

# CSCI 161: Theory of Automata and Languages

## Lecture Notes

Sara Krehbiel  
Santa Clara University

September 2022

### Reading Guide

These lecture notes closely follow the flow of material discussed in class and also refer to the relevant definition and theorem numbers from the course textbook, *Introduction to the Theory of Computation*, 3rd ed., by Michael Sipser. These notes include unworked exercises and examples that we will discuss in class, so you may print a copy at the beginning of the quarter and add your own annotations in class. However, the text in this set of notes is dense, so it is not intended as a read-while-listening, scaffolded worksheet for class. Instead, it serves as a sequentially organized and comprehensive reference to review periodically after class – ideally after each class – to make sure you understood the material presented. Most students learn best when actively taking notes, so I recommend that you maintain your own comprehensive set of notes from lectures and refer to both your notes and these as you study. The explanations typed out here reflect my expectations for what you should understand from the course material, and you can use uncertainty about anything written in these notes as a starting point for questions in office hours.

# Contents

<b>0</b>	<b>Day 1: Course Overview</b>	<b>4</b>
0.1	Motivating Questions . . . . .	4
0.2	Perspective on Computational Tasks: TSP and Optimization vs Decision . . . . .	4
0.3	Equivalence of Optimization and Decision . . . . .	6
0.4	Decidability of TSP . . . . .	6
0.5	A Simpler Computational Task: Addition . . . . .	7
0.6	An Even Simpler Computational Task for Unit 1: Recognizing Strings Starting with ‘1’ . . .	7
0.7	Course Vocabulary . . . . .	8
<b>1</b>	<b>Unit 1: Regular Languages</b>	<b>9</b>
1.1	State Diagram Exercises . . . . .	9
1.2	Deterministic Finite Automata and Regular Languages . . . . .	10
1.3	Our First Proofs: Closure of the Regular Languages Under Union . . . . .	11
1.4	Nondeterministic Finite Automata . . . . .	12
1.5	Equivalence of DFAs and NFAs . . . . .	14
1.6	Closure of the Regular Languages Under the Regular Operations . . . . .	14
1.6.1	Additional Closure Properties for Regular Languages . . . . .	15
1.7	Regular Expressions . . . . .	16
1.8	Equivalence of Regular Expressions and FAs . . . . .	17
1.9	The Pumping Lemma for Regular Languages . . . . .	18
1.9.1	Pumpability of All Regular Languages . . . . .	18
1.9.2	Nonpumpability to Prove a Language is Not Regular . . . . .	19
1.9.3	Logic Pitfalls . . . . .	20
<b>2</b>	<b>Unit 2: Context-Free Languages</b>	<b>22</b>
2.1	Introduction to the Machinery for Context-Free Languages . . . . .	22
2.1.1	Pushdown Automata: FAs With Stacks . . . . .	22
2.1.2	Context-Free Grammars: Generative Analogues . . . . .	22
2.2	Context-Free Grammars . . . . .	23
2.3	CFG Interpretation and Design . . . . .	24
2.3.1	Design Pitfall . . . . .	25
2.4	Ambiguity . . . . .	25
2.5	Chomsky Normal Form . . . . .	27
2.6	The Cocke-Younger-Kasami Parsing Algorithm . . . . .	28
2.6.1	The Dynamic Programming Paradigm . . . . .	29
2.6.2	Pseudocode . . . . .	29
2.6.3	Runtime . . . . .	30
2.7	Pushdown Automata . . . . .	30
2.8	Equivalence of PDAs and CFGs . . . . .	31
2.9	The Pumping Lemma for Context-Free Languages . . . . .	33
2.9.1	Pumpability of All Context-Free Languages . . . . .	33
2.9.2	Nonpumpability to Prove a Language is Not Context Free . . . . .	35

<b>3</b>	<b>Unit 3: Decidability, Recognizability, and the Limits of Computation</b>	<b>37</b>
3.1	The Power of Memory . . . . .	37
3.2	Turing Machines . . . . .	38
3.3	Decidability, Recognizability, Corecognizability, and the Chomsky Hierarchy . . . . .	39
3.4	The DFA Acceptance Problem . . . . .	41
3.4.1	Encoding Machines as Strings . . . . .	42
3.4.2	Decidability of the DFA Acceptance Problem . . . . .	42
3.4.3	Vocabulary Pitfalls . . . . .	43
3.5	Decidability Arguments . . . . .	43
3.5.1	White-Box Reductions . . . . .	43
3.5.2	Decidability of the DFA Language Emptiness Problem . . . . .	44
3.5.3	Decidability of the DFA Equivalence Problem . . . . .	45
3.5.4	Decidability of Languages Related to Unit 2 Mechanics . . . . .	46
3.6	The Turing Machine Acceptance Problem . . . . .	47
3.6.1	The Universal Turing Machine . . . . .	47
3.6.2	Undecidability of the Turing Machine Acceptance Problem . . . . .	48
3.7	Undecidability Arguments . . . . .	49
3.7.1	Black-Box Reductions and Undecidability of the Halting Problem . . . . .	49
3.7.2	Undecidability of the TM Language Emptiness and Equivalence Problems . . . . .	50
3.8	Recognizers for Undecidable Languages . . . . .	51
3.9	Mapping Reductions . . . . .	52
3.9.1	Unrecognizability of the TM Equivalence Problem and its Complement . . . . .	52
3.10	Computation Histories and Linear Bounded Automata . . . . .	53

## 0 Day 1: Course Overview

We start the quarter with an overview of what the course is about, which is fundamentally two things:

- Representing computational problems as **languages**
- Precisely specifying **automata**, computers programmed for specific computational tasks

### 0.1 Motivating Questions

Theory of computation is a sub-discipline of computer science – the first! – driven by three questions:

1. What problems are impossible for computers to solve?
2. What is the relative hardness of problems computers can solve?
3. How do we define a computer?

The *automata* portion of this course answers the third question. We will have three main definitions of computers, which we'll build up from simplest to most complex across three units: finite automata, pushdown automata, and Turing machines. All are associated with rigorous mathematical descriptions, which you will learn to interpret and reasoning about. These will feel like toy machines until the end of the quarter; though the formal model of a Turing machine is not a good representation of how your laptop works, it is powerful enough to describe what your laptop can do!

We will *skip* the second question, which concerns (among other things) the huge open P vs NP problem, asking whether every problem whose answers can be checked efficiently can also be solved efficiently. Interested students should take CSCI 162 for a deeper explanation of this and related complexity problems!

Each automaton we study is associated with a *language* that describes the computational problem it solves. The different types of machines define different classes of problems that are possible for machines of that type to solve. We will cap each unit with an exploration of what our machine of the unit *cannot* do. Because Unit 3 focuses on models of real-life computers, this means that we spend the last third of the class understanding why some problems are impossible for *any* computer to solve.

### 0.2 Perspective on Computational Tasks: TSP and Optimization vs Decision

We often think about computational tasks as optimization problems. To help you shift your intuition from how we usually think about computation to how we will think about it in this class (as decision problems), we'll spend a few minutes discussing the *travelling salesperson problem* (TSP). This won't get any more discussion in this course beyond today, but it is a classic discrete optimization problem important in the fields of computational complexity and optimization – take CSCI 162 and MATH/CSCI 146-147 for more of this!

To understand the TSP problem statement, you have to remember from discrete math that a *graph* is a set of vertices, which may or may not have any physical meaning, and a set of edges, each connecting a pair of vertices. Edges may have weights, representing importance, distance, or other metrics of interest. A *complete weighted graph* is a graph in which each pair of vertices is connected by a weighted edge. A tour on a complete graph is a permutation of its vertices, representing a path that starts and ends at the same vertex, visiting each other vertex exactly once.

The setting for the TSP problem is that a salesperson has many towns in which they hope to sell their wares, and they must travel between towns, visiting each exactly once and returning to their starting point.

An *instance* of this problem is specified by a complete weighted graph. Vertices represent towns, and travel time between any two towns is given by the corresponding edge weight. If the goal is to minimize travel time, we can write the TSP *optimization* problem as follows:

- TSP (opt): Given a complete weighted graph, what is the minimum length of a tour of the graph?

This problem is captured by a **function**  $f_{\text{TSP}_o}$  that maps problem instances to problem solutions. Here a problem instance is an encoding of the “given” portion of the statement, and the solution is the numerical value representing the minimum length of a tour.

Assume that all numerical values are non-negative integers, because computers like discrete quantities. A special type of discrete quantity that computers like are values that can be represented by bits. We don’t want to get bogged down in the details, but be aware that computer architecture requires that we find a way of *encoding* any information as a **binary string**, or sequence of 0s and 1s. Let  $\langle G_w \rangle$  represent a binary encoding of the  $\binom{n}{2}$  weights that constitute an instance  $G_w$  of the TSP optimization problem. Then the codomain of our function should be a non-negative integer representing the length of the shortest tour of  $G_w$ .

So we could describe an example problem instance and corresponding solution as:




$$f_{\text{TSP}_o}(\langle \text{graph} \rangle) = 140$$

Optimization problems also have a decision variant, in which we bundle a candidate answer with the question and ask the computer to check whether the answer is right. The corresponding TSP *decision* problem can be written as follows:

- TSP (dec): Given a complete weighted graph and target length  $T$ , is there a tour of length at most  $T$ ?

This problem is also captured by a function  $f_{\text{TSP}_d}$ , but here the target is part of the instance and the solution is just a ‘yes’ or ‘no’ answer. Some example problem instances and solutions are as follows:



$$f_{\text{TSP}_d}(\langle \text{graph}, 100 \rangle) = \text{N} \quad f_{\text{TSP}_d}(\langle \text{graph}, 140 \rangle) = \text{Y} \quad f_{\text{TSP}_d}(\langle \text{graph}, 180 \rangle) = \text{Y}$$

Because the output of a function for a decision problem is binary, we can describe this problem not as a function but as a set of inputs for the function with a Y answer. The TSP **language** then becomes

$$L_{\text{TSP}} = \{ \langle G_w, T \rangle \mid f_{\text{TSP}_d}(\langle G_w, T \rangle) = \text{Y} \}$$

### 0.3 Equivalence of Optimization and Decision

It's simpler to formalize automata that *recognize languages* rather than *compute outputs of arbitrary functions*, but first we should convince ourselves that this doesn't leave out important computational problems. We'll now argue that computing  $f_{TSP_o}$  outputs requires no more or less complicated machinery than computing  $f_{TSP_d}$  outputs, i.e., recognizing strings in  $L_{TSP}$ . This extends to optimization vs decision in general.

You may have intuition that decision is simpler than optimization. This is not quite right, but the heart of it can be formalized by showing that the decision problem *reduces* to the optimization problem, meaning that *if* there were a way to program a computer  $TSP_o$  to solve the optimization problem, then it could be used to program a computer  $TSP_d$  to solve the decision problem. For now, we describe an automaton with *pseudocode*, which corresponds to an *algorithm* or *Turing machine*, formalized in Unit 3:

```
 $TSP_d(G_w, T)$ :  
if  $TSP_o(G_w) \leq T$ : return Y  
  
else: return N
```

This algorithm is called a *black-box reduction* because it establishes an implication without needing to know whether the hypothesis is correct. Specifically, without knowing whether the optimization problem is solvable, we now know that if it is, then we can solve the decision problem.

Our first reduction does *not* show that decision is *simpler* than optimization – just that it is *no more complex*. Surprisingly, it's also no simpler! That's because the optimization problem reduces to the decision problem. To solve the optimization problem given access to an algorithm for the decision problem, simply try all targets for a given set of weights starting from 0 until you get a Y answer. In pseudocode:

```
 $TSP_o(G_w)$ :  
for each  $T = 0, 1, 2, \dots$ :  
    if  $TSP_d(G_w, T) == Y$ : return  $T$ 
```

Although this reduction requires many calls to  $TSP_d$  to solve  $TSP_o$ , remember that we do not care about efficiency in this class – we care whether a computer can solve the problem.

However, we still haven't shown that a computer can solve either problem. Rather we've shown that if  $TSP_d$  exists, then  $TSP_o$  exists, and if  $TSP_o$  exists, then  $TSP_d$  exists. In the vocabulary of Unit 3,

$f_{TSP_o}$  is a *computable function* if and only if  $L_{TSP}$  is a *decidable language*.

More generally, an optimization problem can be solved by a computer if and only if a decision variant of that problem can be solved by a computer that produces no output other than indicating acceptance of its input. Henceforth, all the automata we study will produce no output other than 'accept' or (maybe) 'do not accept,' and every such automata will have a language it recognizes, consisting of the set of strings it accepts.

### 0.4 Decidability of TSP

We have spent enough time discussing TSP that we should go the one extra step to classify  $L_{TSP}$  as a decidable language, even though we'll wait until Unit 3 to focus on decidability. First, to show that the optimization problem is computable by a Turing machine, we give an algorithm for it:

```

TSPo(Gw):
best = ∞

for each permutation t of cities:
    calculate len = sum of edge lengths corresponding to tour t
    if len < best: best = len

return best

```

This brute force algorithm is not efficient, but it serves our purposes. Moreover, now that we have pseudocode for the subroutine in our first reduction, that implies a well-specified Turing machine that establishes computability of the decision problem as well.

## 0.5 A Simpler Computational Task: Addition

Sometimes there is only one feasible answer, and the decision version is to just check an answer-solution combination to see whether the solution is right. For example, addition is an intuitive computational task with one right answer that can also be converted into a decision problem:

- The addition problem: Given two numbers  $a$  and  $b$ , what is the sum of  $a$  and  $b$ ?
- The addition decision problem: Given an equation  $a + b = c$ , is  $c$  the sum of  $a$  and  $b$ ?

In Unit 2, we'll see that a type of machine called a *pushdown automaton* (PDA) can recognize the language corresponding to the (unary) addition decision problem, which uses 0s to block up the 1s, with the first 0 representing + and the second representing =, with the number of 1s per block denote the values in the equation:

$$L_{\text{add}} = \{1^a 0 1^b 0 1^{a+b} \mid a, b \geq 0\}$$

In Unit 3 we see that any task we could do with a pushdown automaton can be decided by a Turing machine, and because of the optimization/decision equivalence, this means that a Turing machine that can check  $a + b = c$  could be used to build one that computes  $c$  given  $a$  and  $b$ , so in Unit 2 we'll be able to focus on the seemingly easier problem of checking  $a + b = c$ . All that is to say that computers can do addition!

If you're still thinking that decision problems should be easier in general than optimization, that's good intuition! You are conjecturing that P is *not* equal to NP. Most computer scientists agree with you, although the ones that spend their lives focusing on this problem don't expect it to be answered any time soon. However, we are not focused on how easy or hard a problem is, but rather whether it is possible or not, and today we've argued that decision and optimization problems are equivalently solvable by Turing machines.

## 0.6 An Even Simpler Computational Task for Unit 1: Recognizing Strings Starting with '1'

We now define the language of binary strings that start with 1 as follows:

$$L = \{w \in \{0, 1\}^* \mid |w| > 0 \text{ and } w_1 = 1\}$$

Our first concrete Unit 1 task is to build a *deterministic finite automaton* (DFA) that *accepts* any binary string that starts with 1 and does not accept any others. Before building intuition about DFAs to tackle this problem, you should pause to make sure you're keeping all the vocabulary straight.

## 0.7 Course Vocabulary

In order to absorb concepts and then understand and answer questions, it will be essential for you to quickly develop fluency with the vocabulary we've just introduced, summarized below.

- A **string** is a sequence of characters from some character alphabet, often  $\Sigma = \{0, 1\}$ .

Examples: 1101; 11010111; binary encodings of TSP instances.

Notes: Indices in this class start from 1, so  $w_1$  denotes the first character of string  $w$  of length  $|w| \geq 1$  and  $w_{|w|}$  denotes its last character. Indices are positive integers; characters like '0' and '1' in a string should be thought of symbolically rather than numerically. The empty string, denoted  $\varepsilon$ , has length 0.

- A **language** is a set of strings  $L \subseteq \Sigma^*$ , usually sharing some property.

Examples: all binary strings beginning with 1; all valid unary addition equations; all encodings of complete weighted graphs with achievable tour lengths.

Notes: A language is the set of all strings with  $\Upsilon$  answers for an associated decision problem.

- A **function** is a mapping of elements from one set to elements of another.

Examples:

$$f_{\text{TSP}_o}(\langle G_w \rangle) = \text{minimum length of a tour of } G_w;$$

$$f_{\text{TSP}_d}(\langle G_w, T \rangle) = \Upsilon \text{ if } T \text{ is achievable in } G_w \text{ or } \text{N otherwise};$$

$$f(w) = \Upsilon \text{ if } |w| > 0 \text{ and } w_{|w|} = 1 \text{ or } \text{N otherwise}.$$

Notes: A language  $L$  corresponds to a function  $f : \Sigma^* \rightarrow \{\Upsilon, \text{N}\}$  with  $f(w) = \Upsilon$  iff  $w \in L$ .

- An **automaton** is a machine programmed to follow a sequence of steps depending on its *input* to arrive at a corresponding *output*.

Examples: In Unit 3, we will use *pseudocode* to describe Turing machines for solving particular problems, like the TSP examples from today. Before that, we will use *state diagrams* to describe pushdown automata (Unit 2) and deterministic finite automata (Unit 1), coming up next.

Notes: An automaton represents an *algorithm* for some function  $f$  that on input  $w$  calculates  $f(w)$  according to the computational model of the machine. By focusing on decision problems, the only output of our machines will be an acceptance (or not) of an input string.

Tying it all together, a computational task can be interpreted as a function. A function with binary outputs can define a language consisting of the strings that the function maps to  $\Upsilon$ . A sequence of characters make up a string, whereas a set of strings make up a language. While binary functions and machines (automata) both define a language, a function defines a language directly whereas a machine corresponds to an algorithm showing how to arrive at the output (acceptance or not) corresponding to a function's input (a string). The complexity of a language depends on the type of machines that can or cannot recognize exactly the set of strings in the language; individual strings have no inherent associated complexity.



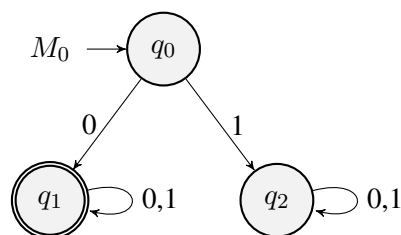
# 1 Unit 1: Regular Languages

We start with the language of binary strings that start with 0. In set-builder notation:

$$L_0 = \{w \in \{0, 1\}^* \mid |w| \geq 1, w_1 = 0\}$$

Star (\*) operates on a set of strings  $L$  by concatenating any number of  $L$ 's elements (maybe with repetition), i.e.,  $L^* = \{w_1 \dots w_k \mid k \geq 0, w_i \in L \forall i = 1, \dots, k\}$ , so  $\{0, 1\}^*$  is the universe of all binary strings.

The language  $L$  defines a computational problem: Given a binary string, does it start with 0? To solve the problem, we must build a machine that specifies how to recognize exactly the set of strings in the language. Without knowing exactly the computation rules, we can start with an intuitive *state diagram* that shows how to digest a string character-by-character before determining whether to accept it as part of the language:



This state diagram represents a **deterministic finite automaton** (DFA). All DFAs share several features:

1. There are a finite number of circles in the state diagram, representing *states*.
2. Arrows are labeled with 0 or 1, representing the *alphabet* of the input the DFA must be able to process.
3. Each state has an arrow leaving it with a 0 and an arrow leaving it with a 1, representing the *transition function* that guides how the DFA changes state on any given input.
4. One state has an unlabeled arrow pointing to it, representing the *start state* of the machine.
5. States can be designated with concentric circles, representing the *final or accepting states*, determining which input strings will be accepted as elements of the language.

We will turn these features into a formal mathematical definition after building our intuition through some more examples.

## 1.1 State Diagram Exercises

Four computational tasks are given in English below. For each task, specify the language that corresponds to the task as a precisely written set of strings, and give a state diagram representing a DFA that solves the task by recognizing the respective language. The alphabet for each language is  $\Sigma = \{0, 1\}$ , so each machines should be able to correctly identify any binary string as either in its corresponding language or not.

