T such that T has a match, if and only if, M accepts w. In the set T there would a special initial tile that must be used first. The string the match would generate would be an accepting trace of M on w. That is, the top and bottom strings of the tiles of T forming the match will look like

$$\#C_1\#C_2\#\ldots\#C_n\#\#,$$

where the C_i are configurations of the TM M. Here C_1 should be the start configuration, and C_n should be an accept configuration. (We are slightly oversimplifying what we are going to do, but do not worry about this quite yet.)

Here, C_i implies the configuration C_{i+1}. As such, C_i and C_{i+1} are almost identical, except for maybe a chunk of 3 or 4 letters. We will set the initial tile to be

$$\begin{array}{|c|}\hline \# \\ \hline \#\mathtt{q_0w}\# \\ \hline \end{array},$$

where q_0w is the initial configuration of M when executed on w. At this point, the bottom string is way long than the shorter string. As such, to get a match, the tiling would have to copy the content of the bottom row to the top row. We will set up the tiles so that the copying would result in copying also the configuration C_0 to the bottom row, while performing the computation of M on C_0. As such, in the end of this process, the tiling would look like

$$\frac{\#C_0\#}{\#C_0\#C_1\#}.$$

The trick is that again the bottom part is longer, and again to get a match, the only possibility is to copy C_1 to the top part, and in the process writing out C_2 on the bottom part, resulting in

$$\frac{\#C_0\#C_1\#}{\#C_0\#C_1\#C_2\#}.$$

Now, how are we going to do this copying/computation? The idea is to introduce for every character $x \in \Sigma_M \cup \{\#\}$, a copying tile

$$\begin{array}{|c|}\hline \mathtt{x} \\ \hline \mathtt{x} \\ \hline \end{array}.$$

Here Σ_M denotes the alphabet set used by M. Similarly, δ_M denotes the transition function of M.

Next, assume we have the transition $\delta_M(q, \mathtt{a}) = (q', \mathtt{b}, \mathtt{R})$, then we will introduce the computation tile

$$\begin{array}{|c|}\hline \mathtt{qa} \\ \hline \mathtt{bq'} \\ \hline \end{array}.$$

Similarly, for the transition $\delta_M(q_2, \mathtt{c}) = (\widehat{q}, \mathtt{d}, \mathtt{L})$, we will introduce the following computation tiles

$$\forall y \in \Sigma_M \quad \begin{array}{|c|}\hline \mathtt{yq_2c} \\ \hline \widehat{\mathtt{q}}\mathtt{yd} \\ \hline \end{array}..$$

Here is what is going on in the ith stage: The bottom row as an additional configuration C_i written in it. To get a match, the tiling has to copy the configuration to the top row. But the copying tiles, only copy regular characters, and it can not copy states. Thus, when the copying process reaches the state character in C_i, it must use the right computation tile to copy this substring to the top row. Then it continues copying the rest of the configuration to the top. Naturally, as the copying goes on from the bottom row to the top row, new characters are added to the bottom row. The critical observation is that the computation tiles guarantee, that the added string to the bottom row is exactly C_{i+1}, since we copied the characters verbatim in the areas of C_i unrelated to the computation of M on C_i, and the computation tile copied exactly the right string for the small region where the computation changes.

Thus, if the starting configuration before the ith stage was

$$\frac{\#C_0\#C_1\#\ldots C_{i-1}\#}{\#C_0\#C_1\#\ldots\#C_i\#}$$

then after the ith stage, the string generated by the tiling (which is trying so hard to match the bottom and top rows) is

$$\frac{\#C_0\#C_1\#\ldots C_{i-1}\#\ \ C_i\#}{\#C_0\#C_1\#\ldots\#C_i\#\ \ \ C_{i+1}\#}.$$

Let this process continue till we reach the accepting configuration $C_n = \alpha q_{\mathrm{acc}} x\beta$, where α, β are some string and x is some character in Σ_M. Here, the tiling we have so far looks like

$$\frac{\#C_0\#C_1\#\ldots C_{n-1}\#}{\#C_0\#C_1\#C_2\#\ldots\#C_n\#}.$$

The question is how do we make this into a match? The idea is that now, since C_n is an accepting configuration, we should now treat the rest of the tiling as a cleanup stage, and slowly reduct C_n to the empty string, as we copy it up and down. How do we do that? Well, let us introduce a delete tile

$$\begin{array}{|c|}\hline \mathtt{q}_{\mathrm{acc}}\mathtt{x} \\ \hline \mathtt{q}_{\mathrm{acc}} \\ \hline \end{array}$$

into our set of tiles T (also known as a pacman tile). Clearly, if we use the copying tiles to copy $C_n = \alpha q_{\mathrm{acc}} x\beta$ to the top row, and erase in the process the character x, using the above "delete x" tile. We get

$$\frac{\#C_0\#C_1\#\ldots C_{n-1}\#\alpha q_{\mathrm{acc}}x\beta\#}{\#C_0\#C_1\#C_2\#\ldots\#C_{n-1}\#\alpha q_{\mathrm{acc}}x\beta\#\alpha q_{\mathrm{acc}}\beta\#}.$$

We can now repeat this process, by introducing such delete tiles for every character of Σ, and also introducing backward delete tiles like

$$\begin{array}{|c|}\hline \mathtt{x}\mathtt{q}_{\mathrm{acc}} \\ \hline \mathtt{q}_{\mathrm{acc}} \\ \hline \end{array}.$$

Thus, by using these delete tiles, we will get the tiling

$$\frac{\#C_0\#C_1\#\ldots C_{n-1}\#\alpha q_{\mathrm{acc}}x\beta\#\alpha q_{\mathrm{acc}}\beta\#\ldots\#q_{\mathrm{acc}}y\#}{\#C_0\#C_1\#C_2\#\ldots\#C_{n-1}\#\alpha q_{\mathrm{acc}}x\beta\#\alpha q_{\mathrm{acc}}\beta\#\ldots\#q_{\mathrm{acc}}y\#q_{\mathrm{acc}}\#}.$$

To finish of the tiling, we introduce a stopper tile

$$\begin{array}{|c|}\hline \mathtt{q}_{\mathrm{acc}}\#\#\# \\ \hline \#\# \\ \hline \end{array}.$$

Adding it to the tiling results in the required match. This is the tiling

$$\frac{\overbrace{\#C_0\#C_1\#\ldots\#C_{n-1}\#\alpha q_{\mathrm{acc}}x\beta\#}^{\text{the accepting trace}}\ \ \overbrace{\alpha q_{\mathrm{acc}}\beta\#\ldots\#q_{\mathrm{acc}}y\#q_{\mathrm{acc}}\#\#\#}^{\text{winding down the match}}}{\underbrace{\#C_0\#C_1\#\ldots\#C_{n-1}\#\alpha q_{\mathrm{acc}}x\beta\#}_{\text{the accepting trace}}\ \ \underbrace{\alpha q_{\mathrm{acc}}\beta\#\ldots\#q_{\mathrm{acc}}y\#q_{\mathrm{acc}}\#\#\#}_{\text{winding down the match}}}.$$

Its now easy to argue that with this set of tiles, if there is a match it must have the above structure which encodes an accepting trace.

Computing the tiles

Let us recap the above description. We are given a string $\langle M, w\rangle$, and we are generating a set of tiles T, as follows. The set containing the initial tile is

$$T_1 = \left\{ \begin{array}{|c|}\hline \# \\ \hline \#\mathtt{q}_0\mathtt{w}\# \\ \hline \end{array} \right\}. \qquad \text{/* initial tile */}$$

The set of copying tiles is

$$T_2 = \left\{ \begin{array}{|c|}\hline \mathtt{x} \\ \hline \mathtt{x} \\ \hline \end{array} \;\middle|\; x \in \Sigma_M \cup \{\#\} \right\}.$$

The set of computation tiles for right movement of the head is

$$T_3 = \left\{ \begin{array}{|c|}\hline \mathtt{qx} \\ \hline \mathtt{yq}' \\ \hline \end{array} \;\middle|\; \forall q, x, q', y \text{ such that } \delta_M(q, x) = (q', y, \mathtt{R}) \right\}.$$

The set of computation tiles for left movement of the head is

$$T_4 = \left\{ \begin{array}{|c|}\hline \mathtt{zqx} \\ \hline \mathtt{q}'\mathtt{zy} \\ \hline \end{array} \;\middle|\; \forall q, x, q', y, z \text{ such that } \delta_M(q, x) = (q', y, \mathtt{L}) \right\}.$$

The set of pacman tiles (i.e., delete tiles) is

$$T_5 = \left\{ \begin{array}{|c|}\hline \mathtt{q}_{\mathrm{acc}}\mathtt{x} \\ \hline \mathtt{q}_{\mathrm{acc}} \\ \hline \end{array}, \; \begin{array}{|c|}\hline \mathtt{x}\mathtt{q}_{\mathrm{acc}} \\ \hline \mathtt{q}_{\mathrm{acc}} \\ \hline \end{array} \;\middle|\; \forall x \in \Sigma_M \right\}.$$

Finally, the set of containing the stopper tiles is

$$T_6 = \left\{ \begin{array}{|c|}\hline \mathtt{q}_{\mathrm{acc}}\#\#\# \\ \hline \#\# \\ \hline \end{array} \right\}.$$

Let $T = T_1 \cup T_2 \cup T_3 \cup T_4 \cup T_5 \cup T_6$. Clearly, given $\langle M, w\rangle$, we can easily compute the set T. Let $AlgTM2MPCP$ denote the algorithm performing this conversion.

We summarize our work so far.

Lemma 31.1.1 *Given a string $\langle M, w\rangle$, the algorithm $AlgTM2MPCP$ computes a set of tiles T that is an instance of MPCP. Furthermore, T contains a match if and only if M accepts w.*

This implies the following result.

Theorem 31.1.2 *The MPCP problem is undecidable.*

Proof: The reduction is from A_{TM}. Indeed, assume for the sake of contradiction that the MPCP problem is decidable, and we are given a decider **decider_MPCP** for it. Next, we use it to build the following decider for A_{TM}.

```
decider_7-A_TM $\left(\langle M, w\rangle\right)$
        $T \leftarrow$ AlgTM2MPCP $(\langle M, w\rangle)$
        $res \leftarrow$ decider_MPCP$\left(T\right)$.
        return $res$
```

Clearly, this is a decider, and it accepts if and only if M halts on w. But this is a contradiction, since A_{TM} is undecidable.

Thus, our assumption (that MPCP is decidable) is false, implying the claim. ■

31.1.2 Reduction to MPCP to PCP

We are almost done, but not quite. Since our reduction only show that MPCP is undecidable, but we want to show that PCP is decidable. An instance of MPCP is a set of tiles (just like an instance of PCP), with the additional constraint that a specific tile is denoted as an initial tile that **must** be used as the first tile in the matching. As such, to convert this instance T of MPCP into PCP we need to somehow remove the need to specify the initial tile.

To this end, let us introduce a new special $\star$ character into the alphabet used by the tiles. Next, given a string $w = x_1 x_2 \ldots x_m$, let us denote

$$\star w = \star x_1 \star x_2 \ldots \star x_m \qquad \star\, w\star = \star x_1 \star x_2 \ldots \star x_m \star \qquad w\star = x_1 \star x_2 \ldots \star x_m \star .$$

In particular, if

$$T = \left\{ \begin{array}{|c|}\hline \mathtt{t}_1 \\ \hline \mathtt{b}_1 \\ \hline \end{array}, \ldots, \begin{array}{|c|}\hline \mathtt{t}_\mathtt{m} \\ \hline \mathtt{b}_\mathtt{m} \\ \hline \end{array} \right\},$$

where $\begin{array}{|c|}\hline \mathtt{t}_1 \\ \hline \mathtt{b}_1 \\ \hline \end{array}$ is the special initial tile, then the new set of tiles would be

$$X = \left\{ \begin{array}{|c|}\hline \star\mathtt{t}_1 \\ \hline \mathtt{b}_1\star \\ \hline \end{array}, \begin{array}{|c|}\hline \star\mathtt{t}_2 \\ \hline \mathtt{b}_2\star \\ \hline \end{array}, \ldots, \begin{array}{|c|}\hline \star\mathtt{t}_\mathtt{m} \\ \hline \mathtt{b}_\mathtt{m}\star \\ \hline \end{array} \right\} \cup \left\{ \begin{array}{|c|}\hline \star\mathtt{t}_1 \\ \hline \star\mathtt{b}_1\star \\ \hline \end{array} \right\}.$$

Note, that in this new set of tiles, the only tile that can serve as the first tile in the match is

$$\begin{array}{|c|}\hline \star\mathtt{t}_1 \\ \hline \star\mathtt{b}_1\star \\ \hline \end{array},$$

since its the only tile that has in the bottom string a $\star$ as the first character. Now, to take care of the balance of stars in the end of the string, we also add the tile

$$Y = X \cup \left\{ \begin{array}{|c|}\hline \star\star \\ \hline \star \\ \hline \end{array} \right\}$$

Its now easy to verify that if the original instance of T of MPCP had a match, then the set Y also has a match.

The important thing about Y is that it does not need to specify what is the special initial tile (this is a minor difference to T, but a difference nevertheless). As such, U is an instance of PCP. We conclude:

Lemma 31.1.3 *Given a string $\langle M, w\rangle$, the algorithm AlgTM2PCP computes a set of tiles T that is an instance of PCP. Furthermore, T contains a match if and only if M accepts w.*

As before, this implies the described result.

Theorem 31.1.4 *The PCP problem is undecidable.*

31.2 Reduction of PCP to $\mathrm{AMBIG}_{\mathsf{CFG}}$

We can use the PCP result to prove a useful fact about context-free grammars. Let us define

$$\mathrm{AMBIG}_{\mathsf{CFG}} = \left\{ \langle G \rangle \;\middle|\; G \text{ is a } \mathsf{CFG} \text{ and } G \text{ is ambiguous} \right\}.$$

We remind the reader that a grammar G is ambiguous if there are two different ways for G to generate some word w.

We will show this problem is undecidable by a reduction from PCP. That is, given a PCP problem S, we will construct a context-free grammar which is ambiguous exactly when S has a match. This means that any decider for $AMBIG_{\mathsf{CFG}}$ could be used to solve PCP, so such a decider can not exist.

Specifically, suppose that S looks like

$$S = \left\{ \left[\frac{t_1}{b_1}\right], \ldots \left[\frac{t_k}{b_k}\right] \right\}.$$

We could define the corresponding grammar as

$\mathsf{D} \rightarrow \mathsf{T} \mid \mathsf{B}$

$\mathsf{T} \rightarrow t_1\mathsf{T} \mid t_2\mathsf{T} \mid \ldots t_k\mathsf{T} \mid t_1 \mid \ldots \mid t_k$

$\mathsf{B} \rightarrow b_1\mathsf{B} \mid b_2\mathsf{B} \mid \ldots b_k\mathsf{B} \mid b_1 \mid \ldots \mid b_k,$

with D as the initial symbol. This grammar is ambiguous if the tops of a sequence of tiles form the same string as the bottoms of a sequence of tiles. However, there is nothing forcing the two sequences to use the same tiles in the same order.

So, we will add some labels to our rules which name the set of tiles we have used. Let us suppose the tiles are named d_1 through d_k. Then we will make our grammar generate strings like $t_i t_j \ldots t_m d_m \ldots d_j d_i$ where the second part of the string contains the labels of the tiles used to build the first part of the string (in reverse order).

So our final grammar H looks like

$\mathsf{D} \rightarrow T \mid B$

$\mathsf{T} \rightarrow t_1\mathsf{T}d_1 \mid t_2\mathsf{T}d_2 \mid \ldots t_k\mathsf{T}d_k \mid t_1d_1 \mid \ldots \mid t_kd_k$

$\mathsf{B} \rightarrow b_1\mathsf{B}d_1 \mid b_2\mathsf{B}d_2 \mid \ldots b_k\mathsf{B}d_k \mid b_1d_1 \mid \ldots \mid b_kd_k.$

Here VD is the initial symbols. Clearly, there is an ambiguous word $winL(H)$ if and only if the given instance of PCP has a match. Namely, deciding if a grammar is ambiguous is equivalent to deciding an instance of PCP. But since PCP is undecidable, we get that deciding if a CFG grammar is ambiguous is undecidable.

Theorem 31.2.1 *The language* $\mathrm{AMBIG}_{\mathsf{CFG}}$ *is undecidable.*

31.3 2D tilings

Show some of the pretty tiling pictures linked on the 273 lectures web page, walking through the following basic ideas.

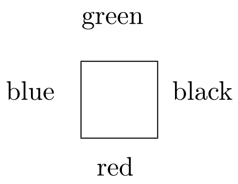
A tiling of the plane is ***periodic*** if it is generated by repeating the contents of some rectangular patch of the plane. Otherwise the tiling is ***aperiodic***.

A set of tiles is ***aperiodic*** if these tiles can tile the whole plane but all tilings generated by these set are aperiodic.

Wang's conjectured, in 1961, that if a set of tiles can cover the plane at all, it can cover the plane periodically.

This is a tempting conjecture, but it is wrong.

In 1966, Robert Berger found a set of 20426 Wang tiles that is aperiodic. A Wang tile is a square tile with colors on its edges. The colors need to match when you put the tiles together.



Various people found smaller and smaller aperiodic sets of Wang tiles. The minimum so far is due to Karel Culik II, and it is made out of 13 tiles.

Other researchers have built aperiodic sets of tiles with pretty shapes, e.g. the Penrose tiles. (Show pretty pictures.)

31.3.1 Tilings and undecidability

Let us restrict our attention to square tiles that are unit squares, as they are easier to understand. Next, consider the problem of deciding whether a set T of such unit square tiles can cover the plane. If there were no aperiodic tile sets, then we could decide this problem as follows

Can T tile the plane?

Enumerate all pairs of integers (m, n). For each pair, do

- Find (by exhaustive search) all ways to cover an m by n box using the tiles in T.
- If there are no ways to tile this box, the plane can not be tiled either. So halt and reject.
- Check whether any of these tilings has matching top and bottom, left and right sides. If so, we can repeat this pattern to tile the whole plane. So halt and accept

However, this procedure will loop forever if the set T has only aperiodic tilings.

In general, we have the following correspondence between 2D tilings and Turing machine behaviors:

tile set	Turing machine
can not cover the plane	halts
has a periodic tiling	loops repeating configurations
has an aperiodic tilings	runs forever without repeating configurations

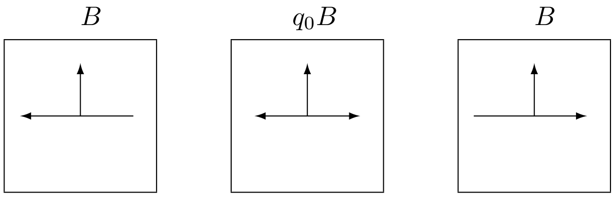
31.4 Simulating a TM with a tiling

In fact, we can reduce A_{TM} to a version of the tiling problem called the ***tiling completion problem***. This problem asks whether, given a set of tiles and a partial tiling of the plane, can we complete this into a tiling of the whole plane.

To decide A_{TM} using a tiling decider, we first hardcode our input string into M, making a TM M_w. We then construct a tiling which covers the whole plane exactly when M_w does not halt on a blank input tape.

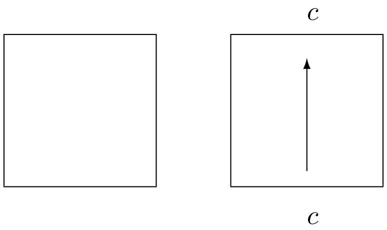
We have three types of tiles: start tiles, copy tiles, and action tiles. These tiles have labels on their edges and also arrows, both of which have to match when the tiles are put together.

Start tiles: Here is the partial tiling we use as input to the problem. B is the blank symbol for M_w and q_0 is its start state.

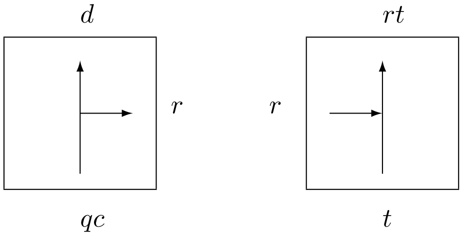


The tile set is engineered so that the rest of this row can only be filled by repeating the left and right tiles from this set.

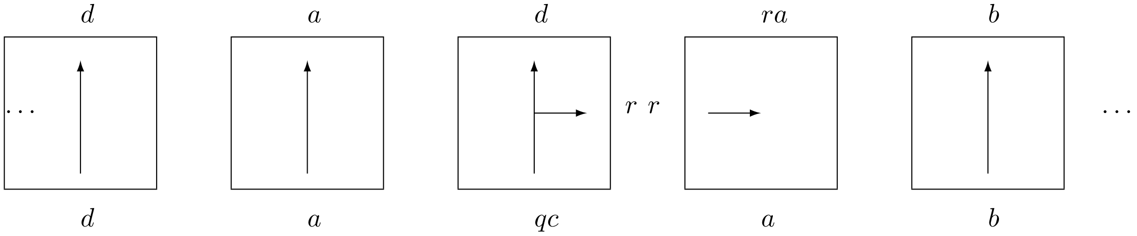
Copy tiles: For every character c in M_w's alphabet, we have a tile that just copies c from one row to the next. We also have an entirely blank tile which must (given the design of this tile set) cover the lower half-plane.



Action tiles: A transition of M_w $\delta(q, c) = (r, d, R)$ is implemented using a "split tile" (left below) and a set of "merge tiles" (right below) for every character t in the tape alphabet.



So a single transition line might look like:



So, this reduction shows that the tiling completion problem is undecidable.

References

Branko Grünbaum and G. C. Shephard (1986) **Tilings and Patterns**, W H Freeman.

Raphael M. Robinson (1971) Undecidability and nonperiodicity for tilings of the plane *Inventiones Mathematicae* 12/3, pp. 177-209.

Chapter 32

Review of topics covered

This review of the class notes was written by Madhusudan Parthasarathy.

32.1 Introduction

The theory of computation is perhaps *the* fundamental theory of computer science. It sets out to define, mathematically, what exactly computation is, what is feasible to solve using a computer, and also what is *not* possible to solve using a computer.

The main objective is to define a computer mathematically, without the reliance on real-world computers, hardware or software, or the plethora of programming languages we have in use today. The notion of a Turing machine serves this purpose and defines what we believe is the crux of all computable functions.

The course is also about weaker forms of computation, concentrating on two classes, regular languages and context-free languages. These two models help understand what we can do with restricted means of computation, and offer a rich theory using which you can hone your mathematical skills in reasoning with simple machines and the languages they define. However, they are not simply there as a weak form of computation— the most attractive aspect of them is that problems formulated on them are *tractable*, i.e. we can build efficient algorithms to reason with objects such as finite automata, context-free grammars and pushdown automata. For example, we can model a piece of hardware (a circuit) as a finite-state system and solve whether the circuit satisfies a property (like whether it performs addition of 16-bit registers correctly). We can model the syntax of a programming language using a grammar, and build algorithms that check if a string parses according to this grammar.

On the other hand, most problems that ask properties about Turing machines are *undecidable*. Undecidability is an important topic in this course. You have seen and proved yourself that several tasks involving Turing machines are unsolvable— i.e. no computer, no software, can solve it. For example, you know now that there is *no software* that can check whether a C-program will halt on a particular input. This is quite amazing, if you think about it. To prove something is possible is, of course, challenging, and you will learn in other courses several ways of showing how something is possible. But to show something is *impossible* is rare in computer science, and you will probably see no other instance of it in any other undergraduate course. To show something is impossible requires an argument quite unlike any other, and you have seen the method of *diagonalization* to prove impossibilities and *reduction* that help you prove infer one impossibility from another. Impossibility results for regular languages and context-free languages are shown using the pumping lemma.

In conclusion, you have formally learnt how to define a computer, and analyze the properties of computable functions, which surely is the theoretical foundation of computer science.

32.2 The players

An alphabet (usually denoted by Σ) for us is always some finite set; words are sequences (strings) of letters in the alphabet. And a language is a set of words over the alphabet.

The main players in our drama have been the four classes of languages: regular language (REG), context-free languages (CFL), Turing-decidable languages (TM-DEC) and Turing-recognizable languages (TM-DEC).

Regular languages are the languages accepted by deterministic finite automata (DFAs) and context-free languages are those languages generated by context-free grammars (CFGs). Turing-decidable languages are those languages L for which there are Turing machines that always halt on every input, and decide whether a word is in L or not.

Turing-recognizable languages are more subtle. A language L is Turing-recognizable if there is a TM M which (a) when run on a word in L, halts eventually and accepts, and (b) when run on a word not in L, M either halts and rejects, or does not halt. In other words, a TM recognizing L has to halt and accept all words in L, and for words not in L, can reject or go off into a loop.

The main things to remember are:

- $\mathsf{REG} \subset \mathsf{CFL} \subset$ TM-DEC $\subset$ TM-RECOG.
- Each of the above inclusions is *strict*: i.e. there is a language that is context-free but not regular, there is a language that is TM-DEC but not context-free, etc.
- There are languages that are not even TM-RECOG.

Regular languages are trivially contained within context-free languages (as DFAs can be easily converted to PDAs). However, it is not easy to see that a CFG/PDA for L can be converted to a TM deciding L. However, this is possible (see Theorem 4.9). TM-DEC languages are clearly TM-RECOG as well, by definition.

For example, if $\Sigma = \{a, b\}$, then

- $\{a^i b^j \mid i, j \in \mathbb{N}\}$ is regular (and hence also a CFL, and TM-DEC and TM-RECOG),
- $\{a^n b^n \mid n \in \mathbb{N}\}$ is a CFL but not regular (but is TM-DEC and TM-RECOG),
- $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ is TM-decidable (and TM-RECOG) but not context-free (nor regular).
- A language that is Turing-recognizable but not Turing-decidable is

$$\mathrm{A}_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM that accepts } w\}.$$

- A language that is not even Turing-recognizable is
  $DOESNOT_{TM} = \{\langle M, w\rangle \mid M \text{ is a TM that does not accept } w\}$.

The notion of what we call an "algorithm" in computer science accords with Turing-decidability. In other words, when we build an algorithm for a decision problem in computer science, we want it to always halt and say 'yes' or 'no'. Hence the notion of a computable function is that it be TM-decidable.

32.3 Regular languages

Let us review what we learnt about regular languages (Chapter 1 of Sipser).

Fix an alphabet Σ. A regular language $L \subseteq \Sigma^*$ is any language accepted by a deterministic finite automaton (DFA).

A DFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states. The (deterministic) transition function is the function $\delta : Q \times \Sigma \to Q$.

A ***non-deterministic finite automaton*** (NFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where Q, q_0 and F are as in a DFA, and the nondeterministic transition function is $\delta : Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$.

A regular expression is formed using the syntax:

$$\epsilon \mid a \mid \emptyset \mid R_1 \cup R_2 \mid R_1 \cdot R_2 \mid R_1^*.$$

Here are some properties of regular languages:

- Non-deterministic finite automata can be converted to equivalent DFAs. This construction is the "subset construction" and is important. See Theorem 1.39 in Sipser. Intuitively, for any NFA, we can build a DFA that tracks the *set of all states* the NFA can be in. Handling ϵ-transitions is a bit complex, and you should know this construction. Hence NFAs are equivalent to DFAs.

- Regular languages are closed under union, intersection, complement, concatenation and Kleene-* (Theorems 1.45, 1.47 and 1.49 in Sipser).

- Regular expressions define exactly the class of regular languages (Theorem 1.54). In other words, any language generated by a regular expression is accepted by some DFA (Lemma 1.55) and any language accepted by a DFA/NFA can be generated by a regular expression (Lemma 1.60).

- So the trinity: DFA $\equiv$ NFA $\equiv$ Regular Expression holds.

- The pumping lemma says that, if L is regular, then there is a $p \in \mathbb{N}$ such that for every $s \in L$ with $|s| > p$, there are words x, y, z with $s = xyz$ such that (a) $|y| > 0$ (b) $|xy| \leq p$ and (c) for every i, $xy^iz \in L$. In mathematical language,

$$\begin{array}{c}\text{L}\\ \text{is}\\ \text{regular}\end{array} \Rightarrow \left[\exists p \in \mathbb{N}.\ \forall s \in L \left[\begin{array}{c} s \in L \\ \wedge \\ |s| > p \end{array} \Rightarrow \left(\exists x, y, z \in \Sigma^* : \begin{array}{c} s = xyz \\ |y| > 0 \\ |xy| \leq p \\ \forall i \;\; xy^iz \in L \end{array}\right)\right]\right]$$

- The contrapositive to the pumping lemma says that, if for every $p \in \mathbb{N}$, there is an $s \in L$ with $|s| > p$, such that for every x, y, z with $s = xyz$, $|y| > 0$, and $|xy| \leq p$, there is some i such that $xy^iz \notin L$, then L is not regular.

  In mathematical language,

$$\left[\begin{array}{l}\forall p \in \mathbb{N} \\ \quad \exists s \in L\end{array} \quad \begin{array}{c} |s| > p \text{ and} \\ \left(\forall x, y, z \in \Sigma^* : \left(\begin{array}{c} s = xyz \\ |y| > 0 \\ |xy| \leq p \end{array}\right) \rightarrow (\exists i.\ xy^iz \notin L)\right)\end{array}\right] \Rightarrow \begin{array}{c} L \\ \text{is not} \\ \text{regular.}\end{array}$$

- The contrapositive to the pumping lemma gives us a way to prove a language is not regular. We take an arbitrary p, and construct a particular word w_p, which depends on p, such that $w_p \in L$ and $|w_p| > p$. Then we show that *no matter* which x, y, z is chosen such that $s = xyz$, $|xy| \leq p$ and $|y| > 1$, there is an i such that $xy^iz \notin L$.

  Knowing how to prove a language non-regular using the pumping lemma is important.

- We can using the above technique, show several languages to be non-regular, for example (see Eg.1.73, 1.74, 1.75, 1.76, 1.77):

  - $\{0^n1^n | n \geq 0\}$ is not regular.
  - $\{w | w$ has an equal number of 0s and 1s $\}$ is not regular.
  - $\{ww | w \in \Sigma^*\}$ is not regular.
  - $\{1^{n^2} | n \geq 0\}$ is not regular.

  Choosing $w_p \in L$ should be done carefully and cleverly. However, the choice of i being 0 or 2 usually works for most example.

  Note that you are allowed to pick w_p (but not p), and allowed to pick i (not x,y or z).

- Deterministic finite automata can be *uniquely minimized.* In other words, for any regular language, there is a unique minimal automaton accepting it (here, by minimal, we mean an automaton with the least number of states). Moreover, given a DFA A, we can build an efficient algorithm to build the minimal DFA for the language $L(A)$. This is not covered in Sipser; see the handout on suffix languages and minimization:

`http://uiuc.edu/class/fa07/cs273/Handouts/minimization/suffix.pdf`

and the minimization algorithm:

`http://uiuc.edu/class/fa07/cs273/Handouts/minimization/minimization.pdf`.

For the final exam, you are not required to know this algorithm, but just know that regular languages have a unique minimal DFA.

Turning to algorithms for manipulating automata, here are some things worth knowing (read Sipser Section 4.1):

- We can build an algorithm that checks, given a DFA/NFA A, whether $L(A) \neq \emptyset$. In other words, the problem of checking emptiness of an automaton is decidable. (see Sipser Theorems 4.1 and 4.2). In fact, this algorithm runs in linear (i.e. $O(n)$) time.

- Automata are closed under operations union, intersection, complement, concatenation, Kleene-*, etc. Moreover, we can build algorithms to do all these closures. That is, we can build algorithms that will take two automata and compute an automaton accepting the union of the languages accepted by the two automata, etc.

  All constructions we did on automata are actually computable algorithmically. For example, we can build algorithms to convert regular expressions to automata, automata to regular expressions, etc.

  Several other questions regarding automata are also decidable: For example:

  - Given two automata A and B, we can decide whether $L(A) \subseteq L(B)$.
    Note that $L(A) \subseteq L(B)$ iff $L(A) \cap \overline{L(B)} = \emptyset$. So we can build the complement C of B, intersect C with A to get D, and check D for emptiness.
  - Given two automata A and B, we can decide if $L(A) = L(B)$, by checking if $L(A) \subseteq L(B)$ and if $L(B) \subseteq L(A)$, which we have show above to be decidable. (See Sipser Theorem 4.5.)

  In general, most reasonable questions about automata are decidable.

32.4 Context-free Languages

A context-free grammar is a 4-tuple (V, Σ, R, S), where V is a finite set of variable, Σ is a finite set of terminals, $S \in V$ is the start variable, and R is a set of rules of the form $A \rightarrow w$ where $A \in V$ and $w \in (V \cup \Sigma)*$.

A context-free grammar generates a set of words over its terminal alphabet Σ, and is called the language generated by the grammar. A word w is generated by a CFG if we can derive, starting with the start symbol S, and using the rules a finite number of times, the word w. A derivation of a word can also be seen as a *parse tree*, where the root is labeled with S, and the leaves, read left to write, give w, and each node with its children encode some derivation rule in R.

A CFG is in Chomsky normal form if every rule is of the form $A \rightarrow BC$ or $A \rightarrow a$, where A, B, C are non-terminals, and a is a terminal. (There's a slightly more complex definition to allow generating ϵ.)

A string is derived ambiguously in a CFG G if it has two or more different left-most derivations (or two or more parse tree derivations) in G. A grammar G is ambiguous if it can derive some word w ambiguously.

A language over Σ is a *context-free language* if it is generated by some context-free grammar with terminal alphabet Σ.

A *pushdown automaton* (PDA) is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, q_0 and F are as in a DFA, Γ is a finite stack alphabet, and the nondeterministic transition function is $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \rightarrow \mathcal{P}(Q \times \Gamma_\epsilon)$.

A pushdown automaton is, by convention, nondeterministic. It uses a LIFO stack to store data, and accepts when it has some run on a word on which it reaches a final state.

Here are some important facts about context-free languages:

- Context-free grammars can be converted to Chomsky normal form (Theorem 2.9).

- Pushdown automata define exactly the class of context-free languages. I.e. PDA $\equiv$ CFG. (Sipser Theorems 2.20).

- Context-free languages are closed under union, concatenation and Kleene-*, but not under intersection or complement. (See the last section in this article for more details.)

- Deterministic pushdown automata are strictly weaker than pushdown automata (since they can be complemented by toggling the final states).

- The membership problem for CFGs and PDAs is decidable. In particular, the CYK algorithm uses dynamic programming to solve the membership problem for CFGs, and in fact produces a parse tree as well, in $O(n^3)$ time.

- The problem of checking if a context-free language generates all words (i.e. if $L(G) = \Sigma^*$) is undecidable. This is proved in Theorem 5.13, using context-free grammars that check computation histories of Turing machines.

- The problem of checking if a context-free language is ambiguous is undecidable (Exercise 5.21 in Sipser), and is proved by a reduction from the Post's correspondence problem. You need to know this fact, not the proof.

- The language $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ is not a context-free language. There is a pumping lemma for context-free languages, and we can use it to show that this language is not context-free. You are not required to know this proof.

32.5 Turing machines and computability

32.5.1 Turing machines

A Turing machine is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject}$ where Q is a finite set of states, Σ is a finite alphabet, Γ is the tape-alphabet that includes Σ and a particular symbol # which is the blank symbol, $q_0 \in Q$ is the initial state, $q_{accept} \in Q$ is the accept state and q_{reject} is the reject state ($q_{accept} \neq q_{reject}$). The transition function is deterministic: $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$.

A Turing machine has access to a one-way infinite tape, and hence has unbounded memory. It can go back and forth on the tape rewriting symbol to do its computation.

A Turing machine halts if it reaches an accepting or rejecting configuration (i.e. reaches a configuration with state q_{accept} or q_{reject}. A Turing machine stops when it reaches such a configuration.

There are two main classes of languages defined using Turing machines:

- A language L is Turing-decidable if there is a Turing machine M such that M halts on all inputs, and accepts all words in L and rejects all words not in L.

- A language L is Turing-recognizable if there is a Turing machine M such that (a) on any word in L, M halts and accepts, and (b) on any word not in L, either M does not halt, or halts and rejects.

The notion of Turing-recognizability and Turing-decidability are robust concepts. Changing the definition of Turing machines in any reasonable way does not change this notion. For example, a multi-tape Turing machine is not more powerful, and can be converted to a single-tape Turing machine. Also, giving two-way-infinite tapes do not make Turing machines more powerful.

A ***nondeterministic Turing machine*** (NTM) is a Turing machine which has a transition function that is non-deterministic. The languages decided and recognized by non-deterministic Turing machines are precisely those decided and recognized by deterministic Turing machines. This proof is done using *dovetailing*: a deterministic Turing machine simulates an NTM by systematically exploring all runs of the NTM for time i steps, for increasing values of i.

By definition, a Turing-decidable language is also Turing-recognizable.

An important language is:
- $A_{TM} = \{\langle M, w\rangle \mid M$ is a Turing machine that halts and accepts $w\}$.
Here are some of the important results:

- A_{TM} is *not* Turing-decidable (Theorem 4.11). This is the fundamental undecidable problem and is shown undecidable using a *diagonalization* method, which takes the code for a purported TM deciding A_{TM} and pits it against itself to lead to a contradiction. Diagonalization is an important technique to prove impossibility results (almost the only technique we know!).

- A_{TM} is Turing-recognizable. This is easy to show: we can build a Turing machine that on input $\langle M, w \rangle$, simulates M on w and accepts if M accepts w. Hence the class TM-DEC is a strict subclass of TM-RECOG.

- A language L is Turing-decidable iff L and $\overline{L}$ are Turing-recognizable (Theorem 4.22).
  If L is decidable, then $\overline{L}$ is decidable as well, and so L and $\overline{L}$ are Turing-recognizable. If L and $\overline{L}$ are both Turing-recognizable, we can build a decider for L by simulating the machines for L and $\overline{L}$ in "parallel", and accept or reject depending on which of them accepts.

- A corollary to the above theorem is that if L is Turing-recognizable and not Turing-decidable, then $\overline{L}$ is not TM recognizable. Hence, $\overline{A_{TM}}$ is not even TM-recognizable (Corollary 4.23).

32.5.2 Reductions

Reductions are a technique to deduce undecidability of problems using another problem that is known to be undecidable.

A language S reduces to a language T if, given a TM deciding T, we can build a TM that decides S.

In other words, if T is decidable, then S is decidable. Which, paraphrased, says that if S is undecidable then T is undecidable.

Hence, to show T is undecidable, we choose a language S that we know is undecidable, and reduce S to T.

Many reductions are from A_{TM}; to show L is undecidable, we try to reduce A_{TM} to L, i.e. assuming we have a decider for L, we show that we can build a decider for A_{TM}. Since A_{TM} has no decider, it follows that L has no decider.

Reduction proofs are important to understand and learn. Reductions from A_{TM} to languages that accept Turing machine descriptions often go roughly like this:

- Assume L has a decider R; we build a decider D for A_{TM}.

- D takes as input $\langle M, w \rangle$.
  D then *modifies* M to construct a TM $N_{M,w}$.
  D then feeds this machine $N_{M,w}$ to R.
  Depending on whether R accepts or rejects, D accepts or rejects (sometimes switching the answer).

Using reductions we can prove several languages undecidable. For example, (see Theorems 5.1, 5.2, 5.3, 5.4)

- $HALT_{TM} = \{\langle M, w \rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$ is undecidable.
  It is Turing-recognizable, though.

- $E_{TM} = \{\langle M, w \rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$ is undecidable.
  It is not even Turing-recognizable since its complement is Turing-recognizable.

- $REGULAR_{TM} = \{\langle M \rangle \mid L(M) \text{ is regular }\}$ is undecidable.

- $EQ_{TM} = \{\langle M_1, M_2 \rangle \mid L(M_1) = L(M_2)\}$ is undecidable.

Rice's theorem generalizes many undecidability results. Consider a class P of Turing machine descriptions. Assume that P is a property of Turing machines that depends only on the language of the Turing machines (i.e. if M and N are Turing machines accepting the *same* language, then either both are in P or both are not in P). Also assume that P is not the empty set nor the set of all Turing machines. Then P is *undecidable.*

Note that if P was the empty set or the set of all Turing machine descriptions, then clearly it is decidable.

Note: There will be a question on reductions in the exam. The reduction will be one which is a direct corollary of Rice's theorem, but you will be asked to give a proof without using Rice's theorem.

32.5.3 Other undecidability problems

There were several other problems that were shown to be undecidable. Knowing these are undecidable is important; you will not be asked for proofs of any of these, however:

- A linear bounded automaton (LBA) is a Turing machine that uses only the space occupied by the input, and does not use any extra cells. The emptiness problem for LBAs in undecidable (Theorem 5.10): i.e. $E_{LBA} = \{\langle M \rangle M$ is an LBA and $L(M) \neq \emptyset\}$ is undecidable.

  However, the membership problem for LBAs is decidable (Theorem 5.9):
  i.e. $A_{LBA} = \{\langle M, w \rangle \mid M$ is an LBA accepting $w\}$ is decidable.

- Checking if a context-free grammar accepts all words is undecidable (Theorem 5.13).

- The Post Correspondence Problem is undecidable (Theorem 5.15).

32.6 Summary of closure properties and decision problems

32.6.1 Closure Properties

Here's a summary of closure properties for the various classes of languages:

	Union	Intersection	Complement	Kleene-*	Homomorphisms
REG	Yes	Yes	Yes	Yes	Yes
CFL	Yes	No	No	Yes	Yes
TM-DEC	Yes	Yes	Yes	Yes	Yes
TM-RECOG	Yes	Yes	No	Yes	Yes

The results for regular languages are in Sipser and class notes.
See also
`http://uiuc.edu/class/fa07/cs273/Handouts/closure/regular-closure.html`.

Sipser doesn't cover closure properties of context-free languages very clearly. However, note that closure under union is easy as it is simple to combine two grammars to realize their union. Non-closure under intersection follows from the fact that $L_1 = \{a^i b^j c^k \mid i = j\}$ and $L_2 = \{a^i b^j c^k \mid j = k\}$ are both context-free, but their intersection $L_1 \cap L_2 = \{a^i b^j c^k \mid i = j = k\}$ is not. Non-closure under complement is easy to see as $L = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ is not context-free but its complement is context-free. Closure under Kleene-* and homomorphisms are easy as one can easily transform a grammar to do these operations.
See
`http://uiuc.edu/class/fa07/cs273/Handouts/closure/cfl-closure.html`
for more detailed proofs.

Turning to TM-DEC, they are closed under union as you can run TM M_1 followed by M_2, and accept if one of them accept. For intersection, you can run them one after the other, and accept if both accept. Closure under complement is easy as we can swap the accept and reject states of a TM. Kleene-* and homomorphisms were not covered, but it is easy to see that TM-DEC languages are closed under these operations (try them as an exercise!).

Finally, TM-RECOG is closed under union as you can run two Turing machines in "parallel", and accept if one of them accepts. The class is closed under intersection, as we can run them one after another, and accept if both accept. (Note the subtleties of the construction here; simulating a TM that recognizes a language has to be done carefully as it may not halt). The class of TM-RECOG languages is not closed under complement (for example, A_{TM} is TM-RECOG but its complement is not). In fact, if L is TM-RECOG and its complement is also TM-RECOG, then L is TM-DEC. Since we know A_{TM} is not TM-DEC, it follows that its complement is not TM-RECOG. TM-RECOG languages are closed under Kleene-* and homomorphisms— we leave these as exercises.

32.6.2 Decision problems

For each class of languages, let's consider four problems—

Membership: Given a language L, and a word w, check if $w \in L$.

Emptiness: Given a language L, check if $L = \emptyset$.

Inclusion: Given two languages L_1 and L_2, check if $L_1 \subseteq L_2$.

Equivalence: Given two languages L_1 and L_2, check if $L_1 = L_2$.

For each of the problems above, we will consider the cases when the language(s) are given as finite automata, PDAs or TM. Note that the problem does not change much if we give it using a grammar instead of a PDA, or a regular expression instead of an automaton, because we can always convert them to PDAs or automata.

	Membership	Emptiness	Inclusion	Equivalence
REG	Yes	Yes	Yes	Yes
CFL	Yes	Yes	No	No
TM-DEC	No	No	No	No
TM-RECOG	No	No	No	No

Notice that regular languages are the most tractable class, and context-free languages have the important property that membership (Theorem 4.7) and emptiness (Theorem 4.8) are decidable. In particular, membership of context-free languages is close to the problem of parsing, and hence is an important algorithmic problem. Context-free languages do not admit a decidable inclusion or equivalence problem (Theorem 5.13 shows that checking if a CFG generates all words is undecidable; we can reduce this to both the problem of inclusion– $L(A) \subseteq L(B)$– and equivalence– $L(A) = L(B)$– by setting A to be a CFG generating all words).

For Turing machines, almost nothing interesting is decidable. However, note that the membership problem for Turing machines (A_{TM}) (but not emptiness problem for Turing machines (E_{TM})) is Turing-recognizable.

Part II

Discussions

Chapter 33

Discussion 1: Review

20 January 2009

Purpose: Most of this material is review from CS 173, though students may have forgotten some of it, especially details of notation. The exceptions are the sections on strings and graph induction.

Before you start, introduce yourself. Promise office hours will be posted very soon and encourage them to come.

33.1 Homework guidelines

- Put each problem on a different sheet of paper, we grade them separately.
- Put your name on each sheet of paper you submit.
- Put the section number on every sheet of paper you submit.
- Homeworks will usually be due on Thursday at 12:30.
- Homeworks are available from the class webpage.

Do not forget to register with the news server and start reading the newsgroups!

33.2 Numbers

What are Z, N (no zero), N_0, (mention quickly Q and $\mathbb{R}$).

33.3 Divisibility

What does it mean for x to divide q? Namely, there exists integer n such that $q = xn$. As such, every number n is divisible by 1 and n (i.e., itself).

An integer number is *even* if it can be divided by 2. A number which is not even, is *odd*.

Question 33.3.1 *Is zero even?*

Well, $0 = 2 * 0$ *and as such,* 0 *is divisible by two.*

Meaning of $p = q \bmod k$. Examples... (i.e., $121 = 73 \bmod 3$).

33.4 $\sqrt{2}$ is not rational

A number x is rational if it can be written as the ratio of two integers $x = \alpha/\beta$. The fraction α/β is *irreducible* if α and β do not have any common divisors (except 1, of course). Note that a rational number has a unique way to be written in irreducible.

Lemma 33.4.1 *An integer k is even if and only if k^2 is even.*

Proof: If k is even, then it can be written as $k = 2u$, where u is an integer. As such, $k^2 = (2u)^2 = 4u^2$ is even, since it can be divided by 2. As for the other possibility, if $k = 2u + 1$, then

$$u^2 = (2k+1)^2 = 4k^2 + 2 * 2k + 1 = 2(2k^2 + 2k) + 1,$$

is odd (since it is the sum of an even number and an odd number), implying the claim. ■

Theorem 33.4.2 *The number $\sqrt{2}$ is not rational (i.e., $\sqrt{2}$ is an irrational number).*

Proof: Assume for the sake of contradiction that $\sqrt{2}$ is rational, and it can written as the irreducible ratio $\sqrt{2} = \alpha/\beta$.

