Models of Computation*

November 26, 2001

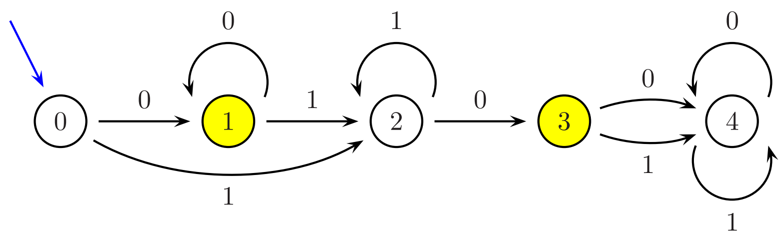
Abstract

This document supplements the COS 126 course material on regular expressions, grammar, and abstract machines.[1] The study of formal languages and abstract machines is central to understanding the *science* in computer science. In this document we address the following fundamental questions:

- What is computation?
- What common features are found in all modern computers?
- Is there a simple model of computation that encompasses all known physical machines?
- Which problems can machines solve?
- Is it possible to design fundamentally more powerful machines?
- How is a computer able to "understand" the programs we write?

To address these questions, we first develop a rigorous notion of computation using formal languages. We use spoken and written language to communicate ideas. Language serves the same purpose for computation – it gives us a rigorous method to state the computational problems that we wish to solve. Programming languages provide a convenient, yet formal, method for describing solution methods.

Once we can precisely express a computation problem using formal language, our goal is to solve it on on a computer. Physical machines are extremely complicated and difficult to analyze. Abstract machines are simplified and idealistic versions of physical machines. By stripping away the unimportant details, we expose the essence of physical machines. This enables a rigorous study of machines, and reveals a surprising commonality among all known physical machines.



[1]Highly motivated students with a strong mathematical background are referred to Sipser [1] for a more in depth treatment of this material.

Contents

1 Formal Languages

Webster[2] defines a langauge as *a systematic means of communicating ideas or feelings by the use of conventionalized signs, sounds, gestures, or marks having understood meaning.* Formal languages associate meaning to symbols, and each language can express its own set of ideas. In order to solve problems on computers, we use language (e.g., C, Java, TOY) to express our algorithmic ideas. Unlike the English language where words and sentences can have multiple meanings, programming languages must be very rigid and unambiguous to ensure that computers do exactly what we program them to do. Yet, they must be flexible enough to allow us to express our algorithmic ideas with ease. In addition, the language must be simple enough so that we can build compilers that will efficiently translate our programs into machine language.

We now define what we mean by a *formal language*. An *alphabet* is a finite set of symbols or characters. The *binary alphabet* is $\{0, 1\}$: it contains the two characters 0 and 1. Computers naturally use this alphabet. A more familiar alphabet consists of the 26 characters: $\{a, b, c, \ldots , z\}$. A *string* over an alphabet is a finite (and ordered) sequence of characters from that alphabet. For example, 01110 is a string over the binary alphabet, *hello* is a string over the alphabet consisting of lower case letters. A *bit* is either 0 or 1 and a *bit string* is a string over the binary alphabet. The *empty string*, which we denote by ϵ, is the string with 0 characters. It plays the role of 0 in a number system. A *language* over an alphabet is an (unordered) collection of strings. E.g., the following is a language over the binary alphabet: $\{01, 111, 10001\}$. Languages can (and often do) contain infinitely many members, e.g., $\{11, 101, 1001, 10001, \ldots\}$ and $\{\epsilon, 01, 0101, 010101, \ldots\}$. They can also be defined implicitly:

- all bit strings with more 1's than 0's
- all positive integers divisible by 17
- all prime numbers
- all ASCII text files that are *legal* C programs, i.e., compile without error
- all 100-city traveling salesperson instances with a tour length of less than 17

There are two complementary language problems that we will address. First, given a language, find a formal and concise way to express it. We will use regular expressions and grammar to do this. For example, we use "context-free grammar" to rigorously define what it means to be a legal C program. Second, given an input string and a formal language, identify whether it is a member of some language. We will develop abstract machines that answer this type of question. For example, a compiler determines whether or not some text file is a valid C program. To do this, it simulates the behavior of a "pushdown automata machine."