1. Recognize binary strings with no 1s.
2. Recognize binary strings that end with 1.
3. Recognize binary strings that have no 1 or end with 1.
4. Recognize binary strings of length three that have a 0 in the middle.

## 1.2 Deterministic Finite Automata and Regular Languages

With some practice thinking of computational problems as languages described as sets of strings and with designing state diagrams that represent DFAs that recognize them, we are ready to formalize our intuition about DFAs as rigorous mathematical objects. The book's definition establishes the five components that specify a DFA, and in text it describes the underlying computational model that determines which strings are accepted by a particular DFA. Because there's no sense in having DFAs without thinking about their languages, our definition bundles the "what" with the "how."

**Definition 1.5.** A **deterministic finite automaton** is a 5-tuple  $M = (Q, \Sigma, \delta, s, F)$ , where

$Q$  is a finite set of objects called states,

$\Sigma$  is a finite set of objects called characters, and  $\Sigma$  is called the alphabet,

$\delta : Q \times \Sigma \rightarrow Q$  is a function called the transition function,

$s \in Q$  is called the start state, and

$F \subseteq Q$  is called the set of final or accepting states.

For any input string  $w \in \Sigma^n, n \geq 0$ , the **computation history** of DFA  $M$  running on  $w$  is a sequence of  $n + 1$  states  $r_0, r_1, \dots, r_n \in Q$  such that  $r_0 = s$  and  $\delta(r_i, w_{i+1}) = r_{i+1}$  for all  $i = 0, \dots, n - 1$ . We say  $M$  **accepts**  $w$  if additionally  $r_n \in F$ .

The **language recognized by DFA  $M$** , denoted  $L(M)$ , is the set of all strings that  $M$  accepts.

Our first state diagram can be described formally as  $M_0 = (Q, \Sigma, \delta, s, F)$  with  $Q = \{q_0, q_1, q_2\}$ ,  $\Sigma = \{0, 1\}$ ,  $s = q_0$ ,  $F = \{q_1\}$ , and  $\delta$  as given by the following transition table:

	0	1
$q_0$	$q_1$	$q_2$
$q_1$	$q_1$	$q_1$
$q_2$	$q_2$	$q_2$

The ultimate goal of this course is to develop enough facility designing different types of automata to be able to *classify* languages in the Chomsky hierarchy. This hierarchy establishes the relative complexity of different languages with respect to what types of machines are able to recognize them. We will build up in complexity as the quarter progresses. Unit 1 focuses only on the simplest class: the **regular languages**.

**Definition 1.16.** A set of strings is a **regular language** if and only if it is recognized by some DFA.

Logically what this means is that in order to prove that a language is regular, it suffices to provide a DFA that accepts all strings in the language and no strings not in the language. Conversely, to prove that a language is *not* regular, it suffices to prove that no DFA could possibly recognize the language in question. This latter task is harder – it is usually easier to show that something exists than to show that it doesn't! – and later in the unit we will develop additional tools to help us prove nonregularity.

**Exercise:** Give a 5-tuple that formally describes a DFA that recognizes the (regular) language of binary strings that end with 1. What is its computation history on 011?

### 1.3 Our First Proofs: Closure of the Regular Languages Under Union

Technically, the DFAs we've seen so far constitute five proofs that those particular languages are regular. Now we prove more general properties of the entire *class* of regular languages: closure under the complement and union operations.

An operation is simply a function applied to one or more elements of a set  $S$  – one for unary operations and two for binary operations. A subset  $R \subseteq S$  is *closed* under some  $n$ -ary operation  $f$  if for any elements  $e_1, \dots, e_n \in R$ , we have  $f(e_1, \dots, e_n) \in R$ .

For languages  $L_1, L_2 \subseteq \Sigma^*$ , complement is the unary operation  $\bar{L}_1 := \Sigma^* \setminus L_1 = \{w \in \Sigma^* \mid w \notin L_1\}$  and union is the binary operation  $L_1 \cup L_2 := \{w \in \Sigma^* \mid w \in L_1 \text{ or } w \in L_2\}$ .

Closure of the regular languages under complement has a 1-sentence proof: If  $A$  is a regular language, then it is recognized by some DFA  $D = (Q, \Sigma, \delta, s, F)$ , and DFA  $D' = (Q, \Sigma, \delta, s, Q \setminus F)$  recognizes  $\bar{A}$ , which therefore must be regular.

The logic of this 1-sentence argument mirrors that of the more involved proof of closure under union:

**Theorem 1.25.** The class of regular languages is closed under union.

Before proving the result, let's translate the statement to make it more concrete. Closure of the class of regular languages under union means that if  $A_1$  and  $A_2$  are any two arbitrary regular languages, then their union  $A_1 \cup A_2$  must be a regular language. To establish this implication, we will give a *constructive proof*, which is one of the cornerstones of the logical reasoning we will work to master in this class. The idea of a constructive proof is as follows:

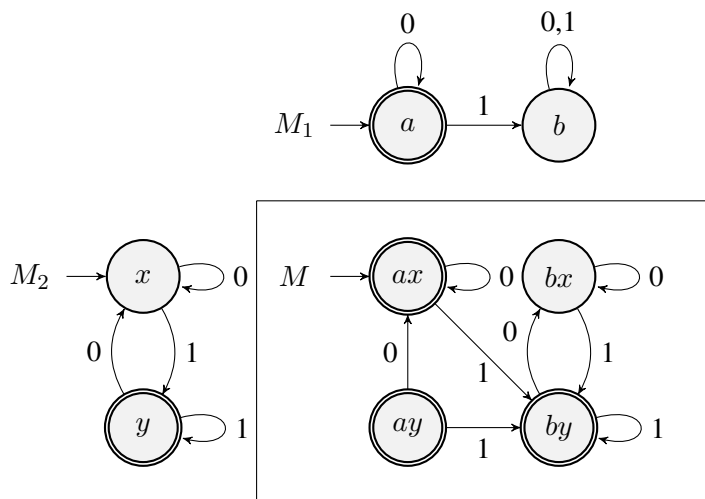
1. Assume the hypothesis of the conditional theorem statement, that  $A_1, A_2$  are regular.
2. Apply the definition of regularity and give variable names for hypothetical objects that must exist.
3. *Construct* a useful object based on your hypothetical objects.
4. Apply the definition of regularity again in the opposite direction to your constructed object.
5. Conclude the conclusion of the conditional theorem statement, that  $A_1 \cup A_2$  is regular.

*Proof of Theorem 1.25.* Assume  $A_1, A_2$  are arbitrary regular languages with the same alphabet. By the definition of regularity, there must exist DFAs  $M_1 = (Q_1, \Sigma, \delta_1, s_1, F_1), M_2 = (Q_2, \Sigma, \delta_2, s_2, F_2)$  with  $L(M_1) = A_1, L(M_2) = A_2$ . Construct a third DFA  $M = (Q, \Sigma, \delta, s, F)$  with  $Q, \delta, s, F$  as follows:

- $Q = Q_1 \times Q_2$ ,
- $s = (s_1, s_2)$ ,
- $F = \{(q_1, q_2) \in Q \mid q_1 \in F_1 \text{ or } q_2 \in F_2\}$ , and
- $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$  for  $(q_1, q_2) \in Q, a \in \Sigma$ .

By construction,  $Q$  keeps track of where both  $M_1$  and  $M_2$  are, with  $\delta$  simultaneously simulating transitions in each machine. Any string accepted by either  $M_1$  or  $M_2$  will be accepted by  $M$ , and any string not accepted by either will not be accepted by  $M$ , so the machine  $M$  recognizes the union of the languages of machines  $M_1$  and  $M_2$ . Because  $L(M) = A_1 \cup A_2$ , the DFA  $M$  establishes the regularity of the language  $A_1 \cup A_2$ .  $\square$

**Example:** To see this general construction applied to a particular pair of languages, recall our DFAs  $M_1$  and  $M_2$  for the languages of binary strings with no 1s and binary strings that end with 1, respectively. Combining their state spaces and transition functions following the proof of Theorem 1.25 yields DFA  $M$  as follows:

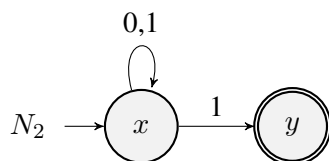


Note that state  $ay$  in  $M$  is *unreachable* from the start state, meaning that no sequence of characters will result in a computation history that visits that state. That's because the meaning of that state is that there are no 1s and the string ends with 1, which is not simultaneously possible! However, if it were, such a string would be in the union language, so it is correct to mark it as an accepting state.

## 1.4 Nondeterministic Finite Automata

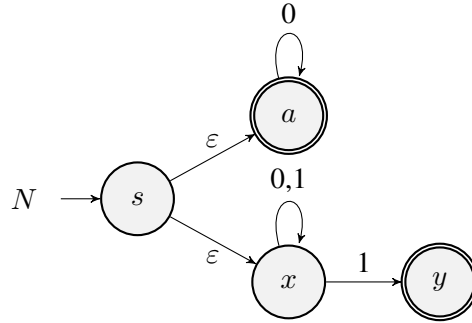
The automata we've seen so far are called *deterministic* because a string determines a unique computation history. We will now add optionality to our finite state machines by allowing *nondeterminism* in two ways: first, a **nondeterministic finite automaton** (NFA) may have multiple options for where it transitions on a given input character, and second, it may make transitions without digesting any input characters at all.

The first type of optionality is particularly useful when you want to transition from looking for something at the beginning of a string to looking for something at the end of a string without knowing when the end is coming. DFA  $M_2$  recognizes binary strings that end with 1, but it involves bouncing back and forth between states. With nondeterminism, we can process this in a more linear fashion:



Note that  $N_2$  can transition from  $x$  to  $y$  when it sees a 1, or it can idle at  $a$ . For a *nondeterministic finite automaton* (NFA) to accept a string, it only needs a single accepting computation history. For example,  $N_2$  has accepting computation history  $(x, x, x, y)$  on string 101, so  $101 \in L(N_1)$  even though  $N_1$  *could* transition to  $y$  right away and then get stuck. At this point, the machine would get stuck, because there are no transition instructions for 0 starting at state  $y$ . We can think of NFAs as testing all possible histories in parallel or always correctly guessing an accepting one if it exists. In either case, NFAs are more of a thought experiment than a real model of computation.

One way to make use of the second new option, to make a transition without digesting input, is an alternate way to construct machines to recognize the union of the languages of two other machines. Recall that Theorem 1.25 provides a way to combine two machines to make a machine that effectively simulates both simultaneously to see whether either one accepts. If we instead allow a machine to choose which smaller machine to run without looking at input, we can build an NFA that always has a way of accepting any string in the union of the two languages, such as strings with no 1s or strings that end with 1.



Recall that we use  $\varepsilon$  to denote the empty string, the string with zero characters. For transition labels on NFA state diagrams, we overload that notation, using  $\varepsilon$  to also denote a non-character.

We now present the formal definition of NFAs and their computation model that defines their languages. It uses two new pieces of notation:  $\Sigma_\varepsilon$  denotes the set  $\Sigma \cup \{\varepsilon\}$ , which is the alphabet augmented with the empty character, and  $\mathcal{P}(S)$  denotes the power set (or set of all subsets) of any set  $S$ .

**Definition 1.37.** A **nondeterministic finite automaton** is a 5-tuple  $N = (Q, \Sigma, \delta, s, F)$ , where

$Q$  is a finite set of objects called states,

$\Sigma$  is a finite set of objects called characters, and  $\Sigma$  is called the alphabet,

$\delta : Q \times \Sigma_\varepsilon \rightarrow \mathcal{P}(Q)$  is a function called the transition function,

$s \in Q$  is called the start state, and

$F \subseteq Q$  is called the set of final or accepting states.

For any input string  $w \in \Sigma^*$ , an **accepting computation history** of NFA  $N$  running on  $w$  is a sequence  $r_0, r_1, \dots, r_m \in Q$  such that  $r_0 = s$ , there exist  $y_1, \dots, y_m \in \Sigma_\varepsilon$  with  $w = y_1 \dots y_m$  and  $r_{i+1} \in \delta(r_i, y_{i+1})$  for all  $i = 0, \dots, m-1$ , and  $r_n \in F$ . We say  $N$  **accepts**  $w$  if it has an accepting computation history on  $w$ .

The **language recognized by NFA  $N$** , denoted  $L(N)$ , is the set of all strings  $w$  that  $N$  accepts.

Examine the following transition table for the union NFA  $N$  above to see how the nondeterministic  $\delta$  can be formalized mathematically:

	0	1	$\varepsilon$
$s$	$\emptyset$	$\emptyset$	$\{a, x\}$
$a$	$\{a\}$	$\emptyset$	$\emptyset$
$x$	$\{x\}$	$\{x, y\}$	$\emptyset$
$y$	$\emptyset$	$\emptyset$	$\emptyset$

## 1.5 Equivalence of DFAs and NFAs

Should we now define a new class of languages with respect to NFAs, the way we defined regular languages to be those associated with DFAs? Instead we will argue that we don't have to by proving that all languages recognized by a DFA are recognized by an NFA and vice versa, i.e., DFAs and NFAs are equivalently powerful models of computation that each define the class of regular languages.

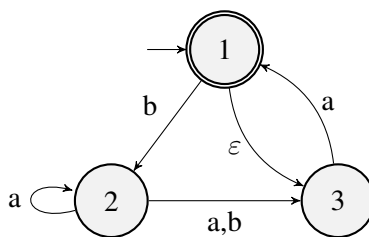
This is a two-part proof. First, we must show that for every DFA there is an NFA that recognizes the same language. This is easy, because an NFA can always opt to not make use of nondeterminism. In fact, any DFA state diagram can be interpreted as an NFA state diagram. Keep the state set, alphabet, start state, and final states the same, and modify the transition table to adjust to the different requirements for NFA transition functions: simply put braces around every entry in the table to interpret the outputs of the function as singleton sets rather than single elements, and add an  $\varepsilon$  column whose entries are all  $\emptyset$ .

Second, we must show that for every NFA there is a DFA that recognizes the same language. This should seem surprising, because NFAs can explore multiple options simultaneously whereas DFAs have a single computation history for each input. Our constructive proof of this fact will reveal how we can build a DFA with enough states to keep track of every possible thing that could be going on in an NFA.

**Theorem 1.39.** Every NFA has an equivalent DFA.

*Proof.* Let  $N = (Q, \Sigma, \delta, s, F)$  be an arbitrary NFA. We construct a DFA  $M = (Q', \Sigma, \delta', s', F')$  to keep track of all possible states the NFA could be in before digesting each input character. Let  $Q' = \mathcal{P}(Q)$  and  $F' = \{R \in Q' \mid R \cap F \neq \emptyset\}$ . The DFA's  $\delta'$  cannot digest  $\varepsilon$ , so we incorporate all NFA  $\varepsilon$  transitions as a second phase of every  $\delta'$  transition on a real input character. For any  $R \in Q'$ , define  $E(R)$  to be the set of states of  $N$  reachable from some state in  $R$  on  $\varepsilon$ -transitions alone. Then for any  $R \in Q'$  and  $a \in \Sigma$ , we define  $\delta'(R, a) = \bigcup_{q \in R} E(\delta(q, a))$ . Because every  $\delta'$  transition represents first digesting a real input character and then 0 or more  $\varepsilon$ -transitions, our DFA should start after permitting the NFA to take  $\varepsilon$ -transitions. Hence, we define  $s' = E(\{s\})$ . By construction,  $L(M) = L(N)$ , establishing our result.  $\square$

**Example 1.41:** The text illustrates how to apply this construction to the particular DFA below.



The general result shows that nondeterminism does not add power to FAs, so regularity can be certified by NFAs as well as DFAs. This fact will be useful in our next task.

## 1.6 Closure of the Regular Languages Under the Regular Operations

Recall that Theorem 1.25 shows that we can take two regular languages and union them together to get a third regular language. Union is one of the three regular operations, the other two being concatenation and star, defined as follows:

**Definition 1.23.** For languages  $A$  and  $B$ , the regular operations are union, concatenation, and star, defined respectively as follows:

$$A \cup B = \{w \mid w \in A \text{ or } w \in B\},$$

$$A \circ B = \{w_1 w_2 \mid w_1 \in A, w_2 \in B\}, \text{ and}$$

$$A^* = \{w_1 \dots w_k \mid k \geq 0, w_i \in A \forall i = 1, \dots, k\}.$$

Theorems 1.45, 1.47, and 1.49 show that the regular languages are closed under the regular operations. We will not formally prove these closure theorems, but we overview the constructions in their proofs for the NFA design strategies they exhibit.

The statement of Theorem 1.45 is identical to that of Theorem 1.25, but instead of building a DFA to recognize the union of two DFA's languages, its proof builds an NFA and appeals to Theorem 1.39, which implies that an NFA is sufficient to conclude regularity. NFA  $N$  from our initial discussion of NFAs shows how to add a start state and two  $\varepsilon$  branches to the start state of other FAs to build an FA that recognizes the union of the other FAs' languages. This captures the idea of the general proof.

To recognize concatenation instead of union, Theorem 1.47 constructs an NFA that processes the underlying FAs in sequence rather than parallel, using nondeterminism in the middle to decide when to transition from the first FA to the second, rather than at the beginning to decide which single FA to execute.

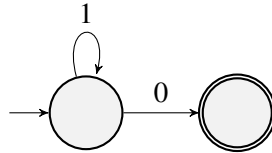
The proof of Theorem 1.49 only has one underlying NFA because star is a unary operation. Link all accepting states to the start state with  $\varepsilon$ -transitions so you can repeatedly loop through the original NFA. Create a new accepting start state,  $\varepsilon$ -linked to the original start, to ensure  $\varepsilon$  is in the new language.

These proof sketches have three key takeaways:

1. Branching introduces options, as in the case of union.
2. Running machines in series enforces order, as in the case of concatenation.
3. Creating loops enables repetition, as in the case of star.

**Exercise:** Design an NFA for strings with an even number of 1s after an occurrence of the substring 01. Is 01 in the language? Is  $\varepsilon$ ? Give an accepting computation history of your NFA on the string 10101011.

**Thought experiment:** Why must we create a new accepting start state instead of making the original start state accepting? Note that this sometimes accepts some strings not in the star language, such as 1 for the DFA below recognizing  $L = \{1^n 0 \mid n \geq 0\}$ , where  $L^* = \{\varepsilon\} \cup \{w0 \mid w \in \Sigma^*\}$ :



### 1.6.1 Additional Closure Properties for Regular Languages

Your homework has you adapt our proof of Theorem 1.25 to additionally show closure under *intersection*.

The *complement* of a language is the set of strings not in the language, i.e.,  $\bar{L} = \{w \in \Sigma^* \mid w \notin L\}$ . We can observe that regular languages are closed under complement with a 1-line constructive proof. Given DFA  $M = (Q, \Sigma, \delta, s, F)$ , the DFA  $M' = (Q, \Sigma, \delta, s, Q \setminus F)$  is such that  $L(M') = \bar{L}(M)$ .

## 1.7 Regular Expressions

Having explored two equivalently powerful types of finite state machines associated with regular languages, we turn to a third type of machinery associated with regular languages: **regular expressions** (regexes).

The proofs of our closure properties instruct how to string simple NFAs together to create regular languages build by combining other regular languages together with regular operations, and this is at the core of how one constructs and reasons about regular expressions. However, while an FA can be seen as a *machine* in that it provides an algorithm that accepts an input string, does some computation, and produces binary output (by way of accepting or rejecting the input string), a regular expression does not have input, rather it is a specially formatted string consisting of alphabet characters combined with special characters  $\emptyset, \varepsilon, \cup, \circ, *$ , and parentheses, specifying a *pattern* shared by all strings in its language. We start with some examples:

Language	A regex for the language	A simplified regex
$\emptyset$ , the empty set of strings	$\emptyset$	$\emptyset$
$\Sigma^*$ , the set of all binary strings	$((0 \cup 1)^*)$	$\Sigma^*$
$L_1$ , binary strings with no 1s	$(0^*)$	$0^*$
$L_2$ , binary strings that end with 1	$((0 \cup 1)^* \circ 1)$	$\Sigma^*1$
$L_1 \cup L_2$	$((0^*) \cup (((0 \cup 1)^*) \circ 1))$	$0^* \cup \Sigma^*1$

There are three types of atomic regex symbols indicating very simple languages that can be combined using the regular operations to generate more complex languages:

- $\emptyset$  corresponds to the language  $\emptyset$  with no strings,
- $\varepsilon$  corresponds to the language  $\{\varepsilon\}$  that consists only of the empty string, and
- $a$  for any  $a \in \Sigma$  corresponds to the language  $\{a\}$  that consists only of a single 1-character string  $a$ .