Let us square both size of this equation. We get that

$$2 = \frac{\alpha^2}{\beta^2}.$$

That is the number 2 is a divisor of α^2. Namely, α^2 is an even number. But then, α must be an even number. So, let $\alpha = 2a$. We have that

$$2 = \frac{(2a)^2}{\beta^2}. \quad \Rightarrow \quad 2\beta^2 = (2a)^2 = 4a^2.$$

As such,

$$\beta^2 = 2a^2.$$

Namely, β^2 is even, which implies, again that β is even. As such, let us write $\beta = 2b$, where b is an integer. We thus have that

$$\sqrt{2} = \frac{\alpha}{\beta} = \frac{2a}{2b} = \frac{a}{b}.$$

Namely, we started from a rational number in irreducible form (i.e., α/β) and we reduced it further to a/b. But this is impossible. A contradiction. Our assumption that $\sqrt{2}$ is a rational number is false. We conclude that $\sqrt{2}$ is irrational. ■

33.5 Review of set theory

Sets, set building notation (just show an example), subset, proper subset, empty set, Venn diagram

Tuples, cross product of two sets.

Sets of sets, power set, sets can contain a mixture of stuff like $\{a, b, \{a, b\}\}$.

Demorgan's Laws

Cardinality of a set

33.6 Strings

strings, empty string, sets of strings

substring, prefix, suffix, string concatenation

What happens when you concatenate the empty string with another string?

Suppose w is a string. Is the empty string a substring of w? (Yes!)

33.7 Recursive definition

Recursive definition (concrete example).

Let U be the set that contains the point $(0, 0)$. Also:

- if $(x, y) \in U$, then $(x + 1, y) \in U$.
- if $(x, y) \in U$, then $(x, y + 1) \in U$

Q: What is U?
A: Clearly, $(0, 1) \in U$, $(0, 2) \in U$,.... $(0, k) \in U$ (for any k).
As such, $(1, k) \in U$ (for any k)
As such, $(2, k) \in U$ (for any k)
As such, $(j, k) \in U$ (for any $j > 0, k > 0$).
Note that $U = N_0 \times N_0$.
(A more complicated example of this is in the homework.)

33.8 Induction example

The following wasn't done in some (all?) of the sections, but you may still find it useful to read.

Consider a graph $G = (V, E)$. The graph G is *connected*, if there is a path between any two vertices of G. A graph is *acyclic* if does not contain any cycle in it. A *leaf* in G is a node that has exactly one edge adjacent to it.

Lemma 33.8.1 *A connected acyclic graph G over $n > 1$ vertices, must contain a leaf.*

Proof: Indeed, start from a vertex $v \in V(G)$, if it is a leaf we are done, otherwise, there must be an edge $e = vu$ of G adjacent to v (since G is connected), and travel on this edge to u. Mark the edge e as used. Repeat this process of "walking" in the graph till you reach a leaf (and then you stop), where the walk can not use an edge that was used before.

If this walk process reached a leaf then we are done. The other possibility is that the walk never ends. But then, we must visit some vertex we already visited a second time (since the graph is finite). But that would imply that G contains a cycle. But that is impossible, since G is acyclic (i.e., it does not contain cycles). ■

Claim 33.8.2 *A connected acyclic graph G over n vertices has exactly $n - 1$ edges.*

Proof: The proof is by induction on n.

The base of the induction is $n = 2$. Here we have two vertices, and since its connected, this graph must have the edge connecting the two vertices, implying the claim.

As for $n > 2$, we know by the above lemma that G contains a leaf, and let w denote this leaf.

Consider the graph H formed by G by removing from it the vertex w and the single edge e' attached to it. Clearly the graph H is connected and acyclic, and it has $n - 1$ vertices. By the induction hypothesis, the graph H has $n - 2$ edges. But then, the graph G has all the edges of H plus one (i.e., e'). Namely, G has $(n - 2) + 1 = n - 1$ edges, as claimed. ■

Chapter 34

Discussion 2: Examples of **DFA**s

27 January 2009

Purpose: This discussion demonstrates a few constructions of DFAs. However, its main purpose is to show how to move from a diagram describing a DFA into a formal description, in particular of the transition function.
This material here (probably) can not be covered in one discussion section.

Questions on homework 1?

Any questions? Complaints, etc?

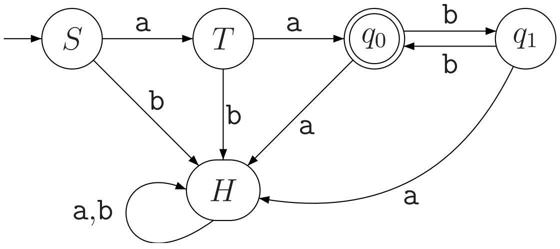
34.1 Languages that depend on k

34.1.1 $\mathtt{aab}^{2i}$

Consider the following language:

$$L_2 = \left\{ \mathtt{aab}^n \;\middle|\; n \text{ is a multiple of } 2 \right\}.$$

Its finite automata is



█ `Advice To TA::` Do not erase this diagram from the board, you would need to modify it shortly, for the next example. ━━ `end` ━━

This automata formally is the tuple $(Q, \Sigma, \delta, S, F)$.

1. $Q = \{S, T, H, q_0, q_1\}$ - states.

2. $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ - alphabet.

3. $\delta : Q \times \Sigma \to Q$, described in the table.

	a	b
S	T	H
T	q_0	H
H	H	H
q_0	H	q_1
q_1	H	q_0

4. S is the start state.

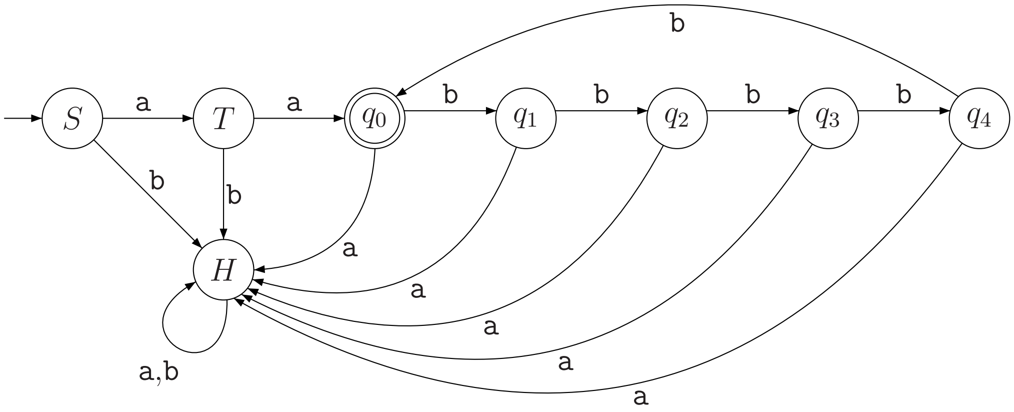
5. $F = \{q_0\}$ is the set of accepting states.

34.1.2 $\texttt{aab}^{5i}$

Consider the following language:

$$L_5 = \left\{ \texttt{aab}^n \;\middle|\; n \text{ is a multiple of } 5 \right\}.$$

Its finite automata is



Advice To TA:: Do not erase this diagram from the board, you would need to modify it shortly, for the next example. **end**

This automata formally is the tuple $(Q, \Sigma, \delta, S, F)$.

1. $Q = \{S, T, H, q_0, q_1, q_2, q_3, q_4\}$ - states.
2. $\Sigma = \{\texttt{a}, \texttt{b}\}$ - alphabet.
3. $\delta : Q \times \Sigma \rightarrow Q$ - see table.
4. S is the start state.
5. $F = \{q_0\}$ is the set of accepting states.

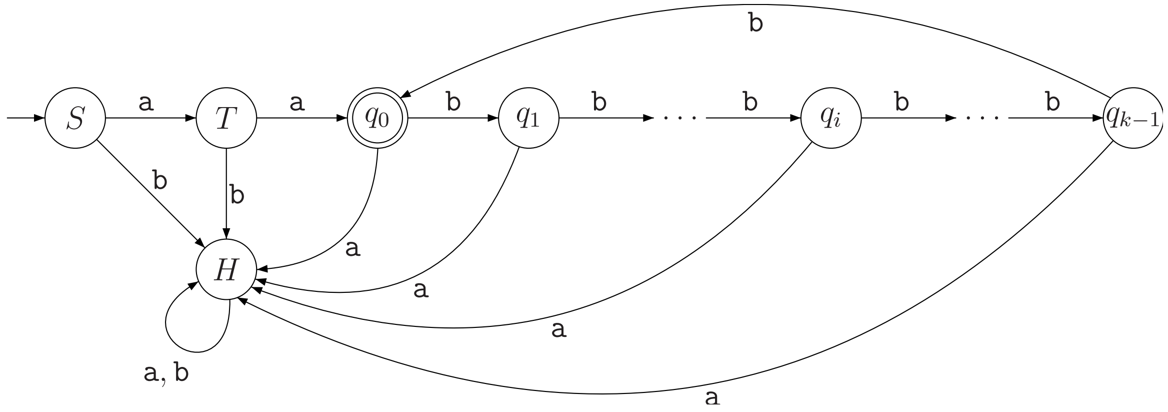
δ	a	b
S	T	H
T	q_0	H
H	H	H
q_0	H	q_1
q_1	H	q_2
q_2	H	q_3
q_3	H	q_4
q_4	H	q_0

34.1.3 $\texttt{aab}^{ki}$

Let k be a fixed constant, and consider the following language:

$$L_k = \left\{ \texttt{aab}^n \;\middle|\; n \text{ is a multiple of } k \right\}.$$

Its finite automata is



This automata formally is the tuple $(Q, \Sigma, \delta_k, S, F)$.

1. States:
   $Q = \{S, T, H, q_0, q_1, \dots, q_{k-1}\}$.
2. $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ - alphabet.
3. $\delta_k : Q \times \Sigma \to Q$ - see table.
4. S is the start state.
5. $F = \{q_0\}$ is the set of accepting states.

	a	b
S	T	H
T	q_0	H
H	H	H
q_0	H	q_1
q_1	H	q_2
$\vdots$	$\vdots$	$\vdots$
q_i	H	q_{i+1}
$\vdots$	$\vdots$	$\vdots$
q_{k-1}	H	q_0

Explicit formula for δ

Advice To TA:: Skip the first two forms in the discussion. Show only the one in Eq. (34.1). end

Another way of writing the transition function δ_k, for the above example, is the following:

$$\begin{aligned}
\delta_k(S, \mathtt{a}) &= T,\\
\delta_k(S, \mathtt{b}) &= H,\\
\delta_k(T, \mathtt{a}) &= q_0,\\
\delta_k(T, \mathtt{b}) &= H,\\
\delta_k(H, \mathtt{a}) &= H,\\
\delta_k(H, \mathtt{b}) &= H,\\
\delta_k(q_i, \mathtt{a}) &= H, \quad \forall i\\
\delta_k(q_i, \mathtt{b}) &= q_{i+1} \quad \text{for } i < k-1,\\
\delta_k(q_{k-1}, \mathtt{b}) &= q_0.
\end{aligned}$$

Another way to write the same information

$$\delta_k(s, x) = \begin{cases}
T & s = S, x = \mathtt{a}\\
H & s = S \text{ or } T, x = \mathtt{b}\\
q_0 & s = T, x = \mathtt{a}\\
H & s = H, x = \mathtt{a} \text{ or } \mathtt{b}\\
H & s = q_i, x = \mathtt{a}, \qquad i = 0, \dots, k-1\\
q_{i+1} & s = q_i, x = \mathtt{b}, \qquad i < k-1,\\
q_0 & s = q_{k-1}, s = \mathtt{b}.
\end{cases}$$

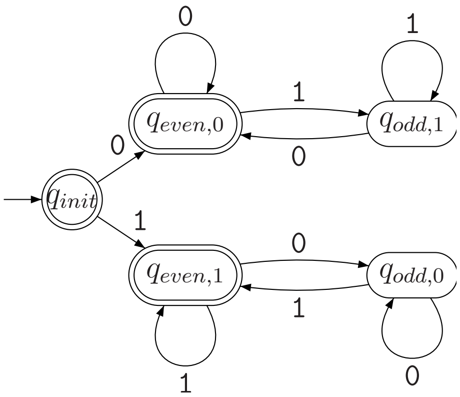
This can be made slightly more compact using the mod notation:

$$\delta_k(s,x) = \begin{cases} T & s = S, x = \texttt{a} \\ H & s = S \text{ or } T, x = \texttt{b} \\ q_0 & s = T, x = \texttt{a} \\ H & s = H, x = a \text{ or } \texttt{b} \\ H & s = q_i, x = \texttt{a}, \qquad \forall i \\ q_{(i+1) \bmod k} & s = q_i, x = \texttt{b}, \qquad \forall i. \end{cases} \tag{34.1}$$

Note, that using good state names would help you to describe the automata compactly (thus q_0 here is not that the starting state). Generally speaking, the shorter your description is, the least work needed to be done, and the chance you make a silly mistake is lower.

34.2 Number of changes from 0 to 1 is even

Let us build a DFA that recognizes all binary strings, such that the number of times the string changes from consecutive zeroes to ones (and vice versa) is even. Thus `0000` is in our language (number of changes is 0, but `0000111111` is not (number of changes is one). We need to remember what was the last character in the input, and whether the number of changes so far was even or odd. As such, we need 4 states. But wait! What about the empty string ϵ. Clearly, it is in the language (i.e., the number of changes between 0 and 1 is zero, which is even). As such, we need a special state to handle it.



1. States: $Q = \{q_{init}, q_{even,0}, q_{even,1}, q_{odd,0}, q_{odd,1}\}$.
2. $\Sigma = \{0, 1\}$ - alphabet.
3. $\delta : Q \times \Sigma \to Q$, see table on the right.
4. q_{init} is the start state.
5. Set of accepting states: $F = \{q_{even,0}, q_{even,1}, q_{init}\}$.

	0	1
q_{init}	$q_{even,0}$	$q_{even,1}$
$q_{even,0}$	$q_{even,0}$	$q_{odd,1}$
$q_{even,1}$	$q_{odd,0}$	$q_{even,1}$
$q_{odd,0}$	$q_{odd,0}$	$q_{even,1}$
$q_{odd,1}$	$q_{even,0}$	$q_{odd,1}$

The resulting finite automata is the tuple $(Q, \Sigma, \delta, q_{init}, F)$.

34.3 State explosion

`Advice To TA::` Show first the longer way to solve this problem. Use your board cleverly, so that you can move from the tedious form into the shorter form, by just erasing things. `end`

Because automatas do not have an explicit way of storing information, we need to encode the information to solve the problem explicitly in the states of the DFA. That can get very tedious, as the following example shows.

Let L be the language of all binary strings such that the third character from the end is zero.

To design an automata that accepts this language, we clearly need to be able to remember the last three characters of the input. To this end, let us introduce the states $q_{000}, q_{001}, q_{010}, q_{011}, q_{100}, q_{101}, q_{110}, q_{111}$. Here the automata would be in state q_{011} if the last three characters are 0, 1 and 1. Naturally, we need to also be able to handle the first one or two characters arriving to the input, which forces us to introduce special states for this case. Here is the resulting automata in formal description. Here, the automata is the tuple $(Q, \Sigma, \delta, e, F)$.

1. States:
   $Q = \{e, q_0, q_1, q_{00}, q_{01}, q_{10}, q_{11}, \quad q_{000}, q_{001}, q_{010}, q_{011}, \quad q_{100}, q_{101}, q_{110}, q_{111}\}$
2. $\Sigma = \{0, 1\}$ - alphabet.
3. $\delta : Q \times \Sigma \to Q$ - see table.
4. e is the start state.
5. $F = \{q_{000}, q_{001}, q_{010}, q_{011}\}$ is the set of accepting states.

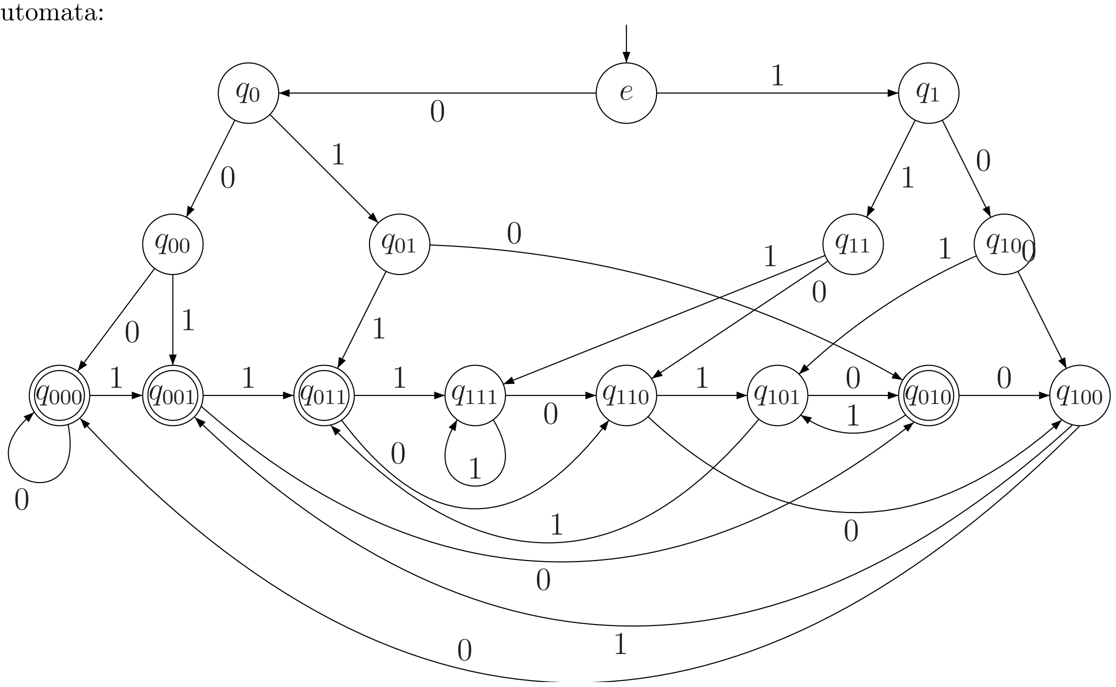
	0	1
e	q_0	q_1
q_0	q_{00}	q_{01}
q_1	q_{10}	q_{11}
q_{00}	q_{000}	q_{001}
q_{01}	q_{010}	q_{011}
q_{000}	q_{000}	q_{001}
q_{001}	q_{010}	q_{011}
q_{010}	q_{100}	q_{101}
q_{011}	q_{110}	q_{111}
q_{100}	q_{000}	q_{001}
q_{101}	q_{010}	q_{011}
q_{110}	q_{100}	q_{101}
q_{111}	q_{110}	q_{111}

`Advice To TA::` Please show the explicit long way of writing the transition table (shown above), and only then show the following more compact way. Its beneficial to see how using more formal representation, can save you a lot of time and space. Say it explicitly in the discussion. `end`

A more sane way to write the transition table for δ is

	0	1
e	q_0	q_1
q_x	q_{x0}	q_{x1}
q_{xy}	q_{xy0}	q_{xy1}
q_{xyz}	q_{yz0}	q_{yz1}

Which is clearly the most compact way to describe this transition function. And here is a drawing of this automata:



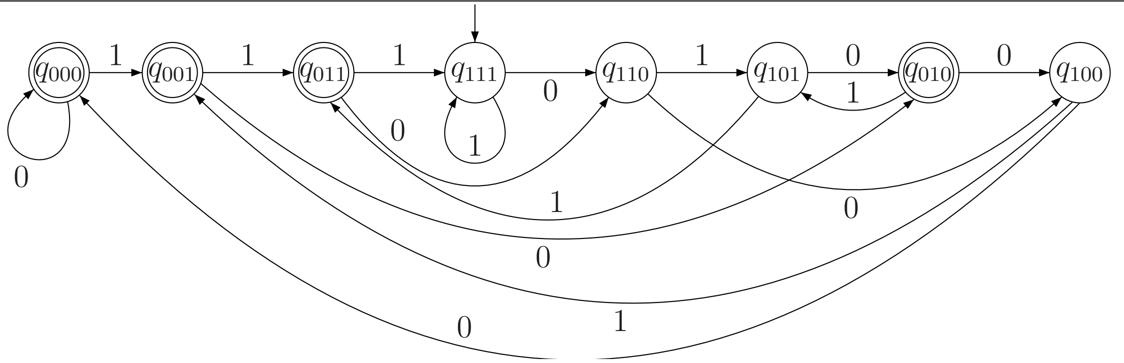
`Advice To TA::` Trust me. You do not want to erase this diagram before doing the next example.. `end`

34.3.1 Being smarter

Since we care only if the third character from the end is zero, we could pretend that when the input starts, the automata already saw, three ones on the input. Thus, setting the initial state to q_{111}. Now, we can get rid of the special states we had before.

1. States:
   $Q = \{q_{000}, q_{001}, q_{010}, q_{011},$
   $q_{100}, q_{101}, q_{110}, q_{111}\}$
2. $\Sigma = \{0, 1\}$ - alphabet.
3. $\delta : Q \times \Sigma \to Q$ - see table.
4. q_{111} is the start state.
5. $F = \{q_{000}, q_{001}, q_{010}, q_{011}\}$ is the set of accepting states.

	0	1
q_{xyz}	q_{yz0}	q_{yz1}



This brings to the forefront several issues: (i) the most natural way to design an automata does not necessarily leads to the simplest automata, (ii) a bit of thinking ahead of time will save you much pain, and (iii) how do you know that what you came up with is the simplest (i.e., fewest number of states) automata accepting a language?

The third question is interesting, and we will come back to it later in the course.

34.4 None of the last k characters from the end is 0

Let L'_k be the language of all binary strings such that none of the least k characters is 0. We can of course adapt the previous automata to this language, by changing the accepting states and the start state. However, we can solve it more efficiently, by remembering what is the maximum length of the suffix of the input seen so far, such that its all one.

In particular, let q_i be the state such that the suffix of the input is a zero followed by i ones. Clearly, q_k is the accept state, and q_0 is the starting state. The transition function is also simple. If the automata sees a 0, it goes back to q_0. If it at q_i and it accepts a 1 it goes to q_{i+1}, if $i < k$. . We get the following automata.

1. States: $Q = \left\{ q_i \;\middle|\; i = 0, \ldots, k \right\}$.
2. $\Sigma = \{0, 1\}$ - alphabet.
3. $\delta_k : Q \times \Sigma \rightarrow Q$, where
$$\delta_k(q_i, x) = \begin{cases} q_0 & x = 0 \\ q_{\min(i+1,k)} & x = 1. \end{cases}$$
4. q_0 is the start state.
5. Set of accepting states: $F = \{q_k\}$.

So, a little change in the definition of the language can make a dramatic difference in the number states needed.

Chapter 35

Discussion 3: Non-deterministic finite automatas

February 3, 2009

Purpose: This discussion demonstrates a few simple NFAs, and how to formally define a NFA. We also demonstrate that complementing a NFA is a tricky business.

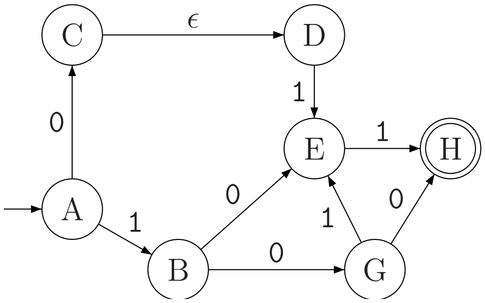
Questions on homework 2?

Any questions? Complaints, etc?

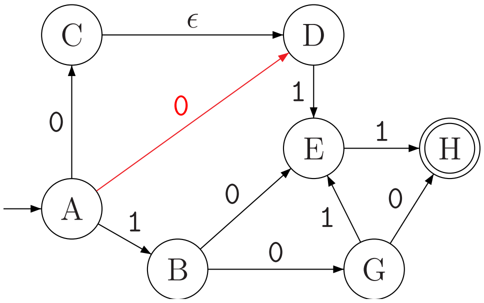
35.1 Non-determinism with finite number of states

35.1.1 Formal description of NFA

M_1 :



In the above NFA, we have $\delta(A, 0) = \{C\}$. Despite the ϵ-transition from C to D. As such, $\delta(A, 0) \neq \{C, D\}$. If $\delta(A, 0) = \{C, D\}$ then the NFA is a different NFA:



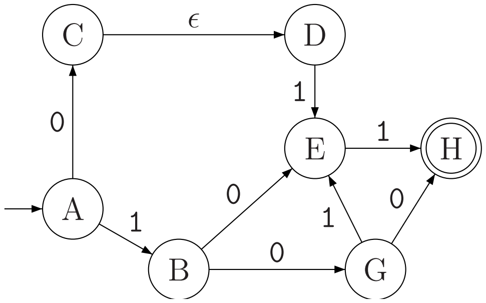
In any case, the NFA M_1 (depicted in first figure) is the 5-tuple $(Q, \Sigma, \delta, A, \mathcal{F})$, where

$$\delta : Q \times \Sigma_\epsilon \rightarrow \mathbb{P}(Q).$$

Here $\Sigma = \{\mathtt{0}, \mathtt{1}\}$, $Q = \{A, B, C, D, E, G, H\}$, and $\mathcal{F} = \{H\}$.

δ	0	1	ϵ
A	$\{C\}$	$\{B\}$	$\emptyset$
B	$\{E, G\}$	$\emptyset$	$\emptyset$
C	$\emptyset$	$\emptyset$	$\{D\}$
D	$\emptyset$	$\{E\}$	$\emptyset$
E	$\emptyset$	$\{H\}$	$\emptyset$
G	$\{H\}$	$\{E\}$	$\emptyset$
H	$\emptyset$	$\emptyset$	$\emptyset$

M_1 :



35.1.2 Concatenating NFAs

We are given two NFAs $M = (Q, \Sigma, \delta, A, F)$ and $M' = (Q', \Sigma, \delta', A', F')$. We would like to build an NFA for the concatenated language $L(M)L(M')$.

First, we can assume that M has a single accepting state f. Indeed, we can create a new accepting state f, add it to Q, make all the states in F non-accepting, but add an ϵ-transition from them to f. Thus, we can now assume that $F = \{f\}$.

Back to our task, of constructing the concatenated NFA, we can just create an ϵ transition from f to A'. Here is the formal construction of the NFA for the concatenated language $N = \left(\mathcal{Q}, \Sigma, \widehat{\delta}, A, F'\right)$, where $\mathcal{Q} = Q \cup Q'$. As for *widehat*δ, we have that

$$\widehat{\delta}(q, x) = \begin{cases} \delta(q, x) & q \in Q \\ \delta'(q, x) & q \in Q' \\ \delta(f, \epsilon) \cup \{A'\} & q = f \text{ and } x = \epsilon. \end{cases}$$

Claim 35.1.1 *The NFA N accepts a string $w \in \Sigma^*$, if and only if there exists two strings $x, y \in \Sigma^*$, such that $w = xy$ and $x \in L(M)$ and $y \in L(M')$.*

Proof: If $x \in L(M)$ then there is an accepting trace (i.e., a sequence of states and inputs that show that x is being accepted by M, and let the sequence of states be $A = r_0, r_1, \ldots, r_\alpha$, and the corresponding input sequence be $x_1, \ldots, x_\alpha \in \Sigma_\epsilon$. Here $x = x_1 x_2 \ldots x_\alpha$ (note that some of these characters might be ϵ).

Similarly, let $A' = r'_0, r'_1, \ldots, r'_\beta$ be accepting trace of M' accepting y, with the input characters $y_1, y_2, \ldots, y_\beta \in \Sigma_\epsilon$, where $y = y_1 y_2 \ldots y_\beta$.

Note, that by our assumption $r_\alpha = f$. As such, the following is accepting trace of $w = xy$ for N:

$$r_0 \underset{x_1}{\rightarrow} r_1 \underset{x_2}{\rightarrow} r_2 \rightarrow \cdots \underset{x_\alpha}{\rightarrow} r_\alpha \underset{\epsilon}{\rightarrow} r'_0 \underset{y_1}{\rightarrow} r'_1 \underset{y_2}{\rightarrow} \cdots \underset{y_\beta}{\rightarrow} r'_\beta.$$

Indeed, its a valid trace, as can be easily verified, and $r'_\beta \in F'$ (otherwise y would not be in $L(M')$.

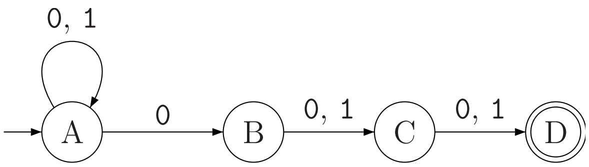
Similarly, given a word $W \in L(N)$, and accepting trace for it, then we can break this trace into two parts. The first part is trace before using the transition $f \rightarrow_\epsilon A'$, and the other is the rest of the trace. Clearly, if we remove this transition from the given accepting trace, we end up with two accepting traces for M and M', implying that we can break w into two strings x and y, such that $x \in L(M)$ and $y \in L(M')$. ■

35.1.3 Sometimes non-determinism keeps the number of states small

Let $\Sigma = \{\mathtt{0}, \mathtt{1}\}$. Remember that the smallest DFA that we built for

$$L_3 = \left\{ x \in \Sigma^* \;\middle|\; \text{the third character from the end of } x \text{ is a zero } \right\}.$$

had 8 states. Note that the following NFA does the same job, by guessing position of third character from the end of string.



M_3:

Q: Is there a language L where we have a DFA for L with smaller number of states that of any NFA for L?

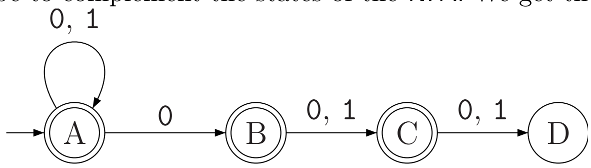
A: No. Because any DFA is also a NFA.

35.1.4 How to complement an NFA?

Given the NFA above M_3, it is natural to ask how to build a NFA for the complement language

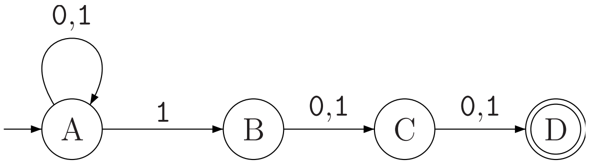
$$\overline{L(M_3)} = \overline{L_3} = \left\{ x \in \Sigma^* \;\middle|\; \text{the third character from the end of } x \text{ is not zero} \right\}.$$

Naively, the easiest thing would be to complement the states of the NFA. We get the following NFA M_4.



M_4:

But this is of course complete and total nonsense. Indeed, the language of $L(M_4) = \Sigma^*$, which is definitely not $\overline{L_3}$. Here is the correct solution.



M_5:

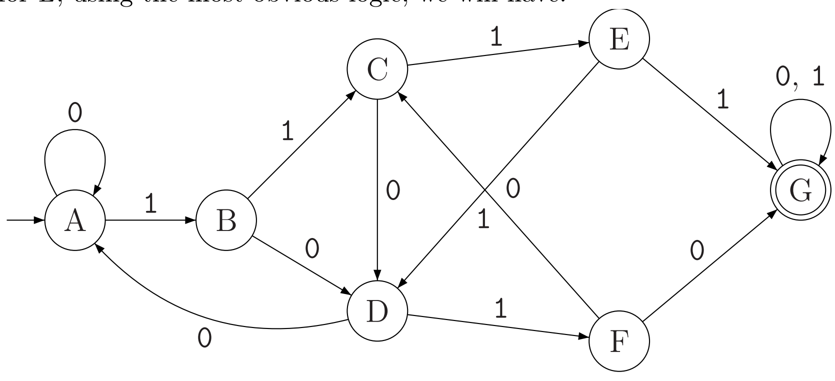
The conclusion of this tragic and sad example is that complementing a NFA is a non-trivial task (unlike DFAs where all you needed to do was to just flip the accepting/non-accepting states). So, for some tasks DFAs are better than DFAs, and vice versa.

35.1.5 Sometimes non-determinism keeps the design logic simple

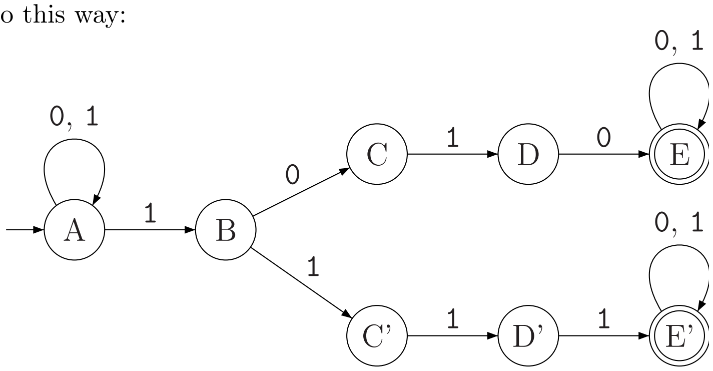
Consider the following language:

$$L = \{x : x \text{ has } 1111 \text{ or } 1010 \text{ as a substring}\}$$

Designing a DFA for L, using the most obvious logic, we will have:



With NFA we can go this way:



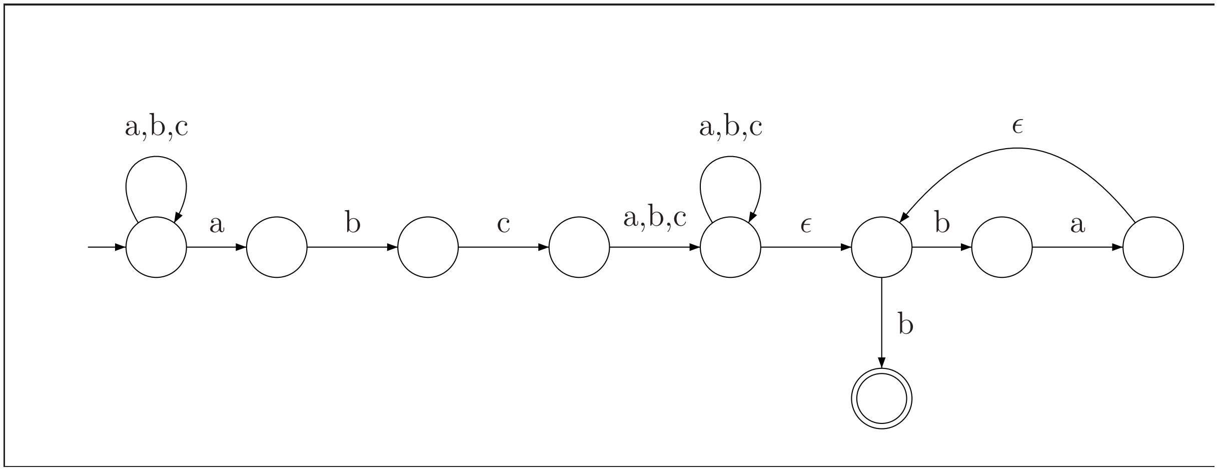
Note that the NFA approach is easily extendable to more than 2 substrings.

35.2 Pattern Matching

Suppose we wanted to build an NFA for the following pattern.

$$abc?(ba)^*b$$

Where ? represents a substring of length 1 or more and * represents 0 of more of the previous expression. The NFA for this pattern would be



35.3 Formal definition of acceptance

Recall that a finite automaton M accepts a string w means there is a sequence of states $r_0, r_1, ..., r_n$ in Q where

1. $r_0 = q_0$
2. $\delta(r_i, w_{i+1}) = r_{i+1}$, for $i = 0, ..., n-1$ and
3. $r_n \in F$

How do we formally show a string w is accepted by M. Lets show that the (automaton on page 1) accepts the string 101.

We show that there must exists states $r_0, r_1, ..., r_3$ statisfying the above three conditions. We claim that the sequence A,B,E,G satisfies the three claims.

1. $A = q_0$

2. $\delta(A, 1) = B$
   $\delta(B, 0) = E$
   $\delta(E, 1) = G$

3. $G \in F$

Chapter 36

Discussion 4: More on non-deterministic finite automatas

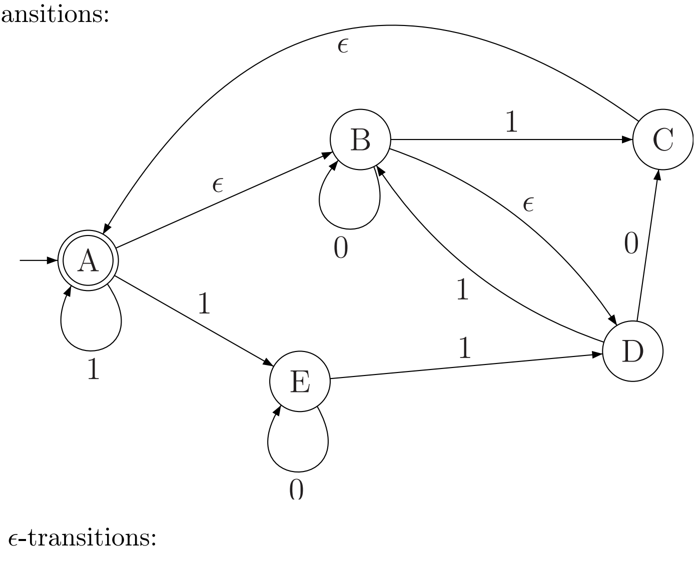
10 February 2008

Questions on homework 3?

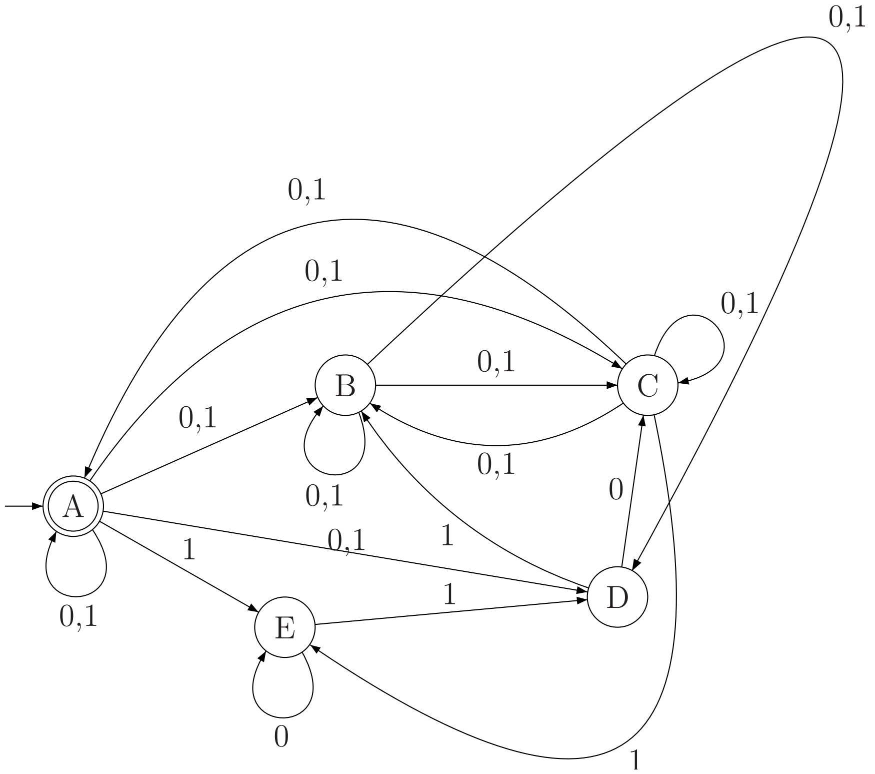
Any questions? Complaints, etc?

36.1 Computing ϵ-closure

An NFA with some ϵ-transitions:



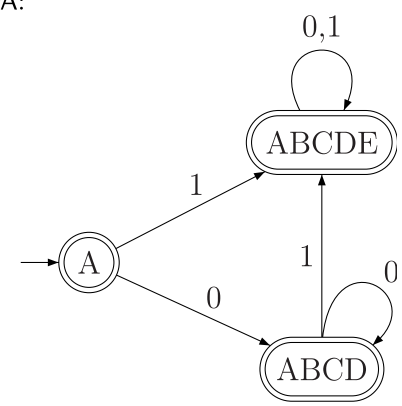
And after removing ϵ-transitions:



General rule: For every state Q and every $a \in \Sigma$, compute the set of all reachable states from Q when we feed just character a.

36.2 Subset Construction

The NFA above converted into a DFA:



Note that to convert an NFA to a DFA, here we first remove ϵ-transitions and then we apply subset construction. You could reverse the order of these two operations (like what Sipser does), but note that the new start state will be the ϵ-closure of old start state.

Chapter 37

Discussion 5: More on non-deterministic finite automatas

17 February 2009

Questions on homework 4?

Any questions? Complaints, etc?

37.1 Non-regular languages

37.1.1 $L(\mathtt{0}^n\mathtt{1}^n)$

Lemma 37.1.1 *The language* $L_1 = \left\{ \mathtt{0}^n\mathtt{1}^n \;\middle|\; n \geq 0 \right\}$ *is not regular.*

Proof: We remind the reader that we already saw that the language $L(\mathtt{a}^n\mathtt{b}^n) = \left\{ \mathtt{a}^n\mathtt{b}^n \;\middle|\; n \geq 0 \right\}$ is not regular. As such, assume for the sake of contradiction that L_1 is regular, and consider the homomorphism $h(\mathtt{0}) = \mathtt{a}$ and $h(\mathtt{1}) = \mathtt{b}$. Since regular languages are closed under homomorphism, and L_1 is regular, it follows that $h(L_1)$ is regular. However, $h(L_1)$ is the language

$$h(L_1) = \left\{ h(\mathtt{0})^n h(\mathtt{1})^n \;\middle|\; n \geq 0 \right\} = \left\{ \mathtt{a}^n\mathtt{b}^n \;\middle|\; n \geq 0 \right\} = L(\mathtt{a}^n\mathtt{b}^n),$$

which is a contradiction, since $L(\mathtt{a}^n\mathtt{b}^n)$ is not regular. ■

37.1.2 $L(\#\mathtt{a} + \#\mathtt{b} = \#\mathtt{c})$

Let $\Sigma = \{1, 2, 3\}$, consider the following language

$$L_2 = \left\{ x \;\middle|\; \#_{\mathtt{a}}(x) + \#_{\mathtt{b}}(x) = \#_{\mathtt{c}}(x) \right\},$$

where $\#_h(x)$ is the number of times the character h appears in x, for $h = \mathtt{a}, \mathtt{b}, \mathtt{c}$.

Direct proof

Lemma 37.1.2 *The language* $L_2 = \left\{ x \;\middle|\; \#_{\mathtt{a}}(x) + \#_{\mathtt{b}}(x) = \#_{\mathtt{c}}(x) \right\}$, *is not regular.*

Proof: Assume for the sake of contradiction, that L_2 is regular, and let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA that accepts it. For $i \geq 0$, let $q_i = \delta(q_0, \mathtt{a}^i)$ be the state M is in, after reading the string $\mathtt{a}^i$ (i.e., a string of length i made out of $\mathtt{a}$s). We claim that if $i \neq j$, then $q_i \neq q_j$. Indeed, If there are $i \neq j$ such that $q_i = q_j$, then we have that M accepts the string $\mathtt{c}^j$, if we start from q_j, since M accepts the string $\mathtt{a}^j\mathtt{c}^j$. But then, it must be that M accepts $\mathtt{a}^i\mathtt{c}^j$. Indeed, after M reads $\mathtt{a}^i$ it is in state $q_i = q_j$, and we know that it accepts $\mathtt{c}^j$, if we start from q_j. But this is a contradiction, since $\mathtt{a}^i\mathtt{c}^j \notin L_2$, for $i \neq j$.

This implies that M has an infinite number of states, which is of course impossible. ■

By closure properties

Here is another proof of Lemma 37.1.2.

Proof: Assume, for the sake of contradiction, that L_2 is regular. Then, since regular languages are closed under intersection, and the language $\mathtt{a}^*\mathtt{c}^*$ is regular, we have that $L_3 = L_2 \cap \mathtt{a}^*\mathtt{c}^*$ is regular. But L_3 is clearly the language

$$L_3 = \left\{ \mathtt{a}^n\mathtt{c}^n \,\middle|\, n \geq 0 \right\},$$

which is not regular. Indeed, if L_3 was regular then $f(L_3)$ would be regular (by closure under homomorphism), which is false by Lemma 37.1.1, where $f(\cdot)$ is the homomorphism mapping $f(\mathtt{a}) = \mathtt{0}$ and $f(\mathtt{b}) = \epsilon$ and $f(\mathtt{c}) = \mathtt{1}$. ■

37.1.3 Not too many as please

Lemma 37.1.3 *The language*

$$L_4 = \left\{ \mathtt{a}^n x \,\middle|\, n \geq 1, x \in \{\mathtt{a}, \mathtt{b}\}^*, \text{ and } x \text{ contains at most } 2n \text{ } \mathtt{a}\text{'s} \right\}..$$

is not regular.

We first provide a direct proof.

Proof: Assume for the sake of contradiction that L_4 is regular, and let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA accepting L_4. Let q_i be the state that M arrives to after reading the string $\mathtt{a}^i\mathtt{b}$. Now, the word $\mathtt{a}^i b \mathtt{a}^{2i} \in L_4$, and as such, $\delta(q_i, \mathtt{a}^{2i})$ is an accepting state. Similarly, $\mathtt{a}^i b \mathtt{a}^{2j} \notin L_4$ if $j > i$, since this string has too many as in its second part. As such $\delta(q_i, \mathtt{a}^{2j})$ is not accepting if $j > i$.

We claim that because of that $q_i \neq q_j$ for any $j > i$. Indeed, $\delta(q_i, \mathtt{a}^{2j})$ is not an accepting state, but $\delta(q_j, \mathtt{a}^{2j})$ is an accepting state. Thus $q_i \neq q_j$, but this implies that M has an infinite number of states, a contradiction. ■

Proof using the pumping lemma

Using the pumping lemma we can show that L_4 is not regular.

Proof: By contradiction, if L_4 is regular, then it must satisfy the pumping lemma. Let p be the pumping length guaranteed by pumping lemma. Looking at definition of L_4, the word $\mathtt{a}^p\mathtt{b}\mathtt{a}^{2p} \in L_4$. Now we must have this decomposition: $\mathtt{a}^p\mathtt{b}\mathtt{a}^{2p} = xyz$, where $|y| \neq 0$ and $|xy| \leq p$, such that for all $k \geq 0$, $s_k = xy^kz \in L_4$. But it is easy to see $s_0 \notin L$. Indeed, s_0 has a run of $p - |y| < p$ of as in its prefix, but after the b, it has a string of length $2p$ made out of b, which is not in L_4. A contradiction. ■

37.1.4 A Trick Example (Optional)

Consider the following language.

$$L_7 = \left\{ wxw^R \,\middle|\, w, x \in \{0,1\}^+ \right\}.$$

Is it regular or not? It seems natural to think that it is not regular. However, it is in fact regular. Indeed, L_7 is the set of all strings where the first and last character are the same, which is definitely a regular language.

Chapter 38

Discussion 6: Closure Properties

20 February 2008

Questions on homework 5?

Any questions? Complaints, etc?

38.1 Closure Properties

38.1.1 sample one

For character a and language L, we define:

$$L/\mathtt{a} = \{xsepx\mathtt{a} \in L\}$$

This operation preserves regularity: consider a DFA for L with set of final states F. Redefine F as follows:

$$F' = \left\{ s \,\middle|\, \delta(s, a) \in F\} \cup \{t \in F : \delta(t, a) = t\} = \{s : \delta(s, a) \in F \right\}$$

and see that this a DFA for $L/\mathtt{a}$.

38.1.2 sample two

Now consider this operation:

$$\min(L) = \left\{ x \,\middle|\, x \in L \text{ and no proper prefix of } x \text{ is in } L \right\}.$$

Again consider a DFA for L and remove all outgoing transitions from final states of that DFA to have a NFA for $\min(L)$.

38.1.3 sample three

Show that L is irregular:

$$L = \left\{ \mathtt{a}^n\mathtt{b}^m\mathtt{c}^r\mathtt{d}^s\mathtt{e}^t \,\middle|\, n + m - r = s + t \right\}$$

map $\mathtt{b}$ to $\mathtt{a}$ and $\mathtt{c}, \mathtt{d}$ to $\mathtt{e}$ to obtain $\left\{ \mathtt{a}^n\mathtt{e}^n \,\middle|\, n \geq 0 \right\}$ which we know is not regular.