2 Regular Expressions (RE)

A *regular expression* is a concise description of a (possibly infinite) language. Languages that can be generated using regular expressions are called *regular languages*. We define regular expressions later, but as an example, the collection of all bit strings that contain 110 as a substring is a regular language, i.e., $\{110, 0110, 1101, 1110, \ldots\}$. Using a different alphabet, regular expressions are useful for finding a particular character pattern in a large document, e.g., searching for all Web documents that contain both the string "Gates" and the string "Macintosh". On Unix systems, you can use the program `grep` to find patterns in your documents, e.g., to find all occurrences of the string `next` in your program `tour.c`, you could type `grep "next" tour.c`. Here's a recursive definition of a *regular expressions*:

1. **base case:** any character in the alphabet is a RE, e.g., in the binary alphabet, 0 and 1 are RE's
2. **parentheses (grouping):** if a is a RE then so is (a)
3. **product (concatenation):** if a and b are RE's then so is ab
4. **plus (or):** if a and b are RE's then so is $a + b$

[2] *Merriam-Webster's Collegiate Dictionary*, tenth edition. Merriam-Webster, Inc., Springfield, Massachusetts, 1993.

5. **closure (replication):** if a is a RE then so is $a*$
6. **the empty string:** we denote by ϵ

Although all of the rules are defined formally, we associate a meaning and interpret each of the basic operations as described below. For convenience, we will assume that we are working with the binary alphabet, i.e., all characters are 0 or 1.

The *product* rule allows regular expressions to concatenate or piece together different regular expressions, lining them up one after the other. E.g., since $a = 0$ and $b = 1$ are regular expressions, we can concatenate them together to form $c = ab = 01$, and conclude that 01 is also a RE. By repeatedly concatenating RE's, we deduce that $d = ccaa = 010100$ is also a RE. The regular expression d matches precisely those strings in the language $\{010100\}$. (Not all languages are particularly interesting.)

The *plus* operation gives regular expressions the power to express the logical OR function. As an example, let $a = 001, b = 111$, and $c = 00$ be regular expressions. Then $x = a + b + c = 001 + 111 + 00$ is a regular expression: it matches precisely those strings in the language $\{001, 111, 00\}$.

The *closure* operation gives regular expressions the power to make zero or more replications of itself. As an example, let $a = 110$. Then $a*$ is a RE: it matches precisely those strings in the language $\{\epsilon, 110,$ $110110,$ $110110110, \ldots\}$. Note that this language has infinitely many strings. Here the empty string ϵ is included since zero replications are allowed.

The *parentheses* operation allows us to combine and group different regular expressions in any order we specify. As an example, the RE $(111 + 00)(101 + 010 + 0)$ generates the language { 000, 00010, 00101, 1110, 111010, 111101}, whereas the RE $(111 + 00101) + (010 + 0)$ generates {0, 010, 00101, 111}. As another example, the RE $110 + 1*$ generates the language $\{\epsilon, 110, 1, 11, 111, \ldots\}$, whereas the RE $(110 + 1)*$ is the same as $\epsilon + (110 + 1) + (110 + 1)(110 + 1) + (110 + 1)(110 + 1)(110 + 1) + \ldots$ and generates the language $\{\epsilon, 1, 110, 110110, 1101, 1110, 11, \ldots\}$.

Each of these rules by itself seems simplistic and not so interesting. However, when we combine the rules together we can obtain much more complicated and diverse RE's. Here are some examples.