We can link arbitrary regexes together with regular operation characters to build complex regexes. The technical regex definition requires Technically, we must add parentheses around every operation to specify the hierarchy of operations, but in practice we recognize a standard precedence order that allows us to drop some parentheses, as in our example  $0^* \cup \Sigma^*1$ . To keep track of both the precedence order and some rough arithmetic analogies, we compare the three regular expressions to their arithmetic counterparts in increasing order of precedence:

Regular Op	Example	Language	Identity	Arithmetic Op	Example	Value	Identity
$\cup$	$(0 \cup 1)$	$\{0, 1\}$	$\emptyset \cup R = R$	Addition	$2 + 3$	5	$0 + x = x$
$\circ$	$(0 \circ 1)$	$\{0, 1\}$	$\varepsilon \circ R = R$	Multiplication	$2 \cdot 3$	6	$1 \cdot x = x$
$*$	$(0^*)$	$\{0^n \mid n \geq 0\}$	$\varepsilon^* = \varepsilon$	Exponentiation	$2^3$	8	$1^x = 1$

In the same way that 0 is the additive identity because  $0 + x = x$ , you can think of  $\emptyset$  as the union identity in that the language of the regex  $\emptyset \cup R$  is just the language of  $R$ . Similarly, 1 is the multiplicative identity because  $1 \cdot x = x$  and  $\varepsilon$  is the concatenation identity in that the language of the regex  $\varepsilon \circ R$  is the language of  $R$ . The analogy between exponentiation and star is a little different, because  $R^* = R^0 \cup R^1 \cup R^2 \cup \dots$ , but thinking of  $\emptyset$  and  $\varepsilon$  as identities helps to reason about star languages.  $R^0 = \varepsilon$  in the same way  $x^0 = 1$ , so  $\varepsilon$  is in the language of  $R^*$  for any regex  $R$ . This is even true for  $R = \emptyset$ ; the language of  $\emptyset^*$  is  $\{\varepsilon\}$  in the same way that  $0^0 + 0^1 + 0^2 + \dots = 1$ .

So why can we simplify  $((0^*) \cup (((0 \cup 1)^*) \circ 1))$  to  $0^* \cup \Sigma^*1$ ? For the same reason that we can write  $((a^x) + (((a + b)^y) \cdot b))) = a^x + (a + b)^y b$ .



We are now ready for the formal definition. As with our FAs, we bundle the definition of a regular expression (the specially formatted string itself) with specification of the language it represents.

**Definition 1.52.** A regular expression over alphabet  $\Sigma$  is a string of one of the following forms:

- $\emptyset$ ,
- $\varepsilon$ ,
- $a$  for some  $a \in \Sigma$ ,
- $(R_1 \cup R_2)$  for regular expressions  $R_1, R_2$ ,
- $(R_1 \circ R_2)$  for regular expressions  $R_1, R_2$ , or
- $(R_1^*)$  for regular expression  $R_1$ .

The language of regular expression  $R$ , denoted  $L(R)$ , is given by

$$L(R) = \begin{cases} \emptyset & \text{if } R = \emptyset \\ \{\varepsilon\} & \text{if } R = \varepsilon \\ \{a\} & \text{if } R = a \text{ for some } a \in \Sigma \\ L(R_1) \cup L(R_2) & \text{if } R = (R_1 \cup R_2) \text{ for regexes } R_1, R_2 \\ L(R_1) \circ L(R_2) & \text{if } R = (R_1 \circ R_2) \text{ for regexes } R_1, R_2 \\ L(R_1)^* & \text{if } R = (R_1)^* \text{ for regex } R_1 \end{cases}$$

We often use a regex to mean its language. For example, I could ask you to design an NFA for  $(ab \cup c)^*$ , which means give an NFA that recognizes the language of  $(ab \cup c)^*$ , i.e., an NFA  $N$  with  $L(N) = L((ab \cup c)^*)$ .

## 1.8 Equivalence of Regular Expressions and FAs

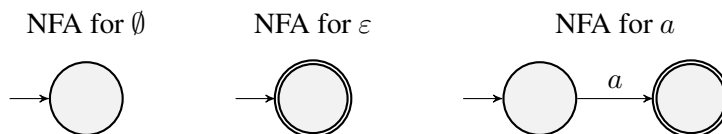
Based on the name alone, you might infer that there is a strong connection between regular languages and regular expressions, and you would be right! This connection is formalized by two more equivalence results in the text, which together constitute a proof that DFAs, NFAs, and regexes are all equally powerful and define the class of regular languages:

**Lemma 1.55.** Every regex has an equivalent NFA.

**Lemma 1.60.** Every DFA has an equivalent regex.

We will handle these results like we handled closure under concatenation and star: rather than write the proofs formally in full generality, we will sketch them to help build our intuition for how to convert between equivalently powerful machinery associated with languages.

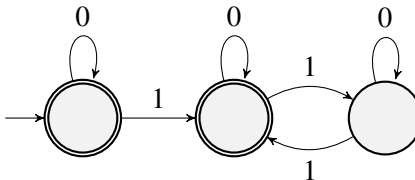
The proof of Lemma 1.55 simply applies the constructions of Theorems 1.45, 1.47, and 1.49 inductively, noting that every regex can be boiled down to atoms, and we have an NFA for each type of atom:



**Exercise:** Give an NFA equivalent to  $(ab \cup c)^*$ .

The proof of Lemma 1.60 is more technical and requires an special-purpose type of machinery called a generalized NFA (GNFA) that we'll only explain through example. Feel free to use GNFA's in your scratch work for homework and exams, but don't mistake them as valid NFAs!

**Exercise:** Give a regex equivalent to the DFA below:



## 1.9 The Pumping Lemma for Regular Languages

Because of the equivalent power of DFAs, NFAs, and regular expressions, we now have three ways to prove that a language  $L$  is regular:

- If there exists a DFA  $M$  with  $L(M) = L$ , then  $L$  is regular.
- If there exists an NFA  $N$  with  $L(N) = L$ , then  $L$  is regular.
- If there exists a regex  $R$  with  $L(R) = L$ , then  $L$  is regular.

Our options so far for proving *nonregularity* are less concrete:

- If there does not exist an FA that recognizes  $L$ , then  $L$  is not regular.

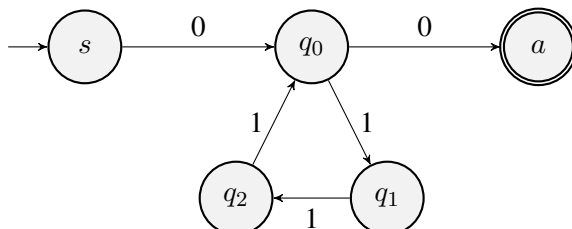
How do you prove that *no* machine for a particular language exists? It is not logically sound to say that you tried and failed to find a machine so it must not exist, because there might still be a trick you haven't seen!

The pumping lemma for regular languages provides a logically sound method for proving that no DFA could possibly exist for a particular language, no matter how clever you are. The way this works is we prove that if the language *is* regular, then it must have some property. Then illustrating that the property does not hold for a particular language means that the language cannot be regular.

### 1.9.1 Pumpability of All Regular Languages

The property that we focus on is the existence of a *pumpable* or looping substring in every long string in a regular language. A pumpable substring can be removed or repeated multiple times to generate an infinite number of other strings in the language. The proof of this property boils down to the pigeonhole principle. If a language is regular, then there is some DFA for it, and a long enough string in the language eventually has to revisit a state in the DFA. That substring and corresponding loop in the state sequence can be removed or repeated and the corresponding string will still end up at an accept state.

**Motivating example:** The following FA recognizes the language  $L = \{01^{3k}0 \mid k \geq 0\}$ . To make it a DFA, add a state  $r$  that receives all the unwritten transitions (from  $s$  on 1, from  $q_1$  and  $q_2$  on 0, from  $a$  on 0 or 1) and self loops on 0 or 1. The string  $00 \in L$  is not long enough to have a pumpable substring, but every string of length at least 4 that is in  $L$  contains a  $q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow q_0$  loop corresponding to substring 111, which can be removed or added arbitrarily many times and the resulting string will still end up at  $a$  and therefore be in the language.



We generalize this in Theorem 1.70 below before showing how to apply the result in the contrapositive direction to prove nonregularity.

**Theorem 1.70** (The pumping lemma). If  $A$  is a regular language, then there exists some  $p > 0$  such that for any string  $s \in A$  with  $|s| \geq p$ , there exist substrings  $x, y, z$  such that  $s = xyz$ ,  $|y| > 0$ ,  $|xy| \leq p$ , and for any  $i \geq 0$  we have  $xy^iz \in A$ . The minimum  $p$  for which this holds is called the *pumping length* of  $A$ .

*Proof.* We start with the hypothesis of the conditional theorem statement and immediately apply the definition of regularity. Assume that language  $A$  is recognized by DFA  $M = (Q, \Sigma, \delta, q_0, F)$ . It suffices to prove the pumping property for a single value of  $p$ , and we will choose  $p = |Q|$ . (The pumping length may be smaller.) We must prove that *every* long enough string in  $A$  pumps, so consider *any arbitrary*  $s \in A$  such that  $|s| \geq p$ .

Now we reason about the computation history of  $M$  on  $s$  to identify how to splice  $s = xyz$  so  $y$  corresponds to a loop in  $M$ . By the pigeonhole principle, the first  $p + 1$  states in the (accepting) computation history  $(r_0, \dots, r_{|s|})$  of  $M$  on  $s$  include a repeated state because there are only  $p = |Q|$  distinct states. In other words, there exist indices  $j, \ell$  with  $0 \leq j < \ell \leq p$  such that  $r_j = r_\ell$ . Then we let  $x = s_1 \dots s_j$ ,  $y = s_{j+1} \dots s_\ell$ ,  $z = s_{\ell+1} \dots s_{|s|}$ . Our selection of  $j, \ell$  guarantees  $|y| > 0$  and  $|xy| \leq p$ , so all that is left is to argue that  $xy^iz \in A$  for all  $i \geq 0$ .

Note that  $x$  takes  $M$  from  $r_0 = q_0$  to  $r_j$ ,  $y$  starting at  $r_j$  returns to  $r_\ell = r_j$ , and  $z$  takes  $M$  from  $r_j$  to  $r_{|s|}$ , which is in  $F$  because we assumed  $M$  accepts  $s$ . Then removing  $y$  means the string  $xy^0z = xz$  makes  $M$  transition from  $q_0$  to  $r_j$  to  $r_n \in F$  and accept, and adding extra copies of  $y$  makes  $M$  visit  $r_j$  extra times but still end up at  $r_n \in F$  and accept. Hence,  $xy^iz \in A$  for all  $i \geq 0$  and the proof is complete.  $\square$

## 1.9.2 Nonpumpability to Prove a Language is Not Regular

The logical equivalence of a conditional and its contrapositive means that if regularity *implies* pumpability (and the pumping lemma says it does), then non pumpability *implies* nonregularity. If you are asked to prove that a language is not regular, you must give an argument that allows you to conclude nonregularity. Non pumpability is one such argument!

First, it requires some care to specify what non pumpability means. Specifically, we have to remember how to negate alternating universal and existential quantifiers as in Math 51. Let's step away from automata for a moment to remember how quantifiers work in less technical settings. Suppose your friend has some strong feelings about soulmates, and you disagree. Your friend thinks that every person has (at least) one

other person that is a perfect match for them. You think that there is (at least) one person for whom no other person is a perfect match. We can formalize this with a 2-variable predicate, where  $M(x, y)$  is true if people  $x$  and  $y$  are a perfect match.

Your friend's belief:  $\forall x \exists y M(x, y)$

Your counter-belief:  $\exists x \forall y \neg M(x, y) \equiv \neg(\forall x \exists y M(x, y))$

Returning to DFAs, we can symbolically describe pumpability and non-pumpability, resp., as follows:

$$\begin{aligned} \exists p > 0 \forall s \in L \text{ with } |s| \geq p \exists x, y, z \text{ with } s = xyz, |y| > 0, |xy| \leq p \forall i \geq 0, xy^i z \in A \\ \forall p > 0 \exists s \in L \text{ with } |s| \geq p \forall x, y, z \text{ with } s = xyz, |y| > 0, |xy| \leq p \exists i \geq 0, xy^i z \notin A \end{aligned}$$

We now have the tools to argue non-pumpability of a particular language to give our first nonregularity proof. For a given language that you believe to be nonregular, you can prove nonregularity by showing that for any arbitrary  $p$ , there is some particular string  $s$  in the language of length at least  $p$  such that no matter how you try to decompose  $s$  into substrings  $x, y, z$ , it will not be a pumpable decomposition.

**Example 1.73:**  $L = \{0^n 1^n \mid n \geq 0\}$  is not regular.

*Proof.* Let  $p > 0$  be arbitrary, and pick  $s = 0^p 1^p$ . Consider any decomposition of  $s$  as  $s = xyz$  with  $|y| > 0, |xy| \leq p$ . Because the first  $p$  characters of  $s$  are 0 and  $y$  is a positive number of them, it must be that  $y = 0^k$  for some  $k = 1, \dots, p$ . Then picking  $i = 0$  we see  $xy^i z = xz = 0^{p-k} 1^p$ . Because there are fewer 0s than 1s,  $xz \notin L$ , so  $L$  has a long string without a pumpable decomposition, thus  $L$  is nonregular.  $\square$

Alternatively, we could have picked  $i = 2$  and argue  $xyyz \notin L$  because it has too many 0s.

**Exercises:** Show that the following languages are nonregular.

- Binary strings with an equal number of 0s and 1s
- Binary palindromes, ie strings that are equal to their reverse
- Binary strings with  $n$  0s followed by  $m < n$  1s

### 1.9.3 Logic Pitfalls

**Warning 1:** Remember that languages can be regular or not; individual strings have no inherent regularity or non-regularity. Additionally, there can be a regular language that contains a nonregular language as a subset. For example, we know the language  $\Sigma^*$  is regular because it is described by regex  $\Sigma^*$  and an NFA with a single accepting state that loops at itself on 0 or 1. This language includes all strings in the nonregular language  $\{0^n 1^n \mid n \geq 0\}$ . So why doesn't nonpumpability of  $s = 0^p 1^p$  demonstrated in Example 1.73 prove nonregularity of  $\Sigma^*$ ? The example shows that  $0^p 1^p$  doesn't pump *within* that nonregular subset, but adding or removing 0s keeps the string in  $\Sigma^*$ , so in fact  $0^p 1^p$  does pump in  $\Sigma^*$ , which is consistent with its regularity.

**Warning 2:**  $0^p 1^p$  pumping in  $\Sigma^*$  is *consistent* with the language's regularity, but it does not *prove* its regularity. You can use (the contrapositive of) the pumping lemma to prove nonregularity; do *not* try to use the pumping lemma to prove regularity. The pumping lemma says that regularity implies pumpability. The contrapositive is logically equivalent – nonpumpability implies nonregularity – but the converse is not true. Showing that strings in a language pump does not allow you to conclude that the language is regular. For that, you need to give a certificate of regularity, like an FA or a regex.

To see why the converse of the pumping lemma is not true, consider the language  $L = L_1 \cup L_2$  with  $L_1 = \{uv \mid u = 1^k \text{ for } k \neq 1, v = \varepsilon \text{ or } v \in \{0, 1\}^* \text{ with } v_1 = 0\}$  and  $L_2 = \{10^n 1^n \mid n \geq 0\}$ .  $L_1$  consists of  $\varepsilon$ , all strings that start with 0, and all strings that start with 11.  $L_2$  consists of strings that start with a single 1 followed by  $n$  0s and  $n$  1s. We claim that their union pumps with  $p = 2$  yet is not regular.

We consider all strings of length at least 2 in  $L$  and show that they all pump. There are several cases:

1. The string is of the form  $10^n 1^n$  for  $n > 0$ . Let  $x = \varepsilon, y = 1, z = 0^n 1^n$ . Pumping up  $y$  results in a string that starts with 11, and pumping it down results in a string that starts with 0.
2. The string starts with 00. Pumping the first 0 maintains a string that starts with 0.
3. The string is 01. Pumping the first 0 up maintains a string that starts with 0; pumping down yields 1.
4. The string starts with 010. Pumping the first 01 maintains a string that starts with 0.
5. The string starts with 011. Pumping the first 0 up maintains a string that starts with 0; pumping down gives a string that starts with 11.
6. The string is 11. Pumping the whole string up maintains a string that starts with 11; pumping down yields  $\varepsilon$ .
7. The string starts with 110. Pumping the first two 1s up maintains a string that starts with 11; pumping down yields a string that starts with 0.
8. The string starts with 111. Pumping the first 1 maintains a string that starts with 11.

All of these resulting strings are in  $L_1 \subseteq L$ , so we've shown that all long strings in the language pump.

Then how do we prove that  $L$  is nonregular if it doesn't contradict the pumping lemma? Closure properties can be another way of contradicting regularity. Let  $L_3$  be the language of  $10\Sigma^*$ . This is regular, so if  $L$  were regular, then  $L \cap L_3$  should be regular as well. But  $L \cap L_3 = \{10^n 1^n \mid n > 0\}$ , which we *can* show contradicts the pumping lemma and is therefore not regular. Hence  $L$  cannot have been regular.

## 2 Unit 2: Context-Free Languages

In ending Unit 1 with the pumping lemma, we found a way to formalize the limits of computation of finite state machines. The language  $\{0^n 1^n \mid n \geq 0\}$  can't possibly be recognized by a DFA because a DFA with  $p$  states would have to loop on  $0^p 1^p$  before ever reaching the 1s, and then it wouldn't be able to tell  $0^p 1^p$  from certain strings with an imbalance in 0s and 1s.

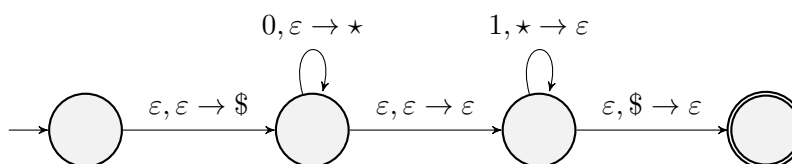
### 2.1 Introduction to the Machinery for Context-Free Languages

#### 2.1.1 Pushdown Automata: FAs With Stacks

Intuitively, DFAs can't accomplish tasks that involve matching different parts of a string because the only way to keep track of their progress when digesting a string left to right is with a finite number of states. In other words, a DFA's current state is its only memory about the portion of the string digested so far.

In Unit 2, we consider finite state machines with an additional source of memory: a memory *stack* that can be written to and read from as the machine reads input and transitions between its states. Because we use the verbs *push* and *pop* to refer to writing to and reading from the first-in-last-out data structure called a stack, we refer to these machines as **pushdown automata** (PDAs). We will formally define PDAs later in the unit, but first let's start with an intuitive example.

To design a PDA for the nonregular language  $\{0^n 1^n \mid n \geq 0\}$  we get around DFAs' inability to keep track of their progress digesting a string by writing a special character (here we choose  $\star$ ) to the PDA's stack every time the machine reads a 0 and then reading a corresponding character from the stack every time it reads a 1. The following state diagram represents this.



The states of the PDA keep track of which stage of the computation the machine is in (reading zeros from input while pushing to the stack versus reading ones while popping). Note that in addition to specifying what character of input is read in order to make each transition, the transition labels also specify what character is popped from the stack and what character is pushed to the stack.