Chapter 39

Discussion 7: Context-Free Grammars

March 3, 2009

Questions on homework or exam?

Any questions? Complaints, etc?

39.1 Context free grammar for languages with balance or without it

39.1.1 Balanced strings

Let $L_{\mathtt{a}=\mathtt{b}} = \left\{ \mathtt{a}^n \mathtt{b}^n \;\middle|\; n \geq 1 \right\}$. Here is a grammar for this language

$$\mathsf{S} \rightarrow \mathtt{a}\mathsf{S}\mathtt{b} \mid \epsilon.$$

39.1.2 Mixed balanced strings

Let $L_{\mathtt{a}=\mathtt{b}}^{\mathrm{mix}}$ be the language of all strings over $\{\mathtt{a}, \mathtt{b}\}$, with equal number of $\mathtt{a}$s and $\mathtt{b}$s, where the $\mathtt{a}$s and $\mathtt{b}$s might be mixed together.

We the following grammar for generating all strings that have the same numbers of a's and b's

$$\mathsf{S} \rightarrow \ \mathtt{a}\mathsf{S}\mathtt{b} \mid \mathtt{b}\mathsf{S}\mathtt{a} \mid \mathsf{SS} \mid \epsilon,$$

where S is the start variable.

■ `Advice To TA::` State the following lemma, and sketch its proof, but do not do the proof in the discussion section. Point the interested students to the class notes. `end`

Lemma 39.1.1 *We have $L(\mathcal{G}) = L_{\mathtt{a}=\mathtt{b}}^{\mathrm{mix}}$.*

Proof: It is easy to see that every string that $\mathcal{G}$ generates has equal number of $\mathtt{a}$'s and $\mathtt{b}$'s. As such, $L(\mathcal{G}) \subseteq L_{\mathtt{a}=\mathtt{b}}^{\mathrm{mix}}$.

We will use induction on the length of string $x \in L(\mathcal{G})$, $2n = |x|$. For $n = 0$ we can generate ϵ by $\mathcal{G}$. For $n = 1$, we can generate both $\mathtt{ab}$ and $\mathtt{ba}$ by $\mathcal{G}$.

Now for $n > 1$, consider a balanced string with length $2n$, $x = x_1 x_2 x_3 \cdots x_{2n} \in L_{\mathtt{a}=\mathtt{b}}^{\mathrm{mix}}$. Let $\#_c(y)$ be the number of appearances of the character c in the string y. Let $\alpha_i = \#_{\mathtt{a}}(x_1 \cdots x_i) - \#_{\mathtt{b}}(x_1 \cdots x_i)$. Observe that $\alpha_0 = \alpha_{2n} = 0$. If $\alpha_j = 0$, for some $1 < j < 2n$, then we can break x into two words $y = x_1 \ldots x_j$ and $z = x_{j+1} \ldots x_{2n}$ that are both balanced. By induction, $y, z \in L(\mathcal{G})$, and as such $\mathsf{S} \Rightarrow^* y$ and $\mathsf{S} \Rightarrow^* z$. This implies that

$$\mathsf{S} \Rightarrow \mathsf{SS} \Rightarrow^* yz = x.$$

Namely, $x \in L(\mathcal{G})$.

The remaining case is that $\alpha_j \neq 0$ for $j = 2, \ldots, 2n-1$. If $x_1 = \mathtt{a}$ then $\alpha_1 = 1$. As such, for all $j = 1, \ldots, 2n-1$, we must have that $\alpha_j > 0$. But then $\alpha_{2n} = 0$, which implies that $\alpha_{2n-1} = 1$. We conclude that $x_1 = \mathtt{a}$ and $x_{2n} = \mathtt{b}$. As such, $x_2 \ldots x_{2n-1}$ is a balanced word, which by induction is generated by $L(\mathcal{G})$. Thus, the x can be derived via $\mathsf{S} \rightarrow \mathtt{aSb} \Rightarrow^* \mathtt{a}x_2x_3 \ldots x_{2n-1}\mathtt{b} = x$. Thus, $x \in L(\mathcal{G})$.

The case $x_1 = \mathtt{b}$ is handled in a similar fashion, and implies that $x \in L(\mathcal{G})$ also in this case. We conclude that $L^{\mathrm{mix}}_{\mathtt{a}=\mathtt{b}} \subseteq L(\mathcal{G})$.

Thus $L^{\mathrm{mix}}_{\mathtt{a}=\mathtt{b}} = L(\mathcal{G})$. ■

39.1.3 Unbalanced pair

Consider the following language:

$$L_{\mathtt{a}\neq\mathtt{b}} = \left\{ \mathtt{a}^n\mathtt{b}^m \;\middle|\; n \neq m \text{ and } \; n, m \geq 0 \right\}.$$

If $n \neq m$ then either $n > m$ or $m > n$, therefore we can design this grammar by first starting with the basic grammar for when $n = m$, and then transition into making more a's or b's.

Let X be the non terminal representing "choosing" to generate more a's than b's and Y be the non-terminal for the other case. One grammar that generates $L_{\mathtt{a}\neq\mathtt{b}}$ will therefore be:

$$\mathsf{S} \rightarrow \mathsf{aSb} \mid \mathsf{aA} \mid \mathsf{bB}, \qquad \mathsf{A} \rightarrow \mathsf{aA} \mid \epsilon, \qquad \mathsf{B} \rightarrow \mathsf{bB} \mid \epsilon.$$

39.1.4 Balanced pair in a triple

Consider the language

$$L_4 = \left\{ \mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \;\middle|\; i = j \text{ or } \; j = k \right\}.$$

We can essentially combine two copies of the previous grammar (with one version that works on b and c) in order to create a grammar that generates L_2:

$$\mathsf{S} \rightarrow \mathsf{S}_{\mathsf{a}=\mathsf{b}}\mathsf{C} \mid \mathsf{AS}_{\mathsf{b}=\mathsf{c}}$$

$$\mathsf{S}_{\mathsf{a}=\mathsf{b}} \rightarrow \mathsf{aS}_{\mathsf{a}=\mathsf{b}}\mathsf{b} \mid \epsilon. \qquad \mathsf{S}_{\mathsf{b}=\mathsf{c}} \rightarrow \mathsf{bS}_{\mathsf{b}=\mathsf{c}}\mathsf{c} \mid \epsilon. \qquad \mathsf{A} \rightarrow \mathsf{Aa} \mid \epsilon \qquad \mathsf{C} \rightarrow \mathsf{Cc} \mid \epsilon.$$

Exercise 39.1.2 *Derive a CFG for the language* $L'_4 = \left\{ \mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \;\middle|\; i = j \text{ or } j = k \text{ or } i = k \right\}$.

39.1.5 Unbalanced pair in a triple

Now consider the related language

$$L_2 = \left\{ \mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \;\middle|\; i \neq j \text{ or } \; j \neq k \right\}.$$

We can essentially combine two copies of the previous grammar (with one version that works on b and c) in order to create a grammar that generates L_2:

$$\mathsf{S} \rightarrow \mathsf{S}_{\mathsf{a}\neq\mathsf{b}}\mathsf{C} \mid \mathsf{AS}_{\mathsf{b}\neq\mathsf{c}} \qquad \mathsf{S}_{\mathsf{a}\neq\mathsf{b}} \rightarrow \mathsf{aS}_{\mathsf{a}\neq\mathsf{b}}\mathsf{b} \mid \mathsf{aA} \mid \mathsf{bB}. \qquad \mathsf{S}_{\mathsf{b}\neq\mathsf{c}} \rightarrow \mathsf{bS}_{\mathsf{b}\neq\mathsf{c}}\mathsf{c} \mid \mathsf{bB} \mid \mathsf{cC}.$$

$$\mathsf{A} \rightarrow \mathsf{Aa} \mid \epsilon \qquad \mathsf{B} \rightarrow \mathsf{Bb} \mid \epsilon \qquad \mathsf{C} \rightarrow \mathsf{Cc} \mid \epsilon.$$

39.1.6 Anything but balanced

Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, and let Let $\overline{L_{\mathtt{a}=\mathtt{b}}} = \Sigma^* \setminus \left\{ \mathtt{a}^n\mathtt{b}^n \,\middle|\, n \geq 1 \right\}$.

The idea is that lets first generate all words that contain b in them, and then later the contain a. The grammar for this language is

$$\mathsf{S}_1 \rightarrow \mathsf{Z}\mathtt{b}\mathsf{Z}\mathtt{a}\mathsf{Z} \qquad \mathsf{Z} \rightarrow \mathtt{a}\mathsf{Z} \mid \mathtt{b}\mathsf{Z} \mid \epsilon.$$

Clearly $L(\mathsf{Z}) \subseteq \overline{L_{\mathtt{a}=\mathtt{b}}}$. The only words we miss, must have all their as before their bs. But these are all words of the form $\mathtt{a}^i\mathtt{b}^j$, where $i \neq j \geq 0$. But we already saw how to generate such words in Section 39.1.3. Putting everything together, we get the following grammar.

$$\begin{aligned}
\Rightarrow & \mathsf{S} \rightarrow \mathsf{S}_1 \mid \mathsf{S}_{\mathtt{a}\neq\mathtt{b}} \\
& \mathsf{S}_1 \rightarrow \mathsf{Z}\mathtt{b}\mathsf{Z}\mathtt{a}\mathsf{Z} \\
& \mathsf{Z} \rightarrow \mathtt{a}\mathsf{Z} \mid \mathtt{b}\mathsf{Z} \mid \epsilon \\
& \mathsf{S}_{\mathtt{a}\neq\mathtt{b}} \rightarrow \mathtt{a}\mathsf{S}_{\mathtt{a}\neq\mathtt{b}}\mathtt{b} \mid \mathtt{a}\mathsf{A} \mid \mathtt{b}\mathsf{B}, \\
& \mathsf{A} \rightarrow \mathtt{a}\mathsf{A} \mid \epsilon, \\
& \mathsf{B} \rightarrow \mathtt{b}\mathsf{B} \mid \epsilon.
\end{aligned}$$

39.2 Similar count

Consider the language

$$L = \left\{ w0^n \,\middle|\, w \in \{\mathtt{a}, \mathtt{b}\}^* \text{ and } \#_{\mathtt{a}}(w) = n \right\},$$

where $\#_{\mathtt{a}}(w)$ is the number of appearances of the character a in w. The grammar for this language is

$$\mathsf{S} \rightarrow \epsilon \mid \mathtt{b}\mathsf{S} \mid \mathtt{a}\mathsf{S}\mathtt{0}.$$

39.3 Inherent Ambiguity

In lecture, the following ambiguous grammar representing basic mathematical statements was discussed:

$$\mathsf{E} \rightarrow \mathsf{E} * \mathsf{E} \mid \mathsf{E} + \mathsf{E} \quad \mathsf{N} \rightarrow \mathtt{0}\mathsf{N} \mid \mathtt{1}\mathsf{N} \mid \mathtt{0} \mid \mathtt{1}.$$

The ambiguity caused because there is no inherent preference from combining expressions with $*$ over $+$ or vise versa. It was then fixed by introducing a preference :

$$\mathsf{E} \rightarrow \mathsf{E} * \mathsf{E} \mid \mathsf{T}, \qquad \mathsf{T} \rightarrow N * T \mid \mathsf{N} \qquad \mathsf{N} \rightarrow \mathtt{0}\mathsf{N} \mid \mathtt{1}\mathsf{N} \mid \mathtt{0} \mid \mathtt{1}.$$

However some languages are inherently ambiguous, no context free grammar without ambiguity can generate it.

Consider the following language:

$$L = \left\{ \mathtt{a}^n\mathtt{b}^n\mathtt{c}^k\mathtt{d}^k \,\middle|\, n, k \geq 1 \right\} \cup \left\{ \mathtt{a}^n\mathtt{b}^k\mathtt{c}^k\mathtt{d}^n \,\middle|\, n, k \geq 1 \right\}.$$

In other words, it is the language of $a^+b^+c^+d^+$ where either:

1. the number of a's equals the number of b's and the number c's equals the number of d's
2. the number of a's equals the number of d's and the number of b's equals the number of c's

One ambiguous grammar that generates it

$$\mathsf{S} \rightarrow \mathsf{X}\mathsf{Y} \mid \mathsf{Z}, \qquad X \rightarrow \mathtt{a}\mathsf{X}\mathtt{b} \mid \epsilon, \qquad \mathsf{Y} \rightarrow \mathtt{c}\mathsf{Y}\mathtt{d} \mid \epsilon, \qquad \mathsf{Z} \rightarrow \mathtt{a}\mathsf{Z}\mathtt{d} \mid \mathsf{T}, \qquad \mathsf{T} \rightarrow \mathtt{b}\mathsf{T}\mathtt{c} \mid \epsilon.$$

The reason why all grammars for this language must be ambiguous can be seen in strings of the form $\left\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n\mathtt{d}^n \;\middle|\; n \geq 1\right\}$. Any grammar needs some way of generating the string in a way that either the `a`'s and `b`'s are equal and the `c` and `d`'s are equal or the `a`'s and `d`'s are equal and the `b`'s and `c`'s are equal. When generating equal `a`'s and `b`'s, it must be still possible to have the same number of c's and d's. When generating equal `a`'s and d's , it must still be possible to have the same number of `b`'s and `c`'s. No matter what grammar is designed, any string of the form $\left\{a^nb^nc^nd^n \;\middle|\; n \geq 1\right\}$ must have at least two possible parse trees.

(This is of course only an intuitive explanation. A formal proof that any grammar for this language must be ambiguous is considerably more tedious and harder.)

39.4 A harder example

Consider the following language:

$$L = \left\{ xy \;\middle|\; x, y \in \{0,1\}^* \text{ where } |x| = |y|, \text{ and } x \neq y \right\}.$$

It should be clear that this language cannot be regular. However, it may not be obvious that we can in fact design a context free grammar for it. x and y are guaranteed to be different if, for some k, the kth character is 0 in x and 1 in y (or vise versa). It is important to notice that we should not try to build x and y separately as, in a CFG, we would have no way to enforce them being of the same length. Instead, we just remember that if the string is of length $2n$, the first n characters are considered x and the second n characters are y. Similarly, notice that we cannot choose k ahead of time for similar reasons.

So, consider the following string

$$w = x_1x_2\dots x_{k-1}\boxed{\mathsf{1}}x_{k+1}\dots x_ny_1y_2\dots y_{k-1}\boxed{\mathsf{0}}y_{k+1}\dots y_n \in L.$$

Then we can rewrite this string as follows

$$w = \overbrace{x_1x_2\dots x_{k-1}}^{k-1\text{ chars}}\boxed{\mathsf{1}}\overbrace{x_{k+1}\dots x_n}^{n-k\text{ chars}}\;\overbrace{y_1y_2\dots y_{k-1}}^{k-1\text{ chars}}\boxed{\mathsf{0}}\overbrace{y_{k+1}\dots y_n}^{n-k\text{ chars}}.$$

In particular, let $z_1z_2\dots z_{n-1} = x_{k+1}\dots x_ny_1y_2\dots y_{k-1}$. Then,

$$\begin{aligned} w &= \overbrace{x_1x_2\dots x_{k-1}}^{k-1\text{ chars}}\,\mathsf{1}\,z_1z_2\dots z_{n-1}\mathsf{0}\overbrace{y_{k+1}\dots y_n}^{n-k\text{ chars}}\\ &= \underbrace{\overbrace{x_1x_2\dots x_{k-1}}^{k-1\text{ chars}}\,\mathsf{1}\,\overbrace{z_1\dots z_{k-1}}^{k-1\text{ chars}}}_{=X}\;\underbrace{\overbrace{z_k\dots z_{n-1}}^{n-1-k+1\text{ chars}}\;\mathsf{0}\overbrace{y_{k+1}\dots y_n}^{n-k\text{ chars}}}_{=Y}. \end{aligned}$$

Now, X is a word of odd length with `1` in the middle (and we definitely know how to generate this kind of words using context free grammars). And Y is a word of odd length, with `0` in the middle. In particular, any word of L can be written as either XY or YX, where X and Y are as above. We conclude, that the grammar for this language is

$$\mathsf{S} \rightarrow \mathsf{XY} \mid \mathsf{YX} \qquad \mathsf{X} \rightarrow \mathsf{DXD} \mid \mathtt{1} \qquad \mathsf{Y} \rightarrow \mathsf{DYD} \mid \mathtt{0} \qquad \mathsf{D} \rightarrow \mathtt{0} \mid \mathtt{1}.$$

Chapter 40

Discussion 8: From PDA to grammar

5 March 2008

Questions on homework 7?

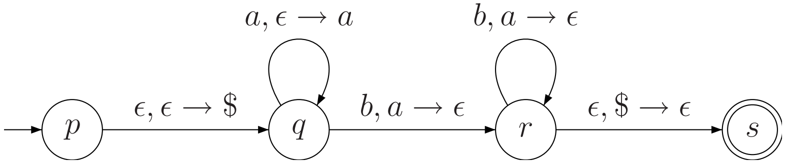
Any questions? Complaints, etc?

40.1 Converting PDA to a Grammar

Note that the following PDA is designed such that it has the required properties for converting into a grammar:

1. It has a single final state.
2. It empties the stack before accepting.
3. Each transition just pushes one symbol or pops one symbol and not both or none.

Note that its language is $L = \{a^n b^n : n \geq 1\}$.


$a, \epsilon \rightarrow a$
$b, a \rightarrow \epsilon$
p
$\epsilon, \epsilon \rightarrow \$$
q
$b, a \rightarrow \epsilon$
r
$\epsilon, \$ \rightarrow \epsilon$
s

The equivalent grammar is (note that in this case we can simplify it to get our familiar grammar for L):

$A_{ps} \rightarrow \epsilon A_{qr} \epsilon$ A_{ps} is the start state
$A_{qr} \rightarrow a A_{qq} b$
$A_{qr} \rightarrow a A_{qr} b$
$A_{pp} \rightarrow \epsilon$
$A_{pp} \rightarrow \epsilon$
$A_{qq} \rightarrow \epsilon$
$A_{rr} \rightarrow \epsilon$
$A_{ss} \rightarrow \epsilon$
$A_{xyz} \rightarrow A_{xz} A_{zy}$ for all $x, y, z \in \{p, q, r, s\}$ (64 rules)

Chapter 41

Discussion 9: Chomsky Normal Form and Closure Properties

12 March 2008

Questions on homework 8?

Any questions? Complaints, etc?

41.1 Chomsky Normal Form

We want to write this grammar in CNF:

$$\begin{aligned} S &\rightarrow ASA \mid aB \\ A &\rightarrow B \mid S \\ B &\rightarrow \epsilon \end{aligned}$$

After removing nullable variables and unreacheables:

$$\begin{aligned} S &\rightarrow AS \mid SA \mid ASA \mid a \\ A &\rightarrow S \end{aligned}$$

Now we remove unit rules:

$$\begin{aligned} S &\rightarrow AS \mid SA \mid ASA \mid a \\ A &\rightarrow AS \mid SA \mid ASA \mid a \end{aligned}$$

returning back ϵ to the language:

$$\begin{aligned} S &\rightarrow AS \mid SA \mid ASA \mid a \mid \epsilon \\ A &\rightarrow AS \mid SA \mid ASA \mid a \end{aligned}$$

finally:

$$\begin{aligned} S &\rightarrow AS \mid SA \mid AT_{SA} \mid a \mid \epsilon \\ A &\rightarrow AS \mid SA \mid AT_{SA} \mid a \\ T_{SA} &\rightarrow SA \end{aligned}$$

41.2 Closure Properties

CFL's are closed under substitution, union, concatenation, Kleene closure and positive closure, reversal, intersection with a regular language, and inverse homomorphism.

Consider $L = \{a^n b^{3n} c^{2n} : n \geq 0\}$. Assume L is CFL. Consider this homomorphism: $h(0) = a, h(1) = bbb, h(2) = cc$. $h^{-1}(L) = \{0^n 1^n 2^n : n \geq 0\}$ and must be CFL, but we know that this is not a CFL. So by contradiction we have shown that L is not a CFL.

Chapter 42

Discussion 10: Pumping Lemma for CFLs

26 March 2008

Questions on homework 9?

Any questions? Complaints, etc?

42.1 Pumping Lemma for CFLs

Show that the following is not a CFL:

$$L = \left\{ ww \,\middle|\, w \in \{a,b\}^* \right\}.$$

Assume L is a CFL with pumping length p. Consider $s = a^p b^p a^p b^p \in L$. Let $s = xyzwv$ be the decomposition guaranteed by pumping lemma. We have several cases:

1. yzw is in the first run of a's: $xy^2zw^2v \notin L$ (why?).
2. yzw is between the first run of a's and b's: $xzv \notin L$ (why?)
3. yzw is in the first run of b's: $xy^2zw^2v \notin L$ (why?).
4. yzw is between the first run of b's and second run of a's: $xzv \notin L$ (why?)
5. yzw is in the second run of a's: $xy^2zw^2v \notin L$ (why?).
6. yzw is between the second run of a's and b's: $xzv \notin L$ (why?)
7. yzw is in the second run of b's: $xy^2zw^2v \notin L$ (why?).

Questions on Practice Midterm

Chapter 43

Discussion 11: High-level TM design

7 April 2008

43.1 Questions on homework?

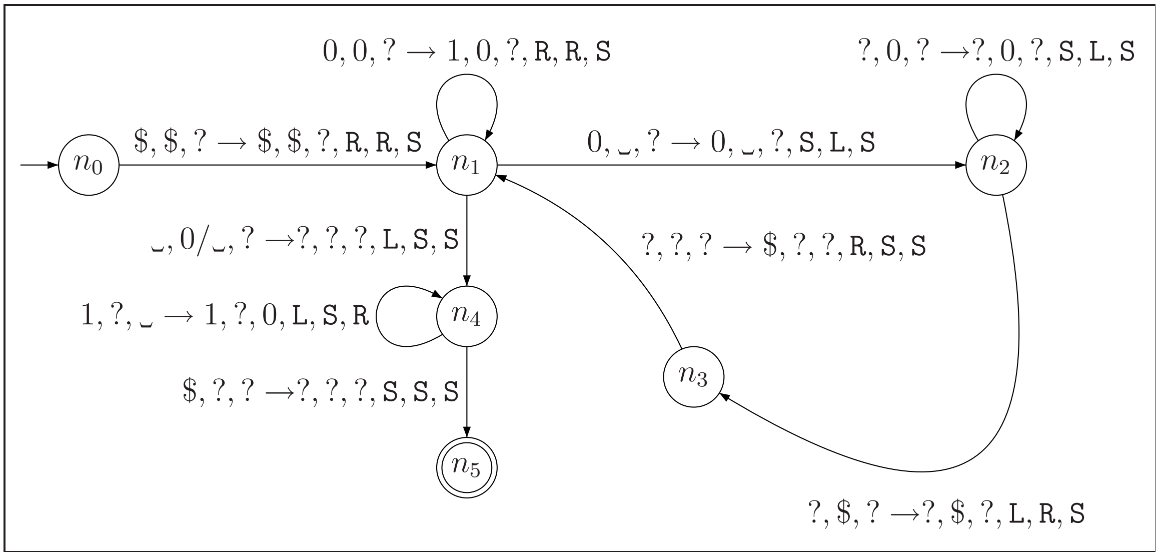
Any questions? Complaints, etc?

43.2 High-level TM design

43.2.1 Modulo

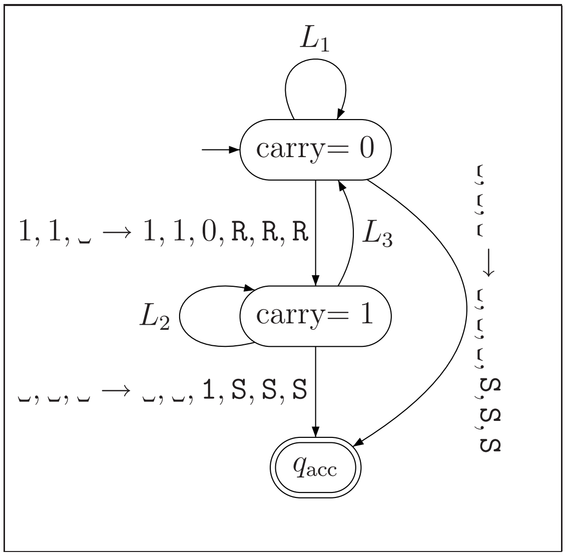
Question 43.2.1 *Design a TM that given* $\$0^{\mathsf{a}}$ *on tape* ✇$_1$ *and* $\$0^{\mathsf{b}}$ *on tape* ✇$_2$*, writes* $0^{a \bmod b}$ *on tape* ✇$_3$*.*

Solution: The idea is to check off every b characters from ✇$_1$ and copy the remaining characters to ✇$_2$. We assume that ␣ is the blank symbol on tapes and S indicates that the head remains stationary.


0, 0, ? → 1, 0, ?, R, R, S
?, 0, ? →?, 0, ?, S, L, S
$, $, ? → $, $, ?, R, R, S
0, ␣, ? → 0, ␣, ?, S, L, S
n0
n1
n2
␣, 0/␣, ? →?, ?, ?, L, S, S
?, ?, ? → $, ?, ?, R, S, S
1, ?, ␣ → 1, ?, 0, L, S, R
n4
n3
$, ?, ? →?, ?, ?, S, S, S
n5
?, $, ? →?, $, ?, L, R, S

The basic idea is that immediately after we move the head of ✇$_2$ back to the beginning of the tape (state n_2), we write a \$ on the first tape (i.e., transition from n_3 to n_1). Thus, conceptually, every time this loop is being performed a block of b charactere of 0 are being chopped of ✇$_1$.

To use this box as a template in our future designs (less and more like Macros in C++), we name this **Mod**(✇$_1$, ✇$_2$, ✇$_3$). ■


L_1
carry= 0
$1, 1, \text{␣} \rightarrow 1, 1, 0, \mathtt{R}, \mathtt{R}, \mathtt{R}$
L_3
L_2
carry= 1
$\text{␣}, \text{␣}, \text{␣} \rightarrow \text{␣}, \text{␣}, 1, \mathtt{S}, \mathtt{S}, \mathtt{S}$
$\text{␣}, \text{␣}, \text{␣} \rightarrow \text{␣}, \text{␣}, \text{␣}, \mathtt{S}, \mathtt{S}, \mathtt{S}$
q_{acc}

L_1 :
$0, 0, \text{␣} \rightarrow 0, 0, 0, \mathtt{R}, \mathtt{R}, \mathtt{R}$
$1, 0, \text{␣} \rightarrow 1, 0, 1, \mathtt{R}, \mathtt{R}, \mathtt{R}$
$0, 1, \text{␣} \rightarrow 0, 1, 1, \mathtt{R}, \mathtt{R}, \mathtt{R}$
$\text{␣}, 0, \text{␣} \rightarrow \text{␣}, 0, 0, \mathtt{S}, \mathtt{R}, \mathtt{R}$
$\text{␣}, 1, \text{␣} \rightarrow \text{␣}, 1, 1, \mathtt{S}, \mathtt{R}, \mathtt{R}$
$0, \text{␣}, \text{␣} \rightarrow 0, \text{␣}, 0, \mathtt{R}, \mathtt{S}, \mathtt{R}$
$1, \text{␣}, \text{␣} \rightarrow 1, \text{␣}, 1, \mathtt{R}, \mathtt{S}, \mathtt{R}$

L_2 :
$1, 0, \text{␣} \rightarrow 1, 0, 0, \mathtt{R}, \mathtt{R}, \mathtt{R}$
$0, 1, \text{␣} \rightarrow 0, 1, 0, \mathtt{R}, \mathtt{R}, \mathtt{R}$
$1, 1, \text{␣} \rightarrow 1, 1, 1, \mathtt{R}, \mathtt{R}, \mathtt{R}$
$\text{␣}, 1, \text{␣} \rightarrow \text{␣}, 1, 0, \mathtt{S}, \mathtt{R}, \mathtt{R}$
$1, \text{␣}, \text{␣} \rightarrow 1, \text{␣}, 0, \mathtt{R}, \mathtt{S}, \mathtt{R}$

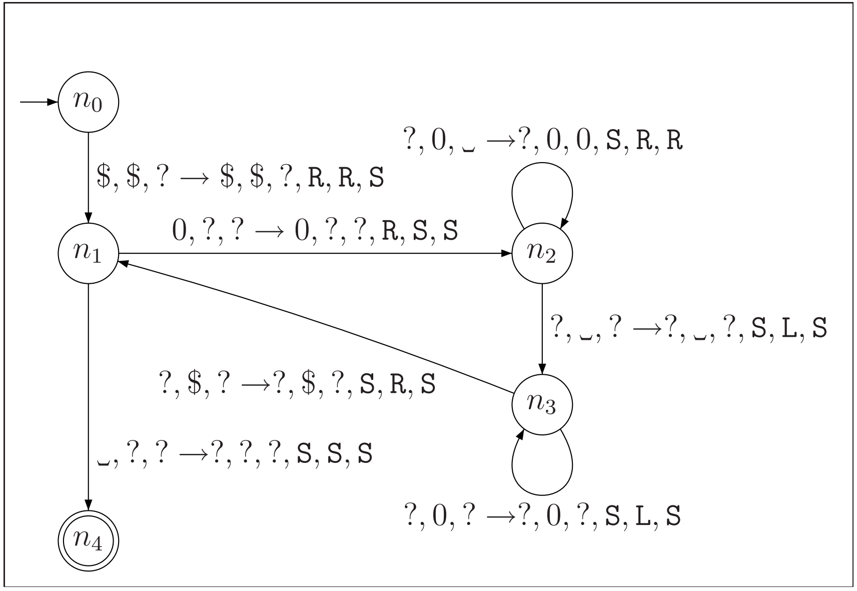
L_3 :
$0, 0, \text{␣} \rightarrow 0, 0, 1, \mathtt{R}, \mathtt{R}, \mathtt{R}$
$\text{␣}, 0, \text{␣} \rightarrow \text{␣}, 0, 1, \mathtt{S}, \mathtt{R}, \mathtt{R}$
$0, \text{␣}, \text{␣} \rightarrow 0, \text{␣}, 1, \mathtt{R}, \mathtt{S}, \mathtt{R}$

Figure 43.1: The TM for the binary addition.

43.2.2 Multiplication

Question 43.2.2 *Design a TM that given $\$0^a$ on tape ✇$_1$ and $\$0^b$ on tape ✇$_2$, writes $\mathtt{0}^{ab}$ on tape ✇$_3$.*

Solution: The idea is to attach a copies of tape ✇$_2$ at the end of tape ✇$_3$.


n_0
$\$, \$, ? \rightarrow \$, \$, ?, \mathtt{R}, \mathtt{R}, \mathtt{S}$
$?, 0, \text{␣} \rightarrow ?, 0, 0, \mathtt{S}, \mathtt{R}, \mathtt{R}$
n_1
$0, ?, ? \rightarrow 0, ?, ?, \mathtt{R}, \mathtt{S}, \mathtt{S}$
n_2
$?, \text{␣}, ? \rightarrow ?, \text{␣}, ?, \mathtt{S}, \mathtt{L}, \mathtt{S}$
$?, \$, ? \rightarrow ?, \$, ?, \mathtt{S}, \mathtt{R}, \mathtt{S}$
n_3
$\text{␣}, ?, ? \rightarrow ?, ?, ?, \mathtt{S}, \mathtt{S}, \mathtt{S}$
$?, 0, ? \rightarrow ?, 0, ?, \mathtt{S}, \mathtt{L}, \mathtt{S}$
n_4

To use this box as a template in our future designs (less and more like Macros in C++), we name this **Mult**(✇$_1$, ✇$_2$, ✇$_3$). ■

43.2.3 Binary Addition

Question 43.2.3 *Design a TM that given w_1^R on tape ✇$_1$ and w_2^R on tape ✇$_2$, writes w_3^R on tape ✇$_3$, where $w_1, w_2, w_3 \in \{0,1\}^*$ and w_3 is the binary addition of w_1 and w_2.*

Thus, if ✇$_1$ = `01` (i.e., this is the number $\mathtt{10}_2 = 2$ and ✇$_2$ = `1011` (i.e., this is the binary number $\mathtt{1101}_2 = 13$ then the output should be `1111` (which is 15).

```
copy from beginning till \$ from ✇$_1$ to ✇$_2$
copy from \$ on tape ✇$_1$ till end to ✇$_3$

/*** ✇$_2$ $= 0^a$ and ✇$_2$ $= 0^b$ ***/

CLEAR(✇$_1$)
CLEAR(✇$_7$)
Mod(✇$_2$, ✇$_3$, ✇$_5$)
/*** ✇$_5$ $= 0^{a \bmod b}$ ***/

do
    if EQ($t_7, t_5$) then accept.
    if GREQ(✇$_1$, ✇$_3$) then reject.
    add one 0 at the end of ✇$_1$
    COPY(✇$_1$, $t_4$)
    Mult(✇$_1$, $t_4$, $t_6$)
    Mod($t_6$, ✇$_3$, $t_7$)
while true
```

Figure 43.2: The algorithm for $c^2 = a \bmod b$.

Solution: We sum starting from the least significant digit, the normal procedure. See Figure 43.1. Here, a transition of the form $0, 1, \llcorner\!\lrcorner \rightarrow 0, 1, 0, \mathtt{R}, \mathtt{R}, \mathtt{R}$ stands for the situation where the TM reads 0 on the first tape, 1 on the second tape and ␣ on the third tape, next it writes 0, 1 and 0 to these three tapes respectively, and move the three heads to the right. ■

43.2.4 Quadratic Remainder

Question 43.2.4 *Design a TM that given $0^a\$0^b$ leaves only 0^c on the tape where $c \in \mathbb{N}_0$ is the smallest number such that $c^2 \equiv a \bmod b$. If such a c does not exist, the TM must reject.*

Solution: To solve this problem we assume that we have some other macros in addition to those we have built already.

- **CLEAR**(t): write ␣ on all the non-␣ characters of the tape t till it encounters a ␣. Then returns the head to the beginning of t.
- **EQ**(✇$_1$, ✇$_2$): checks whether the number of non-space characters on ✇$_1$ and ✇$_2$ are the same.
- **GREQ**(✇$_1$, ✇$_2$): checks whether the length of string of tape ✇$_1$ is more than or equal to that of ✇$_2$.
- **COPY**(✇$_1$, ✇$_2$): copies content of tape ✇$_1$ to tape ✇$_2$.

Note that our TM just needs to check for c among $\{0, 1, 2 \cdots, b-1\}$. We should serialize our macros in the following way (instead of a diagram, we write pseudo-code which is more readable). We use 7 tapes. Figure 43.2 depicts the resulting TM. ■

43.3 MTM

Question 43.3.1 *Show that an* **MTM** *(an imaginary more powerful TM) whose head can read the character under the head and the character to the left of the head (if such a character does not exist, it will read a ␣, i.e. blank character) and can just rewrite the character under the head, is equivalent to a normal TM.*

Solution: First of all it is obvious that an **MTM** can simulate a TM since it can ignore the extra information that it can read using its special head.

Now observe that a TM can simulate an **MTM** this way: For making a move using the transition function of **MTM**, the TM that simulates it must read the character under the head (which a normal TM can) and the character to the left of head (which a normal TM can't). What the simulating TM does is that it remembers the current state of **MTM** in its states (note that we have done several times, this kind of "remembering finite amount of information inside states by redefining states, e.g. extending them to tuples" in class), brings the head to the left and reads that character and remembers it inside its states, moves the head to the right, so now head is in its original place and our TM knows the missed character and can perform the correct move using **MTM**'s transition function. ■

Chapter 44

Discussion 12: Enumerators and Diagonalization

14 April 2009

Questions on homework 8?

Any questions? Complaints, etc?

44.1 Cardinality of a Set

For a finite set X, we denote by $|X|$ the ***cardinality*** of X; that is, the number of elements in A.

Definition 44.1.1 For two arbitrary sets (maybe infinite) X and Y, we have $|X| \leq |Y|$, iff there exists an injective mapping $f : X \rightarrow Y$.

Definition 44.1.2 Two arbitrary sets (maybe infinite) X and Y, are of the same ***cardinality*** (i.e., same "size") if $|A| = |B|$. Namely, there exists an *injective* and ***onto*** mapping $f : X \rightarrow Y$.

Observation 44.1.3 *For two sets X and Y, if $|X| \leq |Y|$ and $|Y| \leq |X|$ then $|X| = |Y|$.*

For $\mathbb{N}$, the set of all natural numbers, we define $|\mathbb{N}| = \aleph_0$. Any set X, with $|X| \leq \aleph_0$, is referred to as a ***countable*** set.

Claim 44.1.4 *For any set A, we have $|X| < |\mathbb{P}(X)|$. That is, $|X| \leq |\mathbb{P}(X)|$ and $|\mathbb{P}(X)| \neq |X|$.*
(Here $\mathbb{P}(X)$ is the power set of X.)

Proof: It is easy to verify that $|X| \leq |\mathbb{P}(X)|$. Indeed, consider the mapping $h(x) = \{x\} \in \mathbb{P}(X)$, for all $x \in X$.

So, assume for the sake of contradiction that $|X| = |\mathbb{P}(X)|$, and let f be a one-to-one and onto mapping from X onto $\mathbb{P}(X)$. Next, consider the set $B = \left\{ x \in X \;\middle|\; x \notin f(x) \right\}$.

Now, consider element $b = f^{-1}(B)$, and consider the question of whether it is a member of the set B or not. Now if $f^{-1}(B) = b \in B$, then by the definition of B, we hvae $bf^{-1}(B) = b \notin B$. Similarly, if $f^{-1}(B) = b \notin B$, then by definition of B, we have $f^{-1}(B) = b \in B$.

A contradiction. We conclude that our assumption that f exists (since X and $\mathbb{P}(X)$ have the same cardinality) is false. We conclude that $|X| \neq |\mathbb{P}(X)|$. ■

Definition 44.1.5 An ***enumerator*** `T` for a language L is a Turing Machine that writes out a list of all strings in L. It has no input tape, only an output tape on which it prints the strings, with some separator character (say, #) printed between them.

The strings can be printed in any order and the enumerator is allowed to print duplicates of a string it already printed. However, sooner or later all strings in L must be printed eventually by `T`. Naturally, all the strings printed by `T` are in L.

44.2 Rationals are enumerable

Consider the set of rational numbers

$$\mathbb{Q} = \left\{ a/b \,\middle|\, a \in \mathbb{Z}, b \in \mathbb{N}, b \neq 0, \text{ and } a, b \text{ are relatively prime} \right\}.$$

We remind the reader that two natural numbers a and B are ***relatively prime*** (or ***coprime***) if they have no common factor other than 1 or, equivalently, if their greatest common divisor is 1. Thus 2 and 3 are coprime, but 4 and 6 are not coprime. Thus, although $2/3 = 4/6$, we will consider only the representation $2/3$ to be in the set $\mathbb{Q}$.

We show that this set is enumerable by giving the pseudo-code for an enumerator for it.

```
EnumerateRationals
  for $i = 1 \dots \infty$ do
      for $x = 0 \dots i - 1$ do
          $y = i - x$
          if $x, y$ are relatively prime then
              print $x/y$ onto the tape followed by #
              print $-x/y$ onto the tape followed by #.
```

It is obvious that every rational number will be enumerated at some point. Any rational number is of the form a/b and as such when $i = a + b$ an d $y = b$ it will enumerate this rational number.

It helps to picture this as travelling along each line $x + y = i$.

44.3 Counting all words

Consider a finite alphabet Σ, and consider the problem of enumerating all words in Σ^*. That is, we want to come up with a way to be able to compute the ith word in Σ^* (let denote it by w_i). We want to do it in such a way that given a word w we can compute the i such that $w_i = w$, and similarly, given i we can compute w_i.

To this end, let Σ_i be all the words in Σ^* of length exactly i. Clearly, $|\Sigma_i| = |\Sigma|^i$. We sort the words inside Σ_i lexicographically. As such, we can now list all the words in Σ^**, by first listing the words in $\Sigma_0 = \{\epsilon\}$, $\Sigma_1 = \Sigma$, and so on.

For example, for $\Sigma = \{\texttt{a}, \texttt{b}\}$, we get the following enumeration of the words of Σ^*:

$$\underbrace{\overbrace{\epsilon}^{=w_1}}_{\Sigma_0}, \underbrace{\overbrace{\texttt{a}}^{=w_2}, \overbrace{\texttt{b}}^{=w_3}}_{\Sigma_1}, \underbrace{\overbrace{\texttt{aa}}^{=w_4}, \overbrace{\texttt{ab}}^{=w_5}, \overbrace{\texttt{ba}}^{=w_6}, \overbrace{\texttt{bb}}^{=w_7}}_{\Sigma_2}, \underbrace{\overbrace{\texttt{aaa}}^{=w_8}, \overbrace{\texttt{aab}}^{=w_9}, \overbrace{\texttt{aba}}^{w_{10}}, \overbrace{\texttt{abb}}^{w_{11}}, \overbrace{\texttt{baa}}^{w_{12}}, \overbrace{\texttt{bab}}^{w_{13}}, \overbrace{\texttt{bba}}^{w_{14}}, \overbrace{\texttt{bbb}}^{w_{15}}}_{\Sigma_3}, \dots$$

It is now easy to verify that given $w \in \Sigma^*$, we can figure out the i such that $w_i = w$. Similarly, given i, we can output the word w_i.

We just demonstrated that there is a one-to-one and onto mapping from $\mathbb{N}$ to Σ^*, and we can conclude the following.

Lemma 44.3.1 *For a finite alphabet Σ, the set Σ^* is countable.*

44.4 Languages are not countable

Let

$$\mathsf{L}_{\mathrm{all}} = \left\{ L \;\middle|\; L \text{ is some language, and } L \subset \{\mathtt{a}, \mathtt{b}\}^* \right\}.$$

Claim 44.4.1 *The set* $\mathsf{L}_{\mathrm{all}}$ *is not countable.*

Proof: We show that this set is not countable by using a diagonalization argument. Assume for the sake of contradiction that $\mathsf{L}_{\mathrm{all}}$ is countable. Then there exists a one-to-one and onto mapping $g : \mathbb{N} \rightarrow \mathsf{L}_{\mathrm{all}}$. Let $L_i = g(i)$, for all i.

We can think about this mapping as follows. We create an infinite table where the ith row is the language of L_i. We also enumerate the columns, where the ith column is the ith word in Σ^* (use the above enumeration scheme. We write 1 in the ith row and jth column of this table if w_j is in the language L_i.

Consider the diagonal language of this table:

	w_1	w_2	w_3	w_4	$\ldots$
L_1	**1**	1	0	0	$\ldots$
L_2	0	**1**	0	1	$\ldots$
L_3	1	0	**1**	1	$\ldots$
L_4	0	1	0	**0**	$\ldots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Formally, $L_d = \left\{ w_i \;\middle|\; w_i \in L_i, i \geq 0 \right\}$.

Let $\overline{L_{\mathrm{diag}}}$ be the complement of L_d. Clearly, $\overline{L_{\mathrm{diag}}}$ is well defined and $\overline{L_{\mathrm{diag}}} \in \mathsf{L}_{\mathrm{all}}$. But then, there must exist a k such that $L_k = g(k) = \overline{L_{\mathrm{diag}}}$.

So consider the kth row in this table (i.e., this is the row that corresponds to the language L_k). We consider if the word $w_k \in L_k$. If $w_k \in L_k = \overline{L_{\mathrm{diag}}}$ then the entry (k, k) in the table must be 1. But in that case $w_k \in L_d$, and as such $w_k \notin \overline{L_{\mathrm{diag}}}$. This is impossible.

The other possibility is that $w_k \notin L_k = \overline{L_{\mathrm{diag}}}$, then the entry at position (k, k) in the table must be 0. But in that case, the $w_k \notin L_d$, but then such $w_k \in \overline{L_{\mathrm{diag}}}$, which is again impossible.

A contradiction. We conclude that our assumption that $\mathsf{L}_{\mathrm{all}}$ is countable is false. ■

Chapter 45

Discussion 13: Reductions

16 April 2008

Questions on homework 12?

Any questions? Complaints, etc?

45.1 Easy Reductions

Question: Reduce A_{TM} to L:

$$L = \left\{ M \;\middle|\; M \text{ accepts } 00 \text{ and doesn't accept } 11 \right\}.$$

Solution: Let N be a decider for L. To decide $\langle M, w\rangle \in A_{\mathrm{TM}}$, build a new TM M' (with input x) that simulates M on w and if M accepts, and if x is 00, accepts (how can it verify this?), otherwise rejects. Now observe that:

$$\langle M, w\rangle \in \mathrm{A}_{\mathsf{TM}} \iff M' \in L$$

Lefthand side can be decided using N.

Question: Reduce A_{TM} to L:

$$L = \{M : L(M), \text{ and } \overline{L(M)} \text{ are both infinite}\}$$

Solution: Let N be a decider for L. To decide $\langle M, w\rangle \in A_{\mathrm{TM}}$, build a new TM M' (with input x) that simulates M on w and if M accepts, and if $|x|$ is even, accepts (how can it verify this?), otherwise rejects. Now observe that:

$$\langle M, w\rangle \in \mathrm{A}_{\mathsf{TM}} \iff M' \in L$$

Lefthand side can be decided using N.

Chapter 46

Discussion 14: Reductions

29 April 2009

Questions on homework ?

Any questions? Complaints, etc?

46.1 Undecidability and Reduction

(Q1) Prove that the language $\mathrm{ODD}_{\mathsf{TM}} = \left\{ \mathtt{T} \;\middle|\; \mathsf{L}(\mathtt{T}) \text{ has odd length strings} \right\}$ is undecidable.

Solution:

Proof: We assume, for the sake of contradiction, that $\mathrm{ODD}_{\mathsf{TM}}$ is decidable. Given $\langle \mathtt{T}, w\rangle$, consider the following procedure.

```
Z(x):
    if $x =$ ab then
        Accept
    $r \leftarrow$ Simulate T on $w$
    if $r =$Accept then Accept

    Reject.
```

Thus $\mathsf{L}(\mathtt{Z}) = \Sigma^*$ (which includes odd length strings) if $\mathtt{T}$ accepts w, and $\mathsf{L}(\mathtt{Z}) = \{\mathtt{ab}\}$ (a set of even length strings) otherwise.

We reduce $\mathrm{A}_{\mathsf{TM}}\mathsf{TM}$ to $\mathrm{ODD}_{\mathsf{TM}}$.

Suppose **isOdd**(T) is a decider for $\mathrm{ODD}_{\mathsf{TM}}$. We build the following decider for A_{TM}.

```
decider$_{10}$-$\mathrm{A}_{\mathsf{TM}}$ $(\mathtt{T}, w)$:
    $N \leftarrow$ compute the encoding of $\mathtt{Z}(x)$ above;
    return isOdd($N$)
```

Clearly, this is a decider for A_{TM}, which is impossible. We conclude our assumption is false, and thus $\mathrm{ODD}_{\mathsf{TM}}$ is not decidable. ■

(Q2) Prove that the language $\mathrm{SUBSET}_{\mathsf{TM}} = \left\{ \langle M, N\rangle \;\middle|\; L(M) \subseteq L(N) \right\}$ is undecidable. Hint: Reduce $\mathsf{EQ}_{\mathsf{TM}} = \left\{ \langle M, N\rangle \;\middle|\; L(M) = L(N) \right\}$ to $\mathrm{SUBSET}_{\mathsf{TM}}$ for this purpose.