- (10)*1
  - `yes`: 1, 101, 10101, 1010101, ...
  - `no`: the empty string, 0, 00, 01, 11, 0101, and any other strings not like the above pattern
- (0+011+101+110)*
  - `yes`: the empty string, 0, 00, 000, 011, 101, 110, 0011, 0101, 0110, 1010, 1100, 011101, 011110, and many more
  - `no`: 1, 01, 10, 11, 111, 001, 010, 100, any string with more than four consecutive 1's, and others
- (1*01*01*01*)*1*
  - `yes`: the empty string, 000, 0100, 0010, 0001, 000000, 0000111010111, 11111, and any string that contains a multiple of three 0's
  - `no`: 0000, 1001, 100111000, and any string that doesn't contain a multiple of three 0's

We're used to $a + b$ meaning addition and ab meaning multiplication, so why do we re-use this notation for OR and concatenation? Many of the properties you are used to for addition and multiplication carry over to regular expressions. For example, the RE $a + b$ generates the same language as $b + a$.[3] But not all of the usual arithmetic properties carry over, so be careful, e.g., ab and ba are, in general, not the same.

Much like programming, the best way to learn how to create and understand regular expressions is by doing examples. There are many problem with solutions later in this text. Try to answer them on your own, if you get stuck, look at the solutions.

[3]Also, $a(bc)$ generates the same language as $(ab)c$ and $a(b + c)$ is the same as $ab + ac$. Here are a few identities associated with the closure operation. Re-applying the closure operation has no effect: $(a*)* = a*$, $a * a* = a*$, $(a * + b*)* = (a + b)*$. For example, $(0*) * (0 * + (10)*)* = 0 * (0 + 10)*$. Also $a + a* = a*$, but $aa* \neq a*$ since the latter includes the empty string.

3 Grammar

A grammar is a formal set of rules that generates all of the "sentences" or "strings" in a language, much like a regular expression. Grammar rules are more general than rules for generating regular expression; hence more complex languages can be expressed using grammar than via RE. Grammar is widely used by linguists to study the underlying structure and meaning of the English language. To effectively translate text between English and French, you can't simply translate on a word-by-word basis. Rather, you have to look at each sentence, and parse it into underlying units (noun clause, verb, etc.) Computer scientists use grammar to translate between computer languages. The C compiler `lcc` translates C code into a machine language. The C grammar rules are precisely specified using grammar (e.g., see Appendix A13, Kernighan and Ritchie), and this is the fundamental link between C programs and the machine.

A *grammar* over an alphabet consists of four elements: terminals, nonterminals, start symbol(s), and production rules.

- **terminals:** characters in the alphabet, e.g., 0 and 1 for the binary alphabet. We use lower case letters (e.g., a, b, c) to represent terminals.

- **nonterminals:** play the role of "variables" in grammar. The course packet uses angle brackets to delimit a nonterminal, e.g., <A>, <B>, <pal>. We also use upper case letters (e.g., A, B, C) to denote nonterminals. Nonterminals are not characters in the alphabet.

- **start symbol(s):** one (or more) special nonterminal

- **production rules:** replacement rules, e.g., the production rule $A \to a$ means that whenever we see the nonterminal A, we could replace it with the terminal a. Production rules can be more complicated. The rule $AbcDA \to EAbc$ means that whenever we see $AbcDA$, we could replace it by $EAbc$. So we could replace $WxYzAbcDAbcDA$ with $WxYzEAbcbcDA$ or, alternatively, with $WxYzAbcDEabc$. The LHS of the production rule must have at least one nonterminal.

A string is *generated* from the grammar if there is a sequence of production rules that produces that string, starting from a start symbol. Note that only terminals are characters in the alphabet. So, you must keep applying production rules until you only have terminals, and have removed all of the nonterminals. (This is where the name terminals comes from; do not confuse terminals with meaning the first or last characters in a string.) The *language generated from the grammar* is the set of all strings that can be generated from the grammar. To make this concrete, we consider a few examples. The first one is a (simplified) grammar to construct English sentences taken from Sipser [1].

terminals	a, the, boy, girl, flower, touches, likes, sees, with
nonterminals	<sentence>, <article>, <noun>, <verb>, <prep>, <cmplx-verb>
	<noun-phrase>, <verb-phrase>, <cmplx-noun>, <prep-phrase>
start	<sentence>

Rules		
<sentence>	→	<noun-phrase><verb-phrase>
<noun-phrase>	→	<cmplx-noun>
<noun-phrase>	→	<cmplx-noun><prep-phrase>
<verb-phrase>	→	<cmplx-verb>
<verb-phrase>	→	<cmplx-verb><prep-phrase>
<prep-phrase>	→	<prep><cmplx-noun>
<cmplx-noun>	→	<article><noun>
<cmplx-verb>	→	<verb>
<cmplx-verb>	→	<verb><noun-phrase>