A technicality is that a PDA can't check whether it's at the bottom of its stack (i.e., nothing in memory), so when we use a stack to balance characters, we often have the first and last steps of computation be to respectively push and pop a special character ( $\$$  by convention), ensuring that we end on an empty stack.

#### 2.1.2 Context-Free Grammars: Generative Analogues

In Unit 1, we showed not only that deterministic and nondeterministic FAs were equivalently powerful, but that regular expressions were also an equivalently powerful generative analogue. A PDA is like an FA in that it turns an input (a string over some alphabet) into an output (acceptance or not), with the set of accepting strings defining its language; a regular expression is a specially formatted string that inductively define a language, and this unit we will see that a **context-free grammar** (CFG) is a set of rules with an underlying input-less model of computation that produces an associated language. Before getting into the specifics, we present a CFG for  $\{0^n 1^n \mid n \geq 0\}$ :

$$S \rightarrow 0S1 \mid \epsilon$$

## 2.2 Context-Free Grammars

First conceived to describe grammatical rules of natural languages, CFGs are instructions for generating a set of strings that match a specific pattern. To specify such instructions, a CFG needs to have a set of *variables*, serving as stand-ins for other strings, a set of *terminals*, the alphabet for the language whose elements can be generated by the CFG, a set of *rules* that describe how to turn each variable into a string of variables and terminals, and a *start variable*, which indicates the starting point of the generative process.

Although the formal definition requires specifying these components as a 4-tuple, like state diagrams for FAs, we can represent this 4-tuple more compactly. For example, the CFG  $[S \rightarrow 0S1 \mid \varepsilon]$  just lists the two rules of the grammar:  $S$  can be replaced with  $0S1$ , or  $S$  can be replaced with  $\varepsilon$ . If we knew our alphabet was  $\Sigma = \{0, 1\}$ , these must be the terminals, so  $S$  is the start and only variable.

Why is  $\{0^n 1^n \mid n \geq 0\}$  the language of this CFG? Rules specify how strings can be *derived* from a CFG's start variable: begin with the start variable, pick one of its rules to replace it with, and continue this process with any resulting variables until there are no variables left. For example, to generate  $0011$ , replace  $S$  with  $0S1$  twice and then replace the remaining  $S$  with  $\varepsilon$  to reach a string of all terminals:

$$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 0011$$

We capture these ideas formally as follows.

**Definition 2.2.** A context-free grammar is a 4-tuple  $G = (V, \Sigma, R, S)$ , where

$V$  is a finite set of objects called variables,

$\Sigma$  is a finite set of objects called terminals,

$R$  is a finite set of rules, each specifying a variable and a string of variables and terminals, and

$S \in V$  is called the start variable.

For any strings  $u, v, w \in (V \cup \Sigma)^*$ , if there is a rule  $A \rightarrow w$ , then we say  $uAv$  **yields**  $uwv$ , written  $uAv \Rightarrow uwv$ , and if there exist  $u_1, \dots, u_k \in (V \cup \Sigma)^*$  for  $k \geq 0$  such that  $u \Rightarrow u_1 \Rightarrow \dots \Rightarrow u_k \Rightarrow w$ , then we say  $u$  **derives**  $w$ , written  $u \Rightarrow^* w$ .

The language generated by CFG  $G$ , denoted  $L(G)$ , is the set of all  $w \in \Sigma^*$  such that  $S \Rightarrow^* w$ .

We know from last unit that the language of our first CFG is nonregular, so we already know that the class of languages associated with CFGs is not the same as the class of languages associated with FAs and regexes. We call the languages in our new class context-free languages (CFLs).

**Definition.** A set of strings is a context-free language if and only if it is generated by some CFG.

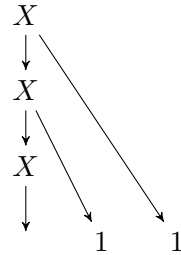
When we return to PDAs we will see that PDAs generalize NFAs and are equivalently powerful to CFGs. That means that the class of context-free languages contains the class of regular languages. You will give an alternate proof of this in your homework by giving an inductively constructive CFG for any regular expression, showing that CFGs generalize regular expressions. We seed that general argument with a specific example.

**Exercise:** Design a CFG for  $(a \cup b)c^*$ .

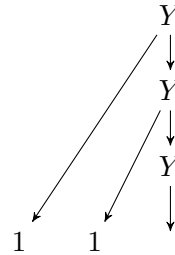
## 2.3 CFG Interpretation and Design

The previous exercise shows that FA loops/regex stars can be represented with CFG rules employing 1-sided recursion. For example, variable  $X$  with rules  $X \rightarrow X1 \mid \varepsilon$  derives all strings in  $1^*$ . The same terminal strings would be derived if we did tail rather than head recursion, i.e.,  $Y \rightarrow 1Y \mid \varepsilon$ . However, we see that the string 11, for example, has different *derivations* and associated *parse trees* under the equivalent CFGs.

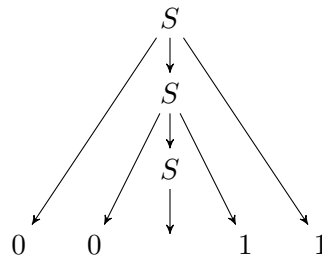
$$X \Rightarrow X1 \Rightarrow X11 \Rightarrow 11$$



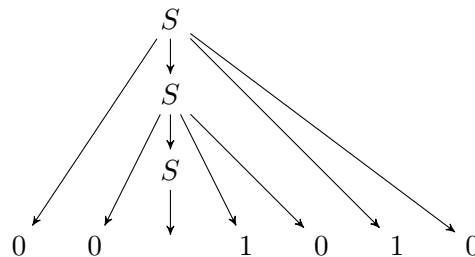
$$Y \Rightarrow 1Y \Rightarrow 11Y \Rightarrow 11$$



FAs and regexes cannot balance characters in one part of a string with characters in another part of the string, but CFGs can, because the substitution rules also allow 2-sided recursion, which we see in our original CFG  $S \rightarrow 0S1 \mid \varepsilon$  generating  $\{0^n 1^n \mid n \geq 0\}$ . The recursive rule generates a 0 on the left and a 1 on the right, placing a new copy of  $S$  (we can think of this as a recursive call) in the middle. We can visualize the parse tree for the derivation  $S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 0011$  as follows:



Note that although recursive CFG rules is (at most) 2-sided, that does not mean that a recursive rule can generate at most 2 terminals at once. For example,  $S \rightarrow 0S10 \mid \varepsilon$  generates the language  $\{0^n (10)^n \mid n \geq 0\}$ , and the parse tree associated with the derivation  $S \Rightarrow 0S10 \Rightarrow 00S1010 \Rightarrow 001010$  looks as follows:





**Exercises:** We now explore CFGs with multiple variables or multiple recursive rules for a single variable through some additional interpretation and design exercises.

1. Consider the 2-variable CFG with rules  $S \rightarrow 0S0 \mid T$ ;  $T \rightarrow 1T \mid 1$ . What are the recursive rules and what are the recursion base cases? What are the strings derived by  $T$ ? By  $S$ ? What is the language of the grammar? Pick a representative string in the language and give a parse tree for it. Is the language regular? Why or why not?
2. Consider the 2-variable CFG with rules  $S \rightarrow AB$ ;  $A \rightarrow 0S0 \mid \varepsilon$ ;  $B \rightarrow 1B \mid 1$ . What are the recursive rules and what are the recursion base cases? What are the strings derived by  $A$  and  $B$ , and what is the language of this grammar? Pick a representative string in the language and give a parse tree for it. Is the language regular? Why or why not?
3. Design a CFG for the language  $\{0^n 1^m \mid m \geq 2n \geq 0\}$ .
4. Design a CFG for the language of binary strings with an equal number of 0s and 1s.

### 2.3.1 Design Pitfall

When matching, use recursion to keep track of your center axis rather than extra variables for modularity. Why is  $S \rightarrow ZN$ ;  $Z \rightarrow Z0 \mid \varepsilon$ ;  $N \rightarrow N1 \mid \varepsilon$  not equivalent to  $S \rightarrow 0S1 \mid \varepsilon$ ? The latter guarantees that 0s and 1s are balanced, while the former allows  $Z$  to generate as many 0s as it wants independent of how many 1s  $N$  generates.

## 2.4 Ambiguity

Part of the reason it is so important for computer scientists to learn about context-free grammars is that they not only describe the grammatical structure of natural languages; they define the rules for programming in high-level programming languages such as C++ and Python. Below is a very incomplete, proof-of-concept gesture at a CFG for valid C++ programs.

```

$$\begin{aligned} S &\rightarrow \text{int main}(A)\{B\} \\ A &\rightarrow A,A \mid TN \mid \varepsilon \\ T &\rightarrow \text{int} \mid \text{double} \mid [\text{rules for other types}] \\ N &\rightarrow [\text{rules to generate valid C++ variable names}] \\ B &\rightarrow B;B \mid [\text{rules to generate conditionals, loops, and valid C++ commands}] \end{aligned}$$

```

It is important in programming language for every identifier in a program to have well-defined scope and exactly one variable that it names, otherwise your compiler would not know how to interpret your code. In other words, *ambiguity* is disallowed by the C++ grammar, because we don't want a program to have multiple ways it could be parsed by a compiler.

In this analogy, a specific C++ program is a string in the language of valid C++ programs, a CFL associated with some CFG. What we mean by ambiguity depends on whether we are talking about a string, grammar, or language. That no valid string is ambiguously derived by the underlying CFG means that the underlying CFG is an unambiguous grammar, and the associated CFL is not inherently ambiguous.

Intuitively, a particular string is derived ambiguously by a particular CFG if it has two different parse trees. But a parse tree is just a two-dimensional visualization of a **derivation** of a string under some grammar, so we formalize ambiguity with respect to derivations. We define a canonical **leftmost derivation** that corresponds to the (unique) depth-first traversal of a parse tree.

**Definition.** A **derivation** of a string  $w \in \Sigma^*$  under a CFG  $G = (V, \Sigma, R, S)$  is a sequence of yield steps  $S \Rightarrow u_1 \Rightarrow \dots \Rightarrow u_k \Rightarrow w$  for  $k \geq 0, u_1, \dots, u_k \in (V \cup \Sigma)^*$ . We call this a **leftmost derivation** if every yield replaces the leftmost variable in the intermediate string according to one of its rules, i.e., for  $i = 1, \dots, k-1$ , we have  $u_i = tAv$  for some  $t \in \Sigma^*, A \in V, v \in (V \cup \Sigma)^*$  and  $u_{i+1} = twv$  for some  $w \in (V \cup \Sigma)^*$  such that  $A \rightarrow w$  is in  $R$ .

We can define ambiguity for strings, grammars, and languages accordingly:

**Definition 2.7.** A string is **ambiguously derived** by a CFG if it has multiple leftmost derivations under that CFG. A CFG is **ambiguous** if it ambiguously derives some string. A CFL is **inherently ambiguous** if every CFG that generates it is ambiguous.

**Exercises:** Practice reasoning about ambiguity with the following exercises.

1. Give a string that is ambiguously derived by  $S \rightarrow 0S11 \mid S1 \mid \varepsilon$ . Is the CFG ambiguous? Is its language inherently ambiguous?
2. Argue that  $S \rightarrow 0S10 \mid \varepsilon$  is unambiguous.

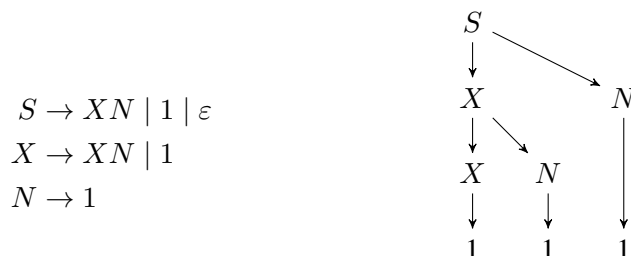
**Thought experiment:** It is much more difficult to prove that a language is inherently ambiguous than that it is not. It just takes one unambiguous grammar to show that a language is not inherently ambiguous. I'll never ask you to prove inherent ambiguity because this would require arguing that every possible grammar for a language is ambiguous – one ambiguous grammar does not imply inherent ambiguity of the language. In fact, you can design a (silly) ambiguous grammar for any CFL. To do so, take the CFG you know must exist by definition of CFLs (assume its start variable is  $S$  and that there is no  $S_0$  or  $E$  variable), replace the start variable with  $S_0$  and add the rules  $S_0 \rightarrow S \mid ES; E \rightarrow \varepsilon$ . This adds ambiguity because any derivation in the original grammar can be prepended with either of the following two leftmost partial derivations in the new grammar:

$$\begin{aligned} S_0 &\Rightarrow S \\ S_0 &\Rightarrow ES \Rightarrow S \end{aligned}$$

Putting aside ambiguity and returning to the setting of using CFGs to represent rules of a programming language, it would be very useful to have a way of checking whether a program is valid and parsing it if so. Compilers are useful! Because there are so many ways to write down a CFG, we will do some normalizing preprocessing to get our grammars into a consistent form before developing a parsing algorithm that relies on that form.

## 2.5 Chomsky Normal Form

Recall our simple 1-variable grammar using 1-sided recursion to generate  $1^*$ :  $X \rightarrow X1 \mid \varepsilon$ . This language is also be generated by the following multi-variable grammar, with its parse tree for 111 depicted on the right:



This grammar is in **Chomsky normal form**, the significance of which will be clarified later when we study the CYK parsing algorithm:

**Definition 2.8.** A CFG  $G = (V, \Sigma, R, S)$  is in **Chomsky normal form** if each of its rules in  $R$  is of one of the following forms:

$$S \rightarrow \varepsilon$$

$$A \rightarrow BC \text{ for } A \in V \text{ and } B, C \in V \setminus \{S\}, \text{ or}$$

$$A \rightarrow a \text{ for } A \in V \text{ and } a \in \Sigma.$$

Our first grammar for  $1^*$  is not in Chomsky normal form because its first rule has a variable alongside a terminal, and this variable is the start variable. The second grammar is, because  $\varepsilon$  only appears as a substitution rule for the start variable, which never appears on the right hand side of any rules, and all other substitutions are either a single terminal or two (non-start) variables.

The language of  $1^*$  wasn't carefully chosen to admit some grammar in Chomsky normal form. Any language with a CFG has one in Chomsky normal form, which we will prove now:

**Theorem 2.9.** Any CFL is generated by a CFG in Chomsky normal form.

*Proof.* This constructive proof can be thought of as an algorithm with the property that for any input CFG, it produces an equivalent output CFG in Chomsky normal form. The algorithm can be summarized in the following 5-step overview:

1. Add a new start variable to remove possible start recursion.
2. Remove  $\varepsilon$ -rules by bubbling them up to  $S$ .
3. Remove unit rules.
4. Break down substitution rules with more than two characters.
5. Fix substitution rules that have two terminals or one variable with one terminal.

We give the algorithm in more detailed pseudocode below. Correctness could be proven formally by showing that each step maintains language equivalence and achieves a property that is never undone, ending in a set of properties that collectively satisfy the Chomsky normal form requirements, but we will not go into the correctness details.

Normalize(CFG  $G = (V, \Sigma, R, S)$ ):

1. Add  $S_0$  to  $V$  and add  $S_0 \rightarrow S$  to  $R$ .
  2. For every  $A \rightarrow \epsilon \in R$  for some  $A \in V \setminus \{S_0\}$ :  
 Remove  $A \rightarrow \epsilon$  from  $R$ .  
 For every occurrence of  $A$  in the RHS of some rule in  $R$ ,  
 add to  $R$  a rule with that occurrence of  $A$  removed.
  3. For every  $A \rightarrow B \in R$  for some  $A, B \in V$ :  
 Remove  $A \rightarrow B$  from  $R$ .  
 For every  $B \rightarrow u$ ,  
 add to  $R$  the rule  $A \rightarrow u$ .
  4. For any  $A \rightarrow u_1 \dots u_k$  in  $R$  with  $k \geq 3$ ,  $u_i \in V \cup \Sigma$  for all  $i = 1, \dots, k$ ,  
 remove it, and add  $A \rightarrow u_1 A_1$ ,  $A_1 \rightarrow u_2 A_2$ ,  $\dots$ ,  $A_{k-2} \rightarrow u_{k-1} u_k$  to  $R$  and  $A_1, \dots, A_{k-2}$  to  $V$ .
  5. For any rule  $A \rightarrow uv$  with  $u \in \Sigma, v \in V \cup \Sigma$ ,  
 add a new variable and rule  $U \rightarrow u$  and replace  $A \rightarrow uv$  with  $A \rightarrow Uv$ .  
 For any rule  $A \rightarrow vu$  with  $v \in V, u \in \Sigma$ ,  
 add a new variable and rule  $U \rightarrow u$  and replace  $A \rightarrow vu$  with  $A \rightarrow vU$ .
- Return  $(V, \Sigma, R, S_0)$ .

Because any CFL has CFG that generates it, the algorithm shows that any CFL has a CFG in Chomsky normal form that generates it.  $\square$

**Exercises:** Convert the following two CFGs into Chomsky normal form.

- $S \rightarrow 0S1 \mid \epsilon$
- $S \rightarrow MX$   
 $M \rightarrow 1M0 \mid \epsilon$   
 $X \rightarrow X0 \mid \epsilon$

## 2.6 The Cocke-Younger-Kasami Parsing Algorithm

With the definition of Chomsky normal form grammars and an algorithm that can convert any CFG into an equivalent CFG in Chomsky normal form, we are ready to develop our parsing algorithm. In its most basic form, the algorithm checks whether a given string can be generated by a given context-free grammar in Chomsky normal form. The algorithm's ideas were first published by Tadao Kasami and separately discovered by Daniel Younger and John Cocke.

### 2.6.1 The Dynamic Programming Paradigm

The algorithm fills out lists of variables stored in a 2D array, keeping track of all possible derivations of the input's substrings from the input grammar. It is a classic example of the *dynamic programming* paradigm, in which an algorithm solves a big problem by combining solutions from its component subproblems.

Main problem: Does the start variable of a grammar  $G$  derive a particular string  $w$ ?  $S \xRightarrow{*} w$

Subproblem: Does a particular variable in  $G$  derive a particular substring of  $w$ ?  $A \xRightarrow{*} w_s \dots w_e$

After defining subproblems, we describe how to solve the smallest such subproblems (look for variables that yield individual characters), give a recurrence for larger problems ( $A \xRightarrow{*} uv$  if there is a rule  $A \rightarrow BC$  where  $B \xRightarrow{*} u$  and  $C \xRightarrow{*} v$ ), and describe the order of solving subproblems (illustrated by the for loops in the pseudocode below).

**Exercise:** Draw a parse tree for 0011 under your Chomsky normal form grammar for  $\{0^n 1^n \mid n \geq 0\}$ . Every node in the parse tree corresponds to a ‘yes’ answer to a subproblem above. Which? How does a parent node inherit its ‘yes’ answer from its children?

### 2.6.2 Pseudocode

In the pseudocode for CYK below, the algorithm takes input string  $w = w_1 \dots w_n \in \Sigma^n$  and a Chomsky normal form CFG  $(V, \Sigma, R, S)$ . It populates the cells of an  $n \times n$  array  $P$  from small subproblems to large ones so that  $P[\ell, s]$  contains variable  $A \in V$  iff variable  $A$  derives the length- $\ell$  substring starting at the  $s$ th character of the input, i.e., if  $A \xRightarrow{*} w_s \dots w_{s+\ell-1}$ . After the table is fully populated,  $P[n, 1]$  contains the start variable  $S$  if and only if  $S \xRightarrow{*} w$ , answering the language membership question.

CYK(CFG  $G = (C, \Sigma, R, S)$ , string  $w \in \Sigma^n, n \geq 0$ ):

If  $n = 0$  and  $S \rightarrow \varepsilon$  is in  $R$ : *accept*.

If  $n = 0$  and  $S \rightarrow \varepsilon$  is not in  $R$ : *reject*.

Initialize  $P$  as an  $n \times n$  matrix of empty lists.

For  $s = 1 \dots n$ :

    Add to  $P[1, s]$  any  $A \in V$  such that  $A \rightarrow w_s$  is in  $R$ .