Solution:

Assume, for the sake of contradiction, that **isSubset**(M, N) is a decider for $\mathrm{SUBSET}_{\mathsf{TM}}$. Then, we have the following decider for $\mathsf{EQ}_{\mathsf{TM}}$.

```
DeciderEQTM$(M, N)$:
    if isSubset$(M, N)$ and isSubset$(N, M)$ then
        return Accept
    else
        return Reject.
```

However, we already know that $\mathsf{EQ}_{\mathsf{TM}}$ is undecidable. A contradiction.

(Q3) Assume that the languages L_1 and L_2 are recognizable languages, where $L_1 \cup L_2 = \Sigma^*$. Now, assume you have a decider for the language $L_1 \oplus L_2$. Show that L_1 is decidable.

Solution:

Let $\mathbf{ORAC}_{xor}$ be a decider for $L_1 \oplus L_2$ and let $\mathtt{T}_1$ (resp. $\mathtt{T}_2$) be a machine that recognizes L_1 (resp. L_2).

```
$\mathbf{Decider}_1(w)$
    $r \leftarrow$ Simulate $\mathbf{ORAC}_{xor}$ on $w$.
    $x_1$ = start a simulation of $\mathtt{T}_1$ on $w$
    $x_2$ = start a simulation of $\mathtt{T}_2$ on $w$
    while (true)
        Advance $x_1$ and $x_2$ by one step
        if $x_1$ accepts then accept
        if $x_1$ rejects then reject
        if $x_2$ accepts then
            return not $r$
        if $x_2$ rejects then
            return $r$
```

‘

Since both L_1 and L_2 are recognizable, and $L_1 \cup L_2 = \Sigma^*$, it follows that one of the two simulations x_1 and x_2 must stops. This implies that the above procedure is indeed a decider. It is now an easy case analysis to verify that this is indeed a decider for L_1.

Indeed, if x_1 accepts than $w \in L_1$ and we are done. Similarly, if x_1 rejects than $w \notin L_1$ and we are done.

If x_2 accepts, but $r =$**reject** this implies that $w \in L_2$ and $w \notin L_1 \oplus L_2$. Namely, it must be that $w \in L_1$ and we accept.

If x_2 accepts, but $r =$**accept** this implies that $w \in L_2$ and $w \in L_1 \oplus L_2$. Namely, it must be that $w \notin L_1$ and we reject.

If x_2 rejects, but $r =$**reject** this implies that $w \notin L_2$ and $w \notin L_1 \oplus L_2$. This implies that $w \notin L_1 \cup L_2$, which is impossible in our case (so this case never happens.)

If x_2 rejects, but $r =$**accept** this implies that $w \notin L_2$ and $w \in L_1 \oplus L_2$. This implies that $w \in L_1$, and we accept.

(Q4) Let $\mathsf{EQ}_{\mathsf{TM}} = \left\{ \langle M, N \rangle \;\middle|\; \mathsf{L}(M) = \mathsf{L}(N) \right\}$. Reduce A_{TM} to $\mathsf{EQ}_{\mathsf{TM}}$ as another way to prove that $\mathsf{EQ}_{\mathsf{TM}}$ is undecidable

Solution:

For a given $\langle \mathtt{T}, w \rangle$, consider $\mathtt{T}_w$ defined as follows

```
$\mathtt{T}_w(y)$ :
    if $y \neq w$ then
        reject
    $r \leftarrow$Simulate $\mathtt{T}$ on $w$
    return $r$
```

And, consider N_w defined as follows:

```
$N_w(y)$ :
    if $y = w$ then
        accept
    else
        reject
```

Let $\mathbf{decider}_{EQ}$ be a decider for $\mathsf{EQ}_{\mathsf{TM}}$. We can design a decider for A_{TM} as follows using $\mathtt{T}_w$ and N_w:

```
$Decider_A(\langle \mathtt{T}, w \rangle)$:
    Compute $\langle \mathtt{T}_w \rangle$ and $\langle N_w \rangle$ from $\langle \mathtt{T}, w \rangle$.
    $r \leftarrow$Simulate $\mathbf{decider}_{EQ}$ on $\langle \mathtt{T}_w, N_w \rangle$.
    return $r$
```

N_w only accepts w and $\mathtt{T}_w$ accepts nothing if $\mathtt{T}$ does not accept w, and $\{w\}$ otherwise. Therefore the two languages will be equal iff $\mathtt{T}$ accepts w.

(Q5) Prove that the following language is not recursively enumerable (namely, it is not recognizable)

$$L = \left\{ \langle \mathtt{T} \rangle \;\middle|\; \mathtt{T} \text{ is a } \mathsf{TM}\text{, and } \mathsf{L}(\mathtt{T}) \text{ is infinite} \right\}.$$

Solution:

We reduce the not recognizable language $\overline{\mathrm{A}_{\mathsf{TM}}}$ to L. Let us check the following routine first (fix $\mathtt{T}$ and w):

```
$\mathtt{T}_w(x)$ :
        Simulate $\mathtt{T}(w)$ for $|x|$ steps.
        if simulation above does not accept then
            accept
        else
            reject
```

Observe that $\mathsf{L}(\mathtt{T}_w)$ is infinite iff $\mathtt{T}(w) \neq$**accept** . So now we have the following reduction: Assume, for the sake of contradiction, that we are given a recognizer **recog**L. We get the following recognizer for $\overline{\mathrm{A}_{\mathsf{TM}}}$.

```
recognizer$_{\overline{\mathrm{A}_{\mathsf{TM}}}}(\langle M, w\rangle)$:
    Compute $\langle \mathtt{T}_w \rangle$
    return recog$L(\langle \mathtt{T}_w \rangle)$
```

So, if $\mathtt{T}$ does not accept w, then $\mathsf{L}(\mathtt{T}_w) = \Sigma^*$ and then **recog**L$(\langle \mathtt{T}_w \rangle)$ would stop and accept.

If $\mathtt{T}$ accepts w, then the language $\mathsf{L}(\mathtt{T}_w)$ is finite, and then **recog**L$(\langle \mathtt{T}_w \rangle)$ might reject (or it might run forever. If **recog**L$(\langle \mathtt{T}_w \rangle)$ halts and rejects, then **recognizer**$_{\overline{\mathrm{A}_{\mathsf{TM}}}}$ would reject.

In any case, we got a recognizer for $\overline{\mathrm{A}_{\mathsf{TM}}}$ which is impossible, since this language is not recognizable.

Chapter 47

Discussion 15: Review

6 May 2009

Questions on homework ?

Any questions? Complaints, etc?

47.1 Problems

(Q1) If B is regular (or CFL), and $A \subseteq B$, can we deduce that B is regular (or CFL)?

Solution:

No, every language is a subset of Σ^* which is regular. More interesting sample is

$$L_1 = \left\{ \mathtt{a}^n \mathtt{b}^n \mathtt{c}^n \;\middle|\; n \geq 0 \right\} \subseteq L_2 = \left\{ \mathtt{a}^n \mathtt{b}^n \mathtt{c}^k \;\middle|\; n, k \geq 0 \right\} \subseteq L_3 = \left\{ \mathtt{a}^i \mathtt{b}^j \mathtt{c}^k \;\middle|\; i, j, k \geq 0 \right\}.$$

The language L_3 is regular, L_2 is not regular but is a CFL and L_1 is not a CFL.

(Q2) Give a direct proof why $L = \left\{ ww \;\middle|\; w \in \Sigma^* \right\}$ is not regular.

Solution:

Proof: Assume, for the sake of contradiction, that this language is regular, and let $\mathtt{D} = (Q, \Sigma, q_{\text{init}}, \delta, F)$ be a DFA for it. Consider the strings $w_i = \mathtt{a}^i\mathtt{b}$, for $i = 1, \ldots, \infty$. Let $q_i = \delta(q_{\text{init}}, w_i)$, for $i = 1, \ldots, \infty$. We claim all the q_is are distinct. Indeed, if for some $i \neq j$, we had $q_i = q_j$ then

$$\begin{aligned} & q' = \delta(q_i, \mathtt{a}^j\mathtt{b}) = \delta(\delta(q_{\text{init}}, \mathtt{a}^i\mathtt{b}), \mathtt{a}^j\mathtt{b}) = \delta(q_{\text{init}}, \mathtt{a}^i\mathtt{b}\mathtt{a}^j\mathtt{b}) \notin F, \\ \text{and} \quad & q' = \delta(q_j, \mathtt{a}^j\mathtt{b}) = \delta(\delta(q_{\text{init}}, \mathtt{a}^j\mathtt{b}), \mathtt{a}^j\mathtt{b}) = \delta(q_{\text{init}}, \mathtt{a}^j\mathtt{b}\mathtt{a}^j\mathtt{b}) \in F. \end{aligned}$$

Which is impossible. As such, the number of states of D is infinite. A contradiction. ■

(Q3) Show that $L = \left\{ \langle \mathcal{G}_1, \mathcal{G}_2, k \rangle \;\middle|\; \exists w, |w| \leq k, w \in L(\mathcal{G}_1) \cap L(\mathcal{G}_2), \text{ and } \mathcal{G}_1, \mathcal{G}_2 \text{ are CFGs} \right\}$ is decidable.

Solution:

For a given $\mathcal{G}_1, \mathcal{G}_2$ and k, the TM produces all strings w of length at most k and checks whether each string is in both $L(\mathcal{G}_1)$ and $L(\mathcal{G}_2)$ (first we convert the grammar into CNF and then we use CYK algorithm).

(Q4) Show that $L = \left\{ \langle \mathcal{G}_1, \mathcal{G}_2 \rangle \;\middle|\; L(\mathcal{G}_1) \setminus L(\mathcal{G}_2) = \varnothing \right\}$ ($\mathcal{G}_1$ and $\mathcal{G}_2$ are grammars) is undecidable.

Solution:

We can reduce $EQ_{CFG} = \left\{ \langle \mathcal{G}_1, \mathcal{G}_2 \rangle \;\middle|\; L(\mathcal{G}_1) = L(\mathcal{G}_2) \right\}$ to L. If we have a decider for L, we can build a decider for EQ_{CFG} by querying it once for $\langle \mathcal{G}_1, \mathcal{G}_2 \rangle$ and once for $\langle \mathcal{G}_2, \mathcal{G}_1 \rangle$. (Note that for any two sets A and B we have that $A = B$ if and only if $A \setminus B = \emptyset$ and $B \setminus A = \emptyset$.)

(Q5) Show that $L = \left\{ \langle \mathtt{P}, w \rangle \;\middle|\; \mathtt{P} \text{ is an } \mathsf{RA} \text{ and } w \in \mathsf{L}(\mathtt{P}) \right\}$ is decidable.

Solution:

Given $\langle \mathtt{P}, w \rangle$, the decider converts $\mathtt{P}$ into a CFG $\mathcal{G}$ (remember we have seen the algorithm for this) and then converts $\mathcal{G}$ into CNF (we also saw this algorithm), and finally using CYK it detects if $w \in \mathsf{L}(\mathcal{G})$ or not.

(Q6) Assume we have a language L and a TM $\mathtt{T}$ with the following property. For every string w with length at least 2, $\mathtt{T}(w)$ halts and outputs k strings $w_1, \cdots, w_k$ where $|w_i| < |w|$ for all i (k is not a constant and depends on string w). We know that $w \in L$ iff for all i, $w_i \in L$.

Assuming that $\mathtt{0} \in L, \mathtt{1}\ /inL, \epsilon \notin L$, design a decider for set L (you can use $\mathtt{T}$ as a subroutine) and prove that your decider works.

Solution:

```
Decider_L(w):
        if |w| ≤ 1 then
            if w = 0 then
                return Yes
            else
                return No
        w_1, ··· , w_k ← T(w)
        for i = 1, . . . , k do
            if Decider_L(w_i) =No then
                return No
        return Yes
```

Claim 47.1.1 *For any string $w \in \Sigma^*$, we have that* **Decider**$_L(w)$ *halts, and furthermore* **Decider**$_L(w) = Yes \iff w \in L$.

Proof: We prove by induction on the length of w.

For the base case, when $|w| \leq 1$, it is easy to see that lines 1,2,3 of the algorithm, return the correct result immediately. And $\mathtt{T}$ halts in this case.

If $|w| > 1$, and assume inductively that the claim holds for any string strictly shorter than w. For w, the algorithm computes (by using a decider) a finite number of strings $w_1, \ldots w_k$, and call recursively on each one of these strings. By induction, we know that each one of these recursive calls returns, since the call is done on shorter input strings. As such, the **Decider**$_L(w)$ stops on w.

Furthermore, it returns true iff **Decider**$_L(w_i) = Yes$ for all i. But by induction hypothesis this happens exactly when $M(w_i) = Yes$ for all i. By the property of machine M, this happens iff $w \in L$, which is the desired result. ■

Part III

Exams

Chapter 48

Exams – Spring 2009

48.1 Midterm 1 - Spring 2009

6 May 2009

INSTRUCTIONS (read carefully)

- Fill in your name, netid, and discussion section time below. Also write your netid on the other pages (in case they get separated).

NAME:

NETID: **DISC:**

- There are 7 problems. Make sure you have a complete exam.
- The point value of each problem is indicated next to the problem, and in the table below.
- Points may be deducted for solutions which are correct but excessively complicated, hard to understand, or poorly explained. Please keep your solutions ***short*** and crisp.
- The exam is designed for one hour, but you have the full two hours to finish it.
- It is wise to skim all problems and point values first, to best plan your time.
- This is a closed book exam. No notes of any kind are allowed. Do all work in the space provided, using the backs of sheets if necessary. See the proctor if you need more paper.
- Please bring any apparent bugs to the attention of the proctors.
- After the midterm is over, discuss its contents with other CS 373 students **only** after verifying that they have also taken the exam (e.g. they aren't about to take the conflict exam).
- We indicate next to each problem how much time we suggest you spend on it. We also suggest you spend the last 25 minutes of the exam reviewing your answers.

Problem 1: Short Answers (8 points)

[10 minutes.]

The answers to these problems should be short and not complicated.

(A) If a DFA M has k states then M must accept some word of length at most $k-1$. True or false?

□ True □ False

And why? (At most 20 words.)

(B) Let L be a finite language. Is the complement language $\overline{L}$ regular?

□ Yes □ No

And why? (At most 30 words.)

(C) Suppose that L is a regular language over the alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$. And consider the language

$$L' = \left\{ 0^i \;\middle|\; i = |w|\,, w \in L \right\}.$$

Is the language L' regular?

□ Yes □ No

And why? (At most 40 words.)

(D) $\mathtt{a}^*\emptyset = \mathtt{a}^*$. True or False?

□ True □ False

(E) Let $L = \left\{ wx \;\middle|\; w \in \Sigma^*, x \in \Sigma^*, |w| = |x| \right\}$. Is L regular?

□ Yes □ No

(F) Let L_i be a regular language, for $i = 1, \ldots, \infty$. Is the language $\bigcap_{i=1}^{\infty} L_i$ always regular? True or false?

□ True □ False

(G) If L_1 and L_2 are two regular languages that are accepted by two DFAs D_1 and D_2, respectively, each with k_1 and k_2 states, then the language $L_1 \setminus L_2$ can always be recognized by a DFA with $k_1 * k_2$ states? True or false?

□ True □ False

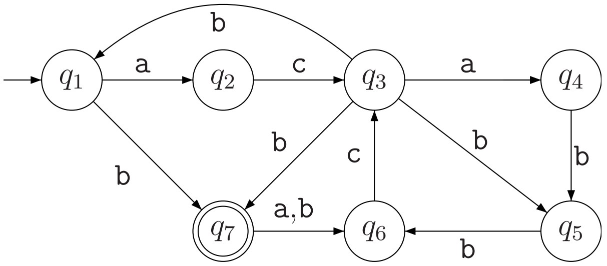
(H) The minimum size NFA for a regular language L, always has strictly fewer states then the minimum size DFA for the language L. True or False?

□ True □ False

Problem 2: NFA transitions (6 points)

[5 minutes.]

Suppose that the NFA $N = (Q, \{\mathtt{a}, \mathtt{b}\}, \delta, q_0, \mathcal{F})$ is defined by the following state diagram:



Fill in the following values:

(A) $\mathcal{F} =$

(B) $\delta(q_2, \mathtt{a}) =$

(C) $\delta(q_3, \mathtt{b}) =$

(D) $\delta(q_6, \texttt{c}) =$

(E) List the members of the set $\left\{ q \in Q \;\middle|\; q_5 \in \delta(q, \texttt{b}) \right\}$: □

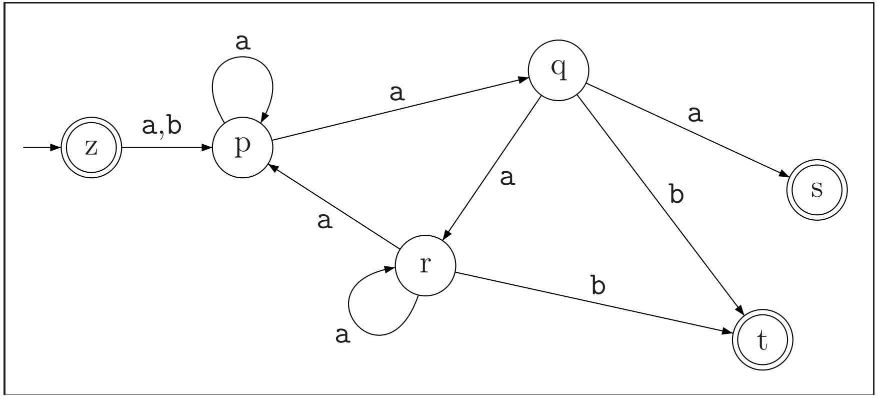
(F) Does the NFA accepts the word acbacbb? (Yes / No)

□ Yes □ No

Problem 3: NFA to DFA (6 points)

[10 minutes.]

Convert the following NFA to a DFA recognizing the same language, using the subset construction. Give a state diagram showing all states reachable from the start state, with an informative name on each state. Assume the alphabet is $\{\texttt{a}, \texttt{b}\}$.



Problem 4: Modifying DFAs (6 points)

[15 minutes.]

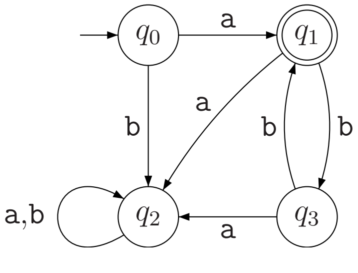
Suppose that $M = (Q, \Sigma, \delta, q_0, F)$ is a DFA, where $\Sigma = \{\texttt{a}, \texttt{b}\}$. Using M, we define a new DFA

$$M' = (Q \cup \{r, s\}, \Sigma \cup \{\#\}, \delta', q_0, F'),$$

where $F' = \{s\}$ and the new transition function δ' is defined as follows

$$\delta'(q,t) = \begin{cases} \delta(q,t) & q \in Q \text{ and } t \in \Sigma \\ s & q \in F, t = \# \\ q_0 & q \in Q \setminus F, t = \# \\ r & q \in \{r, s\}, t \in \{\texttt{a}, \texttt{b}\} \\ s & q \in \{r, s\}, t = \# \end{cases}$$

(A) Assume M is the following DFA:



Draw M' for this case.

(B) In general, define the language of M' in terms of the language of an arbitrary M. (Hint: Make sure that your answer works for the above example!)

Problem 5: NFA construction (8 points)

[15 minutes.]

A string y is a ***subsequence*** of string $x = x_1x_2 \cdots x_n \in \Sigma^*$, if there exists indices $i_1 < i_2 < \cdots < i_m$ such that $y = x_{i_1}x_{i_2} \cdots x_{i_m}$. Note, that the empty word ϵ is a subsequence of every string. For example, aaba is a subsequence of $\mathtt{c\underline{a}dcd\underline{a}c\underline{ba}}$, but abc is not a subsequence of cbacba.

Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA with language L. Describe a construction for an NFA N' such that:

$$L(N') = \left\{ x \;\middle|\; \text{there is a } y \in L, \text{ such that } x \text{ is a subsequence of } y \right\}.$$

(A) Explain the idea of your construction.

(B) Write down your construction formally (in tuple notation, as we did for example in constructing the product of two DFA's in lecture). Namely, you are given the NFA $N = (Q, \Sigma, \delta, q_0, F)$, and you need to specify the new NFA N'. (Use formal notations and few words.)

Problem 6: Proof (8 points)

[20 minutes.]

Consider ***effective regular expressions***. These are regular expressions that are not allowed to use the empty-set notation. Formally, they are defined as follows:

regex	conditions	set represented
a	$\mathtt{a} \in \Sigma$	$\{\mathtt{a}\}$
ϵ		$\{\epsilon\}$
$R + S$	R, S regexps	$L(R) \cup L(S)$
RS	R, S regexps	$L(R)L(S)$
R^*	R a regex	$L(R)^*$

The ***complexity*** $\Delta(R)$ of a regular expression R is the number of operators appearing in R. Thus, $\Delta(\mathtt{ab}) = \Delta(\mathtt{a} \circ \mathtt{b}) = 1$ (we concatenate a with b), $\Delta(\,(\mathtt{a} + \mathtt{b})^*) = 2$, and $\Delta\Big(((\mathtt{a} + \mathtt{b})^*(\mathtt{b} + \mathtt{c} + \epsilon))^*\;\Big) = 6$ (note, that parenthesis are not counted).

Prove formally the following claim.

Claim. *If R is an effective regular expression, then $L(R)$ contains at least one word of length at most $\Delta(R) + 1$.*

Proof:

Problem 7: NFA modification with a proof (8 points)

[20 minutes.]

Fix a finite alphabet Σ. We want to transform an NFA A over Σ into an NFA B over Σ such that B accepts all suffixes of words accepted by A. The suffix language of $L(A)$ is

$$S = \left\{ y \in \Sigma^* \;\middle|\; \exists x \in \Sigma^*, xy \in L(A) \right\}$$

We want to build a NFA B such that $L(B) = S$.

Let $A = (Q, \Sigma, \delta, q_0, F)$ be an NFA where all states in Q are *reachable* from the initial state (i.e. for any state $q \in Q$, there is a set of transitions that connect the initial state q_0 to q). More formally, for any $q \in Q$, there is a word $w \in \Sigma^*$ such that $q \in \delta(q_0, w)$ (where δ is the transition function extended to words).

Let us define $B = (Q \cup \{q_0'\}, \Sigma, \delta', q_0', F)$ where we define

$$\begin{aligned} \delta'(q, a) = &\quad \delta(q, a) && \textit{for all } q \in Q, a \in \Sigma \cup \{\epsilon\} \\ \delta'(q_0', \epsilon) = &\quad Q && \\ \delta'(q_0', a) = &\quad \emptyset && \textit{for all } a \in \Sigma \end{aligned}$$

Prove *formally* that $L(B) = S$.

Hint: The proof goes by showing that $L(B)$ is contained in S, and that $L(B)$ is contained in S. So, you must show both inclusions using precise arguments. You do not need to use complete mathematical notation, but your proof should be precise, correct, short and convincing. (Make sure you are not writing unnecessary text [like copying the hint, or writing text unrelated to the proof].)

48.2 Midterm 2 - Spring 2009

6 May 2009

CS 373, Spring 2009, Midterm 2, 7-9pm, April 2, 2009.

INSTRUCTIONS (read carefully)

- Fill in your name, netid, and discussion section time below. Also write your netid on the other pages (in case they get separated).

NAME:

NETID: **DISC:**

- There are 7 problems, on pages numbered 1 through 8. Make sure you have a complete exam.
- The point value of each problem is indicated next to the problem, and in the table below.
- Points may be deducted for solutions which are correct but excessively complicated, hard to understand, or poorly explained.
- The exam is designed for slightly over one hour, but you have the full two hours to finish it.
- It is wise to skim all problems and point values first, to best plan your time.
- This is a closed book exam. No notes of any kind are allowed. Do all work in the space provided, using the backs of sheets if necessary. See the proctor if you need more paper.
- Please bring any apparent bugs to the attention of the proctors.
- After the midterm is over, discuss its contents with other students in the class **only** after verifying that they have also taken the exam (e.g. they aren't about to take the conflict exam).

Problem 1: Short Answer (12 points)

[10 minutes.]

Answer "yes" or "no" to the following questions. No explanations are required. (All the strings in this problem use the same, fixed alphabet.)

(a) Is the language $\left\{ \mathtt{a}^i \mathtt{b}^{i+j} \mathtt{c}^j \;\middle|\; i, j \geq 0 \right\}$ a context-free language?

Yes: □ **No**: □

(b) If L is a language over Σ^*, and h is a homomorphism, and $h(L)$ is regular. Then L must be regular. Is this statement correct?

Yes: □ **No**: □

(c) Suppose L is a language over $\{\mathtt{0}\}^*$, such that if $\mathtt{0}^i$ is in L then $\mathtt{0}^{i+2}$ is in L. Then L must be regular. Is this statement correct?

Yes: □ **No**: □

(d) If L_1 and L_2 be two languages. If L_1 and L_2 are both context-free, then $L_1 \setminus L_2$ must also be context-free. Is this statement correct?

Yes: □ **No**: □

(e) If L is a language, its subset language is

$$S(L) = \left\{ w \;\middle|\; \text{there exists } x \in L \text{ s.t. } w \text{ is a subsequence of } x \right\}.$$

A string y is a ***subsequence*** of string $x = x_1 x_2 \cdots x_n \in \Sigma^*$, if there exists indices $i_1 < i_2 < \cdots < i_m$ such that $y = x_{i_1} x_{i_2} \cdots x_{i_m}$. Note, that the empty word ϵ is a subsequence of every string. For example, $\mathtt{aaba}$ is a subsequence of $\mathtt{c\underline{a}dcd\underline{a}c\underline{ba}}$, but $\mathtt{abc}$ is not a subsequence of $\mathtt{cbacba}$.

If L is context-free, then the language $S(L)$ is context free. Is this statement correct?

Yes: □ **No**: □

(f) Consider a ***parallel recursive automata*** $\mathtt{D}$ – it is made out of two recursive automatas $\mathtt{B}_1$ and $\mathtt{B}_2$ and it accepts a word w if both $\mathtt{B}_1$ and $\mathtt{B}_2$ accepts w. The parallel recursive automata $\mathtt{D}$ might accept a language that is not context-free. Is this statement correct?

Yes: □ **No**: □

Problem 2: Grammar design (8 points)

[10 minutes.]

Let $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. Let

$$J = \left\{ w \;\middle|\; \#_{\mathtt{a}}(w) = \#_{\mathtt{b}}(w) \text{ or } \#_{\mathtt{b}}(w) = \#_{\mathtt{c}}(w) \right\},$$

where $\#_z(w)$ is the number of appearances of the character z in w. For example, the word $x = \mathtt{baccacbbcb} \in \mathsf{L}(J)$ since $\#_{\mathtt{a}}(x) = 2$, $\#_{\mathtt{b}}(x) = 4$, and $\#_{\mathtt{c}}(x) = 4$. Similarly, the word $y = \mathtt{abbccc} \notin \mathsf{L}(J)$ since $\#_{\mathtt{a}}(y) = 1$, $\#_{\mathtt{b}}(y) = 2$, and $\#_{\mathtt{c}}(y) = 3$.

Give a context-free grammar whose language is J. Be sure to indicate what its start symbol is. (Hint: First provide a CFG for the easier language $K = \left\{ w \in \{\mathtt{a}, \mathtt{b}\}^* \;\middle|\; \#_{\mathtt{a}}(w) = \#_{\mathtt{b}}(w) \right\}$ and modify it into the desired grammar.)

Problem 3: RA design (8 points)

[5 minutes.]

Let

$$J = \left\{ w \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}^* \;\middle|\; \begin{array}{l} w = \mathtt{a}^i \mathtt{b}^i \mathtt{c}^n \mathtt{d}^n \text{ or} \\ w = \mathtt{c}^n \mathtt{a}^i \mathtt{b}^i \mathtt{d}^n \\ \text{for some } i, n \geq 0 \end{array} \right\}.$$

Give the state diagram for a RA whose language is J. Include brief comments explaining the design of your RA, to help us understand how it works.

Problem 4: Short Answer II (8 points)

[8 minutes.]

The answers to these problems should be short and not complicated.

(a) Suppose we know that the language $B = \left\{ \mathtt{a}^n \mathtt{b}^n \mathtt{c}^n \;\middle|\; n \geq 2 \right\}$ is not <u>context-free</u>. Let L be the language

$$L = \left\{ w \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{f}\}^* \;\middle|\; \#_{\mathtt{a}}(w) = \#_{\mathtt{d}}(w) = \#_{\mathtt{f}}(w) \right\},$$

where $\#_z(w)$ is the number of appearances of the character z in w. Prove that L is not context-free using closure properties and the fact that B is not context-free.

(b) Let w be a word of length n in a language generated by a grammar $\mathcal{G}$ which is in Chomsky Normal Form (CNF). First, how many internal nodes does the parse tree for w using $\mathcal{G}$ has?

Secondly, assume that $\mathcal{G}$ has k variables, and $n = |w| > 2^k$. Is the language $\mathsf{L}(\mathcal{G})$ finite or not? Justify your answer.

Problem 5: Not regular (8 points)

[8 minutes.]

Suppose $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and let $L = \left\{ ww \,\middle|\, w \in \Sigma^* \right\}$. That is, every word in L is a concatenation of some string with itself. Provide a direct (and short) proof that L is not regular (a proof not using the pumping lemma is preferable).

Problem 6: Finite or not. (8 points)

[10 minutes.]

Let $\mathcal{G}$ be a context-free grammar over $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, and let m be some fixed number.

(i) Prove that $L_{\geq m} = \left\{ w \in \mathsf{L}(\mathcal{G}) \,\middle|\, |w| \geq m \right\}$ is a context-free language.

(There is a very short proof of this – your answer should be at most 40 words. Longer answers would get no points.)

(ii) Assume that (i) holds, and furthermore, one can compute the grammar of $L_{\geq m}$ given L and m. Describe an algorithm that decides whether the language L is finite or not. Prove that your algorithm works.

Problem 7: Proof. (8 points)

(Extra credit.) [9 minutes.]

Consider the grammar

$$\Longrightarrow \quad \begin{aligned} &\mathsf{S} \to \mathtt{a}\mathsf{S}\mathtt{b} \mid \mathsf{T}, \\ &\mathsf{T} \to \mathtt{c}\mathsf{T} \mid \epsilon. \end{aligned}$$

Prove that the language of this grammar is $L = \left\{ \mathtt{a}^n \mathtt{c}^k \mathtt{b}^n \,\middle|\, n, k \geq 0 \right\}$. Your proof must be formal, correct and ***short***.

Hint: Prove (by induction, naturally) exactly what are the words that can be derived by a tree with i internal nodes, with T as the root (i.e., these are the strings that can be derived from T in i derivation steps). Next, prove a similar claim about all the words that can be generated by a parse tree with i internal nodes, having S as the root.

48.3 Final - Spring 2009

6 May 2009

CS 373: Theory of Computation

Spring 2009 Final Exam, May 11, 2009
13:30-16:30am, SC 1404

NAME:

NETID: **DISC:**

INSTRUCTIONS (read carefully)

- Print your name, netID, and section time in the boxes above.
- The exam contains 9 problems on pages numbered 1 through 10. Make sure you have a complete exam.
- You have three hours.
- The point value of each problem is indicated next to the problem and in the table on the right.
- It is wise to skim all problems and point values first, to best plan your time. If you get stuck on a problem, move on and come back to it later.
- Points may be deducted for solutions which are correct but excessively complicated, hard to understand, hard to read, or poorly explained.
- If you change your answers (especially for T/F questions or multiple-choice questions), be sure to erase well or otherwise make sure your final choice is clear.
- Please bring apparent bugs or unclear questions to the attention of the proctors.
- This is a closed book exam.
- Do all work in the space provided, using the backs of sheets if necessary. See the proctor if you need more paper.

P	Pnts	Score	Grader
1	18		
2	16		
3	9		
4	8		
5	10		
6	10		
7	10		
8	10		
9	9		
	100		

Problem 1: True/False (18 points)

Decide whether each of the following statements is true or false, and check the corresponding box. You do not need to explain or prove your answers. Read the questions very carefully. They may mean something quite different than what they appear to mean at first glance.

(a) Consider a regular language L, an let $L' = \left\{ w \;\middle|\; w \in L \text{ and } w \text{ is a palindrome} \right\}$. The language L' is regular. True or false?

False: □ **True**: □

(b) If L is undecidable and $\overline{L}$ is recognizable, then L is not recognizable.

False: □ **True**: □

(c) There exist two context-free languages L_1 and L_2 such that $L_1 \cap L_2$ is also context-free.

False: □ **True**: □

(d) Let $\texttt{T}$ and $\texttt{Z}$ be two non-deterministic Turing machine deciders. Then there is a TM that recognizes the language $L(\texttt{T}) \cap L(\texttt{Z})$.

False: □ **True**: □

(e) Let $h : \Sigma^* \to \Sigma^*$ be a string sorting operator; that is, it sorts the characters of the given string and returns the sorted string (it uses alphabetical ordering on the characters of Σ). For example, we have $h(\texttt{baba}) = \texttt{aabb}$ and $h(\texttt{peace}) = \texttt{aceep}$. Let L be a language, and consider the sorted language

$$h(L) = \left\{ h(w) \;\middle|\; w \in L \right\}.$$

The statement "if L is regular, then $h(L)$ is also regular" is true or false?

False: □ **True**: □

(f) Let $\texttt{T}$ be a TM decider for some language L. Then $h(L)$ (see previous question) is decidable.

False: □ **True**: □

(g) If languages L_1 and L_2 are context-free then $L_1 \cap L_2$ is decidable.

False: □ **True**: □

(h) If a language L is context-free and there is a Chomsky normal form (CNF) for it with k variables, then either L is infinite, or L is finite and contains only words of length $< 2^{k+4}$.

False: □ **True**: □

(i) If a language L is accepted by a RA $\texttt{D}$ then $\overline{L}$ will be accepted by a RA $\texttt{C}$ which is just like $\texttt{D}$ except that the set of accepting states has been complemented.

False: □ **True**: □

(j) Let $f : \Sigma^* \to \mathbb{N}$ be a function that for a string $w \in \Sigma^*$ it returns some positive integer number $f(w)$. Furthermore, assume that you are given a TM $\texttt{T}$ that, given w, computes $f(w)$ and this TM always halts. Then the language

$$L = \left\{ \langle M, w \rangle \;\middle|\; M \text{ stops on } w \text{ after at most } f(w) \text{ steps} \right\}$$

is decidable.

False: □ **True**: □

Problem 2: Classification (16 points)

For each language L described below, we have listed 2–3 language classes. Mark the most restrictive listed class to which L must belong. E.g. if L must always be regular and we have listed "regular" and "context-free", mark only "regular".

For example, if you are given a language L, and given choices Regular, Context-free, and Decidable, then you must mark Regular if L is regular, you must mark Context-free if L is context-free and *not* regular, and mark Decidable if L is Decidable but *not* context-free.

(a) $L = \left\{ x\mathtt{c}^n \;\middle|\; x \in \{\mathtt{a}, \mathtt{b}\}^*, n = |x| \right\}$.

□ **Regular** □ **Context-free** □ **Decidable**

(b) $L = \left\{ \langle \mathcal{G} \rangle \;\middle|\; \mathcal{G} \text{ is a } \mathsf{CFG} \text{ and } \mathsf{L}(\mathcal{G}) \text{ is not empty} \right\}$.

□ **Decidable** □ **Recognizable** □ **Not recognizable**

(c) $L = \left\{ \mathtt{a}^i\mathtt{b}^j\mathtt{c}^k\mathtt{d}^m \;\middle|\; i+j+k+m \text{ is a multiple of 3 and } i+j = k+m \right\}$

□ **Regular** □ **Context-free** □ **Decidable**

(d) $L = \left\{ \langle w, M_1, M_2 \rangle \;\middle|\; \begin{array}{l} w \text{ is a string,} \\ M_1 \text{ and } M_2 \text{ are } \mathsf{TM}\text{s,} \\ M_1 \text{ accepts } w, \text{ or } M_2 \text{ accepts } w \end{array} \right\}$.

□ **Decidable** □ **Recognizable** □ **Not recognizable**

(e) $L = \{x_1\#x_2\#\ldots\#x_n \mid x_i \in \{a,b\}^* \text{ for each } i \text{ and, for some } r < s, \text{ we have } x_r = x_s^R\}$.

□ **Regular** □ **Context-free** □ **Decidable**

(f) $L = \left\{ \langle \mathcal{G}, \mathtt{D} \rangle \;\middle|\; \mathcal{G} \text{ is a } \mathsf{CFG}, \mathtt{D} \text{ is a } \mathsf{DFA}, \text{and } \mathsf{L}(\mathtt{D}) \cap \mathsf{L}(\mathcal{G}) = \emptyset \right\}$.

□ **Decidable** □ **Recognizable** □ **Not TM recognizable**

(g) $L = \left\{ \langle \mathcal{G}_1, \mathcal{G}_2 \rangle \;\middle|\; \mathcal{G}_1 \text{ and } \mathcal{G}_2 \text{ are } \mathsf{CFG}\text{s and } \left| \mathsf{L}(\mathcal{G}_1) \cap \mathsf{L}(\mathcal{G}_2) \right| > 0 \right\}$.

□ **Decidable** □ **Recognizable** □ **Not TM recognizable**

(h) $L = \left\{ \langle \mathcal{G} \rangle \;\middle|\; \mathcal{G} \text{ is a } \mathsf{CFG} \text{ and } \mathsf{L}(\mathcal{G}) \neq \Sigma^* \right\}$

□ **Decidable** □ **Recognizable** □ **Not recognizable**

Problem 3: Context-free grammars (9 points)

(i) Let $\Sigma = \Big\{ [\,,\,] \Big\}$, and consider the language L_0 of all balanced bracketed words. For example, we have $[\,][\,] \in L_0$ and $[\,[\,][\,]\,] \in L_0$, but $][\notin L_0$.

Give a context-free grammar for this language, with the start symbol S_0.

(ii) Let L_1 be the language of all balanced bracketed words, where the character $\mathtt{x}$ might appear somewhere at most once. For example, we have $[\,\mathtt{x}\,][\,] \in L_1$ and $[\,[\,][\,]\,] \in L_1$, but $][\notin L_1$ and $[\,\mathtt{x}\,]\mathtt{x} \notin L_1$.

Give an explicit context-free grammar for L_1, with the start symbol S_1. You can use the grammar for the language L_0 that you constructed above.

(iii) Let L_2 be the language of all balanced bracketed words, where the character $\mathtt{x}$ might appear somewhere in the word at most twice. For example, we have $\mathtt{x}[\,\mathtt{x}\,][\,] \in L_2$ and $[\,[\,][\,]\,] \in L_2$, but $][\notin L_2$ and $\mathtt{x}[\,\mathtt{x}\,]\mathtt{x} \notin L_2$.

Give a context-free grammar for this language, with the start symbol S_2.

Problem 4: Proof of Non-Regularity (8 points)

Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$. Let

$$L = \Big\{ w \;\Big|\; w \in \Sigma^* \text{ and } \#_{\mathtt{a}}(w) = \#_{\mathtt{b}}(w) \Big\},$$

where $\#_{\mathtt{a}}(w)$ (resp. $\#_{\mathtt{b}}(w)$) is the number of $\mathtt{a}$ (resp. $\mathtt{b}$) appearing in w.

Prove that L is not regular (provide a short and concise proof). We encourage you to use the direct proof for non-regularity, though you are also allowed to use the pumping lemma.

Problem 5: Short answer (10 points)

(a) For the following grammar, give an equivalent grammar to it in Chomsky normal form (CNF).

$$\begin{aligned} \Longrightarrow \mathsf{S} &\rightarrow \mathtt{aTb} \\ \mathsf{T} &\rightarrow \mathtt{aTb} \mid \epsilon. \end{aligned}$$

(b) A ***Quadratic Bounded Automaton*** (QBA) is a Turing machine that can use at most n^2 calls of the tape (assume it has a single tape) given an input string of length n. Let

$$\mathsf{L}_{\mathsf{QBA}} = \Big\{ \langle \mathtt{D}, w \rangle \;\Big|\; \mathtt{D} \text{ is a } \mathsf{QBA} \text{ and } \mathtt{D} \text{ halts on } w. \Big\}.$$

Explain why $\mathsf{L}_{\mathsf{QBA}}$ is Turing decidable, and shortly describe a TM that decides this language.

Problem 6: TM variations (10 points)

Joe the programmer has come up with a new extension of Turing machines that he believes can help decide more languages, called the ***back-tracking Turing machine*** (BTM for short).

A back-tracking Turing machine has, apart from the normal transitions (that allow it read a symbol, rewrite it, and move left or right), a new ***backtrack*** transition. A transition t from state p_1 to p_2 can be labeled as a $\mathsf{backtrack}_q$ transition. If the BTM use the transition t, which is (say)

$$p_1 \xrightarrow[\mathsf{backtrack}_q]{\mathtt{a} \rightarrow ?,?} p_2$$

then it resets the current contents of the tape and the tape head position to that which they were at the last time it was in state q, and then it sets the control to be at state p_2.

In other words, assume the BTM has run through a sequence of configurations $c_1, c_2, \ldots c_k$ and c_k is a configuration where the state is p_1. Then, if it uses the transition t when being in the configuration c_k it can go to a new configuration $c = wp_2w'$, where $c_i = wqw'$ is the last configuration in the sequence where the state was q.

Joe thinks that this greatly enhances a Turing machine, as it allows the Turing machine to roll-back to an earlier configuration by undoing changes.

Note, that a BTM can perform several (say three) consecutive transitions of $\mathsf{backtrack}_x$ one after the other. That would be equivalent to restoring to the second to last configuration in the execution history with the state x in it. Also, observe that BTM are deterministic.

Show that Joe is wrong by giving, for any BTM, a deterministic Turing machine that performs the same job. Keep your description of the deterministic Turing machine to high-level psuedo-code, and do write down the intuition of how the deterministic Turing machine will work.

(Your answer should not exceed a hundred words.)

Problem 7: All together now. (10 points)

Let $L = \left\{ \langle \mathtt{T}, \mathtt{Z} \rangle \;\middle|\; \begin{array}{l} \mathtt{T} \text{ and } \mathtt{Z} \text{ are Turing machines and} \\ \text{there is a word } w \text{ such that both } \mathtt{T} \text{ and } \mathtt{Z} \text{ accept } w \end{array} \right\}$.

Show that L is TM-recognizable, i.e. explain how to construct a Turing machine that accepts $\langle \mathtt{T}, \mathtt{Z} \rangle$ if $\langle \mathtt{T}, \mathtt{Z} \rangle \in L$. Assume that $\mathtt{T}$ and $\mathtt{Z}$ work on input words over the same alphabet. Remember that $\mathtt{T}$ and $\mathtt{Z}$ may run forever on some inputs.

Problem 8: Reduction (10 points)

Recall that $\mathrm{A}_{\mathsf{TM}} = \left\{ \langle M, w \rangle \;\middle|\; M \text{ is a Turing machine and } M \text{ accepts } w \right\}$ and A_{TM} is undecidable.

Let

$$L = \left\{ \langle \mathtt{T} \rangle \;\middle|\; \mathtt{T} \text{ is a Turing machine such that for } \textit{every } w \in \mathsf{L}(\mathtt{T}) \text{ we have } w^R \in \mathsf{L}(\mathtt{T}) \right\}.$$

(Recall that for any word w, w^R denotes the reverse of the word w.) We assume here that the input alphabet of $\mathtt{T}$ has at least two characters in it.
Show that L is undecidable using a reduction from A_{TM}.
Prove this directly, not by citing Rice's Theorem.

Problem 9: Decidability (9 points)

(i) Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, and recall the language

$$L_{eq} = \left\{ w \;\middle|\; w \in \Sigma^* \text{ and } \#_{\mathtt{a}}(w) = \#_{\mathtt{b}}(w) \right\}.$$

Give a CFG for this language.

(ii) Consider the following language

$$\mathsf{L} = \left\{ \langle \mathtt{D} \rangle \;\middle|\; \begin{array}{l} \mathtt{D} \text{ is a } \mathsf{DFA} \text{ and it accepts some string } w \in \{\mathtt{a}, \mathtt{b}\}^* \\ \text{such that } \#_{\mathtt{a}}(w) = \#_{\mathtt{b}}(w) \end{array} \right\}.$$

Show how to construct a decider for L.

You may use any decidability results you have learnt in this course.

48.4 Mock Final Exam - Spring 2009

6 May 2009

CS 373: Theory of Computation, Mock Final Exam, Spring 2009

Version: 2.0

Problem 1: True/False (10 points)

Completely write out "True" if the statement is necessarily true. Otherwise, completely write "False". Other answers (e.g. "T") will receive credit only if your intent is unambiguous. For example, "$x+y > x$" has answer "False" assuming that y could be 0 or negative. But "If x and y are natural numbers, then $x + y \geq x$" has answer "True". You do not need to explain or prove your answers.

1. Let $\mathtt{D}$ be a DFA with n states such that $\mathsf{L}(\mathtt{D})$ is infinite. Then $\mathsf{L}(\mathtt{D})$ contains a string of length at most $2n - 1$.

   False: □ **True**: □

2. Let $L_w = \left\{ \langle \mathtt{T} \rangle \,\middle|\, \mathtt{T} \text{ is a } \mathsf{TM} \text{ and } \mathtt{T} \text{ accepts } w \right\}$, where w is some fixed string. Then there is an enumerator for L_w.

   False: □ **True**: □

3. The set of undecidable languages is countable.

   False: □ **True**: □

4. For a TM $\mathtt{T}$ and a string w, let $CH_{\mathtt{T},w} = \left\{ x \,\middle|\, x \text{ is an accepting computation history for } \mathtt{T} \text{ on w } \right\}$. The language $CH_{\mathtt{T},w}$ is decidable.

   False: □ **True**: □

5. The language $\left\{ \langle \mathtt{T}, w \rangle \,\middle|\, \mathtt{T} \text{ is a linear bounded automaton and } \mathtt{T} \text{ accepts } w \right\}$ is undecidable.

   False: □ **True**: □

6. The language $\left\{ \langle \mathcal{G} \rangle \,\middle|\, \mathcal{G}\text{is a context-free grammar and } \mathcal{G} \text{ is ambiguous} \right\}$ is Turing-recognizable.

   False: □ **True**: □

7. Context-free languages are closed under homomorphism.

   False: □ **True**: □

8. The modified Post's correspondence problem is Turing recognizable.

   False: □ **True**: □

9. There is a bijection between the set of recognizable languages and the set of decidable languages.

   False: □ **True**: □

Problem 2: Classification (20 points)

For each language L described below, classify L as

- **R**: Any language satisfying the information must be regular.
- **C**: Any language satisfying the information must be context-free, but not all languages satisfying the information are regular.
- **DEC**: Any language satisfying the information must be decidable, but not all languages satisfying the information are context-free.
- **NONDEC**: Not all languages satisfying the information are decidable. (Some might be only Turing recognizable or perhaps even not Turing recognizable.)