For  $\ell = 2 \dots n$ :

    For  $s = 1 \dots n - \ell + 1$ :

        For  $p = 1 \dots \ell - 1$ :

            Add to  $P[\ell, s]$  any  $A \in V$  such that  $A \rightarrow BC$  with  $B$  in  $P[p, s]$  and  $C$  in  $P[\ell - p, s + p]$ .

If  $P[n, 1]$  contains  $S$ : *accept*.

Else: *reject*.

**Exercises:** Fill out CYK's subproblem matrix for the string 0011 with your Chomsky normal form grammar for  $\{0^n 1^n \mid n \geq 0\}$ . How can we incorporate pointers to keep track of a parse tree for  $w$  under the grammar while populating the table (without increasing asymptotic runtime)?

Doing the minimum amount of extra work, fill out the subproblem matrix for the string 00011 under the same grammar. How does the algorithm tell you that the first string is in the language and the second is not?

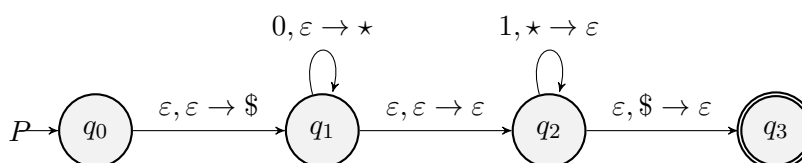
### 2.6.3 Runtime

Though we don't normally think about efficiency in this course, it is at the core of the *practicality* of CYK. Runtime analysis shows that it is  $O(n^3 |G|)$ , where  $n$  is the input string length and  $|G|$  is the size (number of rules) of the grammar. This means that CYK establishes the problem of deciding whether a CFG generates a particular string as being in the class P of problems decidable in polynomial time.

## 2.7 Pushdown Automata

We now return to the mechanics of PDAs that we previewed at the beginning of the unit and show that PDAs generalize NFAs from Unit 1 before establishing their connection to context-free languages by proving their equivalence with context-free grammars.

Recall the state diagram that represents a PDA that uses a memory stack to keep track of how many 0s it reads in order to ensure that it then reads the same number of 1s before accepting:



The only difference between PDAs and NFAs is that PDAs have a memory stack. We must modify the form of the transition function for our PDA definition, because what happens to the machine depends not only on the current state and input character read, but also (possibly) the input character popped from the stack. Then what happens to the machine is not just a set of options for new states to transition to, but also a set of options for what character might be pushed to the stack. In order to formalize this, we need one extra set: a stack alphabet.

In the PDA above, our state set is  $Q = \{q_0, q_1, q_2, q_3\}$  with start state  $s = q_0$  and final states  $F = \{q_3\}$ . Our input alphabet is  $\Sigma = \{0, 1\}$  but our *stack alphabet* is  $\Gamma = \{\$, *\}$  because these are the only characters pushed to or popped from the stack. It is okay but not required for the input and stack alphabets to be disjoint: if we wanted to use 1 to denote the bottom of the stack and 0 to denote the breadcrumbs we leave as we read 0s, the alphabets could be the same! Because a transition is determined by the current state, the next input character (or  $\varepsilon$ ), and the character at the top of the stack (or  $\varepsilon$ ), and it corresponds to a set of options of new state + memory write, the form of the transition function is

$$\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \rightarrow \mathcal{P}(Q \times \Gamma_\varepsilon).$$

In addition to updating the form of  $\delta$  and adding  $\Gamma$  to the set of objects in a PDA tuple, we have a new model of computation that determines the set of strings accepted by a PDA. Now an accepting computation history must not only include a state sequence that obeys  $\delta$ , it must also include a *stack transcript* that logs the contents of the memory stack before and after each transition and satisfies  $\delta$  in conjunction with the state sequence. For example, an accepting computation history of  $P$  on 0011 can be written as follows:

$q_0$	$q_1$	$q_1$	$q_1$	$q_2$	$q_2$	$q_2$	$q_3$
$\varepsilon$	$\$$	$\star \$$	$\star \star \$$	$\star \star \$$	$\star \$$	$\$$	$\varepsilon$

Note that when we right stack contents left-to-right, the leftmost character is on the top. You can alternatively write them vertically, as we will do in class. We are now ready for our formal definition:

**Definition 2.13.** A pushdown automaton is a 6-tuple  $P = (Q, \Sigma, \Gamma, \delta, s, F)$ , where

$Q$  is a finite state set,

$\Sigma$  is a finite input alphabet,

$\Gamma$  is a finite stack alphabet,

$\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \rightarrow \mathcal{P}(Q \times \Gamma_\epsilon)$  is the transition function,

$s \in Q$  is the start state, and

$F \subseteq Q$  is the set of final or accepting states.

For any input string  $w \in \Sigma^*$ , an **accepting computation history** of PDA  $P$  running on  $w$  is a state sequence  $r_0, \dots, r_m \in Q$  and stack transcript  $s_0, \dots, s_m \in \Gamma^*$  such that  $r_0 = s, s_0 = \epsilon$ , there exist  $y_1, \dots, y_m \in \Sigma_\epsilon$  with  $w = y_1 \dots y_m$  and for all  $i = 0, \dots, m-1$  there exist  $a, b \in \Gamma_\epsilon$  and  $t \in \Gamma^*$  such that  $s_i = at, s_{i+1} = bt$  and  $(r_{i+1}, b) \in \delta(r_i, y_{i+1}, a)$ , and  $r_n \in F$ . We say  $P$  **accepts**  $w$  if it has an accepting computation history on  $w$ .

The **language recognized by PDA  $P$** , denoted  $L(P)$ , is the set of all strings  $w$  that  $P$  accepts.

**Exercise:** Give a PDA for  $\{0^n(10)^m \mid m \geq n \geq 0\}$  and an accepting computation history for 01010.

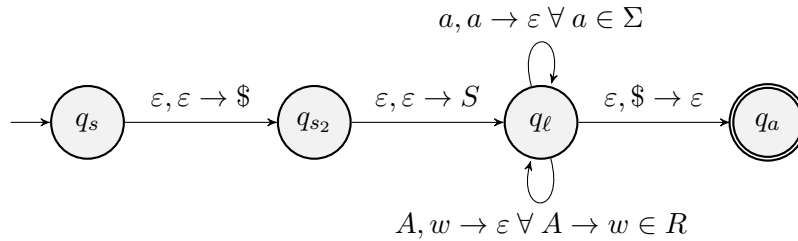
## 2.8 Equivalence of PDAs and CFGs

To show that PDAs and CFGs are equally powerful and interchangeably define the class of CFLs, we give two proof sketches showing how any CFG can be converted into an equivalent PDA and how any PDA can be converted into an equivalent CFG.

**Lemma 2.21.** Every CFG has an equivalent PDA.

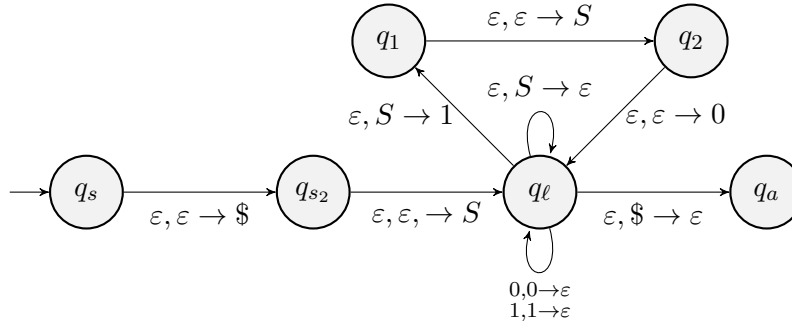
*Proof sketch.* Given an arbitrary CFG  $G = (V, \Sigma, R, S)$ , we construct a PDA  $P = (Q, \Sigma, \Gamma, \delta, s, F)$  to digest any string in the language by simulating its derivation under  $G$ .

We let  $Q = \{q_s, q_{s_2}, q_\ell, q_a\} \cup \{\text{more states depending on } R\}$ ,  $s = q_s$ ,  $F = \{q_a\}$ , and  $\Gamma = \{\$ \} \cup V \cup \Sigma$ . Transition function  $\delta$  enforces that the PDA first pushes a  $\$$  and then  $S$  to the stack, representing the starting point for the derivation of any string in the grammar. Then variables in an in-progress intermediate derivation are substituted according to their rules by taking any number of loops around  $q_2$ , and the input string is digested one alphabet character at a time whenever a matching terminal character is on the top of the stack. Once the derivation is complete and the initial  $\$$  is exposed, the PDA transitions to its accepting state.



The above diagram serves a mnemonic for  $\delta$ , with the self loops around  $q_2$  varying across CFGs. □

**Example:** Our original CFG  $[S \rightarrow 0S1 \mid \varepsilon]$  would need two rule loops (for  $S \rightarrow 0S1$  and  $S \rightarrow \varepsilon$ ) and two alphabet-terminal matching loops (for 0 and 1) around  $q_\ell$  with the following structure.



The characters in the  $S \rightarrow 0S1$  substitution are pushed right to left because the stack ensures that the last character pushed will be the first popped. Then the accepting computation history of this grammar-simulating PDA on 0011 could be written as follows:

$q_s$	$q_{s2}$	$q_l$	$q_1$	$q_2$	$q_l$	$q_l$	$q_1$	$q_2$	$q_l$	$q_l$	$q_l$	$q_l$	$q_l$	$q_a$
$\varepsilon$	\$	$S$	1	$S1$	$0S1$	$S1$	11	$S11$	$0S11$	$S11$	11	1	\$	$\varepsilon$

Note that the construction from the proof of Lemma 2.21 provides an algorithmic way of designing PDAs for a CFL without really understanding PDAs, as long as you are able to design a correct CFG. Unless I ask specifically for a PDA of a certain form, this is a strategy you can always use, but I encourage you to think about more natural uses of a PDA stack, like our first PDA for this language.

**Exercise:** Design a PDA corresponding to the grammar  $[S \rightarrow SS \mid T; T \rightarrow aTb \mid ab]$ .

**Lemma 2.27.** Every PDA has an equivalent CFG.

*Proof sketch.* Given an arbitrary PDA  $P = (Q, \Sigma, \Gamma, \delta, s, F)$ , we construct a CFG whose variables correspond to pairs of states in the PDA and whose rules ensure that  $A_{pq}$  derives the set of strings that could allow the PDA to transition from state  $p$  to state  $q$  beginning and ending on an empty stack. This construction fits the dynamic programming paradigm of CYK, with main and subproblems characterized as follows:

Main problem: Design a grammar whose start variable  $A_{sa}$  derives  $L(P)$ .

Subproblems: Ensure  $A_{pq}$  derives all strings that take  $P$  from  $p$  to  $q$  with no net stack effect.

To apply this strategy, we need to modify the PDA to meet a few technical conditions. First, it should not accept any strings unless its stack is empty. For this, we can use the \$ technique to keep track of the bottom of the stack, remove the accept status from the accepting state(s), and force them to empty their stack before transitioning to a new accept state. Second, we need a single accepting state  $a$ . If there were multiple, we can combine them into one with  $\varepsilon$  transitions. Finally, each transition must either push something to or pop something from the stack, and not both. A transition that includes both a pop and a push can be replaced with a 2-transition path that does this in two steps, and a transition that doesn't use the stack can be replaced with a 2-transition path that pushes and pops a dummy character.

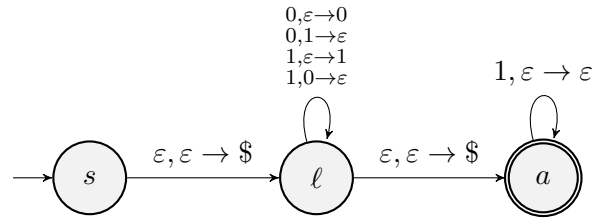
With these restrictions in place,  $P$  can transition from  $p$  to  $q$  with no net stack effect in three ways, and we add rules for variable  $A_{pq}$  accordingly:



1. If  $p = q$ , we can begin and end on an empty stack without reading any input.  
 $\implies$  Add rules  $A_{pp} \rightarrow \varepsilon$  for all states  $p$ .
2. For any state  $r$ , we can get from  $p$  to  $r$  beginning and ending on an empty stack, and then get from  $r$  to  $q$  beginning and ending on an empty stack.  
 $\implies$  Add rules  $A_{pq} \rightarrow A_{pr}A_{rq}$  for all states  $p, q, r$ .
3. We can get from  $p$  to  $q$  beginning and ending on an empty stack without ever having an empty stack in the middle by pushing stack character  $u$  when transitioning to  $r$  while reading input character  $a$ , getting from  $r$  to  $s$  adding things to the stack and picking up everything that was added, and then popping  $u$  when transitioning to  $q$  while reading input character  $b$ .  
 $\implies$  Add rules  $A_{pq} \rightarrow aA_{rs}b$  for all states  $p, q, r, s$ .

These rules collectively capture the recurrence relations between the subproblems and ensure that the grammar generates the desired language.  $\square$

**Exercise:** The following PDA recognizes strings with an equal number of 0s and 1s followed by all 1s. Give a computation history for it on 110100011, modify it so it fits the technical conditions of the proof of Lemma 2.27, and design an equivalent CFG using the construction of the proof.



## 2.9 The Pumping Lemma for Context-Free Languages

Because of the equivalent power of PDAs and CFGs, designing a PDA or CFG for a language suffices to prove that it is context free. In the same way that the pumping lemma for regular languages (Theorem 1.70) provided a sound method for proving that no DFA could possibly exist for a particular language and therefore the language cannot be regular, we now develop a pumping lemma for context-free languages (Theorem 2.34) to provide a sound method for proving that a language is not context free.

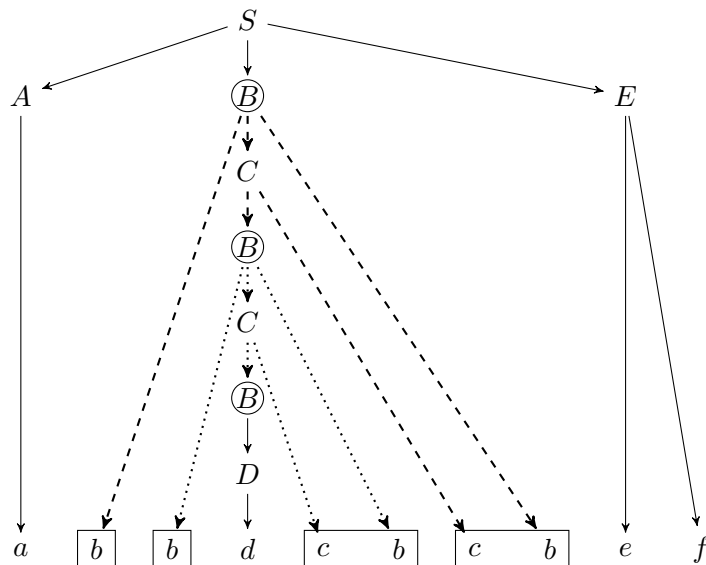
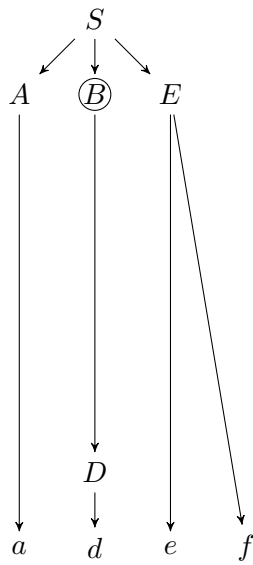
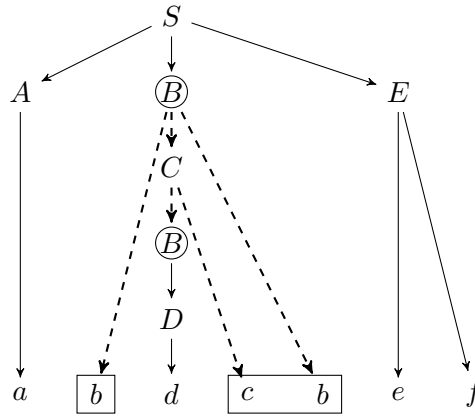
### 2.9.1 Pumpability of All Context-Free Languages

Whereas we focused on substrings cycling in a DFA as our notion of regular pumpability, we will now think about CFG recursion as our notion of context-free pumpability. Again the proof boils down to the pigeonhole principle, but now the pigeons are the variables in  $S$ -to-terminal path in a parse tree, and the holes are the variables. The repetition of a variable in a parse tree will generate *two* substrings that can be pumped down once or up arbitrarily, generating new strings in the language.

**Motivating example:** The following CFG generates the language  $L = \{ab^n d(cb)^n ef \mid n \geq 0\}$  over alphabet  $\Sigma = \{a, b, c, d, e, f, g\}$ . The choice to split the recursion into two steps is to better illustrate the generality of the argument about repeated variables.

$$\begin{aligned} S &\rightarrow ABE \\ A &\rightarrow a \\ B &\rightarrow bCb \mid D \\ C &\rightarrow Bc \\ D &\rightarrow d \\ E &\rightarrow ef \end{aligned}$$

Between the first and second instance of  $B$  in the parse tree of the string  $s = abdcbe f$ , the substring consisting of the leftmost  $\boxed{b}$  and the substring  $\boxed{cb}$  get generated. We correspondingly decompose  $s = uvxyz$  with  $u = a, v = \boxed{b}, x = d, y = \boxed{cb}, z = ef$ . Substrings  $v$  and  $y$  can be pumped down or up by replicating the corresponding subtrees in the parse tree. Below we depict the parse trees for the original string, the string removing the repetition of  $B$ , and the string adding the repetition of  $C$ , which correspond to  $uv^i xy^i z$  for  $i = 1, 0, 2$ , respectively.



Theorem 2.34 below formalizes this idea so we can use it to prove that a language is not context free.

**Theorem 2.34.** If  $A$  is a context-free language, then there exists some  $p > 0$  such that for any string  $s \in A$  with  $|s| \geq p$ , there exist substrings,  $u, v, x, y, z$  such that  $x = uvxyz$ ,  $|vy| > 0$ ,  $|vxy| \leq p$ , and for any  $i \geq 0$  we have  $uv^i xy^i z \in A$ .

*Proof.* Assume that  $A$  is recognized by CFG  $G = (V, \Sigma, R, S)$ , and without loss of generality assume that  $G$  is in Chomsky normal form. Choose  $p = 2^{|V|+1}$ . If  $G$  is in Chomsky normal form, then each variable will generate two other variables or a terminal, so there is some path from the root of the parse tree (the start variable) to a character in  $s$  that passes through at least  $|V| + 1$  variable nodes. By the pigeonhole principle, this path includes a repeated variable.

In particular, for  $s \in A$  with  $|s| \geq p$ , there is a substring  $t$  of  $s$  with length  $\leq p$  that corresponds to the terminal set of the subtree of some variable  $V'$  which is repeated in the subtree. We let  $x$  be the substring of terminals generated by the second occurrence of  $V'$ , we let  $v$  and  $y$  be such that  $t = vxy$ , and we let  $u$  and  $z$  be such that  $s = uvxyz$ . By assumption,  $|vxy| \leq p$ , and  $|vy| > 0$  because the substitution of the first occurrence of  $V'$  must have yielded two variables, at least one which is not  $V'$ , and that variable must have derived a non-empty string of terminals.

Replacing the subtree under the first occurrence of  $V'$  with the subtree under the second occurrence of  $V'$  results in the string  $uxz$ , and iteratively replacing the last occurrence of  $V'$  with the subtree generating  $vxy$  yields  $uv^i xy^i z$  for all  $i > 1$ . We have given a value of  $p$  such that any long enough string in the language has a decomposition satisfying the conclusion of the theorem, and the proof is complete.  $\square$

## 2.9.2 Nonpumpability to Prove a Language is Not Context Free

Review our discussion of negating the conclusion of the Theorem 1.70 from Unit 1. We now wish to prove that a language is not context free by showing that for arbitrary  $p$ , there is some particular string  $s$  in the language of length at least  $p$  such that no matter how you try to decompose  $s$  into substrings  $u, v, x, y, z$ , it will not be a pumpable decomposition.

Recall that a good strategy for proving nonregularity was to pick an  $s$  for which the first  $p$  characters were all the same, say  $a$  to allow you to conclude  $y = a^k$  for some  $k = 1, \dots, p$ . But that was only possible because  $|xy| \leq p$  guaranteed that  $y$  lived in the first  $p$  characters. Now the pumpable substrings could live in the beginning, middle, or end of the string; all our guarantee states is that the left and right pumpable substrings are not too far apart.

In the example below, we use this reasoning to say that the first  $b$  block, the middle  $d$  block, and the last  $cb$  block can't be equally represented in the pumpable substrings if each block is sufficiently large. But to do this, we have to break the decomposition possibilities into a few cases.

**Example:**  $\{ab^n d^n (cb)^n e f \mid n \geq 0\}$  is not context free.

*Proof.* Let  $p > 0$  be arbitrary, and pick  $s = ab^p d^p (cb)^p e f$ . Consider any decomposition of  $s$  as  $s = uvxyz$  with  $|vy| > 0$  and  $|vxy| \leq p$ . There are four cases for the contents of  $v$  and  $y$ :

1.  $v$  or  $y$  contains a  $b$  to the left of the  $ds$ . Pumping down results in more  $(cd)s$  than  $bs$  to the left.
2.  $v$  or  $y$  contains a  $b$  or  $c$  to the right of the  $ds$ . Pumping down results in more  $ds$  than  $bs$  or  $cs$  to the right.
3.  $vy$  is all  $ds$ . Pumping down results in fewer  $ds$  than  $bs$  to the left.
4.  $vy$  is some substring of  $aef$ . Pumping up means one of these characters occurs more than once.

Any decomposition falls under one of these cases. None of the cases pump, so the language is not a CFL.  $\square$

**Exercises:** Classify the following languages, and prove your classifications. In Unit 1 we learned that to classify a language as regular, it suffices to provide a DFA, NFA, or regex for it, and to classify a language as nonregular, it suffices to show that the conclusion of the pumping lemma for regular languages does not hold. But now we know that nonregular languages may either be context free or not. If you have shown that a language is nonregular, you now must additionally either provide a CFG or PDA to show that it is context free, or you must show that the conclusion of the pumping lemma for context-free languages does not hold to prove that it is not context free.

- $\{0^n 1^m 0^k 1^\ell \mid n, m, \ell, k \geq 0\}$
- $\{0^n 1^m 0^m 1^n \mid n, m \geq 0\}$
- $\{0^n 1^m 0^n 1^m \mid n, m \geq 0\}$
- $\{1^a 0 1^b 0 1^{ab} \mid a, b \geq 0\}$

### 3 Unit 3: Decidability, Recognizability, and the Limits of Computation

We ended Unit 2 with the pumping lemma for context-free languages, which allowed us to prove, among other things, the inability of PDAs to recognize the language of unary multiplication. Although this is a completely rigorous and logically sound argument for a limit of the PDA model of computation, you know that *computers* exist that can do multiplication. In this final unit, we introduce our most powerful automaton, a Turing machine (TM), whose computation model allows it to accomplish the same tasks as a modern computer.

#### 3.1 The Power of Memory

Recall that we started Unit 2 by giving our Unit 1 machines more power by way of adding a memory stack. Whereas DFAs can only keep track of a finite number of phases the algorithm can be in, PDAs can remember an arbitrarily large number of things about the input digested so far by recording information on a memory stack. However, PDAs are still limited by the requirement that they process input in a single sweep, and memory access is sufficiently restricted that we can't look at memory recorded early without destroying memory recorded later.

Rather than solving these problems separately, a TM combines the role of input reading and memory access. Like all automata seen so far, a TM has a finite set of states that keeps track of what phase of its process it is in, but unlike all automata seen so far, there is no differentiation between input and memory.

A TM has a memory *tape* consisting of infinitely many sequential *cells*. Memory is accessed by a *tape head*, which may move left and right across the tape. Computation begins with the input written on the tape and the tape head on the left most tape cell. Each computation step involves reading the current memory cell, changing states, writing a character to the current memory cell, and shifting the tape head one cell to the left or right.

Then at a high level, a TM can recognize the unary multiplication language as follows:

Multiply( $w$ ):

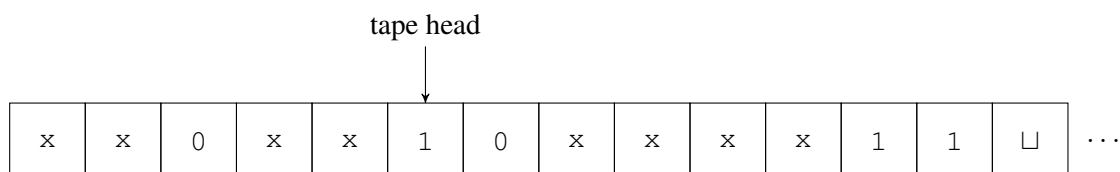
For each of the 1s before the first 0:

Mark it, and for each of the 1s between the 0s, mark it and the next unmarked 1 after the second 0.

Unmark all the 1s between the 0s.

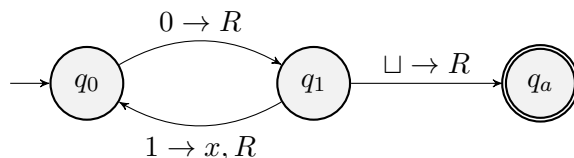
If there is nothing but marked 1s after the second 0: *accept*.

The following diagram indicates the state of memory immediately after the TM described above marks off the second 1 in the second block on the final iteration of the outer for loop:



## 3.2 Turing Machines

Before formalizing our last main type of automaton, let's consider a state diagram of a Turing machine for a regular language that doesn't require all the power of a Turing machine. The machine checks whether a string is of the form  $(01)^n 0$  for some  $n \geq 0$  by scanning memory left to right, marking off all the 1s. Marking 1s is not necessary to check the form of the language, but it helps illustrate TM functionality.



The  $R$ s in the labels indicate that the tape head is always moving right. In general the tape head can move left, which we'll denote with  $L$ . The character read from memory is indicated before the arrow on a transition label. If  $R$  or  $L$  is the only thing after the arrow, then nothing is written, but if there is an additional character, like the  $x$  in the  $q_1$  to  $q_0$  transition, that is the character that replaces the read character.

A *configuration* is a snapshot of the current state of a Turing machine. It lists all the contents of the used portion of memory and indicates the current state of the machine immediately before the memory cell that the tape head currently hovers over. The *accepting configuration history* for this machine on input 010 is as follows:

$q_0 010$   
 $0q_1 10$   
 $0xq_0 0$   
 $0x0q_1$   
 $0x0 \sqcup q_a$

Turing machines are deterministic, but their state diagrams take NFA-style shortcuts, omitting transitions that result in the Turing machine rejecting. For example, the input 000 should be rejected, and the point at which this happens is when, at state  $q_1$  having read the first 0, we are neither done with input nor looking at a 1. Because no 0-labeled arrow is leaving  $q_1$ , there is an undrawn transition from  $q_1$  on 0 to undrawn state  $q_r$ , causing the computation history to end, rejecting the input. The *rejecting configuration history* of the machine on 0000 is:

$q_0 0000$   
 $0q_1 000$   
 $00q_r 00$

Aside from memory, a small but important difference between Turing machines and the automata we've seen so far is that whereas DFAs, NFAs, and PDAs have their states partitioned into accepting and non-accepting states, Turing machines have a single accepting state and a single rejecting state. Computation ends not when the machine finishes reading its input – it's not even clear what this would mean with input being a part of memory and the ability to scan input multiple times – but when it enters an accepting or rejecting state. We are now ready for our formal definitions.

**Definition 3.3.** A Turing machine is a 7-tuple  $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$ , where

$Q$  is a finite state set,

$\Sigma \not\ni \sqcup$  is a finite input alphabet,

$\Gamma \supseteq \Sigma \cup \{\sqcup\}$  is a finite tape alphabet,

$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ , and

$q_0, q_a, q_r \in Q$  with  $q_a \neq q_r$  are the start, accept, and reject states, resp.

A configuration is a string  $uqv$  for  $q \in Q, u, v \in \Gamma^*$ , where  $u$  denotes the memory contents left of the tape head,  $v$  denotes the memory contents under and to the right of the tape head, and  $q$  denotes the current state. For  $a, b, c \in \Gamma, u, v \in \Gamma^*, q_i, q_j \in Q$ ,

$uaq_i bv$  yields  $uq_j acv$  if  $\delta(q_i, b) = (q_j, c, L)$ , or

$uaq_i bv$  yields  $uacq_j v$  if  $\delta(q_i, b) = (q_j, c, R)$ .

The above statements also hold if we remove  $a$  when  $u = \varepsilon$ , representing  $M$  idling at the leftmost memory cell when instructed to move left.

$M$  halts on  $w \in \Sigma^*$  if there is a sequence of configurations  $C_1, \dots, C_k$ ,  $k \geq 1$  such that  $C_1 = q_0 w$  and  $C_i$  yields  $C_{i+1}$  for  $i = 1, \dots, k-1$ , and  $C_k = uq_a v$  or  $C_k = uq_r v$  for some  $u, v \in \Gamma^*$ . If  $C_k = uq_a v$ , we say  $M$  accepts  $w$ . If  $C_k = uq_r v$ , we say  $M$  rejects  $w$ .

The language recognized by TM  $M$ , denoted  $L(M)$ , is the set of strings  $M$  accepts.

With a sketch of an argument that some languages that are not CFLs can be recognized by a TM, we define a new class of languages associated with TMs:

**Definition 3.5.** A language is Turing-recognizable if and only if it is recognized by some TM.

**Exercise:** Give a state diagram for a Turing machine that recognizes valid unary multiplication expressions.

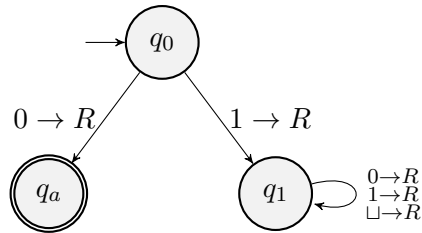
**Exercise:** Describe the language of CFG  $S \rightarrow 1S00 \mid 1S000 \mid \varepsilon$ . Give high-level pseudocode for a Turing machine that recognizes the language, and then provide a state diagram that implements your algorithm.

### 3.3 Decidability, Recognizability, Corecognizability, and the Chomsky Hierarchy

The Turing machines we've seen so far are all special types of Turing machines called deciders because they never get stuck in infinite loops. This means that the languages of these machines are decidable:

**Definition 3.6.** A language is decidable if and only if it is recognized by a TM that halts on all inputs.

The following TM recognizes our very first language from Unit 1, the set of binary strings that start with 0, but it is *not* a decider because it scans right forever on strings that start with 1:



This TM shows the language is Turing-recognizable, but is it also decidable? To show this, we'd have to give a decider, which we can do by omitting  $q_1$  so the machine rejects immediately on any string that does not start with 0.

Because every Turing decider is a Turing machine, all decidable languages are recognizable. But is the converse true? No! It takes some careful logic to prove undecidability, but once we learn how to do this we will be able to describe a language that is demonstrably recognizable but cannot possibly be decidable.

Before we begin our discussion of undecidability, we add one additional class of languages to the mix and describe its relationship to recognizability and decidability:

**Definition.** A language is co-Turing-recognizable if and only if its complement is recognizable.

The relationship between decidability, recognizability, and corecognizability is intuitive and made explicit by the following theorem:

**Theorem 4.22.** A language is decidable if and only if it is Turing-recognizable and co-Turing-recognizable.

You should read Sections 3.2 and 3.3 on your own for the first Unit 3 homework and to get a big-picture sense of the power of the Turing machine computation model. One useful fact is that even though Turing machines have a single tape and are deterministic, we can simulate multiple tapes on a single tape. Reasoning about parallel processing is convenient in the second part of our proof establishing the relationship between decidability, recognizability, and corecognizability.

*Proof of Theorem 4.22.* We prove the two directions of the if and only if separately.

( $\Rightarrow$ ) Assume  $A$  is decided by some TM  $D = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$ . By definition,  $D$  recognizes  $A$ . Flipping the accept and reject states gives  $\bar{D} = (Q, \Sigma, \Gamma, \delta, q_0, q_r, q_a)$ , which recognizes  $\bar{A}$ . Hence  $A$  is both recognizable and corecognizable.

( $\Leftarrow$ ) Assume TM  $R$  recognizes  $A$  and TM  $C$  recognizes  $\bar{A}$ . Any  $w \in \Sigma^*$  causes at least one of  $A$  or  $C$  to halt, so a TM  $D$  can run  $A$  and  $C$  in parallel and it is guaranteed to halt on all inputs and decide  $A$ .  $\square$

More important in Sections 3.2-3 than nondeterminism/parallel computation tricks is the perspective that everything you think of as a (deterministic) algorithm can be implemented – possibly very inefficiently, but we still don't care about efficiency! – as a Turing machine, so we can reason about decidability using algorithms instead of fully-specified state diagrams. In particular, we can use our parsing algorithm from Unit 2 to show that decidable languages contain context-free languages.

**Theorem 4.9.** Any CFL is decidable

*Proof sketch.* Theorem 2.9 showed that every CFL has an associated CFG in Chomsky normal form. An algorithm determine whether a string is in a given CFL could hard-code this grammar, and on input  $w$ , run the CYK algorithm to determine whether the start variable derives  $w$ . This process terminates in finite time with either a yes or no answer, so it describes a decider for the CFL, showing that it is decidable.  $\square$



With Theorem 4.22 and Theorem 4.9, we can draw our full Chomsky hierarchy as follows:

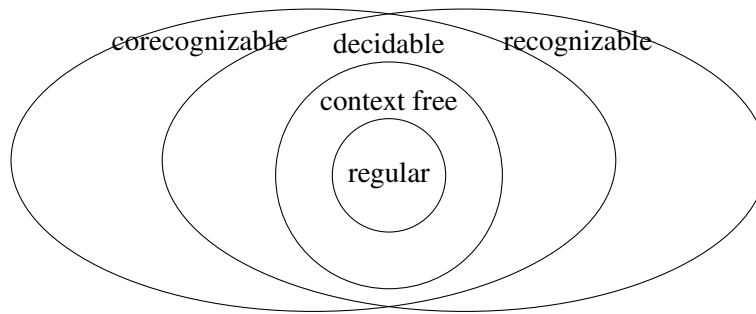


Figure 1: The Chomsky hierarchy

We will turn our focus for the rest of the course first to the decidable segment of the Chomsky hierarchy, and then to the segments that are at most one of recognizable and corecognizable. After discussing encodings of machines as strings, we'll appreciate that all the languages we consider for the rest of the unit will be sufficiently complicated that we *could* use the pumping lemma for CFLs to prove that they are not context free, but we will put these low-level arguments aside to focus on the big picture for the rest of the unit.

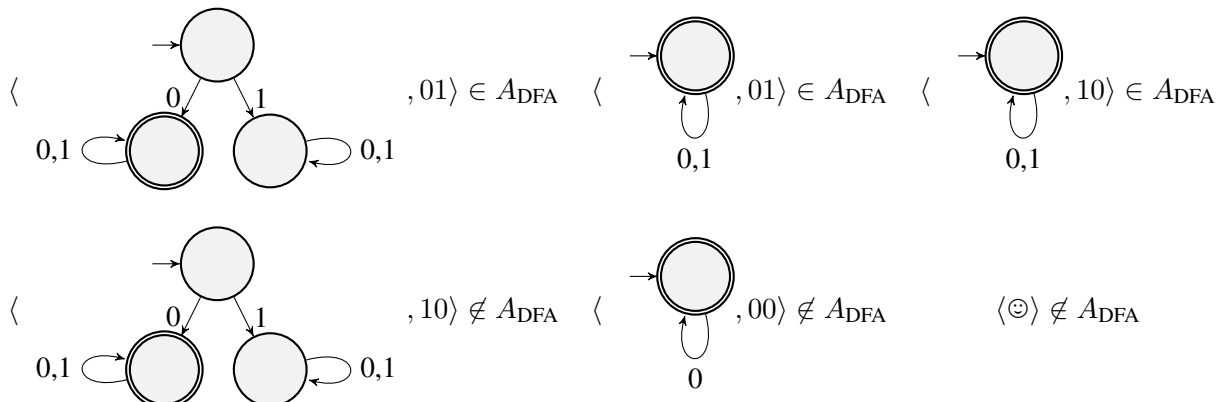
### 3.4 The DFA Acceptance Problem

So far we've given only one example of a language that is decidable but *not* context free: the language of unary multiplication. Now we study a more complicated decidable language to inspire additional languages for high-level decidability and, later, undecidability arguments.

The language makes more sense in the context of its associated decision problem.

**Problem:** Given a DFA and a string, does the DFA accept the string?  
**Language:**  $A_{\text{DFA}} := \{\langle D, w \rangle \mid D \text{ is a DFA and } w \in L(D)\}$

The language is easiest to reason about with some examples of strings in and not in the language. We will talk concretely about how to encode a machine as a binary string, but to keep things more intuitive, we'll just write them as state diagrams for now. The following represent some strings in and not in the  $A_{\text{DFA}}$  language:

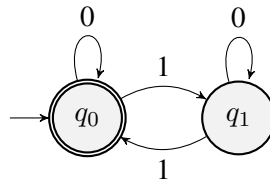


### 3.4.1 Encoding Machines as Strings

To prove that  $A_{\text{DFA}}$  is *decidable* we need to design a Turing machine  $M_{A_{\text{DFA}}}$  that decides it by accepting everything in the top row of examples above and every other element of  $A_{\text{DFA}}$  and rejecting everything in the bottom row of non-examples and every other non-element of  $A_{\text{DFA}}$ . But inputs for  $M_{A_{\text{DFA}}}$  must be binary strings, so we will need rules for encoding each of these diagrams as a binary strings.

This low-level discussion will help us think about languages involving machines as we prepare for higher-level reasoning. Recall that a DFA is a 5-tuple  $(Q, \Sigma, \delta, s, F)$ . Assume  $\Sigma = \{0, 1\}$  and write  $|Q|$  1s followed by a 0 to indicate the number of states. We encode the  $|\Sigma| |Q|$  transitions of  $\delta$  with  $|Q|$  bits each as follows. For a given  $(q_i, a)$  input to  $\delta$  in canonical order, encode  $\delta(q_i, a) = q_j$  with  $|Q| - 1$  zeros and a 1 indicating  $q_j$ . Then encode  $F$  with  $|Q|$  more bits,  $|F|$  of which are 1s. Write  $w$  at the end.

**Example:** Recall our DFA for determining whether there are an even number of 1s:



We encode this along with input 0101 as the following binary string:

11010010110100101

Which bits represents the self-loop at  $q_1$ ?

**Thought experiment:** Why is *no* language that consists of binary encodings of machines context free?

### 3.4.2 Decidability of the DFA Acceptance Problem

To prove decidability of  $A_{\text{DFA}}$ , we must design a Turing machine  $M_{A_{\text{DFA}}}$  that decides it, i.e.,  $M_{A_{\text{DFA}}}$  has to *accept* every string in  $A_{\text{DFA}}$  and *reject* every string not in  $A_{\text{DFA}}$ . Because every algorithm can be implemented as a Turing machine, we can describe  $M_{A_{\text{DFA}}}$  with pseudocode. We write this result and justification as a theorem and proof as follows.

**Theorem 4.1.**  $A_{\text{DFA}}$  is decidable.

*Proof.* Define a Turing machine  $M_{A_{\text{DFA}}}$  as follows:

$M_{A_{\text{DFA}}} =$  On input  $\langle D, w \rangle$ ,

0. If the input is formatted improperly: *reject*.

1. Simulate  $D$  running on  $w$ .

2. If  $D$  ends in an accept state: *accept*.

Else: *reject*.