For each language, circle the appropriate choice (**R**, **C**, **DEC**, or **NONDEC**). If you change your answer be sure to erase well or otherwise make your final choice clear. **Ambiguously marked answers will receive no credit.**

1. **R** **C** **DEC** **NONDEC**

$$L = \left\{ \langle \mathtt{T} \rangle \;\middle|\; \mathtt{T} \text{ is a linear bounded automaton and } \mathsf{L}(\mathtt{T}) = \emptyset \right\}.$$

2. **R** **C** **DEC** **NONDEC**

$$L = \left\{ ww^R w \;\middle|\; w \in \{\mathtt{a}, \mathtt{b}\}^* \right\}.$$

3. **R** **C** **DEC** **NONDEC**

$$L = \left\{ w \;\middle|\; \text{the string } w \text{ occurs on some web page indexed by Google on May 3, 2007} \right\}.$$

4. **R** **C** **DEC** **NONDEC**

$$L = \left\{ w \;\middle|\; \begin{array}{l} w = x\#x_1\#x_2\#...\#x_n \text{ such that} \\ n \geq 1 \text{ and there is some } i \text{ for which } x \neq x_i \end{array} \right\}.$$

5. **R** **C** **DEC** **NONDEC**

$$L = \left\{ \mathtt{a}^i \mathtt{b}^j \;\middle|\; i + j = 27 \bmod 273 \right\}.$$

6. **R** **C** **DEC** **NONDEC**

$$L = L_1 \cap L_2,$$

where L_1 and L_2 are context-free languages

7. **R** **C** **DEC** **NONDEC**

$$L = \left\{ \langle \mathtt{T} \rangle \;\middle|\; \mathtt{T} \text{ is a } \mathsf{TM} \text{ and } \mathsf{L}(\mathtt{T})\, L(M) \text{ is finite} \right\}.$$

8. **R** **C** **DEC** **NONDEC**

$$L = L_1 \setminus L_2,$$

where L_1 is context-free and L_2 is regular.

9. **R** **C** **DEC** **NONDEC**

$$L = L_1 \cap L_2,$$

where L_1 is regular and L_2 is an arbitrary language.

Problem 3: Short answer I (8 points)

(a) Give a regular expression for the set of all strings in $\{0,1\}^*$ that contain at most one pair of consecutive 1's.

(b) Let M be a DFA. Sketch an algorithm for determining whether $L(M) = \Sigma^*$. Do not generate all strings (up to some bound on length) and feed them one-by-one to the DFA. Your algorithm must manipulate the DFA's state diagram.

Problem 4: Short answer II (8 points)

Let $\Sigma = \{a,b\}$, let $G = (V, \Sigma, R, S)$ be a CFG, and let $L = L(G)$. Give a grammar G' for the language $L' = \{wxw^R \mid w \in \Sigma^*, x \in L\}$ by modifying G appropriately.

Problem 5: Nonregularity (8 points)

Show that each of the languages below is not regular using the "direct argument" (you are advised to use the direct argument; however, if you wish to prove this in any other way, including the pumping lemma, you are welcome to do so). Assume $\Sigma = \{a, b\}$.

(a) $L = \left\{ w \;\middle|\; w \in \{\mathtt{a}, \mathtt{b}\}^* \text{ is a palindrome} \right\}$.

(b) $L = \left\{ w \;\middle|\; w \text{ contains at least twice as many } \mathtt{a}\text{'s as } \mathtt{b}\text{'s} \right\}$.

Problem 6: TM design (6 points)

Give the state diagram of a TM M that does the following on input $\#w\#$ where $w \in \{0,1\}^*$. Let $n = |w|$. If n is even, then M converts $\#w$ to $\#0^n\#$. If n is odd, then M converts $\#w$ to $\#1^n\#$. Assume that ϵ is an even length string.

The TM should enter the accept state after the conversion. We don't care where you leave the head at the end of the conversion. The TM should enter the reject state if the input string is not in the right format. However, your state diagram does not need to explicitly show the reject state or the transitions into it.

Problem 7: Subset construction (10 points)

Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA that does not contain any epsilon transitions. The *subset construction* can be used to construct a DFA $M = \{Q', \Sigma, \delta', q_0', F')$ recognizing the same language as N. Fill in the following key details of this construction, using your best mathematical notation:

Q' =????
q_0' =???
F' =????.

Suppose that P is a state in Q' and $\mathtt{a}$ is a character in Σ, then

$$\delta'(P, \mathtt{a}) = ?????$$

Suppose N has ϵ-transitions. How would your answer to the previous question change?

Problem 8: Writing a proof (8 points)

We have seen that $ALL_{CFG} = \left\{ \langle \mathcal{G} \rangle \;\middle|\; \mathcal{G} \text{ is a CFG and } \mathsf{L}(\mathcal{G}) = \Sigma^* \right\}$ is undecidable.

(a) Show that $\overline{ALL_{CFG}}$ is Turing-recognizable.

(b) Is ALL_{CFG} Turing-recognizable? Explain why or why not.

Problem 9: RA modification (8 points)

Let L be a context-free language on the alphabet Σ. Let $(M, main, \{(Q_m, \Sigma \cup M, \delta_m, q_0^m, F_m)\}_{m \in M})$ be an RA recognizing L. Give an RA recognizing the language

$$L' = \left\{ xy \;\middle|\; x \in L \text{ and } y \in \Sigma^* \text{ and } |y| \text{ is even} \right\}.$$

(a) Explain the idea behind the construction.

(b) Give tuple notation for new RA.

Problem 10: Decidability (6 points)

Show that

$$EQINT_{\mathsf{DFA}} = \left\{ \langle A, B, C \rangle \;\middle|\; \begin{array}{l} A, B, C \text{ are DFAs over the same alphabet } \Sigma, \\ \text{and } L(A) = L(B) \cap L(C) \end{array} \right\}.$$

is decidable.

This question does not require detail at the level of tuple notation. Rather, keep your proof short by exploiting theorems and constructions we've seen in class.

Problem 11: Reduction (8 points)

Let $EVEN_{\mathsf{TM}} = \{\langle M \rangle \mid \text{M is a TM and } L(M) \text{ does not contain any string of odd length}\}$. Show that $EVEN_{\mathsf{TM}}$ is undecidable (you may assumeA_{TM} - Turing machine membership - is undecidable).
You may *not* use Rice's Theorem.

Problem 12: Proof (8 points)

Prove that every binary string of even length can be derived from the following grammar:

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

48.5 Quiz 1 - Spring 2009

6 May 2009

Quiz I

(Q1) Consider $L = \left\{ \mathtt{a}^n \mathtt{b}^n \;\middle|\; n \geq 0 \right\}$ and the homomorphism $h(\mathtt{a}) = \mathtt{a}$ and $h(\mathtt{b}) = \mathtt{a}$.

Which of the following statements is true?

(a) L is regular and $h(L)$ is regular.

(b) L is regular and $h(L)$ is not regular.

(c) L is not regular and $h(L)$ is not regular.

(d) L is not regular and $h(L)$ is regular.

(Q2) Let $L = \left\{ x_1 \# x_2 \# ... \# x_k \;\middle|\; k \geq 2,\ \forall i, j\ \ x_i = x_j \text{ and } \forall x_i\ \ x_i \in \{\mathtt{a}, \mathtt{b}\}^* \right\}$. The language is

(a) regular, but not context free.

(b) context free and not regular.

(c) both regular and context free.

(d) neither regular nor context-free.

(Q3) Let $L = \left\{ \mathtt{a}^i \mathtt{b}^j \;\middle|\; (i + j) \bmod 5 = 3 \right\}$, where $x \bmod y$ denotes the remainder of dividing x by y. The language L is

(a) regular, but not context free.

(b) context free and not regular.

(c) both regular and context free.

(d) neither regular nor context-free.

(Q4) Let $L = \left\{ \mathtt{0}^n \mathtt{1} \mathtt{0}^n \;\middle|\; n \geq 0 \right\}$. The language L is

(a) regular, but not context free.

(b) context free, but not regular.

(c) both regular and context free.

(d) neither regular nor context-free.

(Q5) Let $\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{R}, \mathsf{S})$ be any CFG. Consider $\mathcal{G}' = (\mathcal{V} \cup S', \Sigma, \mathcal{R}', \mathsf{S}')$ where $\mathcal{R}' = \mathcal{R} \cup N$ and N is the additional rules

$$N = \left\{ \mathsf{S}' \to c\mathsf{S}'c \;\middle|\; c \in \Sigma \right\} \cup \{\mathsf{S}' \to S\}.$$

The language $L(\mathcal{G}')$ is

(a) $\left\{ wxw \;\middle|\; x \in L(\mathcal{G}), w \in \Sigma^* \right\}$

(b) $\left\{ wxw^R \;\middle|\; x \in L(\mathcal{G}), w \in \Sigma^* \right\}$

(c) $\left\{ w_1 x w_2 \;\middle|\; x \in L(\mathcal{G}), w_1, w_2 \in \Sigma^* \right\}$

(d) None of the above.

(e) All of the above.

(Q6) Suppose L is a context-free language (CFL) and R is regular, then $L \setminus R$ is CFL. This is

(a) True.

(b) False.

(Q7) Consider the grammar $\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{R}, \mathsf{S})$, where $\mathcal{V} = \{\mathsf{S}, \mathsf{X}, \mathsf{Y}\}$ with the rules (i.e., $\mathcal{R}$) defined as

$\mathsf{S} \rightarrow \mathtt{a}\mathsf{S}\mathtt{b} \mid \mathsf{X}$.

$\mathsf{X} \rightarrow \mathtt{c}\mathsf{X}\mathtt{d} \mid \mathsf{Y}$.

$\mathsf{Y} \rightarrow \mathtt{e}\mathsf{Y}\mathtt{f} \mid \epsilon$.

If we associate a language with each variable, what is the language of X ?

(a) $\left\{ \mathtt{e}^n\mathtt{f}^n \;\middle|\; n \geq 0 \right\}$

(b) $\left\{ \mathtt{c}^k\mathtt{e}^n\mathtt{f}^n\mathtt{d}^k \;\middle|\; n \geq 0, k \geq 0 \right\}$

(c) $\left\{ \mathtt{c}^n\mathtt{e}^n\mathtt{f}^n\mathtt{d}^n \;\middle|\; n \geq 0 \right\}$

(d) $\left\{ \mathtt{a}^n\mathtt{c}^n\mathtt{e}^n\mathtt{f}^n\mathtt{d}^n\mathtt{b}^n \;\middle|\; n \geq 0 \right\}$

(Q8) Consider the grammar with the rules

$\mathsf{S} \rightarrow \mathsf{A1B} \mid \mathsf{A2B}$.

$\mathsf{A} \rightarrow \mathsf{0A} \mid \epsilon$.

$\mathsf{B} \rightarrow \mathsf{C} \mid \mathsf{D}$

$\mathsf{D} \rightarrow \mathsf{0D} \mid \mathsf{1D} \mid \epsilon$.

$\mathsf{C} \rightarrow \mathsf{2C} \mid \epsilon$.

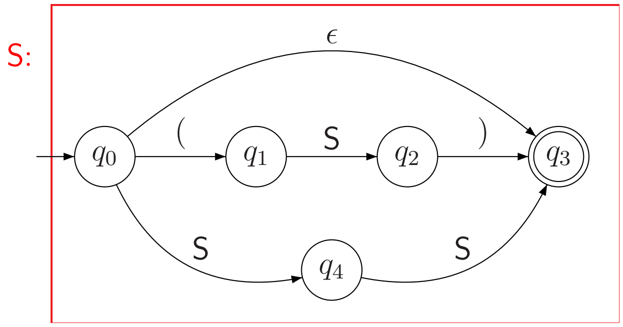
Which of the following strings can be derived in 0 or more steps?

(a) 0012202

(b) 002

(c) $(122222)^2$

(Q9) What is the language of the following recursive-automata?

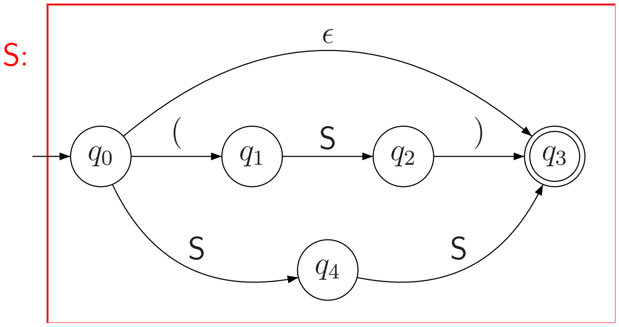


(a) $\left\{ (^n)^n \;\middle|\; n \geq 0 \right\}$

(b) $\left\{ (^n)^n(^m)^m \;\middle|\; n, m \geq 0 \right\}$

(c) $\left\{ w \;\middle|\; w \text{ is a string with balanced parenthesis } \right\}$.

(Q10) Consider the execution trace of the string (()) on the following recursive-automata.



As a reminder, a ***configuration*** of a recursive automata, is a pair (q, s) where q is a state, and s is the "call stack" of the recursive automata. Now, if the stack s contains the elements x, y, z (where z is the top of the stack), we write its content as $\langle x, y, z\rangle$, and the empty stack is denoted by $\langle\rangle$. As such, the initial configuration of the recursive automata is $\bigl(q_0, \langle\rangle\bigr)$.

$\bigl(q_0, \langle\rangle\bigr) \xrightarrow{(} \bigl(q_1, \langle\rangle\bigr) \xrightarrow{?} \bigl(q_0, \langle q_2\rangle\bigr) \xrightarrow{(} \bigl(?, \langle q_2\rangle\bigr) \xrightarrow{\mathsf{S}} \bigl(q_0, \langle ?\rangle\bigr) \xrightarrow{\epsilon} \bigl(q_3, \langle ?\rangle\bigr) \xrightarrow{?} \bigl(?, \langle q_2\rangle\bigr) \xrightarrow{)} \bigl(q_3, \langle q_2\rangle\bigr) \xrightarrow{?} \bigl(?, \langle ?\rangle\bigr) \xrightarrow{)} \bigl(?, \langle ?\rangle\bigr).$

Which one of the following is the correct execution trace?

(a) $\bigl(q_0, \langle\rangle\bigr) \xrightarrow{(} \bigl(q_1, \langle\rangle\bigr) \xrightarrow{\mathsf{S}} \bigl(q_0, \langle q_2\rangle\bigr) \xrightarrow{(} \bigl(q_1, \langle q_2\rangle\bigr) \xrightarrow{\mathsf{S}} \bigl(q_0, \langle q_2, q_2\rangle\bigr) \xrightarrow{\epsilon} \bigl(q_3, \langle q_2, q_2\rangle\bigr) \xrightarrow{\texttt{pop}} \bigl(q_2, \langle q_2\rangle\bigr) \xrightarrow{)} \bigl(q_3, \langle q_2\rangle\bigr) \xrightarrow{\texttt{pop}} \bigl(q_2, \langle\rangle\bigr) \xrightarrow{)} \bigl(q_3, \langle\rangle\bigr).$

(b) $\bigl(q_0, \langle\rangle\bigr) \xrightarrow{(} \bigl(q_1, \langle\rangle\bigr) \xrightarrow{\mathsf{S}} \bigl(q_0, \langle q_2\rangle\bigr) \xrightarrow{(} \bigl(q_1, \langle q_2\rangle\bigr) \xrightarrow{\texttt{pop}} \bigl(q_0, \langle q_2, q_2\rangle\bigr) \xrightarrow{\epsilon} \bigl(q_3, \langle q_2, q_2\rangle\bigr) \xrightarrow{\mathsf{S}} \bigl(q_2, \langle q_2\rangle\bigr) \xrightarrow{)} \bigl(q_3, \langle q_2\rangle\bigr) \xrightarrow{\texttt{pop}} \bigl(q_2, \langle\rangle\bigr) \xrightarrow{)} \bigl(q_3, \langle\rangle\bigr).$

(c) $\bigl(q_0, \langle\rangle\bigr) \xrightarrow{(} \bigl(q_1, \langle\rangle\bigr) \xrightarrow{\mathsf{S}} \bigl(q_4, \langle q_2\rangle\bigr) \xrightarrow{(} \bigl(q_4, \langle q_2\rangle\bigr) \xrightarrow{\mathsf{S}} \bigl(q_0, \langle q_2, q_2\rangle\bigr) \xrightarrow{\epsilon} \bigl(q_3, \langle q_2, q_2\rangle\bigr) \xrightarrow{\texttt{pop}} \bigl(q_2, \langle q_2\rangle\bigr) \xrightarrow{)} \bigl(q_3, \langle q_2\rangle\bigr) \xrightarrow{\texttt{pop}} \bigl(q_2, \langle\rangle\bigr) \xrightarrow{)} \bigl(q_3, \langle\rangle\bigr)$

48.6 Quiz 2 – Spring 2009

6 May 2009

(Q1) Consider the Turing Machine , $TM = \{Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej}\}$ where $Q = \{q_0, q_1, q_2, q_{acc}, q_{rej}\}$, $\Sigma = \{0, 1\}$ and $\Gamma = \{0, 1, B\}$ and the transitions are defined as follows:

$\delta(q_0, 0) = (q_1, 0, R)$; $\delta(q_0, 1) = (q_1, 1, R)$;

$\delta(q_1, 0) = (q_2, 0, R)$; $\delta(q_1, 1) = (q_2, 1, R)$;

$\delta(q_2, B) = (q_{acc}, B, R)$ and all other transitions are to the reject state.

Assuming that we start the TM on some input and the current configuration is $1q_1100$, what is the next configuration?

(a) $11q_200$.
(b) q_21100.
(c) $11q_100$.

(Q2) Consider the following statements about the TM described in the question above. Which of them is correct?

(a) the TM always halts.
(b) there exist inputs on which the TM does not halt.
(c) there exist inputs on which the TM halts and inputs on which it does not halt.

(Q3) How many Turing machines can you have with the same Q, Σ, Γ as described in the first question?

(a) 30^9.
(b) 30^6.
(c) 30^{15}.
(d) infinitely many.

(Q4) Which of the following is true?

The set of Turing recognizable languages is closed under

(a) complementation and intersection.
(b) intersection.
(c) complementation.
(d) neither intersection nor complement.

(Q5) Suppose L is undecidable and suppose L is a Turing recognizable language. Which of the following statements is true?

(a) $\overline{L}$ is not recognizable.
(b) $\overline{L}$ is recognizable.
(c) $\overline{L}$ may or may not be recognizable.

(Q6) Which of the following properties is a property of the language of TM M?

(a) M is a TM that has 481 states.
(b) M is a TM and $|L(M)| = 481$.
(c) M is a TM takes more than 481 steps on some input.

(Q7) Consider the language $L = \left\{\langle M\rangle \,\middle|\, \text{takes no more than 481 steps on some input}\right\}$

Which of the following statements is true?

(a) L is decidable.

(b) L is not decidable.

(c) L may be decidable or may not be decidable.

(Q8) Consider the language $L = \left\{\langle M\rangle \,\middle|\, M \text{ has 481 states}\right\}$

Which of the following statements is true?

(a) L is decidable and recognizable.

(b) L is decidable but not recognizable.

(c) L is not decidable but recognizable.

(d) L is not decidable nor recognizable.

(Q9) For any TM M and word w, consider a procedure $WriteN(M, w)$ that produces the following procedure N as output:

$$N(x)\{\text{simulate } M \text{ on } w;\quad \text{if } M \text{ accepts } w \text{ then accept } x \text{ else reject } x\}$$

Assume the procedure $IsEmpty(M)$ is one that takes a Turing machine M as input and returns ”yes” if $L(M)$ is empty and ”No” otherwise.

Consider this following procedure:

$$Decider(M, w)\{N = \text{WriteN; return IsEmpty(N)}\}$$

What does the above procedure prove?

(a) $L = \left\{M \,\middle|\, L(M) = \emptyset\right\}$ is undecidable as A_{TM} reduces to L.

(b) $L = \left\{M \,\middle|\, L(M) = \Sigma^*\right\}$ is undecidable as A_{TM} reduces to L.

(c) A_{TM} is undecidable as L reduces to A_{TM}.

(d) $L = \left\{M \,\middle|\, L(M) = \emptyset\right\}$ is decidable.

(Q10) Let $G = \left\{\langle M, N\rangle \,\middle|\, L(M) = L(N)\right\}$ Which of the following statements is true?

(a) G is decidable.

(b) G is undecidable by Rices theorem.

(c) G is not decidable as we can reduce $L = \left\{M \,\middle|\, L(M) = \emptyset\right\}$ to it .

(d) G is decidable as we can reduce $L = \left\{M \,\middle|\, L(M) = \emptyset\right\}$ to it .

Chapter 49

Exams – Spring 2008

49.1 Midterm 1

19 February 2008

Problem 1: Short Answer (8 points)

The answers to these problems should be short and not complicated.

1. If an NFA M accepts the empty string (ϵ), does M's start state have to be an accepting state? Why or why not?
2. Is every finite language regular? Why or why not?
3. Suppose that an NFA $M = (Q, \Sigma, \delta, q_0, F)$ accepts a language L. Create a new NFA M' by flipping the accept/non-accept markings on M. That is, $M' = (Q, \Sigma, \delta, q_0, Q - F)$. Does M' accept $\overline{L}$ (the set complement of L)? Why or why not?
4. Simplify the following regular expression $\emptyset^*(a \cup b) \cup \emptyset b^* \cup \epsilon abb$

Problem 2: DFA design (6 points)

Let $\Sigma = \{a, b\}$. Let L be the set of strings in Σ^* which contain the substring bba or the substring aaa.

For example, $aabba \in L$ and $baaab \in L$, but $babab \notin L$. Strings shorter than three characters are never in L.

Construct a DFA that accepts L and give a state diagram showing **all** states in the DFA.

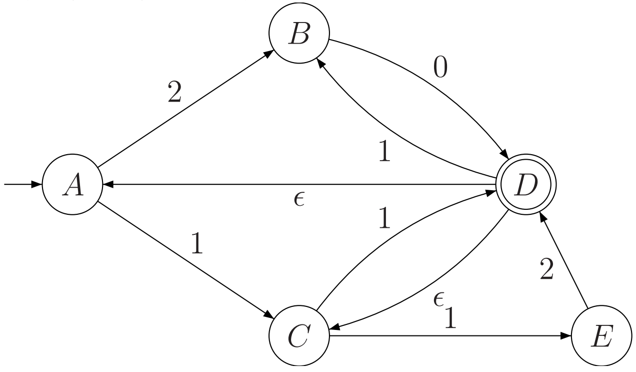
You will receive **zero** *credit if your DFA uses more than* **10** *states or makes significant use of non-determinism.*

Problem 3: Remembering definitions (8 points)

1. Define formally what it means for a DFA $(Q, \Sigma, \delta, q_0, F)$ to accept a string $w = w_1 w_2 \ldots w_n$.
2. Let Σ and Γ be alphabets. Suppose that h is a function from Σ^* to Γ^*. Define what it means for h to be a homomorphism.

Problem 4: NFA transitions (6 points)

Suppose that the NFA $N = (Q, \{0, 1, 2\}, \delta, q_0, F)$ is defined by the following state diagram:



Fill in the following values:

(a) $F =$

(b) $\delta(A, 0) =$

(c) $\delta(C, 1) =$

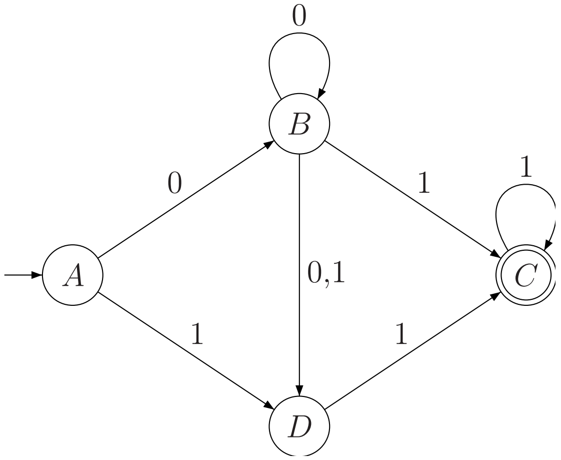
(d) $\delta(D, 1) =$

(e) List the members of the set $\{q \in Q \mid D \in \delta(q, 2)\}$:

(f) Does the NFA accept the word 11120? (Yes / No)

Problem 5: NFA to DFA conversion (6 points)

Convert the following NFA to a DFA recognizing the same language, using the subset construction. Give a state diagram showing all states reachable from the start state, with an informative name on each state. Assume the alphabet is $\{0, 1\}$.



Problem 6: Short Construction (8 points)

1. Give a regular expression for the language L containing all strings in a^*b^* whose length is a multiple of three. E.g. L contains $Taaaabb$ but does not contain ababab or aaabb.

2. Let $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. Give an NFA for the language L containing all strings in Σ^* which have an a or a c in the last four positions. E.g. bbabbb and abbbcb are both in L, but acabbbb is not. Notice that strings of length four or less are in L exactly when they contain an a or a c.

   You will receive **zero** *credit if your NFA contains more than 8 states.*

Problem 7: NFA modification and tuple notation (8 points)

For this problem, the alphabet is always $\Sigma = \{a, b\}$.

Given an NFA M that accepts the language L, design a new NFA M' that accepts the language $L' = \left\{twt \,\middle|\, w \in L, t \in \Sigma\right\}$. For example, if aab is in L, then aaaba and baabb are in L'.

(a) Briefly explain the idea behind your construction, using English and/or pictures.

(b) Suppose that $M = (Q, \Sigma, \delta, q_0, F)$. Give the details of your construction of M', using tuple notation.

49.2 Midterm 2

27 March 2008

Problem 1: Short Answer (12 points)

Answer “yes” or “no” to the following questions. No explanations are required. (All the strings in this problem use the same, fixed alphabet.)

(a) Is the language $\left\{ ww^R w \;\middle|\; w \in \{\mathtt{a}, \mathtt{b}\}^* \right\}$ a context-free language?

Yes: □ **No**: □

(b) If L is a non-regular language over Σ^*, and h is a homomorphism, then $h(L)$ must also be non-regular. Is this statement correct?

Yes: □ **No**: □

(c) Suppose all the words in language L are no more than 1024 characters long. Then L must be regular. Is this statement correct?

Yes: □ **No**: □

(d) If L_1 and L_2 be two languages, the ***xor*** of the two languages is

$$L_1 \oplus L_2 = (L_1 \setminus L_2) \cup (L_2 \setminus L_1) = \left\{ w \;\middle|\; \begin{array}{c} w \in L_1 \text{ and } w \notin L_2 \\ \text{or} \\ w \notin L_1 \text{ and } w \in L_2 \end{array} \right\}.$$

If L_1 and L_2 are both context-free, then $L_1 \oplus L_2$ must also be context-free. Is this statement correct?

Yes: □ **No**: □

(e) If L is a language, its prefix language is

$$P(L) = \left\{ w \;\middle|\; \text{there exists } x \text{ s.t. } wx \in L \right\}.$$

If L is context-free, then the language $P(L)$ is context free. Is this statement correct?

Yes: □ **No**: □

(f) A PDA that is allowed to enter the accept state only if the stack is empty is a ***strict PDA*** . There are context-free languages for which no strict PDA exists. Is this statement correct? **Yes**: □ **No**: □

Problem 2: Grammar design (8 points)

Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$.

Let $J = \left\{ w_1\#w_2\#\ldots\#w_{n-1}\#w_n \;\middle|\; n \geq 2, w_i \in \Sigma^* \text{ for all } i, \text{ and for some } i, |w_i| = |w_{i+1}| \right\}$

In other words, an element of J is a list of at least two strings of a’s and b’s, separated by #’s. In the list, some pair of adjacent strings have the same length. E.g. J contains `b#aa#bbb#aba` but not `a#bbb#b`.

Give a context-free grammar whose language is J. Be sure to indicate what its start symbol is.

Problem 3: PDA design (8 points)

Let

$$J = \left\{ w \in \{\mathtt{a},\mathtt{b}\}^* \;\middle|\; \begin{array}{c} w \text{ contains only } \mathtt{a}\text{'s} \\ \text{or} \\ w \text{ has an equal number of } \mathtt{a}\text{'s and } \mathtt{b}\text{'s} \end{array} \right\}.$$

For example, J contains ϵ, `aaa`, and `aabbba`. But `abbba` is not in J.

Give the state diagram for a PDA whose language is J. Include brief comments explaining the design of your PDA, to help us understand how it works.

Problem 4: Short Answer II (8 points)

The answers to these problems should be short and not complicated.

(a) Suppose we know that the language $B = \left\{ \mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \;\middle|\; n \geq 2 \right\}$ is not context-free. Let L be the language $L = \left\{ \mathtt{a}^n\mathtt{bc}^n\mathtt{bd}^n \;\middle|\; n \geq 0 \right\}$. Prove that L is not context-free using closure properties and the fact that B is not context-free.

(b) Define what it means for a grammar G to be in Chomsky Normal Form (CNF).

Problem 5: Pumping Lemma (8 points)

Suppose $\Sigma = \{\mathtt{a},\mathtt{b}\}$ and let $L = \left\{ \mathtt{a}^n w \;\middle|\; w \in \Sigma^* \text{ and } |w| = n \right\}$. That is, L contains even-length strings whose first half contains only a's. Prove that L is not regular by filling in the missing parts of the following pumping lemma proof.

Suppose that L were regular. Let p be the constant given by the pumping lemma.

Consider the string $w_p = \underset{\textbf{Fill in}}{\boxed{}}$.

Because $w_p \in L$ and $|w_p| \geq p$, there must exist strings x, y, and z such that $w_p = xyz$, $|xy| \leq p$, $|y| > 0$, and $xy^iz \in L$ for every $i \geq 0$.

Since $\underset{\textbf{Fill in}}{\boxed{}}$ is *not* in L, we have a contradiction. Therefore, L must not have been regular.

Problem 6: Formal notation (8 points)

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a PDA recognizing the language L, here L is defined over an alphabet Σ. Define the language L' to be

$$L' = \left\{ x\#y \;\middle|\; x, y \in L \right\}.$$

Describe how to construct a PDA recognizing L'. (You can safely assume that $\#$ is not in the alphabets Σ and Γ used by M and L.)

(a) Describe the ideas behind your construction in words and/or pictures.

(b) When you read the $\#$ from the input, or shortly after you read it, you will need to do something about whatever is left on the stack from reading the x part of the input string. Do you need to push anything onto the stack or pop anything off? If so, what? Namely, describe what your PDA does upon reading $\#$.

(c) Give the details of your construction in formal notation. That is, for the new PDA recognizing L', specify the set of states, the initial and final states, the stack alphabet, the details of the transition function. The input alphabet for the new machine will be $\Sigma \cup \{\#\}$.

Problem 7: Induction (8 points)

Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$. Given any string w in Σ^*, let $A(w)$ be the number of a's in w and $B(w)$ be the number of b's in w.

Suppose that grammar G has the following rules:

$$\mathsf{S} \rightarrow \mathtt{a}\mathsf{S}\mathtt{b} \mid \mathtt{a}\mathsf{SS} \mid \mathtt{ab},$$

where S is the start symbol. Use induction on the derivation length to prove that $A(w) \geq B(w)$ for any string w in $L(G)$.

49.3 Final – Spring 2008

7 May 2008

INSTRUCTIONS (read carefully)

- Print your name, netID, and section time in the boxes above.
- The exam contains 9 problems on pages numbered 1 through 10. Make sure you have a complete exam.
- You have three hours.
- The point value of each problem is indicated next to the problem and in the table on the right.
- It is wise to skim all problems and point values first, to best plan your time. If you get stuck on a problem, move on and come back to it later.
- Points may be deducted for solutions which are correct but excessively complicated, hard to understand, hard to read, or poorly explained.
- If you change your answers (especially for T/F questions or multiple-choice questions), be sure to erase well or otherwise make sure your final choice is clear.
- Please bring apparent bugs or unclear questions to the attention of the proctors.
- This is a closed book exam. You may only consult a cheat sheet, handwritten (by yourself) on a single 8.5 x 11 inch sheet (both sides is ok). You can only use your normal eyeglasses (if any) to read it, e.g. no magnifying glasses.
- Please submit your cheat sheet with the exam.
- Do all work in the space provided, using the backs of sheets if necessary. See the proctor if you need more paper.

Problem 1: True/False (18 points)

Decide whether each of the following statements is true or false, and check the corresponding box. You do not need to explain or prove your answers. Read the questions very carefully. They may mean something quite different than what they appear to mean at first glance.

(a) If L is TM recognizable, and $\overline{L}$ is TM recognizable, then L is TM decidable.

False: □ **True**: □

(b) Let G be a context-free grammar given in Chomsky normal form (CNF). Since the grammar G is in CNF form it must be ambiguous.

False: □ **True**: □

(c) A non-deterministic TM can decide languages that a regular TM cannot decide.

False: □ **True**: □

(d) Suppose that the language $L = \left\{ \mathtt{a}^n \mathtt{b}^n \mathtt{c}^n \;\middle|\; n \geq 0 \right\}$ is not context-free. Let $h(\cdot)$ be a homomorphism. Then the language $h(L)$ cannot be context-free.

False: □ **True**: □

(e) For any $k > 1$, there is no language that is decided by a TM with k tapes, but is undecidable by any TM having $k - 1$ (or less) tapes.

False: □ **True**: □

(f) If a language L is context-free then $\overline{L}$ is TM decidable.

False: □ **True**: □

(g) If a language L is regular and recognized by a DFA with k states, then either L is infinite, or L is finite and contains only words of length $< k$.

False: □ **True**: □

(h) If a language L is accepted by a PDA P then $\overline{L}$ will be accepted by a PDA R which is just like P except that the set of accepting states has been complemented.

False: □ **True**: □

(i) The language $\overline{\mathrm{A}_{\mathsf{TM}}} = \left\{ \langle M, w \rangle \;\middle|\; M \text{ does not accept } w \right\}$ is TM recognizable.

False: □ **True**: □

Problem 2: Classification (16 points)

For each language L described below, we have listed 2–3 language classes. Mark the most restrictive listed class to which L must belong. E.g. if L must always be regular and we have listed "regular" and "context-free", mark only "regular".

(a) $L = \left\{ xw \;\middle|\; x, w \in \{\mathtt{a}, \mathtt{b}\}^* \text{ and } |x| = |w| \right\}$

□ **Regular** □ **Context-free** □ **TM-Decidable**

(b) $L = \left\{ \langle G \rangle \;\middle|\; G \text{ is a } \mathsf{CFG} \text{ and } G \text{ is not ambiguous} \right\}$

□ **TM-Decidable** □ **TM-Recognizable** □ **Not TM recognizable**

(c) $L = \left\{ \mathtt{a}^i \mathtt{b}^j \mathtt{c}^k \mathtt{d}^m \;\middle|\; i + j + k + m \text{ is a multiple of } 13 \right\}$

□ **Regular** □ **Context-free** □ **TM-Decidable**

(d) $L = \left\{ \langle w, M_1, M_2, \ldots, M_k \rangle \;\middle|\; \begin{array}{l} w \text{ is a string,} \\ k \text{ is an odd number larger than 2,} \\ \text{each } M_i \text{ is a } \mathsf{TM}, \\ \text{and a majority of the } M_i\text{'s accept } w \end{array} \right\}$

□ **TM-Decidable** □ **TM-Recognizable** □ **Not TM recognizable**

(e) $L = \{x_1\#x_2\#\ldots\#x_n \mid x_i \in \{a,b\}^*$ for each i and, for some i, x_i is a palindrome$\}$.

□ **Regular** □ **Context-free** □ **TM-Decidable**

(f) $L = \left\{\langle G, D\rangle \,\middle|\, G \text{ is a } \mathsf{CFG}, D \text{ is a } \mathsf{DFA}, \text{and } L(G) \subseteq L(D)\right\}$

□ **TM-Decidable** □ **TM-Recognizable** □ **Not TM recognizable**

(g) $L = \left\{\langle G\rangle \,\middle|\, G \text{ is a } \mathsf{CFG} \text{ and } L(G) \text{ is finite}\right\}$

□ **TM-Decidable** □ **TM-Recognizable** □ **Not TM recognizable**

(h) $L = \left\{\langle G\rangle \,\middle|\, G \text{ is a } \mathsf{CFG} \text{ and } L(G) \neq \Sigma^*\right\}$

□ **TM-Decidable** □ **TM-Recognizable** □ **Not TM recognizable**

Problem 3: Context-free grammars (9 points)

Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and consider the language

$$L = \left\{x\#y^r \,\middle|\, x, y \in \Sigma^* \text{ and } x \neq y\right\}$$

Notice that the lengths of x and y are not required to be equal.

(a) **[4 points]** Give a context-free grammar for L for the case that x and y have the same length, where the start symbol is S.

(b) **[4 points]** Give a context-free grammar for L for the case that x and y have *different* lengths, where the start symbol is X. (You can use portions of the grammar you created for part (a) in your answer, if you want to. Do not re-use variables from your answer to part (a).)

(c) **[1 point]** Give a context-free grammar for L for all cases. Let the start symbol be Y. (You can use portions of the grammar you created for parts (a) and (b) in your answer, if you want to.)

Problem 4: Pumping Lemma (8 points)

Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$. Let

$$L = \left\{ x_1\#x_2\#\ldots\#x_k \,\middle|\, \begin{array}{l} k \geq 2 \\ \text{for all } i, x_i \in \Sigma^* \\ \text{ and, for some } i \neq j, \text{ we have } x_i = x_j \end{array} \right\}.$$

Prove that L is not regular by filling in the missing parts of the following pumping lemma proof.

Suppose that L were regular. Let p be the pumping length given by the pumping lemma.

Consider the string $w_p = \boxed{\qquad\qquad\qquad\qquad\text{‘}}$

Because $w_p \in L$ and $|w_p| \geq p$, there must exist strings x, y, and z such that $w_p = xyz$, $|xy| \leq p$, $|y| > 0$, and $xy^iz \in L$ for every $i \geq 0$.

Since isn't in L, because

As such, we have a contradiction. Therefore, L must not have been regular.

Problem 5: Short answer (10 points)

(a) Give an example of an ambiguous grammar. Show that it is ambiguous by showing two parse trees for the same string w.

(b) Recall that a Linear Bounded Automaton is a Turing machine that cannot move its head off the portion of the tape occupied by the input string. Also recall that

$$\mathrm{A}_{\mathsf{LBA}} = \left\{ \langle M, w \rangle \;\middle|\; M \text{ is an } \mathsf{LBA} \text{ and } M \text{ accepts } w. \right\}.$$

Explain why $\mathrm{A}_{\mathsf{LBA}}$ is Turing decidable.

Problem 6: TM variations (10 points)

A ***unistate TM*** (U_{TM}) is a TM with a single state q, and two tapes. A U_{TM} M starts with the input string w at the left of tape 1, and with tape 2 initially blank. M accepts w if M writes the special character $\bowtie$ anywhere on tape 2.

Prove that if M is any TM, then there is a U_{TM} M' such that $L(M') = L(M)$. (Hint: the tape alphabet of M' will contain many extra symbols.)

(a) First, give the basic idea, but giving enough details so that it is clear you understand what the key issues are.

Next, give the following details:

(b) What is the tape alphabet Γ' of M' in terms of the description of M? _________________

Let δ be the transition function of M, and δ' be the transition function of M', recalling that M' is a two-tape machine. Suppose that in M, $\delta(p, \mathtt{a}) = (p', \mathtt{b}, D)$ where $D \in \{L, R, S\}$ indicates that M moves left, right, or remains in one place, respectively. Then your construction of M' should somehow simulate this transition of M. Below, give values of the place-holder keywords to describe how your simulation works.

(c) For the above transition of M, we create the transition

δ'(state, tape1sym, tape2sym) = (new-state, new-tape1sym, new-tape2sym, direction1, direction2)

where (fill in the blanks)

state = _______ tape1sym = _______ tape2sym = _______

new-state = _______ new-tape1sym = _______ new-tape2sym = _______

direction1 = _______ direction2 = _______

Problem 7: Palindrome (10 points)

Let $L = \Big\{\langle M\rangle \;\Big|\; M \text{ is a Turing machine and } M \text{ accepts at least one palindrome}\Big\}$.

Show that L is TM-recognizable, i.e. explain how to construct a Turing machine that accepts $\langle M\rangle$ exactly when M accepts at least one palindrome. Assume that it is easy to extract M's alphabet Σ from $\langle M\rangle$. Of course, M might run forever on some input strings.

Problem 8: Reduction (10 points)

Recall that $\mathrm{A}_{\mathsf{TM}} = \Big\{\langle M, w\rangle \;\Big|\; M \text{ is a Turing machine and } M \text{ accepts } w\Big\}$ and A_{TM} is undecidable.

Let $L = \Big\{\langle M\rangle \;\Big|\; M \text{ is a Turing machine and } L(M) \text{ is infinite}\Big\}$.
Show that L is undecidable using a reduction from A_{TM}. Prove this directly, not by citing Rice's Theorem.

Problem 9: Decidability (9 points)

We showed in class that $\mathsf{E}_{\mathsf{CFG}}$ is decidable, where E_{CFG} is the language

$$\mathsf{E}_{\mathsf{CFG}} = \Big\{\langle G\rangle \;\Big|\; G \text{ is a } \mathsf{CFG} \text{ and } L(G) = \emptyset\Big\}.$$

Also, let $J = \Big\{\mathtt{a}^n\mathtt{b}^n \;\Big|\; n \geq 0\Big\}$, and consider the language L:

$$L = \Big\{\langle D\rangle \;\Big|\; D \text{ is a } \mathsf{DFA} \text{ and } D \text{ accepts some string in J}\Big\}.$$

Suppose that R is a decider for E_{CFG} and M is a PDA accepting J. Show how to construct a decider for L .

49.4 Mock Final

Spring 2007

This mock final exam was generated from previous semester. Some parts where modified or removed (if they were not relevant).

INSTRUCTIONS (read carefully)

- Print your name and netID here and netID at the top of each other page.
- The exam contains 12 pages and 11 problems. Make sure you have a complete exam.
- You have three hours.
- The point value of each problem is indicated next to the problem and in the table below.
- It is wise to skim all problems and point values first, to best plan your time. If you get stuck on a problem, move on and come back to it later.
- Points may be deducted for solutions which are correct but excessively complicated, hard to understand, hard to read, or poorly explained.
- This is a closed book exam. No notes of any kind are allowed. Do all work in the space provided, using the backs of sheets if necessary. See the proctor if you need more paper.
- Please bring apparent bugs or unclear questions to the attention of the proctors.

Problem 1: True/False (10 points)

Completely write out "True" if the statement is necessarily true. Otherwise, completely write "False". Other answers (e.g. "T") will receive credit only if your intent is unambiguous. For example, "$x+y > x$" has answer "False" assuming that y could be 0 or negative. But "If x and y are natural numbers, then $x + y \geq x$" has answer "True". You do not need to explain or prove your answers.

(a) Let M be a DFA with n states such that $L(M)$ is infinite. Then $L(M)$ contains a string of length at most $2n - 1$.

(b) Let $L_w = \{< M >| M \text{ is a TM and } M \text{ accepts } w\}$, where w is some fixed string. Then there is an enumerator for L_w.

(c) The set of undecidable languages is countable.

(d) For a TM M and a string w, let $CH_{M,w} = \{x \mid x \text{ is an accepting computation history for } M \text{ on } w\}$. Then $CH_{M,w}$ is decidable.

(e) The language $\{< M, w >| M \text{ is a linear bounded automaton and } M \text{ accepts } w\}$ is undecidable.

(f) The language $\{< G >| G \text{ is a context-free grammar and } G \text{ is ambiguous}\}$ is Turing-recognizable.

(g) Context-free languages are closed under homomorphism.

(h) There is a bijection between the set of Turing-recognizable languages and the set of decidable languages.

Problem 2: Classification (20 points)

For each language L described below, classify L as

- **R**: Any language satisfying the information must be regular.
- **C**: Any language satisfying the information must be context-free, but not all languages satisfying the information are regular.
- **DEC**: Any language satisfying the information must be decidable, but not all languages satisfying the information are context-free.
- **NONDEC**: Not all languages satisfying the information are decidable. (Some might be only Turing recognizable or perhaps even not Turing recognizable.)

For each language, circle the appropriate choice (**R**, **C**, **DEC**, or **NONDEC**). If you change your answer be sure to erase well or otherwise make your final choice clear. **Ambiguously marked answers will receive no credit.**

1. **R C DEC NONDEC**
   $L = \{< M >|$ M is a linear bounded automaton and $L(M) = \emptyset\}$.

2. **R C DEC NONDEC**
   $L = \{ww^Rw \mid w \in \{a, b\}^*\}$.

3. **R C DEC NONDEC**
   $L = \{w \mid$ the string w occurs on some web page indexed by Google on May 3, 2007$\}$

4. **R C DEC NONDEC**
   $L = \{w \mid w = x\#x_1\#x_2\#...\#x_n$ such that $n \geq 1$ and there is some i for which $x \neq x_i\}$.

5. **R C DEC NONDEC**
   $L = \{a^ib^j \mid i + j = 27 \bmod 273\}$

6. **R C DEC NONDEC**
   $L = L_1 \cap L_2$ where L_1 and L_2 are context-free languages

7. **R C DEC NONDEC**
   $L = \{< M >| M$ is a TM and $L(M)$ is finite$\}$

8. **R C DEC NONDEC**
   $L = L_1 - L_2$ where L_1 is context-free and L_2 is regular.

9. **R C DEC NONDEC**
   $L = L_1 \cap L_2$ where L_1 is regular and L_2 is an arbitrary language.

Problem 3: Short answer I (8 points)

(a) Give a regular expression for the set of all strings in $\{0, 1\}^*$ that contain at most one pair of consecutive 1's.

(b) Let M be a DFA. Sketch an algorithm for determining whether $L(M) = \Sigma^*$. Do not generate all strings (up to some bound on length) and feed them one-by-one to the DFA. Your algorithm must manipulate the DFA's state diagram.

Problem 4: Short answer II (8 points)

(a) Let $\Sigma = \{a, b\}$, let $G = (V, \Sigma, R, S)$ be a CFG, and let $L = L(G)$. Give a grammar G' for the language $L' = \{wxw^R \mid w \in \Sigma^*, x \in L\}$ by modifying G appropriately.

Problem 5: Pumping Lemma (8 points)

For each of the languages below you can apply the pumping lemma either to prove that the language is non-regular or to prove that the language is not context-free. Assuming that p is the pumping length, your goal is to give a candidate string for w_p that can be pumped appropriately to obtain a correct proof. You ONLY need to give the string and no further justification is necessary. Assume $\Sigma = \{a, b\}$ unless specified explicitly.

(a) $L = \{w \mid w \text{ is a palindrome}\}$. To show L is not regular using the pumping lemma $w_p =$

(b) $L = \{w \mid w \text{ contains at least twice as many } \mathtt{a}\text{'s as } \mathtt{b}\text{'s}\}$. To show L is not regular using the pumping lemma

$w_p =$

Problem 6: TM design (6 points)

Give the state diagram of a TM M that does the following on input $\#w$ where $w \in \{0,1\}^*$. Let $n = |w|$. If n is even, then M converts $\#w$ to $\#0^n$. If n is odd, then M converts $\#w$ to $\#1^n$. Assume that ϵ is an even length string.

The TM should enter the accept state after the conversion. We don't care where you leave the head at the end of the conversion. The TM should enter the reject state if the input string is not in the right format. However, your state diagram does not need to explicitly show the reject state or the transitions into it.

Problem 7: Subset construction (10 points)

Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA that does not contain any epsilon transitions. The *subset construction* can be used to construct a DFA $M = \{Q', \Sigma, \delta', q_0', F')$ recognizing the same language as N. Fill in the following key details of this construction, using your best mathematical notation:

$Q' =$
$q_0' =$
$F' =$

Suppose that P is a state in Q' and a is a character in Σ. Then

$\delta'(P, a) =$

Suppose N has ϵ-transitions. How would your answer to the previous question change?

$\delta'(P, a) =$

Problem 8: Writing a proof (8 points)

We have seen that $ALL_{CFG} = \{< G >\mid G \text{ is a CFG and } L(G) = \Sigma^*\}$ is undecidable.

1. Show that $\overline{ALL_{CFG}}$ is Turing-recognizable.

2. Is ALL_{CFG} Turing-recognizable? Explain why or why not.

Problem 9: PDA modification (8 points)

Let L be a context-free language on the alphabet Σ. Let $M = (Q, \Sigma, \Gamma, \delta, q, F)$ be a PDA recognizing L. Give a PDA M' recognizing the language $L' = \{xy \mid x \in L \text{ and } y \in \Sigma^* \text{ and } |y| \text{ is even}\}$

Note: If you make assumptions about M, then state them clearly and justify.

1. Explain the idea behind the construction.

2. Give tuple notation for M'.

Problem 10: Decidability (6 points)

Show that

$$EQINT_{DFA} = \left\{ \langle A, B, C \rangle \;\middle|\; \begin{array}{l} A, B, C \text{ are } \mathsf{DFA}\text{s over the same alphabet } \Sigma \\ \text{and } L(A) = L(B) \cap L(C) \end{array} \right\}$$

is decidable.

This question does not require detail at the level of tuple notation. Rather, keep your proof short by exploiting theorems and constructions we've seen in class.

Problem 11: Reduction (8 points)

Let $ODD_{TM} = \{\langle M \rangle \mid \text{M is a TM and } L(M) \text{ does not contain any string of odd length}\}$. Show that ODD_{TM} is undecidable using a reduction from A_{TM} (Turing machine membership). You may not use Rice's Theorem.

49.5 Mock Final with Solutions

Spring 2007

Problem 1: True/False (10 points)

Completely write out "True" if the statement is necessarily true. Otherwise, completely write "False". Other answers (e.g. "T") will receive credit only if your intent is unambiguous. For example, "$x+y > x$" has answer "False" assuming that y could be 0 or negative. But "If x and y are natural numbers, then $x + y \geq x$" has answer "True". You do not need to explain or prove your answers.

1. Let M be a DFA with n states such that $L(M)$ is infinite. Then $L(M)$ contains a string of length at most $2n - 1$.

   Solution: True.

2. Let $L_w = \{< M >| M$ is a TM and M accepts $w\}$, where w is some fixed string. Then there is an enumerator for L_w.

   Solution: True.

3. The set of undecidable languages is countable.

   Solution: False.

4. For a TM M and a string w, let $CH_{M,w} = \{x \mid x$ is an accepting computation history for M on $w\}$. Then $CH_{M,w}$ is decidable.

   Solution: True.

5. The language $\{< M, w >| M$ is a linear bounded automaton and M accepts $w\}$ is undecidable.

   Solution: False.

6. The language $\{< G >| G$ is a context-free grammar and G is ambiguous$\}$ is Turing-recognizable.

   Solution: True.

7. Context-free languages are closed under homomorphism.

   Solution: True.

8. The modified Post's correspondence problem is Turing recognizable.

   Solution: True.

9. There is a bijection between the set of Turing-recognizable languages and the set of decidable languages.

   Solution: True.

Problem 2: Classification (20 points)

For each language L described below, classify L as

- **R**: Any language satisfying the information must be regular.
- **C**: Any language satisfying the information must be context-free, but not all languages satisfying the information are regular.
- **DEC**: Any language satisfying the information must be decidable, but not all languages satisfying the information are context-free.
- **NONDEC**: Not all languages satisfying the information are decidable. (Some might be only Turing recognizable or perhaps even not Turing recognizable.)

For each language, circle the appropriate choice (**R**, **C**, **DEC**, or **NONDEC**). If you change your answer be sure to erase well or otherwise make your final choice clear. **Ambiguously marked answers will receive no credit.**

1. **R C DEC NONDEC**
   $L = \{< M >|$ M is a linear bounded automaton and $L(M) = \emptyset\}$.

   Solution: NONDEC

2. **R C DEC NONDEC**
   $L = \{ww^Rw \mid w \in \{a, b\}^*\}$.

   Solution: DEC

3. **R C DEC NONDEC**
   $L = \{w \mid$ the string w occurs on some web page indexed by Google on May 3, 2007$\}$

   Solution: R

4. **R C DEC NONDEC**
   $L = \{w \mid w = x\#x_1\#x_2\#...\#x_n$ such that $n \geq 1$ and there is some i for which $x \neq x_i\}$.

   Solution: C

5. **R C DEC NONDEC**
   $L = \{a^ib^j \mid i + j = 27 \bmod 273\}$

   Solution: R

6. **R C DEC NONDEC**
   $L = L_1 \cap L_2$ where L_1 and L_2 are context-free languages

   Solution: DEC

7. **R C DEC NONDEC**
   $L = \{< M >| M$ is a TM and $L(M)$ is finite$\}$

   Solution: UNDEC

8. **R C DEC NONDEC**
   $L = L_1 - L_2$ where L_1 is context-free and L_2 is regular.

   Solution: C

9. **R C DEC NONDEC**
   $L = L_1 \cap L_2$ where L_1 is regular and L_2 is an arbitrary language.

   Solution: NONDEC

Problem 3: Short answer I (8 points)

1. Give a regular expression for the set of all strings in $\{0,1\}^*$ that contain at most one pair of consecutive 1's.

Solution:

First, a regular expression for the set of strings that have no consecutive 1's is:
$R = 0^*.(10.0^*)^*.(\epsilon + 1)$.

The regular expression for the set of strings that contain at most one pair of consecutive 1's is just:

$$R.R = 0^*.(10.0^*)^*.(\epsilon + 1).\ 0^*.(10.0^*)^*.(\epsilon + 1)$$

since a string w has at most one pair of consecutive 1's iff $w = xy$, for some x, y, where x and y have no pair of consecutive ones.

There are, of course, many solutions to this question; whatever your solution is, be careful to check border cases. For example, check if you allow initial 0's, final 0's, allow 1 to occur in the beginning and the end, allow ϵ, and also test your expression against some random strings, some in the language and some outside it.

2. Let M be a DFA. Sketch an algorithm for determining whether $L(M) = \Sigma^*$. Do not generate all strings (up to some bound on length) and feed them one-by-one to the DFA. Your algorithm must manipulate the DFA's state diagram.

Solution:

If M does not accept all of Σ^*, then there must be a word $w \notin L(M)$. On word w, M would reach a unique state which is non-final. Also, if there is some way to reach a non-final state from the initial state, then clearly the DFA does not accept the word that labels the path to the non-final state. Hence it is easy to see that the DFA M does not accept Σ^* iff there is a path from the initial state to a non-final state (which can be the initial state itself).

The algorithm for detecting whether $L(M) = \Sigma^*$ proceeds as follows:

1. Check whether the input is a valid DFA (in particular, make sure it is complete; i.e. from any state, on any input, there is a transition to some other state).
2. Consider the transition graph of the DFA.
3. Do a depth-first search on this graph, searching for a state that is not final.
4. If any non-final state is reached on this search, report that M does not accept Σ^*; otherwise report that M accepts Σ^*.

An alternate solution will be to flip the final states to non-final and non-final states to final (i.e. complementing the DFA), and then checking the emptiness of the resulting automaton by searching for a reachable final state from the initial state.

Problem 4: Short answer II (8 points)

1. Let $\Sigma = \{a, b\}$, let $G = (V, \Sigma, R, S)$ be a CFG, and let $L = L(G)$. Give a grammar G' for the language $L' = \{wxw^R \mid w \in \Sigma^*, x \in L\}$ by modifying G appropriately.

Solution:

The grammar $G' = (V', \Sigma, R', S')$ where

- $V' = V \cup \{S'\}$ where S' is a new variable, not in V;
- R' consists of all rules in R as well as the following rules:
  - One rule $S' \longrightarrow aS'a$, for each $a \in \Sigma$
  - The rule $S' \longrightarrow S$

Intuitively, S' is the new start variable, and for any word $wxw^R \in L'$, S' generates it by first generating the w and w^R parts, and then calling S to generate x.

2. Let $P = \{ \left[\frac{a}{c}\right] \left[\frac{a}{aa}\right], \left[\frac{cba}{b}\right], \}$ be an instance of the Post correspondence problem. Does P have a match? Show a match or explain why no match is possible.

Solution:

Yes, the PCP has a match: $\left[\frac{a}{aa}\right] \left[\frac{a}{c}\right] \left[\frac{cba}{b}\right], \left[\frac{a}{aa}\right]$

The top word and the bottom word are both $aacbaa$.

Problem 5: Pumping Lemma (8 points)

For each of the languages below you can apply the pumping lemma either to prove that the language is non-regular or to prove that the language is not context-free. Assuming that p is the pumping length, your goal is to give a candidate string for w_p that can be pumped appropriately to obtain a correct proof. You ONLY need to give the string and no further justification is necessary. Assume $\Sigma = \{a, b\}$ unless specified explicitly.

(a) $L = \{w \mid w \text{ is a palindrome}\}$. To show L is not regular using the pumping lemma

Solution:

$w_p = a^p bba^p$

(b) $L = \{w \mid w \text{ contains at least twice as many } \texttt{a}\text{'s as } \texttt{b}\text{'s}\}$. To show L is not regular using the pumping lemma.

Solution:

$w_p = b^p a^{2p}$

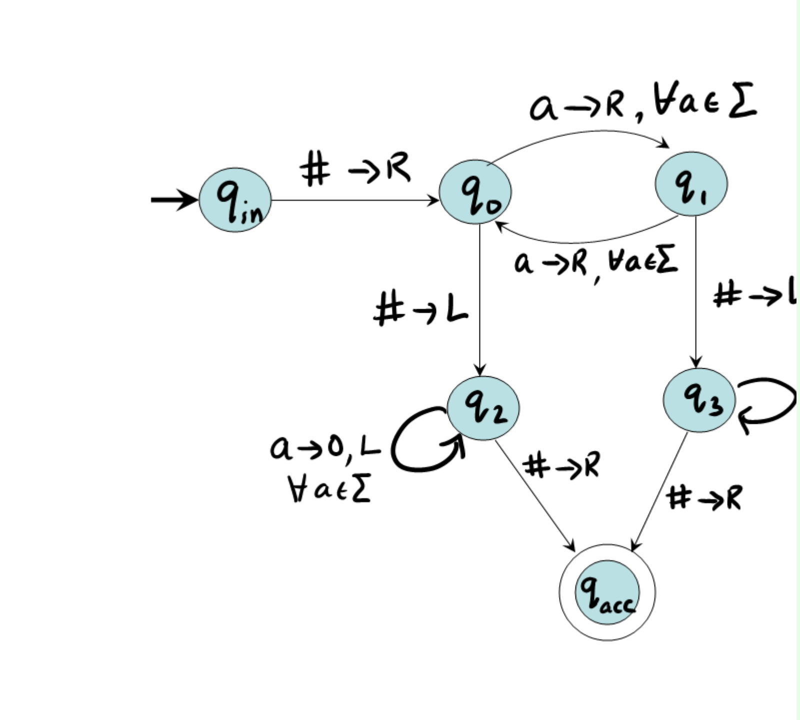
Problem 6: TM design (6 points)

Give the state diagram of a TM M that does the following on input $\#w$ where $w \in \{0,1\}^*$. Let $n = |w|$. If n is even, then M converts $\#w$ to $\#0^n$. If n is odd, then M converts $\#w$ to $\#1^n$. Assume that ϵ is an even length string.

The TM should enter the accept state after the conversion. We don't care where you leave the head at the end of the conversion. The TM should enter the reject state if the input string is not in the right format. However, your state diagram does not need to explicitly show the reject state or the transitions into it.

Solution:

The Turing machine's description is:



In the diagram, the initial state is q_{in}, and the accept state is q_{acc}; the reject state is not shown, and we assume that all transitions from states that are not depicted go to the reject state. We are assuming that the blank tape-symbol is #.

Intuitively, the TM first reads the tape content w, moving right, alternating between states q_0 and q_1 in order to determine whether $|w|$ is even or odd. If $|w|$ is even, it ends in state q_0, moves left rewriting every letter in w with a 0, till it reaches the first symbol on the tape, and then accepts and halts. If $|w|$ is odd, it does the same except that it rewrites letters in w with 1's.

Problem 7: Subset construction (10 points)

Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA that does not contain any epsilon transitions. The *subset construction* can be used to construct a DFA $M = \{Q', \Sigma, \delta', q'_0, F')$ recognizing the same language as N. Fill in the following key details of this construction, using your best mathematical notation:

Solution:

$Q' = \mathcal{P}(Q)$
$q'_0 = \{q_0\}$
$F' = \{R \subseteq Q \mid R \cap F \neq \emptyset\}$

Suppose that P is a state in Q' and a is a character in Σ. Then

$\delta'(P, a) = \{q \in Q \mid \exists p \in P, q \in \delta(p, a)\}$

Suppose N has ϵ-transitions. How would your answer to the previous question change?

$\delta'(P, a) = \{q \in Q \mid \exists p \in P, q \in E(\delta(p, a))\}$
where

$$E(R) = \{s \mid s \text{ can be reached from some state in } R \text{ using zero or more } \epsilon\text{-edges}\}$$

is the epsilon-closure of the set R.

Problem 8: Writing a proof (8 points)

We have seen that $ALL_{CFG} = \{< G >\mid G \text{ is a CFG and } L(G) = \Sigma^*\}$ is undecidable.

1. Show that $\overline{ALL_{CFG}}$ is Turing-recognizable.

Solution:

$\overline{ALL_{CFG}} = \{\langle G\rangle \mid G \text{ is not the encoding of a CFG, or, } G \text{ is a CFG and } L(G) \neq \Sigma^*\}$.

First, recall that membership of a word in the language generated by a grammar is decidable, i.e.. for any grammar G and any word w, we can build a TM that decides whether G generates w.

Also, for any Σ, we can fix some ordering of symbols in Σ, and enumerate all words in Σ^* in the lexicographic ordering. In particular, we can build a TM that can construct the i'th word in Σ^* for any given i.

We can build a TM recognizing $\overline{ALL_{CFG}}$ as follows:

```
Input is $\langle G\rangle$.
Check if $\langle G\rangle$ is a proper encoding of a CFG; if not, halt and accept.
Set $i := 1$;
while (true) do {
    Generate the $i$'th word $w_i$ in $\Sigma^*$, where $\Sigma$ is the set of terminals in $G$.
    Check if $w_i$ is generated by $G$. If it is not, halt and accept.
    Increment $i$;
}
```

The TM above systematically generates all words in Σ^* and checks if there is any word that is not generated by G; if it finds one it accepts $\langle G\rangle$.

Note that if $\langle G\rangle$ is either not a well-formed grammar or $L(G) \neq \Sigma^*$, then the TM will eventually halt and accept. If $L(G) = \Sigma^*$, the TM will never halt, and hence will never accept.

2. Is ALL_{CFG} Turing-recognizable? Explain why or why not.

Solution:

ALL_{CFG} is not Turing-recognizable. We know that ALL_{CFG} is not Turing-decidable (the universality problem for context-free grammars in undecidable), and we showed above that $\overline{ALL_{CFG}}$ is Turing-recognizable. Also, for any language R, if R and $\overline{R}$ are Turing-recognizable, then R is Turing-decidable. Hence, if ALL_{CFG} was Turing-recognizable, then ALL_{CFG} would be Turing-decidable, which we know is not true. Hence ALL_{CFG} is not Turing-recognizable.

Problem 9: PDA modification (8 points)

Let L be a context-free language on the alphabet Σ. Let $M = (Q, \Sigma, \Gamma, \delta, q, F)$ be a PDA recognizing L. Give a PDA M' recognizing the language $L' = \{xy \mid x \in L \text{ and } y \in \Sigma^* \text{ and } |y| \text{ is even}\}$

Note: If you make assumptions about M, then state them clearly and justify.

1. Explain the idea behind the construction.

Solution:

The idea would be to construct the PDA M' which will essentially simulate M, and from any of the final states of M, nondeterministically jump on an ϵ-transition to a new state that checks whether the rest of the input is of even length.

2. Give tuple notation for M'.

Solution:

$M' = (Q', \Sigma, \Gamma, \delta', q, \{p_0\})$ where $Q' = Q \cup \{p_0, p_1\}$ where p_0 and p_1 are two new states (that are not in Q), and the transition function $\delta' : Q' \times \Sigma_\epsilon \times \Gamma_\epsilon \rightarrow \mathcal{P}(Q' \times \Gamma_\epsilon)$ is defined as follows:

- For every $q \in Q$, $a \in \Sigma$, $d \in \Gamma_\epsilon$, $\quad \delta'(q, a, d) = \delta(q, a, d)$
- For every $q \in Q$, $d \in \Gamma$, $\quad \delta'(q, \epsilon, d) = \delta(q, \epsilon, d)$

- For every $q \in Q$,
  $\delta'(q, \epsilon, \epsilon) = \delta(q, \epsilon, \epsilon) \cup \{(p_0, \epsilon)\}$, if $q \in F$, and
  $\delta'(q, \epsilon, \epsilon) = \delta(q, \epsilon, \epsilon)$, if $q \notin F$.
- For every $a \in \Sigma$
  $\delta'(p_0, a, \epsilon) = \{(p_1, \epsilon)\}$, and $\delta'(p_1, a, \epsilon) = \{(p_0, \epsilon)\}$.
- For every $a \in \Sigma_\epsilon$, $d \in \Gamma_\epsilon$, where either $a = \epsilon$ or $d \neq \epsilon$,
  $\delta'(p_0, a, d) = \emptyset$, and $\delta'(p_1, a, d) = \emptyset$.

Problem 10: Decidability (6 points)

Show that

$$EQINT_{DFA} = \left\{ \langle A, B, C \rangle \;\middle|\; \begin{array}{l} A, B, C \text{ are DFAs over the same alphabet } \Sigma \\ \text{and } L(A) = L(B) \cap L(C) \end{array} \right\}$$

is decidable.

This question does not require detail at the level of tuple notation. Rather, keep your proof short by exploiting theorems and constructions we've seen in class.

Solution:

We know that for any two DFAs A and B over an alphabet Σ, we can build a DFA D whose language is the intersection of $L(A)$ and $L(B)$. We also know that we can build a TM that can complement an input DFA, and we know that we can build a TM that checks whether the language of a DFA is empty (this TM can, for example, do a depth-first search from the initial state of the DFA, and explore whether any final state is reachable).

Also, note that $L(A) \subseteq L(B)$ iff $L(A) \cap \overline{L(B)} = \emptyset$.

So, we can build a Turing machine that takes as input $\langle A, B, C \rangle$, and first checks whether these are all DFAs (over the same alphabet Σ). Now, if they are, we must first check if $L(A) \subseteq L(B) \cap L(C)$. We can first build a DFA D that accepts $L(B) \cap L(C)$. Now we need to check if $L(A) \subseteq L(D)$, which is the same as checking if $L(A) \cap \overline{L(D)} = \emptyset$. We can do this by first complementing D to get a DFA E, and then building a DFA F that accepts the intersection of the languages of A and E, and then finally checking if the language of F is empty. We do a similar check to verify whether $L(B) \cap L(C) \subseteq L(A)$.

Hence, the decider for $EQINT_{DFA}$ works as follows.

1. Input is $\langle A, B, C \rangle$.
2. Check if A, B and C are DFAs over the same alphabet Σ. If not, reject.
3. Build the automaton D accepting $L(B) \cap L(C)$.
4. Complement D to get DFA E.
5. Build the DFA F that accepts $L(A) \cap L(E)$.
6. Check if $L(F) = \emptyset$. If it is not, then reject (as $L(A) \not\subseteq L(B) \cap L(C)$).
7. Complement A to obtain the DFA G.
8. Construct DFA H accepting $L(G) \cap L(D)$.
9. Check if $L(H) = \emptyset$. If it is, accept, else reject (as $L(B) \cap L(C) \not\subseteq L(A)$).

Problem 11: Reduction (8 points)

Let $ODD_{TM} = \{\langle M \rangle \mid \text{M is a TM and } L(M) \text{ does not contain any string of odd length}\}$. Show that ODD_{TM} is undecidable using a reduction from A_{TM} (Turing machine membership). You may not use Rice's Theorem.

Solution:

Let $A_{TM} = \{\langle M, w \rangle \mid M \text{ is a TM that accepts } w\}$. We know that A_{TM} is undecidable. Let us reduce A_{TM} to ODD_{TM} to prove that ODD_{TM} is undecidable. In other words, given a decider R for ODD_{TM}, let us show how to build a decider for A_{TM}.

The decider D for A_{TM} works as follows:

```
1. Input is $\langle M, w \rangle$.
2. Construct the code for a TM $N_{M,w}$ that works as follows:
.     a. Input to $N_{M,w}$ is a word $x$
.     b. Simulate $M$ on $w$; accept $x$ if $M$ accepts $w$.
3. Feed $N_{M,w}$ to $R$, the decider of $ODD_{TM}$.
4. Accept if $R$ rejects; reject if $R$ accepts.
```

For any TM M and word w, if M accepts w, then $N_{M,w}$ is a TM that accepts all words, and if M does not accept w then $N_{M,w}$ accepts no word. Hence M accepts w iff $L(N_{M,w})$ contains an odd-length word.

Hence D accepts $\langle M, w \rangle$ iff R rejects $\langle N_{M,w} \rangle$ iff $L(N_{M,w})$ contains an odd-length word iff M accepts w. Hence D decides A_{TM}.

Note that D does not simulate M on w; it simply constructs the code of a TM $N_{M,w}$ (which if run may simulate M on w). So D simply constructs $N_{M,w}$ and runs R on it. Since R is a decider, it halts always, and hence D also always halts.

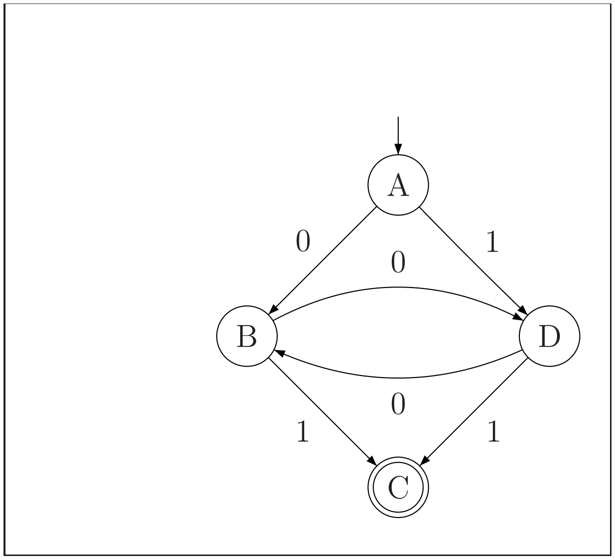
Since A_{TM} reduces to ODD_{TM}, i.e. any decider for ODD_{TM} can be turned into a decider for A_{TM}, we know that if ODD_{TM} is decidable, then A_{TM} is decidable as well. Since we know A_{TM} is undecidable, ODD_{TM} must be undecidable.

49.6 Quiz 1

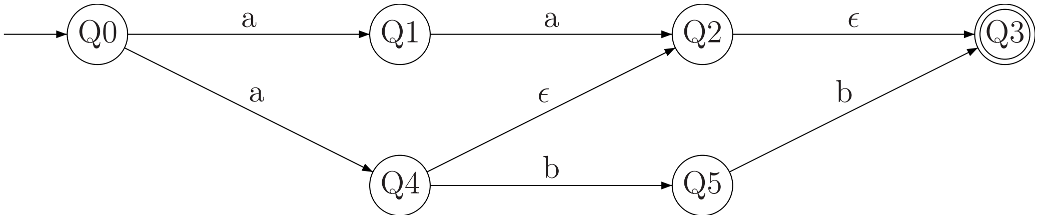
6 February 2008

This quiz has 3 pages containing 7 questions. None requires a long answer. No proofs are required; explanations are only required when explicitly requested. Ensure your answers are legible. You have 20 minutes to finish.

1. (2 points) To formally define a DFA, what five components do you need to specify?
2. (3 points) Suppose that $A = \{aa, bb\}$ and $B = \{1, 2\}$. List the members of $B \times \mathbb{P}(A)$.
3. (2 points) Is the following a valid state diagram for a DFA? Explain your answer.



4. (6 points) Here is the state diagram for an NFA.



Suppose the transition function is named δ. Fill in the following output values for the transition function:

(a) $\delta(Q0, a) =$

(b) $\delta(Q4, a) =$

(c) $\delta(Q4, \epsilon) =$

5. (5 points) Give the state diagram of an NFA which recognizes the language represented by the regular expression $(a+\epsilon)(ba)^*ba$. (It's not necessary to follow any specific mechanical construction.)

6. (3 points) Give a regular expression for the following language

$$L = \{w \mid |w| \text{ is of even length}\}$$

7. (4 points) Let Σ be some alphabet. Suppose that $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ is a DFA recognizing L_1, $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ is a DFA recognizing L_2, and $N_3 = (Q_3, \Sigma, \delta_3, q_3, F_3)$ is a DFA recognizing L_3. We can use the product construction to build a DFA M recognizing $(L_1 \cup L_2) - L_3$. Each state of M is a triple of states (q_1, q_2, q_3), where $q_1 \in Q_1$, $q_2 \in Q_2$, and $q_3 \in Q_3$.

   (a) Precisely describe the set of final states for M.

   (b) Suppose that δ is the transition function for M. Give a formula for δ in terms of δ_1, δ_2, and δ_3. That is, if c is any character in Σ, then

   $\delta((q_1, q_2, q_3), c) =$

49.7 Quiz 2

5 March 2008

This quiz has 3 pages containing 7 questions. None requires a long answer. No proofs are required; explanations are only required when explicitly requested. Ensure your answers are legible. You have 20 minutes to finish.

1. (2 points) Given a context-free grammar G with start symbol S, generating language L, explain how to construct a grammar for L^*.

2. (5 points) Finish the following statement of the pumping lemma.

   If L is a regular language, there is an integer p such that, if w is any string in L of length at least p, then we can divide w into substrings $w = xyz$ such that

   (1) $|x| \geq 1$

   (2)

   (3)

3. (3 points) Suppose that δ is the transition function for a PDA. What types of objects does δ take as input? What type of object does δ produce as its output value?

4. (4 points) Let our alphabet be $\Sigma = \{a, b, c\}$. Give a context-free grammar that generates $L = \{a^n b^j c^n \mid n \geq 0, j \geq 1\}$. Just give the rule(s) for each grammar. Your start symbol should be S and all variables should be uppercase letters.

5. (2 points) Let $\Sigma = \{0, 1\}$. Let $L = \{0^n x 1^n \mid x \in \Sigma^*, n \geq 1\}$. Is L regular? Why or why not?

6. (5 points) Suppose we know that the language $B = \{a^n b^n \mid n \geq 0\}$ is not regular. Let $L = \{a^n b^j c^n \mid j \geq 1, n \geq 1\}$. Prove that L is not regular using closure properties and the fact that B isn't regular.

7. (4 points) Let G be the grammar with start symbol S and the following rules:

   $$S \rightarrow AS \mid B$$
   $$A \rightarrow ab \mid b$$
   $$B \rightarrow c$$

   Give a parse tree and a leftmost derivation for the string abbc.

49.8 Quiz 3

23 April 2008

This quiz has 6 questions. None requires a long answer. No proofs are required; explanations are only required when explicitly requested. Ensure your answers are legible. You have 20 minutes to finish.

1. (6 points) Mark each of the following claims as true or false:

   (a) The language $\{< M, w >| M$ is a TM and M halts on input $w\}$ is TM recognizable.

   (b) The language $\{< G >| G$ is a CFG and $L(G) = \Sigma^*\}$ is TM decidable. (Σ is the alphabet given in the description of G.)

   (c) There are more possible languages than there are different Turing machines.

2. (4 points) Suppose that a Turing machine M contains the transition $\delta(q, 0) = (r, 2, R)$ (q and r are states; 0 and 2 are tape symbols.) If M is now in configuration $021q03$, what will its next configuration be?

3. (3 points) An LBA is a Turing machine with one important restriction. What's the restriction?

4. (4 points) Briefly explain why the following statement is true. (Don't give a proof, just outline the key ideas.)

   If L and $\overline{L}$ are recognizable, then L is decidable.

5. (4 points) Suppose that an extended TM is like a normal TM, except its transitions have the option of staying in place (S) rather than always having to move left or right. Show how to simulate the transition $\delta(s, a) = (r, b, S)$ using transitions of a normal TM. (Either write out the transitions as equations or draw a fragment of a state diagram.)

6. (4 points) Let M be a TM and w be a string. Define a TM M_w (with input alphabet Σ) as follows:

   - Input $= x$
   - If x is a palindrome, then accept.
   - Otherwise, simulate M on w.
   - Accept if M accepts. Reject if M rejects.

   What are the possible values for $L(M_w)$?

Part IV

Homeworks

Chapter 50

Spring 2009

50.1 Homework 1: Problem Set 1

Spring 09

This homework contains four problems (and one extra credit problem). Please follow the homework format guidelines posted on the class web page:

`http://www.cs.uiuc.edu/class/sp09/cs373/`

In particular, submit each problem on a **separate sheet of paper**, put your name on each sheet, and write your discussion section time and day (e.g. Tuesday 10am) in the upper righthand corner. These details may sound picky, but they make the huge pile of homeworks much easier to grade quickly and more importantly, since we return them in the discussion sections, easier for you to get them back.

Note, that this homework should be done on your own. The next homeworks in this class could be done will working in groups. See webpage for details.

1. We are all individuals.
   [**Category**: Proof, **Points**: 10]

   The following is an inductive proof of the following statement:[1]

   "All students enrolled in CS373 in Spring 2009 have hair of the same color"
   (Which I hope you realize is not true.).

   Proof: For any subset S of students enrolled, we will argue that their hair is of the same color. The proof is by induction on the size of S.

   Base case: Base case is when $|S| = 1$. A set containing only one student clearly satisfies the statement.

   Inductive step: Assume the statement is true for all subsets of students S', where $|S'| = k$. We will show that the statement is true for all subsets of students S, where $|S| = k + 1$.

   Let $|S| = k + 1$. Remove one student, say x, from S to get a set $S_{\setminus x}$. By the inductive assumption, all students in $S_{\setminus x}$ have hair of the same color. Now remove another student y from S to get a set $S_{\setminus y}$; again, all students in $S_{\setminus y}$ have hair of the same color. Now, since x's color agrees with the rest of the students in the set $S_{\setminus y}$ and y's color agrees with the rest of the students in $S_{\setminus x}$, x and y must also have hair of the same color!

   Let us illustrate the argument for a particular k, say 5. Assume $S = \{x, y, z, a, b\}$. Then $S \setminus \{x\} = \{y, z, a, b\}$ is a 4-element set, and hence the students in this set have the same hair color. Similarly $S \setminus \{y\} = \{x, z, a, b\}$ all have the same hair color. Hence, since x's hair color is the same as, say z, and y's color also agrees with z's, x and y have the same hair color, and hence the students in $S = \{x, y, z, a, b\}$ all have the same hair color.

[1] We will assume for the sake of simplicity that each student has hair, and this hair is of only one color!

Show why the above induction argument is wrong. In particular, illustrate one set for which the inductive argument fails.

2. Trees and edges.
[**Category**: Proof, **Points**: 10]

The following is an inductive proof of the statement that in every tree $T = (V(T), E(T))$, $|E(T)| = |V(T)| - 1$, i.e a tree with n vertices has $n - 1$ edges.

Proof: The proof is by induction on $|V(T)|$.

Base case: Base case is when $|V(T)| = 1$. But a tree with a single vertex has no edge, so $|E(T)| = 0$. Therefore in this case the formula is true since $0 = 1 - 1$.

Inductive step: Assume that the formula is true for all trees T where $|V(T)| = k$. We will prove that the formula is true for trees with $k + 1$ nodes. A tree T with $k + 1$ nodes can be obtained from a tree T' with k nodes by attaching a new vertex to a leaf of T'. This way we add exactly one vertex and one edge to T', so $|V(T)| = |V(T')| + 1$ and $|E(T)| = |E(T')| + 1$. Since $|V(T')| = k$ by induction hypothesis we have $|E(T')| = |V(T')| - 1$.

Combining the last three relations we have $|E(T)| = |E(T')|+1 = |V(T')|-1+1 = |V(T)|-1-1+1 = |V(T)| - 1$, which means that the formula is true for tree T.

Show that the above is *not* a correct inductive proof! You must argue why it is not correct, and in particular produce a tree which the above argument does not cover.

3. Number of leaves.
[**Category**: Proof, **Points**: 10]

Give a (correct) inductive proof of the following claim:

Let T be a *full binary tree* over n vertices (that is, every node in T other than the leaves has two children). Then T has $(n + 1)/2$ leaves.

4. True/false/whatever.
[**Category**: Notation, **Points**: 20]

Answer each of the following with true, false or meaningless. The notation \ denotes “set-minus” or “set-difference”.

D1) $\varnothing \in \{\varnothing, 1\}$

D2) $\varnothing \subseteq \{\varnothing, 1\}$

D3) $1 \subseteq \{\varnothing, 1\}$

D4) $\{1\} + \{1, 2\} = \{1, 2\}$

D5) $\{1, 2\} \setminus \{1\} = \{2\}$

D6) $\{1, 2\} \setminus \{0\} = \{1, 2\}$

D7) $\{1, 2\} \cap \{3, 4\} = \{\}$

D8) $\{1, 2\} \cap \{3, 4\} = \{\varnothing\}$

D9) $\{1, 2\} \cup \{1, 3\} = \{1, 1, 2, 3\}$

D10) $\{1, 2\} \cup \{1, 3\} = \{1, 2, 3\}$

D11) $\{1, \{1\}, \{\{1\}\}\} = \{1\}$

D12) $\{1, \{1\}, \{\{1\}\}\} = \{1, 1, 1\}$

D13) $\{1\} \in \{1, \{1\}, \{\{1\}\}\}$

D14) $\{1\} \subseteq \{1, \{1\}, \{\{1\}\}\}$

D15) $\{\{1\}\} \in \{1, \{1\}, \{\{1\}\}\}$

D16) $\{A, B\} \times \{C, D\} = \{(A, B), (C, D)\}$.

D17) $\Big(\{A, B\} \times \{C, D\}\Big) \cap \Big(\{C, D\} \times \{A, B\}\Big) = \{\}$.

D18) $|\{A, B, C\} \times \{D, E\}| = 6$.

D19) $\{A, B\} \times \{\} = \{A, B\}$.

D20) $\{A, B\} \setminus \{B, A\} = \{\}$.

5. Getting to 100.
   [**Category**: Proof, **Points**: 10]

   (Extra credit.)

   We are given one copy of every digit in the list $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$. You are asked to form numbers from these digits (you can use each digit only once, and you must use each digit once), such that the sum of these numbers add up to 100.

   For example, a valid set of numbers you can form from the digits is $\{23, 17, 40, 5, 6, 8, 9\}$ but it adds up to 108, not 100.

   Prove that this is impossible; i.e. no matter how you form the numbers, you *can not* find a way for them to add up to exactly 100.

50.2 Homework 2: DFAs

Spring 09

1. DFA building
   [**Category**: Construction, **Points**: 10]

   Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$. Let L be the set of words w in Σ^* such that w has an even number of $\mathtt{b}$'s and an odd number of a's and does not contain the substring $\mathtt{ba}$.

   (a) **[1 Points]** Give a regular expression for the language of all strings over Σ that do not contain $\mathtt{ba}$ as a substring.

   (b) **[1 Points]** Give a regular expression for the language L above (shorter expression is better, naturally).

   (c) **[8 Points]** Construct a deterministic finite automaton for L that has at most 5 states. Make sure that your DFA is complete (has transitions from all actions on all states).

   If you find this hard, you can also give a DFA with more states that accepts L for partial credit.

   Hint: You may want to try to enumerate some elements in L, and see whether you can simplify the description of L.

2. Control.
   [**Category**: Construction, **Points**: 10]

   We want to build the controller for an automatic door, that opens for some time when a sensor senses that a person is approaching the door.

   Let $\Sigma = \{\mathit{tick}, \mathit{approach}, \mathit{open}, \mathit{close}\}$

   Every odd event is a "tick" or an "approach", and every even event is the controller's response to it, "open" or "close". An "open" event directs the door to open if it is closed, and to remain open if it is open. Similarly, a "close" event directs the door to close or remain closed. An "approach" event happens when the the sensor detects that a person is approaching the door, and a "tick" event denotes the passage of one second of time. We want the door to open immediately after detecting an "approach" event, and remain open for exactly 2 ticks after the last "approach" event, at which point the door must close. (Thus, the input has even length and is made out of pair of word. Each pair is a command, followed by current status report.)

Build an automaton that accepts all valid sequences of this controller behavior. Your automaton should be over the alphabet Σ, and, for example, must accept

$$tick.close.approach.open.tick.open.tick.open.tick.close,$$

and accepts

$$approach.open.tick.open.approach.open.tick.open.tick.open.tick.close.$$

But it rejects the word

$$tick.open,$$

and rejects the word

$$tick.close.approach.open.tick.open.tick.close.$$

3. Recursive definitions
[**Category**: Construction, **Points**: 10]

Consider the following recursive definition of a set S.

(a) $(1, 58) \in S$

(b) If $(x, y) \in S$ then $(x + 2, y) \in S$

(c) If $(x, y) \in S$ then $(-x, -y) \in S$

(d) If $(x, y) \in S$ then $(y, x) \in S$

(e) S is the smallest set satisfying the above conditions.

Give a nonrecursive definition of the set S. Explain why it is correct.

4. Set Theory
[**Category**: Proof, **Points**: 10]

For any two arbitrary sets X and Y, we have that $(X \setminus (X \cap Y)) \cap (Y \setminus (Y \cap X))$ is an empty set.

(a) Explain informally why this is true, using words and/or a Venn diagram.

(b) Prove it formally.

5. No such thing.
[**Category**: Proof, **Points**: 10]

(Extra credit.)

Let L_{same} be the language of all strings over $\{0, 1, 2\}$ that have the same number of 0s as the sum of the number of 1s and 2s. Provide a formal (correct) proof that no finite automata (i.e., DFA) can accept L_{same}.

50.3 Homework 3: DFAs II

Spring 09

1. Intersect 'em
[**Category**: Category: Construction, **Points**: 10]

You are given two NFAs $A_1 = (P, \Sigma, \delta_1, p_0, F_1)$ and $A_2 = (Q, \Sigma, \delta_2, q_0, F_2)$.

Construct an NFA that will accept the language $L(A_1) \cap L(A_2)$ with no more than $|P| * |Q|$ states. Also, prove that it indeed accepts the language of the intersection as stated above.

Hint: You may want to think of a product construction. And you may want to look at the proof of the correctness of the product construction.

2. Express yourself
[**Category**: Category: Comprehension, **Points**: 10]

(a) What is the language of each of the following regular expressions?
Note: A clear, crisp and one-level-interpretable English description is acceptable, like "This is the set of all binary strings with at least three zeros and at most a hundred ones", or like $\{0^n(10)^m : n \text{ and } m \text{ are integers}\}$*. A vague, recursive or multi-level-interpretable description is not acceptable, like: "This is the set of binary strings such that they start and end by 1, and the rest of string starts and ends by 0, and the remainder of string is a smaller string of the same form!", or "This is the set of strings like* 010, 00100, 0001000 *and so on!!".*

(1) $(0^* + 1 + 1^*)^*$
(2) $(1 + \epsilon)(00^*1)^*0^*$
(3) $0^*(1 + 000^*)^*0^*$
(4) $(0^*1^*)000(0 + 1)^*$
(5) $1^* + 0^* + (00^*11^*00^*11^*)^* + (11^*00^*11^*00^*)^*$

(b) Write down a regular expression for each case below, that represents the desired language. (Binary strings are strings over the alphabet $\{0, 1\}$.)

(1) All binary strings that their third character from the end is zero.
(2) All binary strings which have 111 as a substring.
(3) All binary strings that have 00 as a substring but do not contain 011 as a substring.
(4) All binary strings such that in every prefix of the string, the number of one's and zero's differ by at most one.
(5) All binary strings such that every pair of consecutive zero's appears before any pair of consecutive one's.

3. Modifying DFAs
[**Category**: Comprehension, **Points**: 5+5]

Suppose that $M = (Q, \Sigma, \delta, q_0, F)$ and $N = (R, \Sigma, \gamma, r_0, G)$ are two DFAs sharing the common alphabet $\Sigma = \{a, b\}$.

(a) Define a new DFA $M' = ((Q \cup \{q_x\}) \times \{0, 1\}, \Sigma \cup \{\#\}, \delta', (q_0, 0), F')$ whose transition function is defined as follows

$$\delta'((q, i), t) = \begin{cases} (\delta(q, t), i) & q \in Q \text{ and } t \in \Sigma \\ (q_0, 1) & q \in F, i = 0, t = \# \\ (q_x, i) & \text{otherwise.} \end{cases}$$

and where $(q_j, i) \in F'$ iff $q_j \in F$ and $i = 1$
Describe the language accepted by M' in terms of the language accepted by M.

(b) Show how to design a DFA N' over $\Sigma \cup \{\#\}$ that accepts the language

$$L' = \left\{ x\#y\#z \;\middle|\; x \in \overline{L(M)}\ , \ y \in L(N) \text{ and } z \in a^* \right\}$$

Define your DFA formally using notation similar to the definition of M' in part (a).

4. Multiple destinations
[**Category**: Proof, **Points**: 7+3]

Let $L = aa^* + bb^*$.

(a) Prove that *any DFA* accepting L must have more than one final state.
(b) Show that L is acceptable by an NFA with only one final state.

5. Equality and more.
[**Category**: Construction, **Points**: 10]

Let $\Sigma = \{\mathtt{0}, \mathtt{1}, \$\}$. For any $n \in \mathbb{N}$, let the language L_n be:

$$L_n = \left\{ w_1 \$ w_2 \;\middle|\; w_1, w_2 \in \{\mathtt{0}, \mathtt{1}\}^*,\ |w_1| = |w_2| = n, \text{ and } w_1 = w_2 \right\}.$$

(a) Exhibit a DFA for L_3.

(b) For any fixed k, specify the DFA accepting L_k.

(c) Let L be the language

$$L = \left\{ w_1 \$ w_2 \;\middle|\; w_1, w_2 \in \{\mathtt{0}, \mathtt{1}\}^*, \text{ and } w_1 = w_2 \right\}.$$

Argue,as precisely as you can (a proof would be best), that L is not regular.

(d) We can express the language L as $\cup_{k=1}^{\infty} L_k$. It is tempting to reason as follows: The language L is the union of regular languages, and hence is regular.

What is the flaw in this argument, and why is it *wrong* in our case?

6. How big?
[**Category**: Proof, **Points**: 10]

(Extra credit.)

Let $\Sigma = \{\mathtt{0}, \mathtt{1}, \$\}$, and consider the language

$$L_n = \left\{ w_1 \$ w_2 \;\middle|\; w_1, w_2 \in \{\mathtt{0}, \mathtt{1}\}^*,\ |w_1| = |w_2| = n, \text{ and } w_1 = w_2 \right\}.$$

(i) Prove that any DFA accepting L_n must have at least $2(2^{n+1} - 1)$ states.

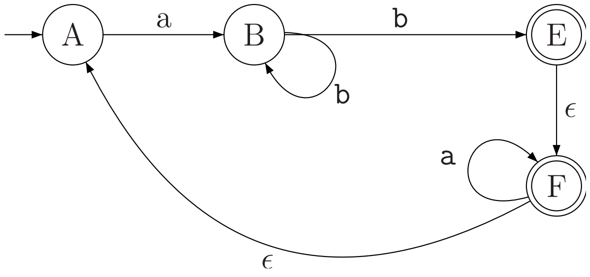
(ii) Prove that any NFA accepting L_n must have at least $2(2^{n+1} - 1)$ states.

50.4 Homework 4: NFAs

Spring 09

1. NFA interpretation.
[**Category**: Notations., **Points**: 10]

Consider the following NFA M.

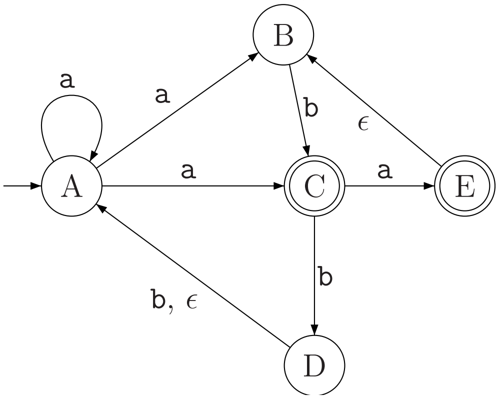


(a) Give a regular expression that represents the language of M. Explain briefly why it is correct.

Note: You needn't go through the process of converting this to a regular expression using GNFAs; you can guess (correctly) the regular expression.

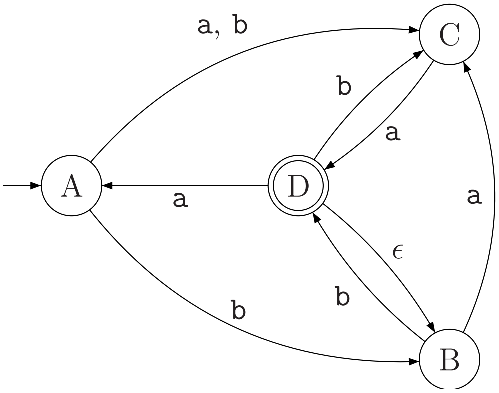
(b) Recall the definition of an NFA accepting a string w (Sipser p. 54). Show formally that M accepts the string $w = \mathtt{abba}$

(c) Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$. Give the formal definition of the following NFA N (in tuple notation). Make sure you describe the transition function completely (for every state and every letter).



2. NFA to DFA.
[**Category**: Construction., **Points**: 10]

Convert the following NFA to an equivalent DFA (with no more than 10 states). You must construct this DFA using the subset-construction as done in class. Draw the DFA diagram, and also write down the DFA in tuple notation (there is no need to include states that are unreachable from the initial state).



3. NFA Construction by Guessing.
[**Category**: Construction, **Points**: 10]

Let $\Sigma = \{\mathtt{0}, \mathtt{1}\}$ be an alphabet. Let

$$L_n = \left\{ \alpha c x c \beta \;\middle|\; c \in \Sigma, \alpha, \beta \in \Sigma^*, x \in \Sigma^n, \text{ and } |\alpha| \text{ is divisible by 3 and } |\beta| \text{ is divisible by 3 } \right\}.$$

(a) **[3 Points]** Draw the NFA for L_2.

(b) **[7 Points]** Formally define the NFA $M_n = (Q_n, \Sigma, \delta_n, q_{init,n}, F_n)$ for L_n for any n. In particular, you must define formally $Q_n, \delta_n, q_{init,n}, F_n$ using simple set notations.

4. NFA to Regex.
[**Category**: Construction, **Points**: 10]

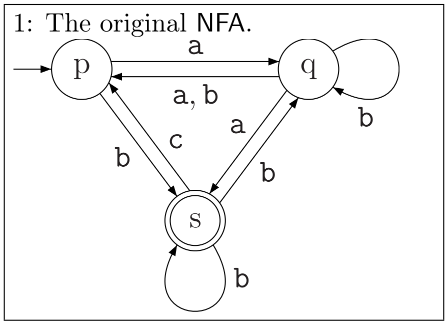
In lecture 8 (Sipser pp. 69–76), we saw a procedure for converting a DFA to a regular expression. This algorithm also works even if the input is an NFA.

For the following NFA, use this procedure to compute an equivalent regular expression. So that everyone does the same thing (and we don't create a grading nightmare), you should do this by:

– Adding new start and end states, and then

– removing states p, q, s **in that order**.

Provide detailed drawing of the GNFA after each step in this process.

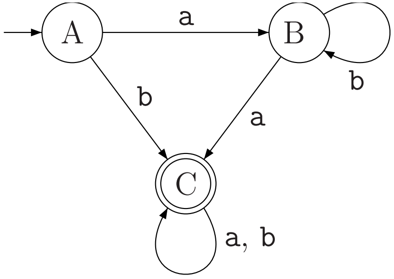


Note, that in this problem you will get interesting self-loops. For example, one can travel to from q to p and then back q. This creates a self-loop at q when p is removed.

5. NFA to Regex by other means.
[**Category**: Proof., **Points**: 10]

(**Extra credit.**)

There is another technique one can use to compute a regular expression from an NFA. As a concrete example, consider the following NFA M seen in class (lecture 8, the examples section).