Note that for any  $\langle D, w \rangle \in A_{\text{DFA}}$ , i.e.,  $w \in L(D)$ , Step 1 will entail simulating  $|w|$  transitions in  $D$  resulting in  $D$  ending in an accept state, so  $\langle D, w \rangle \in L(M_{A_{\text{DFA}}})$ . Conversely, if  $\langle D, w \rangle \notin A_{\text{DFA}}$ , then formatting errors will be caught in Step 0, or Step 1 will terminate in finite time on a non-accepting state. This shows that  $M_{A_{\text{DFA}}}$  halts on every input and  $L(M_{A_{\text{DFA}}}) = A_{\text{DFA}}$ , proving our decidability result.  $\square$

### 3.4.3 Vocabulary Pitfalls

As a *language*,  $A_{\text{DFA}}$  consists of a set of strings. Each string in this particular language happens to encode a *machine*, in this case a DFA  $D$ , along with another string  $w$ . To prove decidability of the  $A_{\text{DFA}}$  language, we need to give another *machine*, in this case a TM  $M_{A_{\text{DFA}}}$  that decides the *language*  $A_{\text{DFA}}$ . Deciding the language means that  $M_{A_{\text{DFA}}}$  must take any input of the form  $\langle D, w \rangle$  and output the answer to the decision question of whether the encoded DFA  $D$  accepts its specified input string  $w$ .

This will only get more complicated as the unit continues! Always practice using course vocabulary correctly, including bundling verbs with the correct type of noun. You could “run  $M_{A_{\text{DFA}}}$ ” because  $M_{A_{\text{DFA}}}$  is an automaton, but you cannot “run  $A_{\text{DFA}}$ ,” because  $A_{\text{DFA}}$  is a language, and languages can be recognized by a machine, but they themselves cannot be run. Likewise, a machine recognizes a language by accepting that set of strings, but it only runs on a single input, so you could “run  $M_{A_{\text{DFA}}}$  on  $\langle D, w \rangle$ ” but you cannot “run  $M_{A_{\text{DFA}}}$  on  $A_{\text{DFA}}$ ,” because a language is not a valid input for a machine.

## 3.5 Decidability Arguments

We reasoned about the finiteness of the DFA simulation process to prove decidability of  $A_{\text{DFA}}$ , and now we will continue to reason about computational models and other properties studied in Units 1 and 2 for several more decidability arguments.

### 3.5.1 White-Box Reductions

Consider a simple modification to the  $A_{\text{DFA}}$  language:

$$A_{\text{NFA}} := \{ \langle N, w \rangle \mid N \text{ is an NFA and } w \in L(N) \}.$$

We can demonstrate the decidability of  $A_{\text{NFA}}$  by *reducing* it to the decidability of  $A_{\text{DFA}}$ , which is guaranteed by Theorem 4.1. The notation  $A_{\text{NFA}} \rightarrow A_{\text{DFA}}$  is read as “the  $A_{\text{NFA}}$  problem reduces to the  $A_{\text{DFA}}$  problem,” and a reduction argument like this consists of the following three steps:

1. Convert the  $A_{\text{NFA}}$  *problem instance* into a related  $A_{\text{DFA}}$  problem instance.

In this case, we are given  $\langle N, w \rangle$ , where  $N$  is an NFA, and we will construct a DFA  $D$  with  $L(D) = L(N)$  and bundle it with  $w$  to create an  $A_{\text{DFA}}$  problem instance  $\langle D, w \rangle$ .

2. Run the  $A_{\text{DFA}}$  problem instance on the  $A_{\text{DFA}}$  decider,  $M_{A_{\text{DFA}}}$ .
3. Convert the answer to the  $A_{\text{DFA}}$  problem instance into an answer to the original  $A_{\text{NFA}}$  problem instance.

In this case,  $D$  accepts  $w$  iff  $N$  accepts  $w$ , so we forward the answer from  $M_{A_{\text{DFA}}}$ .

We could still write down this argumentation if we had not yet established decidability of  $A_{\text{DFA}}$ , but the implication would be different. Rather than using decidability of  $A_{\text{DFA}}$  to conclude decidability of  $A_{\text{NFA}}$ , we would be saying that *if* someone were to figure out how to decide  $A_{\text{DFA}}$ , *then* that algorithm could be used as a *black-box subroutine* in our decider for  $A_{\text{NFA}}$ . The machine  $M_{A_{\text{DFA}}}$  is seen as a black box (one you can’t open up and examine) unless it is written down somewhere.

Of course we *have* written down  $M_{A_{\text{DFA}}}$ , in the proof of Theorem 4.1 establishing decidability of  $A_{\text{DFA}}$ , so our reduction does not just give a hypothetical argument. For anyone who has access to our proof of Theorem 4.1, the argument structure above should be viewed as a *white-box reduction* proving decidability of  $A_{\text{NFA}}$ . We formalize this result and proof below.

**Theorem 4.2.**  $A_{\text{NFA}}$  is decidable.

*Proof.* Define a Turing machine  $M_{A_{\text{NFA}}}$  as follows:

$M_{A_{\text{NFA}}} =$  On input  $\langle N, w \rangle$ , where  $N$  is an NFA,

1. Follow the construction in the proof of Theorem 1.39 to encode a DFA  $D$  with  $L(D) = L(N)$ .
2. Run  $M_{A_{\text{DFA}}}$  from the proof of Theorem 4.1 on  $\langle D, w \rangle$ .
3. If  $M_{A_{\text{DFA}}}$  accepts: *accept*.  
Else: *reject*.

Steps 1 and 3 take finite time, and so does Step 2 (because  $M_{A_{\text{DFA}}}$  is a decider). Hence  $M_{A_{\text{NFA}}}$  decides its language. By construction of  $M_{A_{\text{DFA}}}$ ,  $\langle D, w \rangle$  gets accepted in Step 2 iff  $w \in L(D)$ , so by Step 3,  $\langle N, w \rangle$  gets accepted by  $M_{A_{\text{NFA}}}$  iff  $w \in L(D) = L(N)$  as required.  $\square$

Note that we didn't explicitly write a Step 0 that checks whether the input string is properly formatted and rejects if not; that is implied by the "where  $N$  is an NFA" annotation on the input line. We will use this convention moving forward.

Note also that we could use the exact same reductive reasoning, relying on the proof of Lemma 1.55 that shows how to convert a regex to an NFA and then using the proof of Theorem 1.39 to convert the NFA to a DFA to prove decidability of  $A_{\text{regex}} := \{\langle R, w \rangle \mid R \text{ is a regex and } w \in L(R)\}$ .

### 3.5.2 Decidability of the DFA Language Emptiness Problem

Instead of formalizing decidability of  $A_{\text{regex}}$ , we turn to a language capturing a different property of DFAs.

Problem: Given a DFA, is the language of the DFA empty?  
Language:  $E_{\text{DFA}} := \{\langle D \rangle \mid D \text{ is a DFA and } L(D) = \emptyset\}$

We prove this by applying a breadth-first traversal of the states of an input DFA to see if there exists a sequence of input characters that would be accepted by the machine. This approach doesn't rely on any previous algorithms, so we write it as a non-reductive algorithm.

**Theorem 4.4.**  $E_{\text{DFA}}$  is decidable.

*Proof.* Define a Turing machine  $M_{E_{\text{DFA}}}$  as follows:

$M_{E_{\text{DFA}}} =$  On input  $\langle D \rangle$ , where  $D$  is an NFA,

1. Treating the transitions of  $D$  as directed edges, do a breadth-first traversal of the states of  $D$ , starting from the start state and marking every state that gets visited.
2. If any final state of  $D$  is marked: *reject*.  
Else: *accept*.

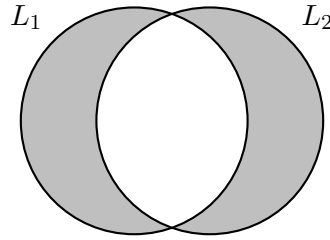
Step 1 takes finite time because it is a traversal over a finite graph, and Step 2 is a simple conditional. Hence  $M_{E_{\text{DFA}}}$  decides its language. The only strings accepted are those for which *no* sequence of characters result in the DFA entering an accept state, certifying emptiness of the language, so  $M_{E_{\text{DFA}}}$  decides  $E_{\text{DFA}}$  as required.  $\square$

### 3.5.3 Decidability of the DFA Equivalence Problem

We consider one additional DFA problem and its associated language:

**Problem:** Given two DFAs, are they equivalent?  
**Language:**  $EQ_{DFA} := \{\langle D_1, D_2 \rangle \mid D_1, D_2 \text{ are DFAs and } L(D_1) = L(D_2)\}$

In general, there will be strings in all four regions of the Venn diagram below depicting the relationship between two languages,  $L_1 = L(D_1)$  and  $L_2 = L(D_2)$ . However, if the two languages are the same, there will be no strings in the (shaded) *symmetric difference* of the two sets, the set of elements in exactly one of the sets.



In other words, asking whether  $D_1, D_2$  are equivalent is the same as asking whether the language of a machine that recognizes  $L(D_1) \oplus L(D_2)$  is empty. The proof of Theorem 4.4 shows us how to recognize whether the language of a machine is empty, pointing us toward a proof by reduction. But given  $D_1, D_2$ , how can we construct a machine that recognizes  $L(D_1) \oplus L(D_2)$ ? Start by observing that we can write the symmetric difference using set operations as follows, noting that  $\cap$  is higher precedence than  $\cup$ :

$$L_1 \oplus L_2 = L_1 \cap \overline{L_2} \cup L_2 \cap \overline{L_1}$$

In Unit 1, we proved several closure properties for regular languages by showing that given two DFAs we can construct a third that recognizes the union or intersection of their languages, and given one DFA we can construct another that recognizes the complement language. We can combine these techniques with  $M_{EQ_{DFA}}$  from the proof of Theorem 4.4 to prove decidability of  $EQ_{DFA}$  as follows.

**Theorem 4.5.**  $EQ_{DFA}$  is decidable.

*Proof.* Define a Turing machine  $M_{EQ_{DFA}}$  as follows:

$M_{EQ_{DFA}} =$  On input  $\langle D_1, D_2 \rangle$ , where  $D_1, D_2$  are DFAs,

1. Construct a DFA  $D$  that recognizes  $L(D_1) \cap \overline{L(D_2)} \cup L(D_2) \cap \overline{L(D_1)}$  using the union, intersection, and complement DFA construction techniques from Unit 1.
2. Run  $M_{EQ_{DFA}}$  on  $\langle D \rangle$ .
3. If  $M_{EQ_{DFA}}$  accepts: *accept*.  
 Else: *reject*.

All steps take finite time, and the emptiness of the symmetric difference language is equivalent to the equivalence of the original DFAs' languages, so  $M_{EQ_{DFA}}$  decides  $EQ_{DFA}$ .  $\square$

### 3.5.4 Decidability of Languages Related to Unit 2 Mechanics

Having problem that the DFA acceptance problem is decidable, we want to do the same for PDA acceptance. We showed  $A_{\text{DFA}}$  was decidable in the proof of Theorem 4.1 by simulating an input DFA running on an input string and reporting whether or not the DFA ended in an accept state.

Why is the picture more complicated for PDAs? Recall that PDAs generalize *nondeterministic* FAs, so the simulation rules are less clear. We could make use of the effective parallel processing power of Turing machine to simulate multiple computation histories at once, but what if the string will never be accepted by the PDA but certain features, like an  $\varepsilon$  self-loop, allow the computation history to go on forever? The simulating Turing machine would not be a decider and we would not be able to conclude decidability. We can gain more traction if we reason about nondeterminism in Turing machines, but this requires a more sophisticated understanding of Turing machine mechanics, and we can avoid this by instead focusing on CFGs.

Once we show CFG acceptance is decidable, we could use our proof sketch for Lemma 2.27 to convert a PDA acceptance problem instance into a CFG acceptance problem instance with the same answer to show decidability of the PDA acceptance. That last sentence is a black-box  $A_{\text{PDA}} \rightarrow A_{\text{CFG}}$  reduction right now, but let's make it white box by proving decidability of  $A_{\text{CFG}}$ !

**Theorem 4.7.** The language  $A_{\text{CFG}} := \{\langle G, w \rangle \mid G \text{ is a CFG and } w \in L(G)\}$  is decidable.

*Proof.* Define a Turing machine  $M_{A_{\text{CFG}}}$  as follows:

$M_{A_{\text{CFG}}} =$  On input  $\langle G, w \rangle$ , where  $G$  is a CFG,

1. Convert  $G$  into an equivalent CFG  $G'$  in Chomsky normal form.
2. Run the CYK parsing algorithm on  $G'$  and  $w$ .
3. If CYK accepts: *accept*.

Else: *reject*.

This machine makes use of the proof of Theorem 2.9, showing how to convert any grammar into Chomsky normal form, as a preprocessing step for the CYK parsing algorithm, which serves as the main subroutine. Because all of these steps terminate in final time and correctly answer the  $A_{\text{CFG}}$  problem,  $M_{A_{\text{CFG}}}$  certifies decidability of  $A_{\text{CFG}}$ .  $\square$

**Exercise:** Compare the statements of Theorem 4.7 and our earlier result Theorem 4.9. Draw a diagram of  $M_{A_{\text{CFG}}}$  illustrating the structure of its inputs, meaning of its outputs, and its general contents. Give an diagram that illustrates Theorem 4.9, and formalize the proof of this result using pseudocode notation like we've been using for our decidability arguments.

Having shown decidability of the CFG acceptance problem, we now consider the CFG language emptiness problem. Recall  $M_{E_{\text{DFA}}}$  determined whether a DFA had an empty language by doing breadth-first search from its start state to see whether a final state was ever visited. Using that as inspiration but modifying it to accommodate the structure of a grammar's rules rather than a DFA state diagram, we construct a decider for  $E_{\text{CFG}}$  that explores the grammar variable-by-variable, starting with variables that generate all-terminal strings and checking whether we ever reach the start variable, indicating that deriving a string of all terminals is possible and hence the language is non-empty.

**Theorem 4.8.** The language  $E_{\text{CFG}} := \{\langle G \rangle \mid G \text{ is a CFG and } L(G) = \emptyset\}$  is decidable.

*Proof.* Define a Turing machine  $M_{E_{\text{CFG}}}$  as follows:

$M_{E_{\text{CFG}}} =$  On input  $\langle G \rangle$ , where  $G$  is a CFG,

1. Mark all terminal characters on the right-hand side of a substitution rule of  $G$ .
2. Iteratively identify and mark variables with a fully marked substitution rule, marking all instances of that variable on in any substitution rule, terminating once no new variables have been marked.
3. If the start variable gets marked: *reject*.

Else: *accept*.

Because variable and rule sets are finite, this traversal will eventually terminate. The start variable gets marked if and only if there is some derivation from it to an all-terminals string, i.e.,  $L(G) \neq \emptyset$ , so our TM  $M_{E_{\text{CFG}}}$  decides the language  $E_{\text{CFG}}$ .  $\square$

**Thought experiment:** Recall that we showed  $EQ_{\text{DFA}} \rightarrow E_{\text{DFA}}$  by constructing a DFA to recognize the symmetric difference of the languages of the two input DFAs, and the emptiness of this difference certified equivalence of the original machines. This reduction established decidability of the DFA equivalence problem. We does the exact same approach not apply to show  $EQ_{\text{CFG}} \rightarrow E_{\text{CFG}}$ , establishing decidability of  $EQ_{\text{CFG}}$ ?

In fact, it turns out the  $EQ_{\text{CFG}}$  is *not* decidable. We will need to build new techniques for undecidability arguments to show this.

### 3.6 The Turing Machine Acceptance Problem

The Turing machine acceptance problem is exactly analogous to the DFA acceptance problems. Given a Turing machine and a string, does the Turing machine accept the string? As a language,

$$A_{\text{TM}} := \{\langle M, w \rangle \mid M \text{ is a Turing machine with } w \in L(M)\}.$$

We can think of Turing machines as arbitrary programs, so decidability of this acceptance problem would have a powerful implication. It would mean that there is some program, say  $M_{A_{\text{TM}}}$ , that can look at any other program  $M$  and input  $w$  and say definitively whether or not  $M$  accepts  $w$ .

#### 3.6.1 The Universal Turing Machine

Our proof of Theorem 4.1 built  $M_{A_{\text{DFA}}}$  to decide  $A_{\text{DFA}}$  by simulating  $D$  and forwarding the result. This idea underlies the universal Turing machine conceived by Alan Turing in the 1930s, well before the existence of modern computers:

**Definition.** The universal Turing machine is the Turing machine  $U$  defined as follows:

$U =$  On input  $\langle M, w \rangle$ , where  $M$  is a Turing machine,

1. Simulate  $M$  running on  $w$ .
2. If  $M$  accepts  $w$ : *accept*.

Else: *reject*.

Why can't we use this as a proof of decidability of  $A_{TM}$ ? Recall that TMs in general may loop on some inputs not in their language. If  $M$  halts on  $w$ , then Step 1 of  $U$  will eventually terminate and  $U$  will correctly identify  $\langle M, w \rangle \in A_{TM}$  or  $\langle M, w \rangle \notin A_{TM}$ . However, if  $M$  loops on  $w$ , Step 1 will never terminate, and  $U$  will loop on  $\langle M, w \rangle$ .  $U$  recognizes  $A_{TM}$ , but this proves *Turing-recognizability* of  $A_{TM}$ , not decidability.

### 3.6.2 Undecidability of the Turing Machine Acceptance Problem

Returning to the DFA acceptance problem, Turing machine  $M_{A_{DFA}}$  had a computational advantage over its input DFAs. By contrast, a hypothetical  $A_{TM}$  decider  $M_{A_{TM}}$  has no computational advantage over its inputs. In fact, it has a handicap:  $M_{A_{TM}}$  must be a decider, whereas its input TMs are allowed to loop.

We will show that no amount of clever TM design can circumvent this handicap. The pumping lemmas for regular and then context-free languages showed certain looping properties of strings in those languages, and then contradicting that property allowed us to show a language was not in the associated class. That approach will not work here, because the infinite memory model of Turing machines means that even long strings may have distinct configurations at every step of computation.

Instead, we will give a different type of proof by contradiction, showing that a black-box  $A_{TM}$  decider  $M_{A_{TM}}$  could be used to build a well-defined Turing machine with ill-defined behavior. The new machine we build will analyze what happens when we feed a program into itself as input. This is an abstract thought experiment that becomes more concrete if we think about compilers. Part of the job of a C++ compiler, for example, is to look at a file and report whether the file encodes a valid C++ program. Compilers themselves are programs, so we could ask a compiler to analyze itself. If our C++ compiler were written in C++, it would accept itself, but if it were written in Python, it would reject itself. This makes perfect sense for compilers, but we will see that the thought experiment becomes a logical contradiction if  $A_{TM}$  were decidable.

**Theorem 4.11.**  $A_{TM}$  is undecidable.

*Proof.* Assume for contradiction that  $A_{TM}$  is decidable by some black-box TM  $M_{A_{TM}}$ . We define another Turing machine  $D$  that uses  $M_{A_{TM}}$  as a black-box subroutine as follows:

$D =$  On input  $\langle M \rangle$ , where  $M$  is a Turing machine:

1. Encode  $M$  with an encoding of itself as a string, i.e.,  $\langle M, \langle M \rangle \rangle$ .
2. Run  $M_{A_{TM}}$  on  $\langle M, \langle M \rangle \rangle$ .
3. If  $M_{A_{TM}}$  accepts: *reject*.  
Else: *accept*.

$D$  is a decider because  $M_{A_{TM}}$  terminates in finite time. In reversing the output of  $M_{A_{TM}}$ , the language of  $D$  is the set of programs that do not accept themselves. Now consider whether  $D$  is in its own language.

If  $\langle D \rangle \in L(D)$ , that means  $D$  accepted at Step 3. Working backwards, it must be that  $M_{A_{TM}}$  rejected  $\langle D, \langle D \rangle \rangle$ . But this would contradict the assumed correctness of  $M_{A_{TM}}$ , which is supposed to accept machines bundled with inputs in their language. On the other hand, if  $\langle D \rangle \notin L(D)$ , that means  $D$  rejected at Step 3 because  $M_{A_{TM}}$  accepted  $\langle D, \langle D \rangle \rangle$ , which would contradict the assumed correctness of  $M_{A_{TM}}$ , which is supposed to reject machines bundled with inputs not in their language.