Consider, for each state in the above automata, the language that this automata would accept if we set this state to be the initial state. The language accepted from state A, denoted by $L(A)$, which we will write as $\mathcal{A}$ to make the notations cleaner. Clearly, a word in $\mathcal{A}$ is either $\mathtt{a}$ followed by a word that is in $\mathcal{B}$ (the language the automaton accepts when B is the initial state), or its $\mathtt{b}$ followed by a word in $\mathcal{C}$ (the language the automaton accepts with C as initial state). We can write this observation as an equation over the three languages:

$$\mathcal{A} = \mathtt{a}\mathcal{B} \,\cup\, \mathtt{b}\mathcal{C}.$$

As a concrete example, in the above automata, C is a final state, which implies that $\epsilon \in \mathcal{C}$. As such, by the above equation, $\mathcal{A}$ must contain the word $\mathtt{b} = \mathtt{b}\epsilon \in \mathtt{b}\mathcal{C}$. Now, since A is the initial state of M, it follows that $L(M) = \mathcal{A}$. This implies that $\mathtt{b} \in L(M)$. (Thats a very indirect way to see that, but it would be useful for us shortly.)

(A) Write down the system of three equations for the languages in the above automata. (Note, that one gets one equation for each state of the NFA.)

(B) Let r, s be two regular expression over some alphabet Σ, and consider an equation of the form

$$\mathcal{D} = r\mathcal{D} \cup s.$$

let $\mathcal{D}$ be the minimal language that satisfies this equation. Give a regular expression for the language $\mathcal{D}$. Prove your answer.

(C) For a character $\mathtt{a} \in \Sigma$, what is the smallest language $\mathcal{E}$ satisfying the equation $\mathcal{E} = \mathtt{a}\mathcal{E}$? Prove your answer.

(D) The above suggests a way to get the regular expression for a language. Take the system of equations from (A), and eliminate from it (by substitution) the variable $\mathcal{B}$. Then, eliminate from it the variable $\mathcal{C}$. After further manipulation, you remain with an equation of the form

$$\mathcal{A} = \textit{something},$$

where "something" does not contain any of our variables (i.e., $\mathcal{B}$, and $\mathcal{C}$). Now, convert the right side into a regular expression describing all the words in $\mathcal{A}$.
Carry out this procedure in detail for the above NFA M (specifying all the details in your solution). What is the regular expression of the language of M that results from your solution?

50.5 Homework 5: On non-regularity.

Spring 09

(Q1) Irregular.
[**Category**: Proof., **Points**: 10]

Let $\Sigma = \{\mathtt{1}, \#\}$ and let

$$L = \left\{ w \,\middle|\, w = x_1\#x_2\#...\#x_k \text{ for } k \geq 0, \text{ where } x_i \in \mathtt{1}^* \text{ and } x_i \neq x_j \text{ for } i \neq j \right\}.$$

Provide a direct proof that L is not a regular language.

(Q2) Inverse homomorphism.
[**Category**: Proof., **Points**: 10]

Let $h : \Sigma^* \to \Gamma^*$ be a homomorphism, and L be a regular language over Γ^*. Show that the inverse homomorphism language $h^{-1}(L)$ is regular. Here

$$h^{-1}(L) = \left\{ w \,\middle|\, h(w) \in L \right\}.$$

For example, for the language $L = \left\{ (\mathtt{aaa})^i \,\middle|\, i \geq 0 \right\}$, and the homomorphism $h(\mathtt{0}) = \mathtt{aa}$ and $h(\mathtt{1}) = \mathtt{aa}$, the inverse homomorphism language is

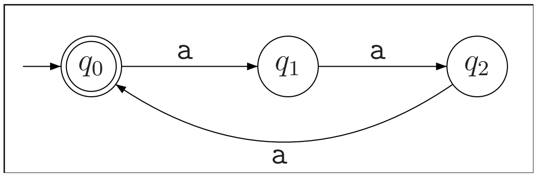
$$L' = \left\{ (\mathtt{0} + \mathtt{1})^i \,\middle|\, i \geq 0 \text{ is a multiply of } 3 \right\},$$

since for any $w \in L'$, we have that $h(w) \in L$. Note, that the inverse homomorphism is not a unique mapping. A word $w \in L$ might have several inverse words $w_1, \ldots, w_j \in L'$, such that $h(w_1) = h(w_2) = \cdots = h(w_j) = w$. For example, for the word $\mathtt{a}^6 \in L$, we have $h^{-1}(\mathtt{a}^6) = \{\mathtt{000}, \mathtt{001}, \mathtt{010}, \mathtt{011}, \mathtt{100}, \mathtt{101}, \mathtt{110}, \mathtt{111}\}$, since, for example, $h(\mathtt{000}) = h(\mathtt{101}) = \mathtt{a}^6$.

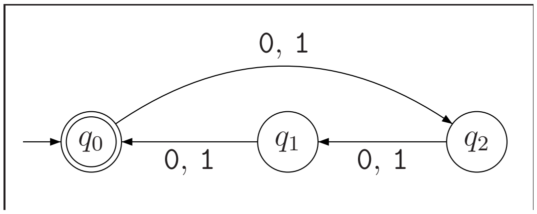
Similarly, it might be that a word $w \in L$ has no inverse in $h^{-1}(L)$. For example, $\mathtt{aaa} \in L$, but has no inverse in L', since $h(w)$ is always an even length word.

To prove that if L is regular then $h^{-1}(L)$ is regular, assume you are given a DFA $\mathtt{D}$ such that $L = L(\mathtt{D})$. Show how to modify this DFA $\mathtt{D} = (Q, \Gamma, \delta, q_0, F)$ into a DFA $\mathtt{C}$ for $h^{-1}(L)$. Describe formally the construction, and prove *formally* that $h^{-1}(L) = L(\mathtt{C})$. (Hint: If $w = w_1 w_2 \ldots w_k \in h^{-1}(L)$ then $h(w) = h(w_1)h(w_2)\ldots h(w_k) \in L$.)

For example, for the languages show above, we have the following two DFAs.



DFA for $L == \left\{ (\mathtt{aaa})^i \,\middle|\, i \geq 0 \right\}$



DFA for $L' = h^{-1}(L)$

This implies that regular languages are closed under inverse homomorphism.

(Q3) Irregularity via closure properties.
[**Category**: Proof., **Points**: 10]

Use (*only*) closure properties to show that the following languages are not regular, using a proof by contradiction. You can use languages that you saw in class that are not regular as a starting (or ending) point to your arguments.

(a) $L_1 = \left\{ \mathtt{a}^n \mathtt{b}^m \mathtt{c}^n \;\middle|\; n, m \geq 0 \right\}$

(b) $L_2 = \left\{ \mathtt{a}^n \mathtt{b}^n \mathtt{c}^n \;\middle|\; n \geq 0 \right\}$.

(c) $L_3 = \left\{ \mathtt{a}^n \mathtt{b}^{3n} \;\middle|\; n \geq 0 \right\}$.
(Hint: Use the result from question 2.)

(d) $L_4 = \left\{ \mathtt{a}^n \mathtt{b}^n \mathtt{a}^n \;\middle|\; n \geq 0 \right\}$.
(Hint: Use the result from question 2 and (b).)

(e) $L_5 = \left\{ w \in (\mathtt{a} + \mathtt{b})^* \;\middle|\; \#_{\mathtt{a}}(w) = \#_{\mathtt{b}}(w) \right\}$, where $\#_c(w)$ is the number of times the character c appears in w.

(f) $L_6 = \left\{ w \in (\mathtt{a} + \mathtt{b})^* \;\middle|\; \#_{\mathtt{a}}(w) \neq \#_{\mathtt{b}}(w) \right\}$.

(Q4) Palindromes are irregular.
[**Category**: Proof., **Points**: 10]

A ***palindrome*** is a string that if you reverse it, remains the same. Thus `tattarrattat`[2] is a palindrome. Let us consider the empty string ϵ to be a palindrome.

(a) Give a direct proof (without using the pumping lemma) that L_{pal}, the language of all palindromes over the alphabet $\{\mathtt{a}, \mathtt{b}\}$ is not regular. Your proof should show that any DFA for this language has an infinite number of states.

(b) (Hard?) Prove using only closure properties that L_{pal} is not a regular languages.

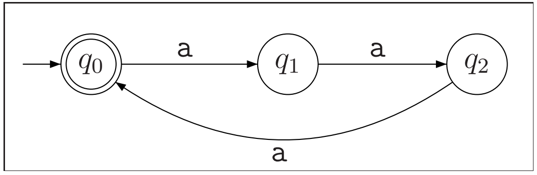
(Q5) CFGs are as strong as regular languages.
[**Category**: Proof., **Points**: 10]

Let L be a regular language recognized by a DFA $\mathtt{D} = (Q, \Sigma, \delta, q_0, F)$. We claim that we can construct a CFG for L as follows. We introduce a variable X_i for every state $q_i \in Q$. Given a transition $q_i \xrightarrow{\mathtt{c}} q_j$ in $\mathtt{D}$, we introduce the rule

$$\mathsf{X}_i \rightarrow c\mathsf{X}_j$$

in the grammar. We also introduce the rule $\mathsf{X}_i \rightarrow \epsilon$, if $q_i \in F$. Finally, we set X_0 to be the starting variable for the resulting grammar. Let $G = G(\mathtt{D})$ denote this grammar.

For example, the grammar for the following DFA,



is the following grammar

[2] Which is the longest palindromic word in the Oxford English Dictionary is tattarrattat, coined by James Joyce in Ulysses for a knock on the door.

$$\text{(G)} \qquad \boxed{\begin{array}{l} \Rightarrow \; \mathsf{X}_0 \rightarrow \mathsf{aX}_1 \mid \epsilon \\ \quad\;\; \mathsf{X}_1 \rightarrow \mathsf{aX}_2 \\ \quad\;\; \mathsf{X}_2 \rightarrow \mathsf{aX}_0 \end{array}}$$

Prove that in general this construction works. That is, prove formally that for any DFA D, we have that the language of $L_G = L(G(\mathtt{D}))$ is equal to $L(\mathtt{D})$.

(Hint: Do not use induction in your proof. Instead, argue directly about accepting traces and derivations of words.)

(Q6) What * has to do with it, anyway?
[**Category**: Proof., **Points**: 10]

(Extra credit – really hard.)

Let $L \subseteq \left\{ \mathtt{0}^i \,\middle|\, i \geq 0 \right\}$ be an arbitrary language. Prove, that the language L^* is regular.

(Note, that this is quite surprising if L is not regular.)

(Hint: Consider first the case when L contains two words. Then prove it for any finite L. Finally, prove it for the general case. Credit would be given only for a solution for the general case, naturally.)

50.6 Homework 6: Context-free grammars.

Spring 09

(Q1) What is in, what is out?
[**Category**: Understanding., **Points**: 10]

For each of the following grammars answer the following:

(a) Is abcd in the language of the grammar? If so give an accompanying derivation and parse tree.

(b) Is acaada in the language of the grammar, if so give an accompanying derivation and parse tree.

(c) What is the language generated by the grammar and explain your answer.

(Note: assume $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$ and the start symbol is S for both grammars.)

$$\mathcal{G}_1: \begin{array}{l} \mathsf{S} \rightarrow \;\mathsf{aSd} \mid \mathsf{A} \mid \mathsf{C} \\ \mathsf{A} \rightarrow \mathsf{aAc} \;\mid\; \mathsf{B} \\ \mathsf{B} \rightarrow \mathsf{bBc} \;\mid\; \epsilon \\ \mathsf{C} \rightarrow \mathsf{bCd} \mid \; \mathsf{B} \end{array} \qquad \Bigg\| \qquad \mathcal{G}_2: \begin{array}{l} \mathsf{S} \rightarrow \mathsf{B} \mid \; \mathsf{AA} \\ \mathsf{A} \rightarrow \mathsf{cA} \mid \; \mathsf{dB} \\ \mathsf{B} \rightarrow \mathsf{aSa} \mid \epsilon. \end{array}$$

(Q2) Building grammars.
[**Category**: Construction., **Points**: 10]

Show that the following languages are context-free by giving a context-free grammar for each of them.

(a) $L = \left\{ \mathtt{a}^i \mathtt{b}^j \,\middle|\, 2i \leq j \leq 3i, \quad i, j \in N \right\}$.

(Hint: Build a grammar for the case that $j = 2i$ and the case $j = 3i$, and think how to fuse these two grammars together to a single grammar.)

(b) $L' = \left\{ w \in \{\mathtt{0}, \mathtt{1}\}^* \,\middle|\, \begin{array}{l} \text{w contains equal number of occurrences of} \\ \text{substring } \mathtt{01} \text{ and } \mathtt{10} \end{array} \right\}$.

Thus 101 contains a single 01 and a single 10 and as such belongs to the language, while 1010 does not as it contains two 10's and one 01.

(Q3) Prove this.
[**Category**: Proof, **Points**: 10]

Consider the following proof.

Lemma 50.6.1 *If L_1 and L_2 are both context-free languages, then $L_1 \cup L_2$ is a context-free language.*

Proof: Let $\mathcal{G}_1 = (\mathcal{V}_1, \Sigma, \mathcal{R}_1, \mathsf{S}_1)$ and $\mathcal{G}_2 = (\mathcal{V}_2, \Sigma, \mathcal{R}_2, \mathsf{S}_2)$ be context free grammars for L_1 and L_2, respectively, where $\mathcal{V}_1 \cap \mathcal{V}_2 = \emptyset$. Create a new grammar

$$\mathcal{G} = (\mathcal{V}_1 \cup \mathcal{V}_2, \Sigma, \mathcal{R}, \mathsf{S}),$$

where $\mathsf{S} \notin \mathcal{V}_1 \cup \mathcal{V}_2$ and $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2 \cup \left\{ \mathsf{S} \to \mathsf{S}_1,\ \mathsf{S} \to \mathsf{S}_2 \right\}$.

We next prove that $L(\mathcal{G}) = L_1 \cup L_2$.

$\mathbf{L}(\mathcal{G}) \subseteq \mathbf{L_1} \cup \mathbf{L_2}$:

Consider any $w \in L(\mathcal{G})$, and any derivation of w by $\mathcal{G}$. It must be of the following form:

$$\mathsf{S} \to \mathsf{S}_i \to X_1 X_2 \to \ldots \to w,$$

where i is either 1 or 2. Assume, without loss of generality, that $i = 1$, and observe that if we remove the first step, this derivation becomes

$$\mathsf{S}_1 \to X_1 X_2 \to \ldots \to w.$$

Namely, $\mathsf{S}_1 \overset{*}{\Longrightarrow} w$ using grammar rules only from $\mathcal{R}_1$. We conclude that $w \in L(\mathcal{G}_1) = L_1$, as S_1 is the start symbol of $\mathcal{G}_1$.

The case $i = 2$ is handled in a similar fashion.

Thus, we conclude that $w \in L_1 \cup L_2$, implying that $L(\mathcal{G}) \subseteq L_1 \cup L_2$.

$\mathbf{L_1} \cup \mathbf{L_2} \subseteq \mathbf{L}(\mathcal{G})$:

Consider any word $w \in L_1 \cup L_2$, and assume without limiting generality, that $w \in L_1$. As such, we have that $\mathsf{S}_1 \underset{\mathcal{G}_1}{\overset{*}{\Longrightarrow}} w$. But $\mathsf{S} \to \mathsf{S}_1$ is a rule in $\mathcal{G}$, and as such we have that

$$\mathsf{S} \to \mathsf{S}_1 \underset{\mathcal{G}_1}{\overset{*}{\Longrightarrow}} w.$$

Namely, $\mathsf{S} \underset{\mathcal{G}}{\overset{*}{\Longrightarrow}} w$, since all the rules of $\mathcal{G}_1$ are in $\mathcal{G}$. We conclude that $w \in L(\mathcal{G})$.

Putting the above together, implies the lemma. ■

Provide a detailed formal proof to the following claim, similar in spirit and structure to the above proof.

Lemma 50.6.2 *If $L_!$ and L_2 are both context-free languages, then $L_1 L_2$ is a context-free language.*

(Q4) Edit with some mistakes.
[**Category**: Construction., **Points**: 10]

The ***edit distance*** between two strings w and w', is the minimal number of edit operations one has to do to modify w into w'. We denote this distance between two strings x and y by $\mathsf{EditDist}(x, y)$. We allow the following edit operations: (i) insert a character, (ii) delete a character, and (iii) replace a character by a different character.

For example, the edit distance between `shalom` and `halo` is 2. The edit distance between `har-peled` and `sharp␣eyed` is 4:

$$\texttt{har-peled} \implies \texttt{shar-peled} \implies \texttt{sharpeled} \implies \texttt{sharp␣eled} \implies \texttt{sharp␣eyed}.$$

For the sake of simplicity, assume that $\Sigma = \{\mathtt{a}, \mathtt{b}, \$\}$. For a parameter k, describe a CFG for the language

$$L_k = \left\{ x\$y^R \;\middle|\; x, y \in \{\mathtt{a}, \mathtt{b}\}^*, \mathsf{EditDist}(x, y) \leq k \right\}.$$

For example, since $\mathsf{EditDist}(\mathtt{aba}, \mathtt{bab}) = 2$, we have that $\mathtt{aba\$bab} \in L_2$, but $\mathtt{aba\$bab} \notin L_1$. Similarly, $\mathsf{EditDist}(\mathtt{aaaa}, \mathtt{abb}) = 3$, and as such $\mathtt{aaaa\$bba} \in L_3$, but $\mathtt{aaaa\$bba} \notin L_2$.

(Hint: What is the language L_0? Try to give a grammar to L_1 before solving the general case.)

Provide a short argument why your CFG works.

(Q5) Speedup theorem for CFGs.
[**Category**: Proof., **Points**: 10]

Assume you are given a CFG $\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{R}, \mathsf{S})$, such that any word $w \in L(\mathcal{G})$ has a derivation of w with $f(n)$ steps, where $n = |w|$. Here $f(n)$ is some function.

Prove the following claim.

Claim 50.6.3 *There exists a grammar $\mathcal{G}'$ such that $L(\mathcal{G}) = L(\mathcal{G}')$, and furthermore, any derivation of a word $w \in L(\mathcal{G})$ of length n requires at most $\lceil f(n)/2 \rceil$ derivation steps.*

For example, consider the grammar $\mathsf{S} \rightarrow \mathtt{a}\mathsf{S}\mathtt{b} \mid \epsilon$.

A word $\mathtt{a}^n\mathtt{b}^n$ can be derived using $n+1$ steps. Its speeded up version is

$$\mathsf{S}' \rightarrow \mathtt{aa}\mathsf{S}'\mathtt{bb} \mid \mathtt{ab} \mid \epsilon.$$

Now, $\mathtt{a}^n\mathtt{b}^n$ has derivation of length $\lceil (n+1)/2 \rceil$ by the grammar of S'.

(Hint: Consider a parse tree of height h in the grammar, and think how to modify the grammar, so that you get a parse tree for the same word with height only $\lceil h/2 \rceil$.)

50.7 Homework 7: Context-free grammars II

Spring 09

(Q1) Building Recursive Automata.
[**Category**: Construction, **Points**: 10]

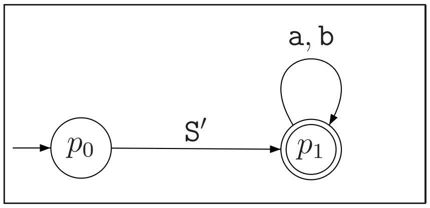
For each of the following languages construct a recursive automaton for it, and briefly describe why it works. Also, for each of these languages, pick a word of length at least 6 in the language and show the run of your automaton on it, including stack contents at each point of the run:

(a) $L_1 = \left\{ \mathtt{a}^i\mathtt{b}^j \;\middle|\; i \geq 0, 3i \geq j \geq 2i \right\}$ over the alphabet $\{\mathtt{a}, \mathtt{b}\}$.

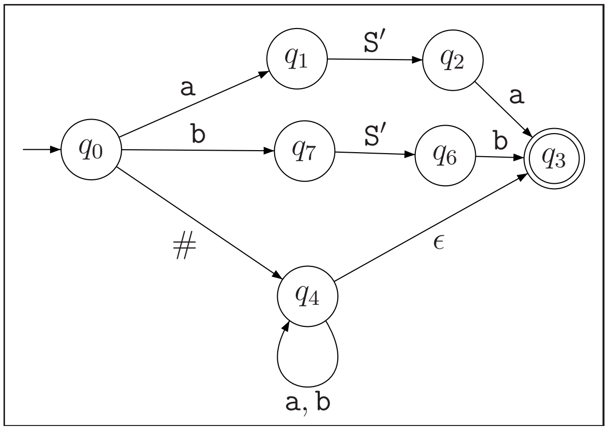
(b) $L_2 = \left\{ w_1\$w_2\$\mathtt{0}^i\$\mathtt{1}^j \;\middle|\; w \in \{\mathtt{a}, \mathtt{b}\}^*, i = |w_2| \text{ and } j = |w_1| \right\}$
Here L_2 is defined over the alphabet $\{\mathtt{a}, \mathtt{b}, \$, \mathtt{0}, \mathtt{1}\}$.

(Q2) Understanding Recursive Automata.
[**Category**: Understanding, **Points**: 10]

For the following recursive automaton with initial module S, give the language of the automata precisely.



S :



S′ :

(Q3) CYK Parsing.
[**Category**: Understanding, **Points**: 10]

For the following CNF grammar (with start symbol S) and the following string:

book the flight through champaign

(assume spaces differentiate the various terminals and nonterminals [not single characters]), generate a valid parse tree using the CYK parsing algorithm. Turn in the filled out chart (matrix) as well.

S → NP VP | X2 PP | VERB NP | VERB PP | VP PP | *book*
NP → *i* | *she* | *me* | *champaign* | DET NOMINAL
DET → *the* | *a*
NOMINAL → *book* | *flight* | *meal* | NOMINAL NOUN | NOMINAL PP
NOUN → *book* | *flight* | *champaign*
VP → *book* | *include* | *prefer* | VERB NP | X2 PP | VERB PP | VP PP
VERB → *book* | *flight*
X2 → VERB PP
PP → PREPOSITION NP
PREPOSITION → *through*

In the above, the terminals are
$\Sigma = \{book, i, she, me, champaign, the, a, flight, meal, include, prefer, through\}$.

(Q4) Closure Properties of CFLs
[**Category**: Proof, **Points**: 10]

For each of the following languages, prove that they are not context free using closure properties discussed in class to generate a known non-context free language such as $\{a^n b^n c^n \mid n \geq 0\}$. You can assume that CFLs are closed under homomorphisms and inverse homomorphisms as well.

(a) $L_1 = \left\{ w \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}^* \;\middle|\; \#_a(w) = \#_b(w) = \#_c(w) = \#_d(w) \right\}$, where $\#_x(w)$ denotes the number of appearances of the character x's in w, for $x \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$.

(b) $L_2 = \left\{ \mathtt{0}^i \# \mathtt{0}^{2i} \# \mathtt{0}^{3i} \;\middle|\; i \geq 0 \right\}$.

(Q5) Shuffle
[**Category**: Proof, **Points**: 10]

For a given language, L, let

$$\text{Shuffle}(L) = \left\{ w \;\middle|\; y \in L,\ |y| = |w| \text{ and, } w \text{ is a permutation of letters in } y \right\}.$$

For instance if $L = \{\mathtt{ab}, \mathtt{ada}\}$, then $\text{Shuffle}(L) = \{\mathtt{ab}, \mathtt{ba}, \mathtt{aad}, \mathtt{ada}, \mathtt{daa}\}$.

Prove that if L is a regular language, then Shuffle(L) is not necessarily a CFL. In other words, prove that the statement “For every regular language L, the language Shuffle(L) is a CFL.” is false.

50.8 Homework 8: Recursive Automatas

Spring 09

(Q1) Building Recursive Automata.
[**Category**: Construction, **Points**: 10]

The language of Boolean expressions is defined as the language of the grammar $\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{R}, \mathsf{B})$ where $\mathcal{V} = \{\mathsf{B}\}$, $\Sigma = \{\mathtt{0}, \mathtt{1}, (,), \vee, \neg\}$ and B has the following rules

$$\mathsf{B} \rightarrow \mathsf{0} \mid \mathsf{1} \mid (\mathsf{B} \vee \mathsf{B}) \mid (\neg B).$$

A Boolean expression evaluates to 0 or 1 in the natural sense: 0 is false, 1 is true, $\vee$ represents the boolean disjunction (***or***) and $\neg$ represents the boolean negation (***not***). For example, the expression $(\neg(0\ \vee\ 1))$ evaluates to 0, and the expression $\Big(\neg((\neg 1)\ \vee\ (\neg 1))\Big)$ evaluates to 1.

(a) Let L be the following language over the alphabet $\Sigma' = \Sigma \cup \{=\}$.

$$L = \left\{ \langle exp \rangle = b \;\middle|\; \langle exp \rangle \text{ is a boolean expression that evaluates to } b \right\}.$$

(Eg. The word “$(\neg(0\ \vee\ 1)){=}0$” is in L but the word $(\neg(0\ \vee\ 1)){=}1$ is not in L.)
Construct a recursive automaton for L and briefly describe why it works.

(b) For the string $w = (0 \vee 1) = 1$ show the run of your automaton on it, including stack contents at each point of the run (i.e., list the sequence of configurations in the accept trace for w for your RA). Use the formal definition given in the lecture notes.

(Q2) Recursive Automata with Finite Memory.
[**Category**: Proof, **Points**: 10]

You are working on a computer, which has a limited stack size of (say) 5. You know this means that you can have a call depth of at most 5 recursive calls.

(a) Argue that the language accepted by any RA on this machine is regular.
More formally, given a RA

$$\mathtt{D} = \left(\mathcal{M}, \textbf{main}, \left\{ (Q_m, \Sigma \cup \mathcal{M}, \delta_m, q_0^m, F_m) \;\middle|\; m \in \mathcal{M} \right\}\right),$$

describe an NFA $\mathtt{C} = (Q', \Sigma, \delta', q_0, F')$ that will accept the same language.

(b) What is Q'?

(c) What is q_0? What is F'?

(d) What is δ'?

(Hint: Think about using configurations of the RA in your construction. See the class notes for the formal definition of what is configuration. It might also be useful for you the review the concept of acceptance for a RA.)

50.9 Homework 9: Turing Machines

Spring 09

(Q1) Building Turing Machines.
[**Category**: Construction, **Points**: 50]

Give the state diagram for a Turing Machine for the following language.

$$L = \left\{ \$\mathtt{a}^{2n}\mathtt{b}^{n}\mathtt{c}^{3n} \;\middle|\; n \geq 0 \right\}.$$

For example, the input may look like $abc.

You do not need to draw transitions that lead to the (implicit) reject state. Hence, any transition that is not present in your diagram will be assumed to lead to the reject state. Indicate which symbol (e.g. ␣) you are using for the special blank symbol that fills the end of the tape.

(Q2) Building Turing Machines II.
[**Category**: Construction, **Points**: 50]

Give the state diagram for a Turing Machine for the following language.

$$L = \left\{ \$\mathtt{a}^{2^n}\#\mathtt{b}^{n} \;\middle|\; n \geq 0 \right\}.$$

For example, the input may look like $aaaa#bb or $aaaaaaaa#bbb

You do not need to draw transitions that lead to the (implicit) reject state. Hence, any transition that is not present in your diagram will be assumed to lead to the reject state. Indicate which symbol (e.g. ␣) you are using for the special blank symbol that fills the end of the tape.

50.10 Homework 10: Turing Machines II

Spring 09

(Q1) High Level TM Design.
[**Category**: Construction of machines, **Points**: 10]

A perfect number is a positive integer that is the sum of its proper positive divisors, that is, the sum of the positive divisors excluding the number itself. For example, 6 is the first perfect number, because $1, 2, 3$ are its proper positive divisors, and $6 = 1 + 2 + 3$.

Design a Turing Machine that for a given input tape with n cells containing 0's, marks the positions which are perfect numbers.

Specifically, cells at the non-perfect-numbered positions are left containing 0, cells at perfect-numbered locations are left containing X. For example, consider the input 0000000000. This represents the first 10 numbers. So the Turing machine should halt with 00000X0000 on the tape.

(Q2) TM Encoding.
[**Category**: Comprehension, **Points**: 10]

In this problem we demonstrate a possible encoding of a TM using the alphabet $\{0, 1, ;\}$ where ; is used as a seperator. We encode $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\mathrm{acc}}, q_{\mathrm{rej}})$ as a string representing $|Q|$, $|\Sigma|$, $|\Gamma|$, q_0, q_{acc}, q_{rej} and then the δ. Each of the first five quantities is represented using a numbering system where n is represented by a 1 followed by n $0's$.

Thus is $|Q| = n$, this means we have n states numbered 1 to n.

If $|\Sigma| = n$, this means we have n symbols in the alphabet. We adopt the convention that these symbols are 0 to $n - 1$ where each can be represented using the numbering system mentioned above.

We use a similar scheme for Γ with the restriction that the blank symbol is assigned the largest number.

q_0, q_{acc} and q_{rej} are the next 3 numbers represented.

D is either 10 to mean move left or 100 to mean move right.

The remaining string represents δ as follows. Each transition $(q, \mathtt{a}) \rightarrow (q', \mathtt{b}, D)$ is represented as the concatenation of the represntation of each of the 5 quantities and the reresentation of δ is simply the concatenation of the representation of each transitions. We use the convention that transitions not mentioned go to the reject state in the encoded machine.

Here is the representation of a mystery Turing machine M, using this encoding.

1000000;
1000;
10000;
10;
100000;
1000000;
10; 100; 100; 100; 100;
100; 1; 100; 1; 100;
100; 10; 100; 10; 100;
100; 1000; 1000; 1000; 10;
1000; 1; 100000; 10; 100;
1000; 10; 10000; 1; 10;
10000; 10; 10000; 1; 10;
10000; 100; 100000; 10; 10;
10000; 1; 100000; 10; 10

(a) Draw a state diagram for this TM (omitting the reject state).

(b) What is the language of this TM? Give a brief justification.

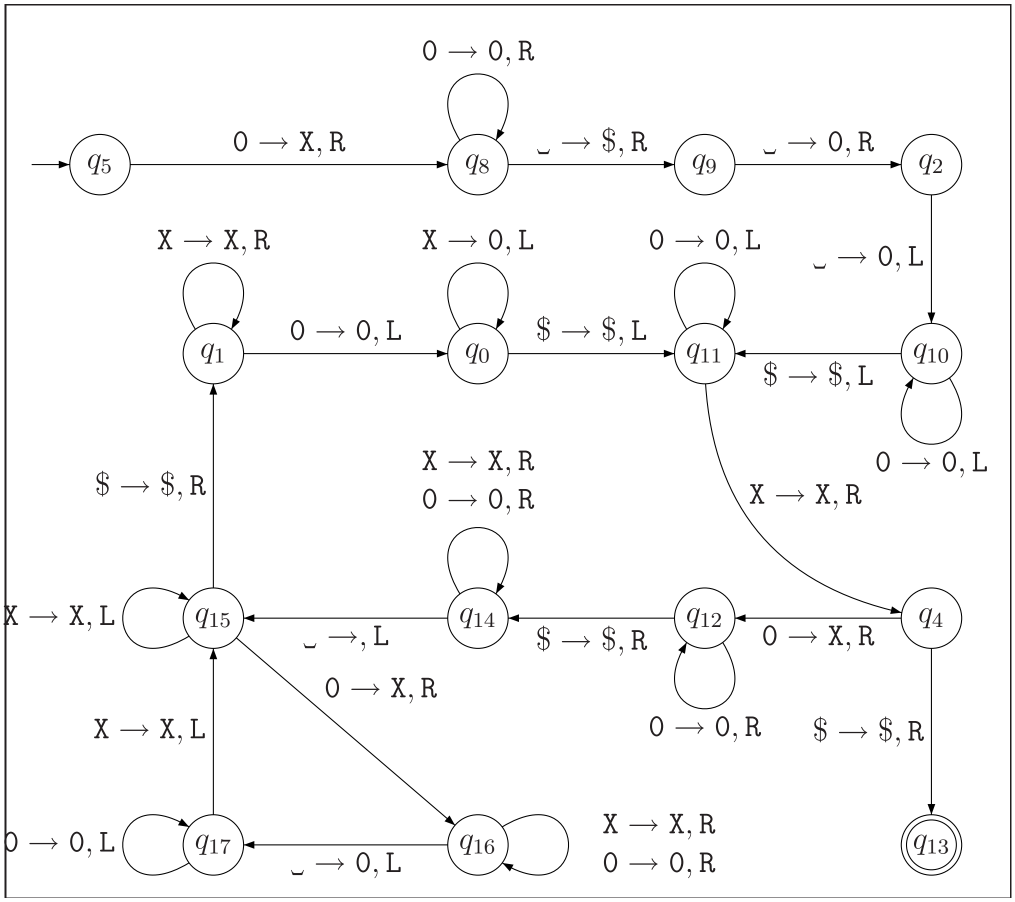
(Q3) TM with infinite number of tapes.
[**Category**: Construction, **Points**: 20]

An ITM is a special TM with one head and infinite number of tapes (one sided tapes). The tapes are numbered $0, 1, 2, 3, \cdots$. The cells on each tape are also numbered from left to right with $0, 1, 2, 3, \cdots$. When the machine starts, the head is on cell number 0 of tape number 0. If the head is on cell number i of tape number j, it can overwrite that cell and move either to cell $i + 1$ or $i - 1$ (if $i \geq 1$) of tape j or move to cell i of tape $j + 1$ or tape $j - 1$ (if $j \geq 1$).

Prove that ITM's and normal TM's are equivalent.
(you should prove that every normal TM can be simulated using an ITM -this should be the easy part- and every ITM can be simulated using a normal TM).

(Q4) What does it do?
[**Category**: Comprehension, **Points**: 10]

What does the following TM do?



50.11 Homework 11: Enumerators and Diagonalization

Spring 09

(Q1) Enumerators
[**Category**: Construction, **Points**: 10]

(a) Design an enumerator that will list all positive integral solutions to a polynomial inequality in three variables, which will be given as input on the input tape of the Turing machine.

I.e., you need to give all positive integral solutions to a given inequality $P(x, y, z) \geq c$. An example of such an inequality is $2x^2y^2 + xy + z \geq 5$.

You may use pseudo code to describe your solution.

(b) Modify the enumerator above to give all integral (positive or negative) solutions.

(Q2) Decision and Enumeration.
[**Category**: Construction, **Points**: 12]

(a) Show that L_1 is a decidable language:

$$L_1 = \{\langle M, x, i\rangle : \text{Machine } M \text{ will accept string } x \text{ in no more than } i \text{ steps}\}$$

(b) Show that L_2 is a decidable language:

$$L_2 = \{\langle M, x, i\rangle : M(x) = \text{“yes” and head of } M \text{ will use only the first } i \text{ cells of its tape}\}$$

(c) Explain how to build an enumerator for L_3:

$$L_3 = \{\langle M, x\rangle : M \text{ accepts } x \text{ in no more than } 2^{|x|} \text{ steps}\}$$

(Q3) Aliens in outer space.
[**Category**: Understanding, **Points**: 4]

It is widely believed by certain people that intelligent aliens exist in outer space. Some of these people will be happy to discuss seriously Invisible Pink Unicorns[3], a tendency that dramatically undercuts their credibility. On the other hand, there are people that conjecture the existence of intelligent life on earth, despite overwhelming evidence to the contrary.

One could of course try to resolve the existence of aliens in outer space question by building spacecrafts, go out, look around, etc. But this is an expensive proposition that would require much time and expense. As such, resolving the question of existence of aliens in outer space seems to be quite challenging.

A cheaper solution would be to build a TM that would resolve the question. The TM would print "Yes" on the tape if aliens in outer space exist, and then it would stop. Similarly, it aliens do not exist then it would print "No way" on the tape and stop.

Argue that the existence of aliens in outer space is a decidable problem. That is, there exists a TM that always stops and prints "Yes" iff there are aliens in outer-space.

(Q4) I am a liar.
[**Category**: Puzzle, **Points**: 4]

A town has two kinds of people, visually indistinguishable, called Grubsies and Greepies. Greepies always tell the truth; Grubsies always lie (names can be deceiving, you see).

You come to the town, and chance upon a person (who could be a Grubsy or a Greepy) and you want to find out whether a particular road leads to Wimperland (assume all people in the town know the answer to the question).

Find a single YES/NO question that you can ask the person so that you can figure out whether the road leads to Wimperland.

Grubsies and Greepies are fictional; any resemblance to person or persons living or dead or undead is purely coincidental.

Hint: think of the diagonalization proof of undecidability of the membership problem for Turing machines.

[3] See `http://en.wikipedia.org/wiki/Invisible_Pink_Unicorn`.

50.12 Homework 12: Preparation for Final

Spring 09

1. Language classification.
[**Category**: Understanding, **Points**: 10]

   Suppose that we have a set of Turing machine encodings defined by each of the following properties. That is, we have a set

   $$L = \left\{ \langle M \rangle \,\middle|\, M \text{ is a } \mathsf{TM} \text{ and } M \text{ has property } P \right\},$$

   and we are considering different ways to fill in P. Assume that the Turing machines M have only a single tape.

   (A) P is "M accepts an input string of length 3."

   (B) P is "on blank input, M halts in at most 300 transitions, leaving the entire tape blank."

   (C) P is "M's code has exactly 3 states to which there are no transitions ."

   (D) P is "M accepts no string of length 3."

   For each of these languages, determine whether it is Turing decidable, Turing recognizable, or not Turing recognizable. Briefly justify your answers.

2. Reduction I.
[**Category**: Construction, **Points**: 10]

   Define the language L to be

   $$L = \left\{ M \,\middle|\, M \text{ is a } \mathsf{TM} \text{ and } L(M) \text{ is decidable but not context free} \right\}.$$

   Show that L is undecidable by reducing A_{TM} to L. (Do the reduction directly. Do not use Rice's Theorem.)

3. Reduction II
[**Category**: Construction, **Points**: 10]

   Define the language L to be

   $$L = \left\{ M \,\middle|\, M \text{ is a } \mathsf{TM} \text{ and } \forall n \ \exists x \in L(M) \text{ where } |x| = n \right\}.$$

   Show that L is undecidable by reducing A_{TM} to L. (Do the reduction directly. Do not use Rice's Theorem.)

4. Enumerate this.
[**Category**: Construction, **Points**: 10]

   Construct an enumerator for the following set:

   $$L = \left\{ \langle \mathtt{T} \rangle \,\middle|\, \mathtt{T} \text{ is a Turing Machine and } |\mathsf{L}(\mathtt{T})| \geq 3 \right\}.$$

5. DFAs are from Mars, TMs are from Venus.
[**Category**: Understanding / Proof., **Points**: 10]

Consider the language

$$L = \left\{ \langle \mathtt{D}, w \rangle \;\middle|\; \mathtt{D} \text{ is a DFA and it accepts } w \right\}.$$

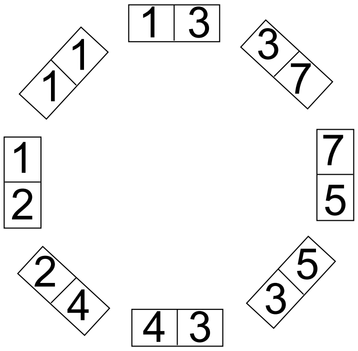
We will prove that L is undecidable by reducing A_{TM} to it:

> *Proof:* For every $\langle \mathtt{D}, w \rangle$, let $\mathtt{T}_{\mathtt{D}}$ be a TM that simulates DFA $\mathtt{D}$. So $\mathtt{D}$ will accept w iff $\mathtt{T}_{\mathtt{D}}(w)$ halts and accepts w, which is exactly equivalent to $\langle \mathtt{T}_{\mathtt{D}}, w \rangle \in \mathrm{A}_{\mathsf{TM}}$; that is,
>
> $$\langle \mathtt{D}, w \rangle \in L \quad \Longleftrightarrow \quad \langle \mathtt{T}_{\mathtt{D}}, w \rangle \in \mathrm{A}_{\mathsf{TM}}.$$
>
> This completes the reduction. But since A_{TM} is undecidable, L should be undecidable too. ■

Why this result is strange? Is it because we did something wrong in the above proof? Is this proof correct? Explain briefly.

6. Using reductions when building algorithms.
[**Category**: Construction., **Points**: 10]

Reductions are not only a technique to prove hardness of problems, but more "importantly" it is used to solve problems by transforming them into other problems that we know how to solve. This is an extremely useful technique, and the following is an example of such a reduction in the algorithmic context.



A set of domino pieces has a solution iff one can put them around a circle (face up), such that every two adjacent numbers from different pieces match. The figure on the right shows a set of pieces which has a solution.

Assume we have an algorithm (i.e., think of it as a black box) **isDomino** which given a set of dominos, can tell use whether they have a solution or they do not have a solution (it returns "yes" or "no" answer). Using **isDomino**, describe an algorithm which given a set of dominos outputs "no" if there is no solution and if there is a solution, prints out one possible solution (that is, prints out an ordered list of input dominos, such that one can put them in the same order around a circle properly). For example if the input to your algorithm is $(1,2)\#(3,5)\#(4,3)\#(1,3)\#(1,1)\#(2,4)$ $\#(3,7)$ $\#(7,5)$, it may output $(1,3)\#(3,7)\#(7,5)\#(3,5)\#(4,3)\#(2,4)\#(1,2)\#(1,1)$ (which is the solution depicted in the above figure).

(We emphasize that your solution must use **isDomino** to construct the solution. These is a direct algorithm to solve this problem, but we are more interested in the reduction here, than in an efficient solution.)

Chapter 51

Spring 2008

51.1 Homework 1: Overview and Administrivia

Spring 08

1. Set theory (10 points)

   Let $A = \{1, 2, 3\}$, $B = \{\varnothing, \{1\}, \{2\}\}$ and $C = \{1, 2, \{1, 2\}\}$. Compute $A \cup B$, $A \cap B$, $B \cap C$, $A \cap C$, $A \times B$, $A \times C$, $C - A$, $C - B$, $A \times B \times C$, and $\mathbb{P}(B)$. ($\mathbb{P}(B)$ is the power set of B.)

2. Sets of Strings (10 points)

   (a) Let $A = \{\text{apple, orange, grape}\}$ and $B = \{\text{green, blue}\}$. What is the set

   $$C = \{ba \mid a \in A, b \in B\}.$$

   (b) Let X be the string uiuc. List all substrings of X.

3. Induction and Strings (10 points)

   (a) Let our alphabet Σ be $\{a, b, c\}$. For any non-negative integer n, how many strings of (exactly) length n are there (over the alphabet Σ)?

   (b) Prove your claim in (a) by induction on n.

4. Recursive definitions (10 points)

   Consider the following recursive definition of a set S.

   (a) $(3, 5) \in S$
   (b) If $(x, y) \in S$, then $(x + 2, y) \in S$
   (c) If $(x, y) \in S$, then $(-x, y) \in S$
   (d) If $(x, y) \in S$, then $(y, x) \in S$
   (e) S is the smallest set satifying the above conditions.

   Give a nonrecursive definition of the set S. Explain why it is correct.

5. Set Theory (10 points)

   Suppose that A, B, and C are sets. We claim that if $A \cup B = A \cup C$ and $A \cap B = A \cap C$, then $B = C$.

   (a) Explain informally why this is true, using words and/or a Venn diagram.

   (b) Prove it formally, using standard identities such as DeMorgan's Laws.

51.2 Homework 2: Problem Set 2

Spring 08

1. Building a DFA.

   Let $\Sigma = \{0, 1\}$. Give DFA state diagrams for the following languages.

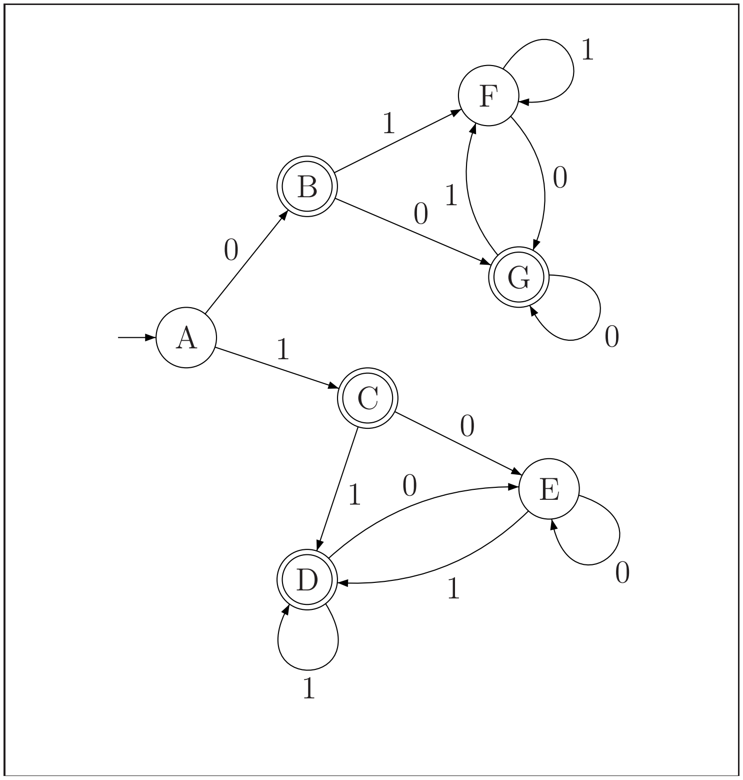
   (a) $L = \{w \mid w$ contains the substring 001 $\}$

   (b) $L = \{a \mid$ the length of a is not divisible by 2 and not divisible 3 $\}$.

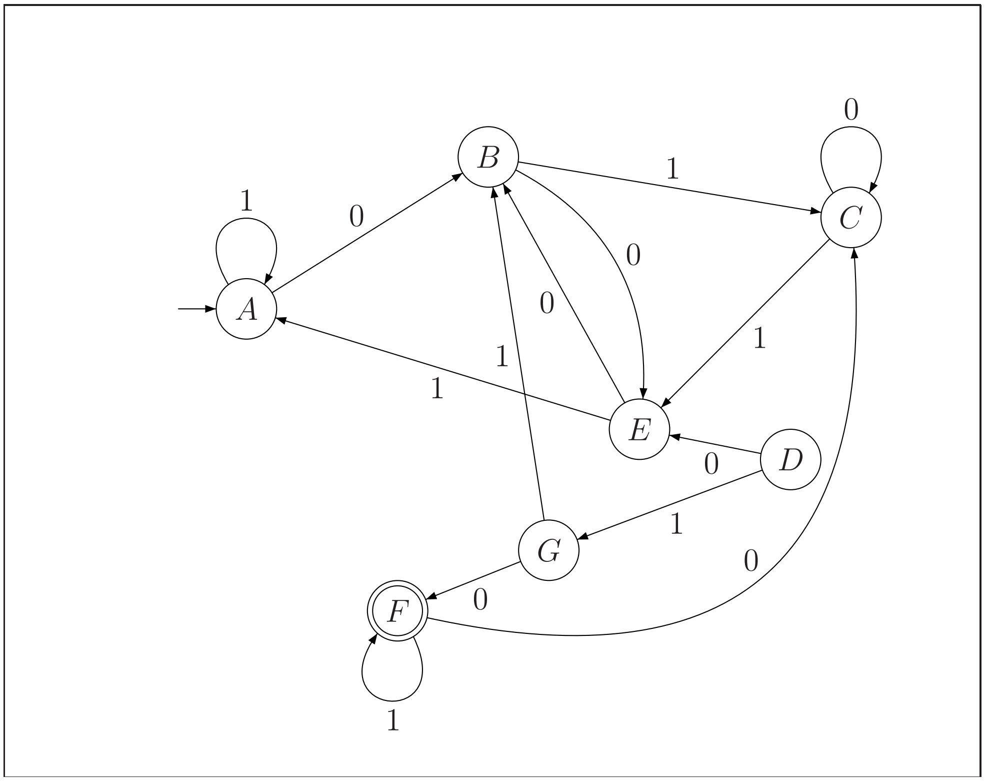
   (c) $L = \Sigma^* - \{\epsilon\}$

2. Interpreting a DFA

   (a) What is the language of the following DFA? That is, explicitly specify all the strings that the following DFA accepts. Briefly explain why your answer is correct.