Either case results in a contradiction, so a decider with  $D$ 's functionality cannot exist, meaning that our assumption of  $A_{TM}$ 's decidability must have been false.  $\square$

**Exercise:** Where in the Chomsky hierarchy does  $A_{TM}$  fall, and what are the components of the proof?



### 3.7 Undecidability Arguments

The logical structure of the proof of Theorem 4.11 is that by assuming decidability of  $A_{TM}$ , we can build a machine to do an impossible task – e.g., deciding whether a given machine does not accept itself – allowing us to conclude that  $A_{TM}$  is undecidable. In other words, the task of deciding whether a machine does not accept itself *reduces* to deciding whether a machine accepts a given input. Because the former is logically impossible, so must be the latter.

We can prove undecidability of a given language  $L$  by assuming it is decidable, and using that to build a machine that does a known impossible task. Deciding  $A_{TM}$  is now a known impossible task! So we can prove that  $L$  is undecidable by showing  $A_{TM} \rightarrow L$ , or  $L_{undec} \rightarrow L$  for any known undecidable language  $L_{undec}$ .

#### 3.7.1 Black-Box Reductions and Undecidability of the Halting Problem

Our first undecidability reduction will be  $A_{TM} \rightarrow HALT_{TM}$ , where  $HALT_{TM}$  is the language associated with the halting problem.

**Problem:** Given a Turing machine and a string, does the machine halt on the string?  
**Language:**  $HALT_{TM} := \{\langle M, w \rangle \mid M \text{ accepts } w \text{ or } M \text{ rejects } w\}$

To give an  $A_{TM} \rightarrow HALT_{TM}$  reduction, we have to consider how knowing whether or not a machine halts on a string can tell us whether or not it accepts the string. We control problem instance conversion and answer conversion, and because the problem instances are similarly structured, we decide to backload the work, using membership in  $HALT_{TM}$  as an indicator of the safety of simulating the machine running on is input to see whether it accepts or rejects.

**Theorem 4.2.**  $HALT_{TM}$  is undecidable

*Proof.* ( $A_{TM} \rightarrow HALT_{TM}$ ) Assume TM  $M_{HALT}$  decides  $HALT_{TM}$  and construct TM  $D$  as follows:

$D =$  On input  $\langle M, w \rangle$ , where  $M$  is a TM,

1. Keep  $\langle M, w \rangle$  on the input tape.
2. Run  $M_{HALT}$  on  $\langle M, w \rangle$ .
3. If  $M_{HALT}$  accepts:
  - Simulate  $M$  running on  $w$ .
  - If  $M$  accepts: *accept*.
  - Else: *reject*.
- Else: *reject*.

$D$  is a decider because Steps 1 and 2 terminate in finite time by the assumption that  $M_{HALT}$  is a decider, and Step 3 terminates in finite time because simulating  $M$  on  $w$  only possibly loops forever in cases that  $M_{HALT}$  would reject. Moreover it decides the correct language, because acceptance occurs exactly when  $M$  halts on  $w$  and simulation results in an acceptance. Existence of  $D$  contradicts undecidability of  $A_{TM}$ , so we conclude that our assumption about the decidability of  $HALT_{TM}$  was false.  $\square$

**Comprehension check:** What piece of knowledge do we combine with an  $A \rightarrow B$  reduction to conclude that  $B$  is undecidable? What is the black box in the reduction?

### 3.7.2 Undecidability of the TM Language Emptiness and Equivalence Problems

We likewise give a reduction  $A_{TM} \rightarrow E_{TM}$  to show undecidability of the TM language emptiness problem. This reduction relies on a new technique of modifying the input program itself to produce a new input program for the black-box decider.

To make sense of this, imagine a program `virus_injector` that, when provided with code for a mobile app returns a modified app with extra code added to install a virus. Although the app is a program itself, the `virus_injector` program does not run the app – it changes its functionality. Suppose another program `virus_detector` checks apps for viruses. We treat it as a black box in that we make no assumptions about how it checks code for viruses. The black-box `virus_detector` may or may not run its input programs on certain input strings, but based only on its functionality, we can know that  $\text{virus\_injector}(a) \in L(\text{virus\_detector})$  without ever running app  $a$ .

The  $A_{TM} \rightarrow E_{TM}$  reduction below converts an  $A_{TM}$  input  $\langle M, w \rangle$  into an  $E_{TM}$  input  $\langle M' \rangle$  by programming  $M'$  to depend on  $M$  and  $w$  in such a way that  $L(M') = \emptyset$  if and only if  $w \notin L(M)$ .

**Theorem 5.2.** The language  $E_{TM} := \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$  is undecidable.

*Proof.* ( $A_{TM} \rightarrow E_{TM}$ ) Assume TM  $M_{E_{TM}}$  decides  $E_{TM}$  and construct TM  $D$  as follows:

$D =$  On input  $\langle M, w \rangle$ , where  $M$  is a TM,

1. Construct TM  $M'$  based on  $M$  and  $w$  as follows:

$M' =$  On input  $w'$ ,  
Run  $M$  on  $w$ .  
If  $M$  accepts: *accept*.  
Else *reject*.

2. Run  $E_{TM}$  on  $\langle M' \rangle$ .
3. If  $E_{TM}$  accepts: *reject*.  
Else: *accept*.

$D$  does not run  $M$  Step 1, rather it encodes it into another Turing machine  $M'$  that has the property that  $L(M') = \Sigma^*$  if  $M$  accepts  $w$  and  $L(M') = \emptyset$  if  $M$  either loops on or rejects  $w$ . Step 2 identifies which case we are in, and Step 3 reverses the answer to correctly decide  $A_{TM}$ . Because  $A_{TM}$  is undecidable, we conclude that the TM  $M_{E_{TM}}$  deciding  $E_{TM}$  cannot exist.  $\square$

Our next reduction proving undecidability of the TM equivalence problem instead reduces from  $E_{TM}$ .

**Theorem 5.4.** Language  $EQ_{TM} := \{\langle M_1, M_2 \rangle \mid M_1, M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$  is undecidable.

*Proof.* ( $E_{TM} \rightarrow EQ_{TM}$ ) Assume TM  $M_{EQ_{TM}}$  decides  $EQ_{TM}$  and construct TM  $D$  as follows:

$D =$  On input  $\langle M \rangle$ , where  $M$  is a TM,

1. Construct TM  $M_\emptyset$  whose start state is its reject state, i.e., it immediately rejects any input.
2. Run  $M_{EQ_{TM}}$  on  $\langle M, M_\emptyset \rangle$ .
3. If  $M_{EQ_{TM}}$  accepts: *accept*.  
Else: *reject*.

$D$  halts in finite time and  $L(M) = L(M_\emptyset)$  iff  $L(M) = \emptyset$ , i.e.,  $D$  decides  $E_{TM}$ , so  $EQ_{TM}$  is undecidable.  $\square$

### 3.8 Recognizers for Undecidable Languages

So far we have shown that  $A_{TM}$ ,  $HALT_{TM}$ ,  $E_{TM}$ ,  $EQ_{TM}$  all live outside the decidable segment of the Chomsky hierarchy (Figure 1). We also know that the universal Turing machine recognizes  $A_{TM}$ , classifying it in the recognizable but not corecognizable wedge. In which of the three undecidable regions does each of the other languages live?

First we argue that  $HALT_{TM}$  is recognizable. We do this by just slightly modifying the universal Turing machine. Instead of checking whether  $M$  accepts or rejects  $w$  after it (maybe) terminates, simply accept if the simulation stops. This machine *never* rejects inputs, but because machines that do not halt on their input get stuck in simulation, they will never be accepted, and the correct language will be recognized.

Next we argue that  $E_{TM}$  cannot be recognizable. That is because our reduction from the proof of Theorem 5.2 flips an  $E_{TM}$  answer to give an  $A_{TM}$  answer. Suppose instead of trying to decide  $A_{TM}$ , we were trying to recognize  $\overline{A_{TM}}$ , which is also impossible. Convert the input in the same way and forward  $E_{TM}$  acceptances as  $\overline{A_{TM}}$  acceptances. This uses an  $E_{TM}$  recognizer to recognize  $\overline{A_{TM}}$ , contradicting the recognizability of  $E_{TM}$ .

We now know that  $E_{TM}$  *cannot* live in the recognizable-but-not-decidable wedge with  $A_{TM}$  and  $HALT_{TM}$ , but does it corecognizable or not? A recognizer for  $\overline{E_{TM}}$  establishes corecognizability of  $E_{TM}$ , which we provide now. A recognizability proof is the same as a decidability proof, but we relax the requirements of the TM we design to allow it to loop forever on inputs that it is not required to accept. In particular, if  $L(M) = \emptyset$ , our recognizer for  $\overline{E_{TM}}$  will keep testing inputs forever.

**Theorem.**  $E_{TM}$  is co-Turing-recognizable.

*Proof.* Define a Turing machine  $C_{E_{TM}}$  as follows:

$C_{E_{TM}}$  = On input  $\langle M \rangle$ ,

0. If  $M$  is not a properly formatted TM: *accept*.

1. For  $n = 0, 1, \dots$ :

For all  $w$  of length  $\leq n$ :

Simulate  $M$  on  $w$  for up to  $n$  steps.

If  $M$  accepts within  $n$  steps: *accept*.

We want to show that  $L(C_{E_{TM}}) = \overline{E_{TM}}$ . Step 0 correctly accepts everything that is not even a well-formatted input for the  $E_{TM}$  problem. The only other strings it should accept are TMs that accept at least one input.  $C_{E_{TM}}$  certainly does not accept anything else, because it only accepts after simulating an accepting computation history. If  $M$ 's language is not empty,  $n$  will eventually grow large enough to find one of its elements in the loop, and  $C_{E_{TM}}$  will correctly accept.  $\square$

We can similarly argue that  $EQ_{TM}$  cannot be recognizable, because an  $EQ_{TM}$  recognizer would contradict the unrecognizability of  $E_{TM}$ . But can we modify our  $\overline{E_{TM}}$  recognizer to build a  $\overline{EQ_{TM}}$  recognizer, showing  $EQ_{TM}$  is corecognizable? We might try simultaneously running two machines on inputs of increasing size, and accepting the machine pair as non-equivalent if one accepts a string that the other does not accept. The problem with this is that after a finite number of steps, we don't know whether non-acceptance means that the machine is looping forever or just hasn't accepted yet. In fact we will see that  $EQ_{TM}$  is our first language that is neither recognizable nor corecognizable, and the framework of mapping reductions will help us establish its non-corecognizability.

### 3.9 Mapping Reductions

Let's take a step back and look at the logic of our classification of  $E_{TM}$ . Before we ever proved that it was corecognizable, we argued that our  $A_{TM} \rightarrow E_{TM}$  reduction meant that it could not possibly be recognizable. That is because the simplicity of Step 3 in our  $A_{TM} \rightarrow E_{TM}$  proof of Theorem 5.2 means that this reduction can be reframed as a **mapping reduction**  $\overline{A_{TM}} \leq_m E_{TM}$ , which formalizes the idea that recognizability of  $E_{TM}$  would contradict unrecognizability of  $\overline{A_{TM}}$ . Intuitively, a mapping reduction is just a reduction in which Step 3 forwards the answer from the black box that was run in Step 2. The formal definition is below.

**Definitions 5.17 and 5.20.** A function  $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$  is a **computable function** if there exists a Turing machine that, on any input  $w \in \Sigma^*$ , halts with  $f(w)$  on its tape.

Language  $A$  is **mapping reducible** to language  $B$ , written  $A \leq_m B$  if there is a computable function  $f$  such that  $w \in A \iff f(w) \in B$  for any  $w \in \Sigma^*$ . If so,  $f$  is called the **reduction** from  $A$  to  $B$ .

In other words,  $f$  handles Step 1, which is converting an  $A$  input  $w$  into a  $B$  input  $f(w)$ . Step 2 is to run an assumed  $B$  decider or recognizer on  $f(w)$ , and then Step 3 is to forward the answer from Step 2, which gives the correct answer to the  $A$  problem because  $w \in A \iff f(w) \in B$ .

**Example:** With purely syntactic modifications to the proof of Theorem 5.2, we can argue that  $\overline{A_{TM}} \leq_m E_{TM}$  with reduction  $f(\langle M, w \rangle) = \langle M' \rangle$  for  $M'$  as in our original reduction, with the technical caveat that improperly formatted  $M$  should be converted to  $M_\emptyset$  so that  $\overline{A_{TM}}$  answers line up with  $E_{TM}$  answers.

Why does this mapping reduction show that  $E_{TM}$  is not recognizable? Because an  $\overline{A_{TM}}$  recognizer could be built by applying  $f$  to its input and running an  $E_{TM}$  recognizer, and  $\overline{A_{TM}}$  is unrecognizable. Theorem 5.28 and its contrapositive Corollary 5.29 in the text generalize this. Theorem 5.22 and Corollary 5.23 are special cases of decidability implications we've already discussed for reductions in general.

**Exercise:** Show  $A_{TM} \leq_m HALT_{TM}$ . Why do we have to modify our  $A_{TM} \rightarrow HALT_{TM}$  reduction?

**Exercise:** For your homework you will write the proof of Theorem 5.3 in the book establishing the undecidability of the language  $REGULAR_{TM}$  as a mapping reduction. What additional work is needed to fully classify  $REGULAR_{TM}$ ?

#### 3.9.1 Unrecognizability of the TM Equivalence Problem and its Complement

Our simple  $E_{TM} \rightarrow EQ_{TM}$  reduction from the proof of Theorem 5.4 shows  $E_{TM} \leq_m EQ_{TM}$  with reduction  $f(\langle M \rangle) = \langle M, M_\emptyset \rangle$ . Unrecognizability of  $E_{TM}$  implies unrecognizability of  $EQ_{TM}$ , but we still don't know whether  $EQ_{TM}$  is corecognizable. To show that it is not, we would need to mapping reduce to  $EQ_{TM}$  from a language that is not corecognizable. The proof of Theorem 5.30 chooses  $A_{TM}$  for this second reduction.

**Theorem 5.30.**  $EQ_{TM}$  is neither recognizable nor corecognizable.

*Proof.* We give one mapping reduction to show unrecognizability and another to show counrecognizability.

$E_{TM} \leq_m EQ_{TM}$  with reduction  $f(\langle M \rangle) = \langle M, M_\emptyset \rangle$ .  $E_{TM}$  is not recognizable, so neither is  $EQ_{TM}$ .

$A_{TM} \leq_m EQ_{TM}$  with reduction  $f(\langle M, w \rangle) = \langle M', M_{\Sigma^*} \rangle$ , for  $M'$  a TM that runs  $M$  on  $w$  and forwards  $M$ 's output as in the proof of Theorem 5.2 and for  $M_{\Sigma^*}$  a TM that accepts all inputs. This is a mapping reduction because  $L(M') = \Sigma^* \iff w \in L(M)$ .  $A_{TM}$  is not corecognizable, so neither is  $EQ_{TM}$ .  $\square$

### 3.10 Computation Histories and Linear Bounded Automata

At this point we have fully classified the acceptance, language emptiness, and equivalence problems for DFAs, CFGs, and TMs, with the exception of  $EQ_{CFG}$ . In your homework, you were asked to show that  $EQ_{CFG}$  is corecognizable by giving a recognizer inspired by our  $\overline{E_{TM}}$  recognizer.

A pre-recorded lecture shows that  $EQ_{CFG}$  is undecidable by giving a reduction from  $A_{TM}$  that, given  $\langle M, w \rangle$ , constructs a PDA that accepts everything but accepting computation histories of  $M$  on  $w$ .

Theorem 5.10 uses a simpler application of the technique of reasoning about computation histories by studying **linear bounded automata** (LBAs), a special type of Turing machine with bounded memory:

**Definition 5.6.** A **linear bounded automaton** is a Turing machine whose tape head is not allowed to leave the portion of the tape initially occupied by input.

We're omitting discussion of what happens when the tape is at the rightmost memory cell and instructed to move right, but we don't need that much detail for our purposes. The key distinction between LBAs and TMs is that a TM's infinite memory means that there may be infinitely many *distinct* configurations running on a particular input, whereas an LBA  $M$  has a finite number of configurations, which we can quantify by reasoning about all possible states  $M$  could be in, tape head locations, and character sequences on the tape:

**Lemma 5.8.** For LBA  $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$  and string  $w \in \Sigma^*$ ,  $M$  running on  $w$  has at most  $|Q| |w| |\Gamma|^{|w|}$  distinct configurations.

Like TMs, LBAs may loop forever, but unlike TMs, we can detect LBA loops. The pigeonhole principle says that if an LBA runs for more than  $|Q| |w| |\Gamma|^{|w|}$  steps, there is a repeated configuration and the LBA will never break free of that loop. This allows us to decide  $A_{LBA}$  whereas we couldn't decide  $A_{TM}$ .

**Theorem 5.9.** The language  $A_{LBA} := \{\langle M, w \rangle \mid M \text{ is an LBA and } w \in L(M)\}$  is decidable.

*Proof.* Define a TM  $M_{A_{LBA}}$  as follows:

$M_{A_{LBA}} =$  On input  $\langle M, w \rangle$ , where  $M$  is an LBA,

1. Run  $M$  on  $w$  for  $|Q| |w| |\Gamma|^{|w|}$  steps, where  $Q, \Gamma$  are the state set and memory alphabet of  $M$ , resp.
2. If  $M$  has accepted: *accept*.

Else: *reject*.

By Lemma 5.8,  $M_{A_{LBA}}$  only needs finite time to check whether LBA  $M$  accepts  $w$ , deciding  $A_{LBA}$ . □

The ability to identify loops doesn't help us brute force an answer to the LBA language emptiness problem, because a brute force approach would still entail simulating on an infinite number of strings even if the length of each simulation could be bounded. However, this is just intuition about why one particular approach won't work. To prove undecidability, we need a reduction. Recall that our  $A_{TM} \rightarrow E_{TM}$  reduction took  $A_{TM}$  candidate string  $\langle M, w \rangle$  and encoded an  $M'$  that simulated  $M$  on  $w$ , resulting in  $L(M') = \emptyset \iff w \notin L(M)$ . This approach won't prove  $E_{LBA}$  is undecidable, because  $M'$  is not an LBA: simulating  $M$  on  $w$  requires more than 0 memory cells, which is all  $M'$  would have access to on input  $w' = \varepsilon$ , noting that  $|\varepsilon| = 0$ . Instead, we want an LBA that somehow checks for a property of  $M, w$  embedded in its input so that it doesn't need extra memory. Of all possible input strings  $\Sigma^*$ , one of them *may* encode a configuration sequence that is an accepting computation history of  $M$  on  $w$ . This is the property that LBA  $M'$  checks for in the reduction below, which we present as a mapping reduction for practice.

**Theorem 5.10.** The language  $E_{\text{LBA}} := \{\langle M \rangle \mid M \text{ is an LBA and } L(M) = \emptyset\}$  not recognizable.

*Proof.*  $\overline{A_{\text{TM}}} \leq_m E_{\text{LBA}}$  with the reduction  $f(\langle M, w \rangle) = \langle M' \rangle$ , where  $M'$  is an LBA that only accepts inputs that encode an accepting computation history of  $M$  on  $w$ . There is one such input if  $w \in L(M)$  and none if  $w \notin L(M)$ , so  $\langle M' \rangle \in E_{\text{LBA}} \iff \langle M, w \rangle \in \overline{A_{\text{TM}}}$ .  $E_{\text{LBA}}$  can't be recognizable because  $\overline{A_{\text{TM}}}$  isn't.  $\square$

Corecognizability of  $E_{\text{LBA}}$  follows using the recognizer for  $E_{\text{TM}}$  because LBAs are just special cases of TMs, and this completes our classification of  $E_{\text{LBA}}$ .

You will think about the differences between TMs and LBAs again in classifying  $EQ_{\text{LBA}}$  as part of your final homework, though doing so does not explicitly require reasoning about computation histories.