   (b) What about this one? Again, briefly justify your answer.



3. Sets and trees

Define a “set-based binary tree” (SBBT) as follows:

- Every positive integer is an SBBT.
- If x and y are SBBTs, $x \neq y$, then $\{x, y\}$ is also an SBBT.

For example, $R = \{\{3, 2\}, 4\}$ and $P = \{3, \{\{7, 9\}, \{8, 3\}\}$ are SBBTs. But $\{2, \{4, 5\}, 27\}$ and $\{2, \{3\}\}$ are not SBBTs. Let T be the set of all SBBTs.

(a) Let's define the following function f mapping SBBTs to sets of integers:

$$f(X) = \begin{cases} \bigcup_{Y \in X} f(Y) & \text{if } X \text{ is a set} \\ \{X\} & \text{if } X \text{ is an integer} \end{cases}$$

Notice that $f(Y)$ is always a set, for any input Y. The operation $\bigcup_{Y \in X} f(Y)$ unions together the sets $f(Y)$, for all the items Y that are in the set X.

For the SBBTs P and R defined above, compute $f(P)$ and $f(R)$. Give a general description of what f does.

(b) Similarly, we can define a function g mapping SBBTs to integers:

$$g(X) = \begin{cases} \sum_{Y \in X} g(Y) & \text{if } X \text{ is a set} \\ 1 & \text{if } X \text{ is an integer} \end{cases}$$

Give the values for $g(P)$ and $g(R)$, as well as a general description of what g does.

(c) For certain SBBTs, $g(X) = |f(X)|$. For which SBBTs does this equation work? Explain why it's not true in general.

4. Balanced strings

A string over $\{0, 1\}$ is *balanced*, if it has the same number of zeros and ones.

(a) Provide a pseudo-code (as simple as possible) for a program that decides if the input is balanced. You can use only a single integer typed variable in your program, and one variable containing the current character (of type `char`). You can read the next character using a library function called, say, `get_char`, which returns -1, if it reached the end of the input. Otherwise, `get_char` returns the next character in the input stream.

In particular, the program prints "accept" if the input is a balanced string, and print "reject", otherwise.

(b) For any fixed prespecified value of k, describe how to construct an automata that accepts a balanced string if, in every prefix of the input string, the absolute difference in the number of zeros and ones does not exceed k. How many states does your automata needs?

(c) Provide an intuitive explanation of why the number of states in automata for the problem of part (B), must have at least, say, k states.

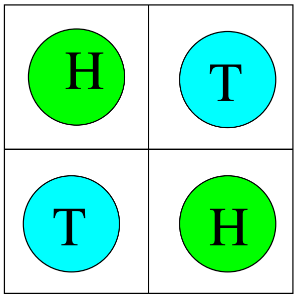
(d) Argue, that there is no *finite* automata that accepts only balanced strings.

For bonus credit, you can provide formal proofs for the claims above.

5. Bonus problem (Coins)

A journalist, named Jane Austen, unfortunately (for her) interviews one of the presidential candidates. The candidate refuses to let Jane end the interview going on and on about the candidate plans how to solve all the problems in the world. In the end, the candidate offers Jane a game. If she wins the game she can leave.

The game board is made out of 2×2 coins:



At each round, Jane can decide to flip one or two coins, by specifying which coins she is flipping (for example, flip the left bottom coin and the right top coin), next the candidate goes and rotates the board by either $90, 180, 270$, or 0 degrees. (Of course, rotation by 0 degrees is just keeping the coins in their current configuration.)

The game is over when all the four coins are either all heads or all tails. To make things interesting, Jane does not see the board, and does not know the starting configuration.

Describe an algorithm that Jane can deploy, so that she always win. How many rounds are required by your algorithm?

51.3 Homework 3: DFAs and regular languages

Spring 08

1. Building and interpreting regular expressions.

Give regular expressions for the following languages over the alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$.

(a) $L_a = \left\{ w \;\middle|\; |w| \text{ is a multiple of 3 and } w \text{ starts with } \mathtt{aa} \right\}$.

(b) $L_b = \left\{ w \;\middle|\; w \text{ contains the substring } \mathtt{bbb} \text{ or } \mathtt{bab} \right\}$.

(c) $L_c = \left\{ w \,\middle|\, w \text{ does not contain two consecutive } \texttt{b}\text{'s} \right\}$.

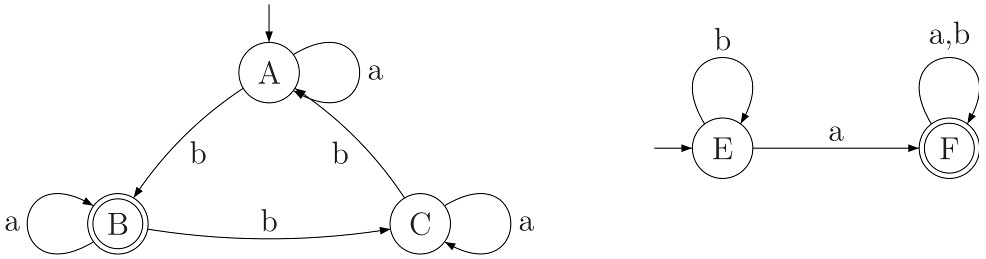
Describe the languages represented by the following regular expressions:

(d) $(a \cup b \cup \epsilon)((a \cup b)(a \cup b)(a \cup b))^*(a \cup b \cup \epsilon)$.

(e) $(aa \cup bb \cup ab \cup ba \cup \epsilon)^+$

2. Product construction

(a) Consider the following two DFAs. Use the product construction (pp. 45–47 in Sipser) to construct the state diagram for a DFA recognizing the intersection of the two languages.



(b) When the product construction was presented in class (and in Sipser), we assumed that the two DFAs had the same alphabet. Suppose that we are given two DFAs M_1 and M_2 with different alphabets. E.g. $M_1 = (Q_1, \Sigma_1, \delta_1, q_1, F_1)$ and $M_2 = (Q_2, \Sigma_2, \delta_2, q_2, F_2)$. To build a DFA M that recognizes $L(M_1) \cup L(M_2)$, we need to add two additional sink states s_1 and s_2. We send the first or the second element of each pair to the appropriate sink state if the incoming character is not in the alphabet for its DFA.

Write out the new equations for M's transition function δ and its set of final states F.

3. Algorithms for modifying DFAs.

Suppose that $M = (Q, \Sigma, \delta, q_0, F)$ and $N = (R, \Sigma, \gamma, r_0, G)$ are two DFAs sharing the common alphabet $\Sigma = \{a, b\}$.

(a) Define a new DFA $M' = (Q \cup \{q_X, q_R\}, \Sigma \cup \{\#\}, \delta', q_0, \{q_X\})$ whose transition function is defined as follows

$$\delta'(q,t) = \begin{cases} \delta(q,t) & q \in Q \text{ and } t \in \Sigma \\ q_X & q \in F, t = \# \\ q_X & q = q_X \\ q_R & \text{otherwise.} \end{cases}$$

Describe the language accepted by M' in terms of the language accepted by M.

(b) Show how to design a DFA N' which accepts the language

$$L' = \left\{ ttw \,\middle|\, (t = a \text{ and } w \in L(M)) \text{ or } (t = b \text{ and } w \in L(N)) \right\}.$$

Define your DFA using notation similar to the definition of M' in part (a).

4. Shared channel.

Funky Computer Systems, who have now gone out of business, submitted the lowest bid for wiring the DFAs supporting the Siebel center classrooms. These idiots wired two of the DFAs M and N so that their inputs come in on a shared input channel. When you try to submit a string w to M and a string y to N, this single channel receives the characters for the two strings interleaved. For example, if $w = \texttt{abba}$ and $y = \texttt{cccd}$, then the channel will get a string like $\texttt{abbcccad}$ or $\texttt{acbccbad}$.

Fortunately, these two DFAs have alphabets that do not overlap, so it's possible to sort this out. Your job is to design a DFA that accepts a string on the shared channel exactly when M and N would have accepted the two input strings separately.

Specifically, let $M = (Q, \Sigma, \delta, q_0, F)$ and $N = (R, \Gamma, \gamma, r_0, G)$, where $\Sigma \cap \Gamma = \emptyset$. Your new machine M' should read strings from $(\Sigma \cup \Gamma)^*$. It should be designed using a variation on the product construction, i.e. its state set should be $Q \times R$.

Give a formal definition of M' in terms of M and N. Also briefly explain the ideas behind how it works (very important especially if your formal notation is buggy).

5. Union in/out game.

   A palindrome is a string that if you reverse it, remains the same. Thus `tattarrattat`[1] is a palindrome. You could consider the empty string ϵ to be a palindrome. But, for this problem, let's consider only strings containing at least one character.

   (a) Let L_k be the language of all palindromes of length k, over the alphabet $\Sigma = \{a, b\}$. Show a DFA for L_4.

   (b) For any fixed k, specify the DFA accepting L_k.

   (c) Let L be the language of all palindromes over Σ. Argue,as precisely as you can, that L is not regular.

   (d) We can express the language L as $\cup_{k=1}^{\infty} L_k$. It's tempting to reason as follows: the language L is the union of regular languages, and as such it is regular.

   What is the flaw in this argument, and why is it *wrong* in our case?

51.4 Homework 4: NFAs

Spring 08

1. NFA design/subset construction.

   (a) Design an NFA for the following language:

   $$L = \left\{ x \,\middle|\, x \text{ is a binary string that has 1101 or 1100 or 0001 as a substring} \right\}.$$

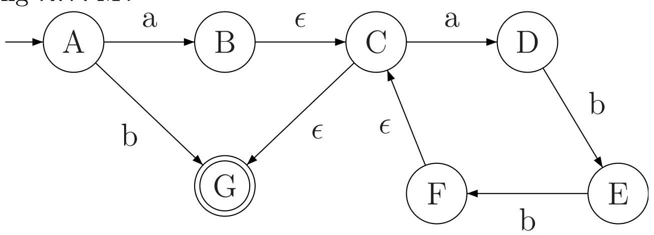
   Design this initial NFA in a modular way, using ϵ-transitions.

   (b) Remove all ϵ-transitions from your NFA. (Namely, present an equivalent NFA with no ϵ-transitions.)

   (c) Convert your ϵ-free NFA into a DFA using subset construction.

2. NFA interpretation/formal definitions.

   Consider the following NFA M.

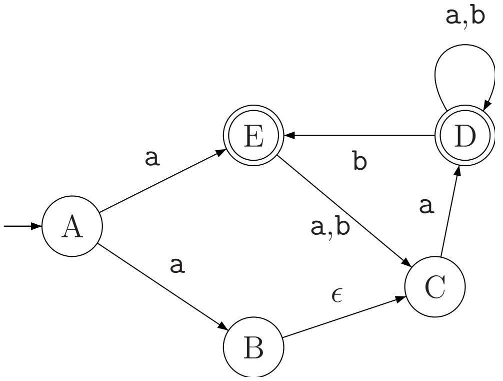


   (a) Give a regular expression that represents the language of M. Explain briefly why it is correct.

[1] Which is the longest palindromic word in the Oxford English Dictionary is tattarrattat, coined by James Joyce in Ulysses for a knock on the door.

(b) Recall the definition of an NFA accepting a string w (Sipser p. 54). Show formally that M accepts the string $w = \texttt{aabb}$

(c) Let $\Sigma = \{\texttt{a}, \texttt{b}\}$. Give the formal definition of the following NFA N.



3. NFA DESIGN WITH GUESSING.

Let $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}\}$ and define the language L

$$L = \left\{ x_1 \# x_2 \# \ldots \# x_n \;\middle|\; \forall i, x_i \in \Sigma^2, \text{ and } \exists i, j \text{ such that } i \neq j \text{ and } x_j = x_k \right\}.$$

That is, each x_i is a string of two characters from Σ. And two of the x_i's need to be identical, but you don't know which two are identical. So the language contains ab#bb#cc#ab and ac#bb#ac#ab, but not aa#ac#bb.

Design an NFA that recognizes L. This NFA should "guess" when it is at the start of each matching string and verify that its guess is correct.

4. NFA MODIFICATION.

The 2SWP operation on strings interchanges the character in each odd position with the character in the following even position. That is, if the string length k is even, the string $w_1 w_2 w_3 w_4 \ldots w_{k-1} w_k$ becomes $w_2 w_1 w_4 w_3 \ldots w_k w_{k-1}$. E.g. abcbac becomes babcca. If the string has odd length, we just leave the last (unpaired) character alone. E.g. abcba becomes babca.

Given a whole language L, we define $\mathsf{2SWP}(L)$ to be $\left\{ \mathsf{2SWP}(w) \;\middle|\; w \in L \right\}$.

Show that regular languages are closed under the 2SWP operation. That is, show that if L is a regular language, then $\mathsf{2SWP}(L)$ is regular. That is, suppose that L is recognized by some DFA M. Explain how to build an NFA N which accepts $\mathsf{2SWP}(L)$.

51.5 Homework 5: NFA conversion

Spring 08

Problem 1: DFA/NFA TO REGEX.

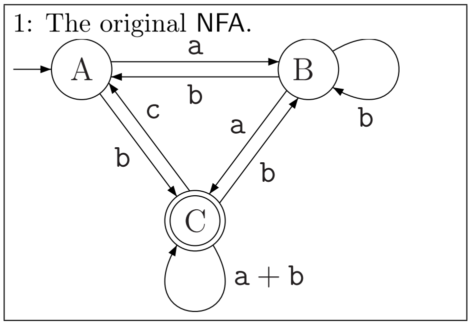
In lecture 8 (Sipser pp. 69–76), we saw a procedure for converting a DFA to a regular expression. This algorithm also works even if the input is an NFA.

For the following NFA, use this procedure to compute an equivalent regular expression. So that everyone does the same thing (and we don't create a grading nightmare), you should do this by:

– Adding new start and end states, and then

– removing states A, B, C **in that order**.

Provide detailed drawing of the GNFA after each step in this process.


1: The original NFA.

Note, that in this problem you will get interesting self-loops. For example, one can travel to from B to A and then back B. This creates a self-loop at B when A is removed.

51.6 Homework 6: Non-regularity via Pumping Lemma

Spring 08

1. Pumping lemma problems:

   (a) Use pumping lemma to prove that L is not regular, where:

   $$L = \{a^k b^m : k \leq m \text{ or } m \leq 2k\}$$

   (b) Prove that the following language satisfies pumping lemma:

   $$L = \{00, 11\}$$

   From the fact that it satisfies the pumping lemma, can we deduce that L is regular? Why or why not?

2. Decide of the following languages are regular. If they are regular, give a DFA, NFA or regular expression for the language. If it is not regular, then give a proof using either closure properties or the pumping lemma.

   (a) $L_1 = \{xy \mid x$ has the same number of a's as $y\}$

   (b) $L_2 = \{w \mid w$ has three times the number of a's than b's$\}$

   (c) $L_3 = \{xy \mid x$ has the same number of a's as y and $|x| = |y|\}$

3. Let T be the language $\{0^n 1^n : n \geq 0\}$. Use closure properties to show that the following languages are not regular, using a proof by contradiction and the fact that T is known not to be regular.

   (a) $L = \{a^n b^m c^{n+m} : n \geq m \geq 0\}$

   (b) $J = \{0^n 1^n 2^n : n \geq 1\}$

4. (bonus) Show that if L is regular then $t(L)$ is regular where:

   $$t(L) = \{x : \text{for some string } y,\ |y| = |x| \text{ and } xy \in L\}$$

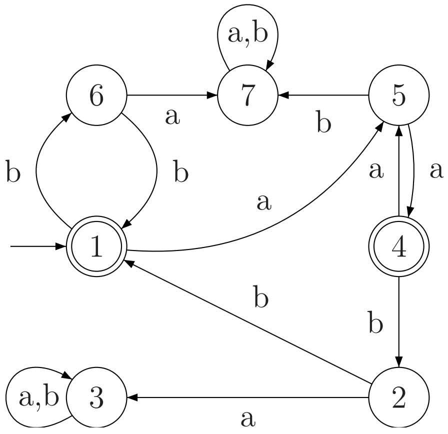
   so t is an operation that preserves regularity.

51.7 Homework 7: CFGs

Spring 08

1. Suffix languages.

   Consider the following DFA:



   (a) Write down the suffix language for each state.

   (b) Draw a DFA that has the same language as the one above, but has the minimal number of states.

2. Context-free grammar design

   Give context-free grammars generating the following languages:

   (a) $L_1 = \left\{ \mathtt{a}^n \mathtt{b}^p \;\middle|\; 0 < p < n \right\}$.

   (b) $L_2 = \{\ a^n b^n c^m d^m \mid n, m \in \mathbb{N}\ \}$

   (c) $L_3 = \left\{ \mathtt{a}^n \mathtt{b}^m \mathtt{c}^p \;\middle|\; n = m \text{ or } m = p \right\}$.

   (d) $L_4 = \mathtt{a}(\mathtt{ab}^*)^*$.

3. Context-free grammar interpretation.

   (a) What is the language of this grammar? The alphabet is $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$ and start symbol is T.

$$S \rightarrow \mathtt{a}S\mathtt{b} \mid \epsilon$$
$$T \rightarrow S \mid \mathtt{c}\,T \mid T\,\mathtt{d}$$

   (b) Answer the same question for this grammar, with same alphabet and start symbol.

$$S \rightarrow \mathtt{a}S\mathtt{b} \mid \epsilon$$
$$T \rightarrow S \mid \mathtt{c}S \mid S\mathtt{d}$$

   (c) Answer the same question for this grammar, with same alphabet and start symbol.

$$S \rightarrow T\,\mathtt{b}$$
$$T \rightarrow \mathtt{aa}S \mid \mathtt{cd}$$

4. NFA PATTERN MATCHING.

   Pattern-search programs take two inputs: a pattern given by the user and a file of text. The program determines whether the text file contains a match to the pattern, typically using some variation on NFA/DFA technology. Fully developed programs, such as `grep`, accept patterns containing regular-expression operators (e.g. union) and also other convenient shorthands. Our patterns will be much simpler.

   Let's fix an alphabet $\Sigma = \{\texttt{a}, \texttt{b}, ...\texttt{z}, \sqcup\}$. Let $\Gamma = \Sigma \cup \{?, [,], *\}$. A **pattern** will be any string in Γ^*.

   A string w matches a pattern p if you can line up the characters in the two strings such that:

   - When p contains a character from Σ, it must be paired with an identical character in w.
   - The character ? in p can match any substring x in w, where x contains at least one character.
   - When p contains a substring of the form $[w]*$, this can match zero or more repetitions of whatever w matches.

   For example, the pattern "$\texttt{fleck}$" matches only the string "$\texttt{fleck}$". The pattern "$\texttt{margaret?fleck}$" will match anything containing "$\texttt{margaret}$" and "$\texttt{fleck}$", separated by at least one character. The pattern "$\texttt{i} \sqcup \texttt{ate} \sqcup [\texttt{many}\sqcup] * \texttt{donuts}$" matches strings like

   "$\texttt{i} \sqcup \texttt{ate} \sqcup \texttt{donuts}$" and

   "$\texttt{i} \sqcup \texttt{ate} \sqcup \texttt{many} \sqcup \texttt{many} \sqcup \texttt{donuts}$"

   Instances of []* can be nested. So the pattern $\texttt{cc}[\texttt{bb}[\texttt{a}] * \texttt{bb}] * \texttt{dd}$ matches strings like $\texttt{ccdd}$ or $\texttt{ccbbaaaaabbdd}$ or $\texttt{ccbbabbbbabbdd}$.

   A text file t contains a match to a pattern p if t contains some substring w such that w matches p.

   Design an algorithm which converts a pattern p to an NFA N_p that searches for matches to p. That is, the NFA N_p will read an input text file t and accept t if and only if t contains a match to p. N_p searches for only one fixed pattern p. However you must describe a general method of constructing N_p from any input pattern p.

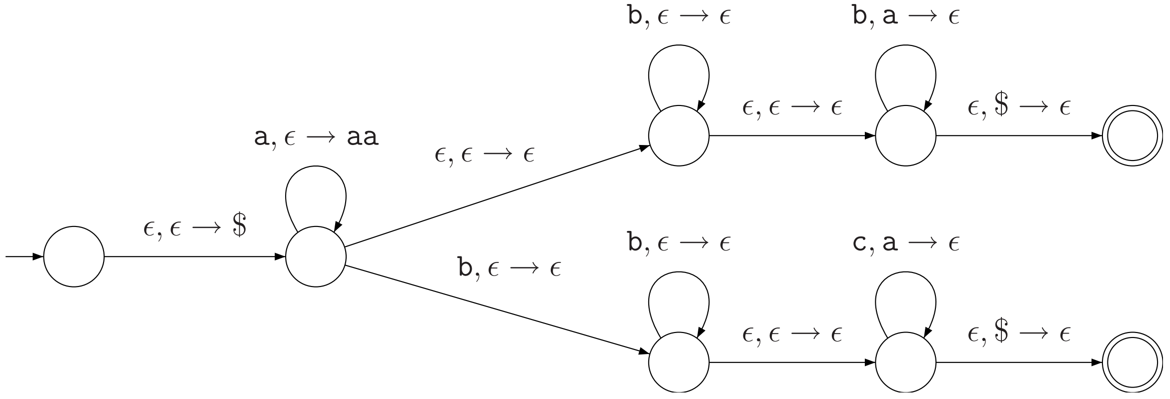
   You can assume that your input pattern p has been checked to ensure that it's well-formed and that we have a function m which matches open and close brackets. For example, you can assume that an open bracket (]) at position i in the pattern is immediately followed by a star (*). You can also assume that there is a matching open bracket ([) at position $m(i)$ in the pattern. The function m is a bijection, so if there is an open bracket at position j in the pattern, $m^{-1}(j)$ returns the corresponding close bracket.

51.8 Homework 8: CFGs

Spring 08

1. EXTRACT LANGUAGE FROM PDA.

   Give the language of the following PDA .



2. Converting CFG to PDA.

   For each of the following languages (with alphabet $\Sigma = \{\texttt{a}, \texttt{b}\}$) construct a pushdown automaton recognizing that language, following the general construction for converting a context-free grammar to a PDA (lecture 13, pp. 115–118 in Sipser).

   For each language, also give a parse tree for the word w, a leftmost derivation for w, and the first 10 configurations (state and stack contents) for the PDA as it accepts w.

   a) The word $w = \texttt{aababbaabbbb}$ and the grammar with start symbol S:

      SP$\texttt{a}TT\texttt{b}$
      TP$\epsilon \mid T\texttt{ab} \mid \texttt{a}T\texttt{b} \mid \texttt{b}$

   b) The word $w = \texttt{babaabaabaaba}$ and the grammar with start symbol S:

      SP$AB \mid B$
      AP$BB \mid B$
      BP$CC \mid \texttt{b}$
      CP$\texttt{aba}$

3. Language to PDA.

   Let $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}\}$ and consider the language

$$L = \left\{ \texttt{a}^i\texttt{b}^j\texttt{c}^k \;\middle|\; i \neq j \text{ or } j \neq k \right\}.$$

   Design a PDA for L. Present your PDA as a state diagram, with brief comments about how it works.

4. Context-free grammar design.

   Let $\Sigma = \{\texttt{a}, \texttt{b}\}$.

   A pair of strings (x, w) is an S-pair if they are identical except that two characters have been swapped. Formally, if $x = c_1c_2 \ldots c_n$ and $w = d_1d_2 \ldots d_n$, then there are two character positions i and j such that $c_i = d_j$ and $c_j = d_i$, and $c_k = d_k$ for k other than i or j.

   (a) Notice that for any word $x \in \Sigma^*$, it holds that (x, x) is always a S-pair. Briefly explain why.

   (b) Let $L_S = \left\{ wx^R \;\middle|\; x, w \in \Sigma^* \text{ and } (x, w) \text{ is an S-pair} \right\}$.
   Give a context-free grammar that generates L_S.

   (c) Let

$$L_T = \left\{ w_1\#w_2\# \ldots w_i\# \;\middle|\; \begin{array}{l} w_i \in \Sigma^* \text{ for all } i \text{ and} \\ \exists i, j \text{ such that } i < j \text{ and } (w_i, w_j^R) \text{ is an } S\text{-pair} \end{array} \right\}.$$

   Give a context-free grammar that generates L_T.

5. Context-free grammar. (Bonus)

   The following grammar (with start symbol S) is ambiguous:

$$S \Rightarrow \texttt{a}S \mid \texttt{a}S\texttt{b}S \mid \epsilon$$

   (a) Show that the grammar is ambiguous, by giving two parse trees for some string w.

   (b) Give an efficient test to determine whether a string w in $L(S)$ is ambiguous. Explain informally why your test works.

51.9 Homework 9: Chomsky Normal Form

Spring 08

1. Chomsky Normal Form.

   (a) Remove nullable variables from the following grammar (with start symbol S):

$$\begin{aligned} \mathsf{S} \rightarrow & \ \mathsf{aAa} \mid \mathsf{bBb} \mid \mathsf{BB} \\ \mathsf{A} \rightarrow & \ \mathsf{C} \\ \mathsf{B} \rightarrow & \ \mathsf{S} \mid \mathsf{A} \\ \mathsf{C} \rightarrow & \ \mathsf{S} \mid \epsilon \end{aligned}$$

   (b) This grammar (with start symbol S) has no nullable variables. Generate its Chomsky normal form.

$$\begin{aligned} \mathsf{S} \rightarrow & \ \mathsf{ASB} \\ \mathsf{A} \rightarrow & \ \mathsf{aAS} \mid \mathsf{a} \\ \mathsf{B} \rightarrow & \ \mathsf{SbS} \mid \mathsf{A} \mid b \end{aligned}$$

2. Context free language closure properties.

   We know (example 2.37 in Sipser) that the following language is not context free:

$$C = \left\{ \mathsf{a}^i \mathsf{b}^j \mathsf{c}^k \;\middle|\; 0 \leq i \leq j \leq k \right\}.$$

   Using closure properties of context-free languages, and the fact that C is not context-free, prove that the following languages are not context free:

   (a) $J = \left\{ \mathsf{a}^i \mathsf{b}^j \mathsf{c}^{k-1} \;\middle|\; 1 \leq i \leq j \leq k \right\}$

   (b) $K = \left\{ \mathsf{a}^i \mathsf{c}^j \mathsf{b}^k \mathsf{c}^n \;\middle|\; 0 \leq i \leq j \leq k, 0 \leq n \right\}$

3. Grammar-based induction.

   Let G be the grammar with start symbol S, terminal alphabet $\Sigma = \{\mathsf{a}, \mathsf{b}\}$ and the following rules:

   $\mathsf{S} \rightarrow \mathsf{aX} \mid \mathsf{Y}.$

   $\mathsf{X} \rightarrow \mathsf{aS} \mid \mathsf{a}.$

   $\mathsf{Y} \rightarrow \mathsf{bbY} \mid \mathsf{aa}.$

   Claim version 1. *If w is a string in $L(G)$, then w contains an even number of a's.*

   It's actually easier to prove the following, stronger and more explicit claim:

   Claim version 2. *For any n, show that if a string $w \in \Sigma^*$ can be derived from either S or Y in n steps, then w contains an even number of a's.*

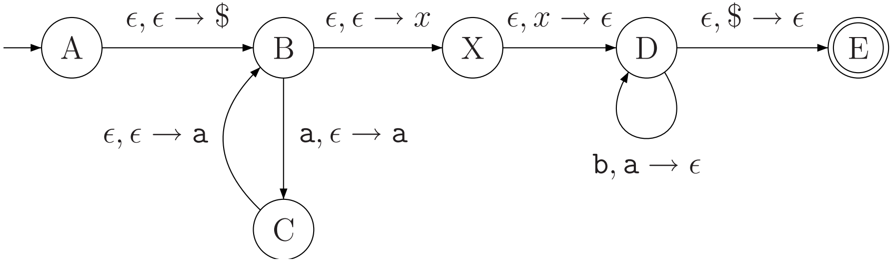
   (a.) The original claim involved strings in $L(G)$, i.e. strings that can be derived from the start symbol S.

   Why did we extend the claim to include derivations starting with the variable Y? Why didn't we extend it even further to include derivations starting with the variable X?

   (b.) Prove version 2 of the claim using strong induction on the derivation length n.

4. PDA TO CFG CONVERSION.

Consider the following PDA.


A
$\epsilon, \epsilon \rightarrow \$$
B
$\epsilon, \epsilon \rightarrow x$
X
$\epsilon, x \rightarrow \epsilon$
D
$\epsilon, \$ \rightarrow \epsilon$
E
$\epsilon, \epsilon \rightarrow$ a
a, $\epsilon \rightarrow$ a
b, a $\rightarrow \epsilon$
C

Recall the proof that a PDA is a context free language on page 119–120 of Sipser (and in the notes for Lecture 14). For the above PDA.

(a.) Generate rules defined by the first bullet point of the proof on page 120.

(b.) What is the start variable?

(c.) How many rules are generated by the second bullet point. Explain why your answer is correct.

5. GIVE ME THAT OLD TIME PDA, ITS GOOD ENOUGH FOR ME, IT WAS GOOD ENOUGH FOR MY FATHER.

Tony had just released into the market a new model of PDA called Blu-PDA (Sushiba released a competing PDA product called HD-PDA, but thats really a topic for a different exercise).

Instead of a stack, like the good old PDA, the new Blu-PDA has a queue. You can push/pop characters from both sides of the queue (thus, the Blu-PDA can see both characters stored in the front and back of the queue when making a transition decision [and the current input character of course]). Since Tony is targeting this product to the amateur market, they decided to limit the queue size to 25 (if the queue size exceeds 25, then it stops and reject the input). Tony claims that the new Blu-PDA is a major breakthrough and a considerably stronger computer than a PDA.

(Of course, if the Blu-PDA does strange things like reading characters from an empty queue, or popping characters from an empty queue, then it immediately rejects the input.)

(a) So, given a Blu-PDA, is it equivalent to a PDA, DFA, or is it stronger than both?

(b) Explain clearly why your answer is correct.

(c) (5 point bonus) Prove your answer in detail.

51.10 Homework 10: Turing Machines

Spring 08

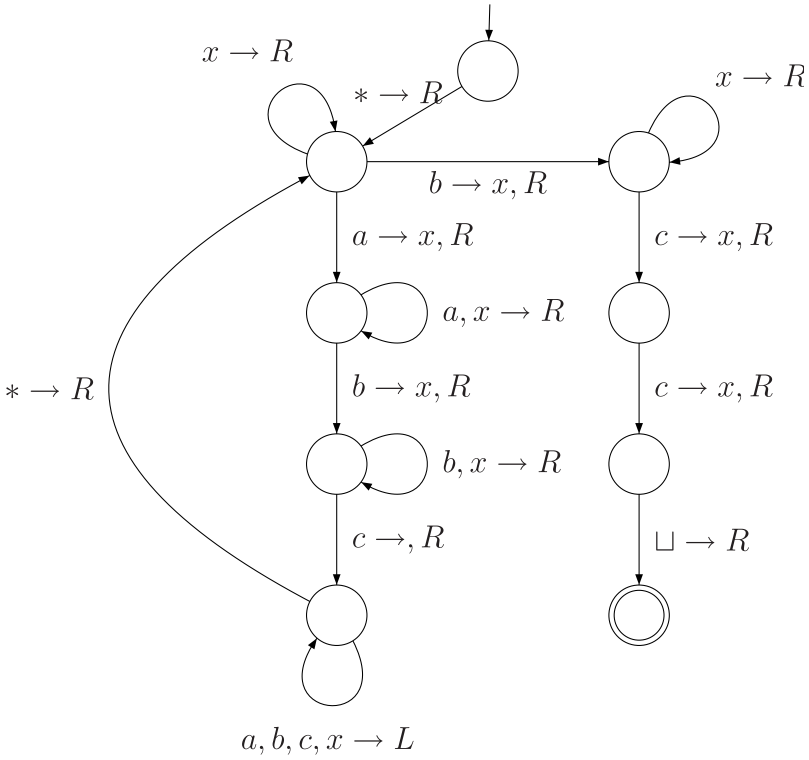
1. TURING MACHINES.

Give the state diagram for a Turing Machine for the following language.

$$L = \{a^n b^{n+1} c^{n+2} | n \geq 0\}$$

To simplify your design, you can assume the beginning of the string is marked with $*$. (Inputs that don't start with a $*$ should be rejected.) For example, the input may look like $*abc$.

You do not need to draw transitions that lead to the (implicit) reject state. Hence, any transition that is not present in your diagram will be assumed to lead to the reject state. Indicate which symbol (e.g. ⊔ or B) you are using for the special blank symbol that fills the end of the tape.



2. Context-free Pumping Lemma. (bonus)

   Use pumping lemma for CFLs to show that L is not a CFL:

$$L = \left\{ \mathtt{a}^i \mathtt{b}^j \mathtt{c}^k \;\middle|\; i < j < k \right\}.$$

51.11 Homework 11: Turing Machines

Spring 08

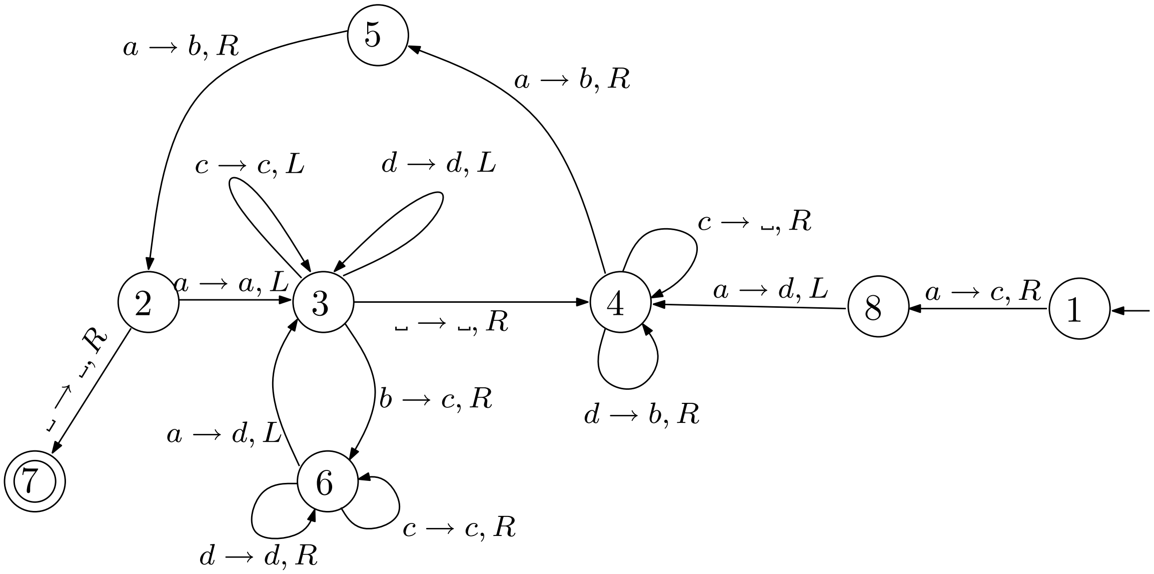
1. Turing machine tracing.

   In the following Turing machine $\Sigma = \{\mathtt{a}\}$ and $\Gamma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{B}\}$.

   (a) What is the language of this TM?

   (b) Informally and briefly explain why this TM accepts the language you claimed in the previous part.

   (c) Trace the execution of this TM as it processes string aaaaaaaaa (i.e., a sequence of 9 as) by providing the sequence of configurations it goes through (i.e., tape & state in each step - use the configuration notation shown in class).



2. High Level TM Design.

Design a Turing Machine that for a given input tape with n cells containing `0`'s, marks the positions which are composite numbers. Specifically, cells at the prime-numbered positions are left containing `0`, cells at composite-numbered locations are left containing `X`, and the cell at the first (leftmost) location is left containing `U` (for unit). For example, consider the input `0000000000`. This represents the first 10 numbers. So the Turing machine should halt with `U00X0X0XXX` on the tape.

3. Turing machine encodings.

In this problem we demonstrate a possible encoding of a TM using the alphabet $\{0, 1, ;, |\}$ where $|$ is the newline character. We encode $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ as a string $n|i|j|t|r|s|w$ where n, i, j, t, r, s are integers in binary representing $|Q|$, $|\Sigma|$, $|\Gamma|$, q_0, q_{acc}, q_{rej} and w represents Σ as described below. We adopt the convention that states are numbered from 0 to $n - 1$, the input alphabet symbols are numbered from 0 to $i - 1$, and the tape alphabet symbols are numbered from 0 to $j - 1$ with $j - 1$ representing the special blank symbol (therefore $j > i$).

The string w represents δ as follows. Each transition $(q, \mathtt{a}) \to (q', \mathtt{b}, D)$ is represented as a 5-tuple $q; \mathtt{a}; q'; \mathtt{b}; D$ where $q, \mathtt{a}, q', \mathtt{b}$ are integers in binary and D is either 0 (move left) or 1 (move right). We adopt the convention that only useful transitions are represented and any transition not represented leads to the reject state. The string w consists of 5-tuples separated by $|$. Thus $w = w_1|w_2|...|w_p$ where p is the number of useful transitions.

Here is the representation of a mystery Turing machine M, using this encoding. For ease of reading, we have shown | as an actual line break and given the integers in decimal rather than binary.

```
8
2
3
7
3
5
7;1;0;2;1
0;1;0;1;1
0;0;1;1;1
1;1;1;1;1
1;0;0;0;1
0;2;2;2;0
6;2;3;2;1
2;1;2;1;0
2;0;6;0;0
6;1;6;1;0
6;0;4;0;0
4;0;4;0;0
4;1;4;1;0
4;2;0;2;1
```

(a) Draw a state diagram for this TM (omitting the reject state).

(b) What is the language of this TM? Give a brief justification.

4. Turing machine simulation.

A PlaneTM is a TM that instead of a tape (line), uses a quadrant of the plane for recording. Consider all the points in the plane with coordinates $P = \Big\{(x, y) \;\Big|\; x, y \in \mathbb{Z},\;\; x, y \geq 0\Big\}$. We assume that each point of P indicates a memory cell in the plane and a PlaneTM has a head that can change the content of the cell that it is on top of it and after that the head transfers to a neighbor cell.

Formally a PlaneTM is $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$, where Q is the set of states, Σ and Γ are the input and tape alphabets, $q_0, q_{\text{acc}}, q_{\text{rej}}$ are the initial, accept and reject states, respectively. The initial position of the head is always cell $(0, 0)$. The input sequence is written from left to right on the cells $(0, 0), (1, 0), (2, 0), \cdots$; and the rest of cells are filled with ␣ (blank symbol, ␣ $\in \Gamma$). Transition function, $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{\mathtt{U}, \mathtt{D}, \mathtt{L}, \mathtt{R}\}$, where $\mathtt{U}$ (resp. $\mathtt{D}$) indicates that the head should be moved up (resp. down).

If $\delta(q, \mathtt{a}) = (q', \mathtt{b}, D)$ then the PlaneTM moves from state q when seeing character $\mathtt{a}$ under the tape head, to the state q', replace character $\mathtt{a}$ with character $\mathtt{b}$. Furthermore, the head is moved to the cell above/below/left/right of the current cell if D is $\mathtt{U}, \mathtt{D}, \mathtt{L}, \mathtt{R}$, respectively.

As with our normal Turing machines, the head simply stays put if the commanded move would take it off the edge of the quadrant.

When a PlaneTM powers up, its input (which contains no blanks) occupies a rectangular region in the lower left corner. The other cells (i.e. the infinite areas to the right and above the input) are filled with blanks.

(a) Show that every TM can be simulated using a PlaneTM.

(b) Show that every PlaneTM can be simulated using a TM.

From this, we can conclude that TM's and PlaneTM's are equivalent.

51.12 Homework 12: Enumerators

Spring 08

1. Decidable problems.

   Prove that L is a decidable language:

   $$L = \left\{ \langle D, k \rangle \;\middle|\; \begin{array}{l} D \text{ accepts no string of length } \leq k, \\ \text{and } D \text{ is a } \mathsf{NFA} \end{array} \right\}.$$

2. Enumerators I.

   An enumerator for a language L is a Turing machine that writes out a list of all strings in L. See p. 152–153 in Sipser.

   The enumerator has no input tape. Instead, it has an output tape on which it prints the strings, with some sort of separator (e.g. #) between then. The strings can be printed in any order and duplicates of the same string are ok. But each string in L must be printed eventually.

   Design an enumerator that writes all tuples of the form (n, p) where $n \in \mathbb{N}$, $p \in \mathbb{N}$, and n is a multiple of p.

3. Enumerators II.

   If L and J are two languages, define $L \oplus J$ to be the language containing all strings that are in exactly one of L and J. That is

   $$L \oplus J = \{ w \mid w \in L \text{ and } w \notin J \text{ or } w \in J \text{ and } w \notin L \}$$

   Suppose that you are given two context-free grammars G and H.

   (a) Design an enumerator that will print all strings in $L(G) \oplus (H)$.

   (b) Is $L(G) \oplus (H)$ context-free? TM recognizable? TM decidable? Briefly justify your answer.

   (c) Recall that

   $$EQ_{CFG} = \left\{ \langle G, H \rangle \;\middle|\; G \text{ and } H \text{ are } \mathsf{CFG}\text{'s and } L(G) = L(H) \right\}.$$

   We have mentioned in class that EQ_{CFG} is undecidable. Why is this problem harder than the ones you just solved in parts (a) and (b)?

51.13 Homework 13: Enumerators

Spring 08

1. Language classification.

   Suppose that we have a set of Turing machine encodings defined by each of the following properties. That is, we have a set

   $$L = \left\{ \langle M \rangle \;\middle|\; M \text{ is a } \mathsf{TM} \text{ and } M \text{ has property } P \right\},$$

   and we are considering different ways to fill in P. Assume that the Turing machines M have only a single tape.

   (a) P is "there is an input string which M accepts after no more than 327 transitions."

   (b) P is "on blank input, M halts leaving the entire tape blank."

   (c) P is "M's code has no transitions into the reject state."

(d) P is "on input UIUC, M never changes the contents of the even-numbered positions on its tape." (That is, it can read the even-numbered positions, but not write a different symbol onto them.)

For each of these languages, determine whether it is Turing decidable, Turing recognizable, or not Turing recognizable. Briefly justify your answers.

2. Reduction I.

   Define the language L to be

   $$L = \left\{ \langle M \rangle \;\middle|\; M \text{ is a } \mathsf{TM},\ L(M) \text{ is context free but is not regular} \right\}.‘$$

   Show that L is undecidable by reducing A_{TM} to L. (Do the reduction directly. Do not use Rice's Theorem.)

3. Reduction II.

   Define the language L to be

   $$L = \left\{ M \;\middle|\; M \text{ is a } \mathsf{TM} \text{ and } 100 \leq |L(M)| \leq 200 \right\}.$$

   Show that L is undecidable by reducing A_{TM} to L. (Do the reduction directly. Do not use Rice's Theorem.)

4. Interleaving.

   Suppose that we have Turing machines M and M' which enumerate languages L and L', respectively.

   (a) Describe how to construct an enumerator P for the language $L \cup L'$. The code for P will need to make use of the code for M and M' (e.g. call it as a subroutine or run it in simulation using U_{TM}).

   (b) Suppose that the languages L and L' are infinite and suppose that M and M' enumerate their respective languages in lexicographic order. Explain how to modify your construction from part (a) so that the output of P is in lexicographic order.

5. Confusing but Interesting(?) Reduction. (bonus)

   Reduce L to A_{TM} (notice the different direction of reduction, in particular don't reduce A_{TM} to L):

   $$L = \{\langle M, w \rangle : \text{execution of } M \text{ with input } w \text{ never stops}\}$$

51.14 Homework 14: Dovetailing, etc.

Spring 08

1. Dovetailing

   (a) Briefly sketch an algorithm for enumerating all Turing machine encodings. Remember that each encoding is just a string, with some specific internal syntax (e.g. number of states, then number of symbols in Σ etc).

   (b) Now consider a language L that contains Turing machines which take other Turing machines as input. Specifically,

   $$L = \left\{ \langle M \rangle \;\middle|\; M \text{ halts on some input } \langle N \rangle \text{ where } N \text{ is a } \mathsf{TM} \right\}.$$

   As concrete examples of words in this language, consider the following TM M_1 and M_2.

If M_1 is a decider that checks if a given input $\langle X \rangle$ (that encodes a TM) halts in ≤ 37 steps, then $\langle M_1 \rangle$ is in L.

Suppose that M_2 halts and rejects if its input $\langle X \rangle$ is not the encoding of a TM, and M_2 spins off into an infinite loop if $\langle X \rangle$ is a TM encoding. Then, M_3 is not in L.

Show that L is (nevertheless) TM recognizable by giving a recognizer for it.

2. Language classification revisited.

Suppose that we have a set of Turing machine encodings defined by each of the following properties. That is, we have a set

$$L = \left\{ \langle M \rangle \;\middle|\; M \text{ is a } \mathsf{TM} \text{ and } M \text{ has property } P \right\},$$

and we are considering different ways to fill in P. Assume that the Turing machines M have only a single tape.

(a) P is "M accepts some word $w \in \Sigma^*$ which has $|w| \leq 58$".

(b) P is "M does not accept any word $w \in \Sigma^*$ which has $|w| \leq 249$".

(c) P is "M stops on some string w containing the character $\mathtt{a}$ in ≤ 37 steps."

(d) P is "M stops on some string $w \in \left\{ \mathtt{a}^n \mathtt{b}^n \;\middle|\; n \geq 0 \right\}$".

(e) Given some additional (fixed) TM M', the property P is "there is a word w such that both M and M' accepts it."

For each of these languages, determine whether it is Turing decidable, Turing recognizable, or not Turing recognizable. Briefly justify your answers.

3. LBAs emptiness revisited. (15 points)

Consider a TM M and a string w. Suppose that \$ and c are two fixed haracters that are not in M's tape alphabet Γ. Now define the following language

$$L_{M,w} = \left\{ z = w\$\$\$c^i \;\middle|\; i \geq 1 \text{ and } M \text{ accepts } w \text{ in at most } i \text{ steps} \right\}.$$

(a) Show that given M and w, the language $L_{M,w}$ can be decided by an LBA. That is, explain how to build a decider $D_{M,w}$ for $L_{M,w}$ that uses only the tape region where the input string z is written, and no additional space on the tape.

(b) M accepts w if and only if $L(D_{M,w}) \neq \emptyset$. Explain briefly why this is true.

(c) Assume that we can figure out how to compute the encoding $\langle D_{M,w} \rangle$, given $\langle M \rangle$ and w. Prove that the language

$$\mathsf{E}_{\mathsf{LBA}} = \left\{ \langle M \rangle \;\middle|\; M \text{ is a } \mathsf{LBA} \text{ and } L(M) = \emptyset \right\}.$$

is undecidable, using (a) and (b).

Bibliography

[EZ74] A. Ehrenfeucht and P. Zeiger. Complexity measures for regular expressions. In *Proc. 6th Annu. ACM Sympos. Theory Comput.*, pages 75–79, 1974. `http://portal.acm.org/citation.cfm?id=803886`.

[Sip05] M. Sipser. *Introduction to the Theory of Computation*. 2ed edition, February 2005.

Index