Rules		
<article>	→	a
<article>	→	the
<noun>	→	boy
<noun>	→	girl
<noun>	→	flower
<verb>	→	touches
<verb>	→	likes
<verb>	→	sees
<prep>	→	with

By applying the rules in an appropriate order we see that the following strings are in the language:

```
a boy sees
the boy sees a flower
a girl with a flower likes the boy
the girl touches the boy with the flower
```

For example we can apply the following production rules, starting with the start symbol <sentence>. This procedure is called a *derivation*.

$$
\begin{array}{rcl}
\langle\text{sentence}\rangle & \to & \langle\text{noun-phrase}\rangle\langle\text{verb-phrase}\rangle \\
& \to & \langle\text{cmplx-noun}\rangle\langle\text{verb-phrase}\rangle \\
& \to & \langle\text{article}\rangle\langle\text{noun}\rangle\langle\text{verb-phrase}\rangle \\
& \to & \text{a } \langle\text{noun}\rangle\langle\text{verb-phrase}\rangle \\
& \to & \text{a boy } \langle\text{verb-phrase}\rangle \\
& \to & \text{a boy } \langle\text{cmplx-verb}\rangle \\
& \to & \text{a boy } \langle\text{verb}\rangle \\
& \to & \text{a boy sees}
\end{array}
$$

Some strings in the language have multiple derivations (e.g., `the girl touches the boy with the flower`). Note that there are also two different English meanings to this sentence. (See Grammar Exercise 1.) Such strings are called *ambiguous*. In the English language, ambiguous sentences are usually not too troublesome. In a programming language, they would be disastrous.

3.1 Type III Grammar (regular)

A Type III grammar consists only of the following two types of production rules: $A \to b$, $C \to De$. The grammar can contain any number of terminals and nonterminals. But, each production rule has only a single nonterminal on the left of the arrow. On the right of the arrow, there can be either a single terminal or a single nonterminal followed by a single terminal. This restriction significantly handicaps the grammar's ability to generate languages. Despite this, any regular language can be generated using a Type III grammar.

Example. Consider the following Type III grammar, which generates strings such as: `b, bb, aa, aba, bab, bbaaaa`. This grammar generates all strings with an even number of a's. Think of `E` as representing an even number of `a`'s and `O` as representing an odd number of `a`'s.

terminals	a, b
nonterminals	E, O
start	E

Rules
$E \to Oa$
$O \to Ea$
$E \to Eb$
$O \to Ob$
$E \to \epsilon$

3.2 Type II Grammar (context-free)

A Type II grammar consists only of production rules of the following type:: $A \to b$, $B \to cAbD$, and $C \to BcdEcAb$. As in Type III, each production rule has a single nonterminal on the left. However, an arbitrary combination of terminals and nonterminals is allowed on the right. We denote this class of production rules by $A \to \alpha$. Here α denotes any fixed sequence of terminals and nonterminals.

Type II grammar is at least as powerful as a Type III grammar (since all Type III production rules are of this form). In fact, it is more expressive and can generate some interesting and familiar languages. Most notably, the language consisting of all syntactically valid C programs can be generated using a context-free grammar[4] (e.g., see Kernighan and Ritchie, Appendix A13).

Example. Consider the following Type II grammar, which generates string such as: `aa, bb, abba, bbbb`, `abbbba`. This grammar generates all even length *palindromes*, i.e., strings that look the same if read forwards and backwards. We note that this language cannot be generated with a Type III grammar.

[4]This is almost true. The part of the grammar involving `typedef` is not.

terminals	a, b
nonterminals	S
start	S

Rules
$S \rightarrow aSa$
$S \rightarrow bSb$
$S \rightarrow \epsilon$

3.3 Type I Grammar (context-sensitive)

A Type I grammar is allowed to use any Type II or III production rules. It can also include rules of the following form: $\beta B \gamma \rightarrow \beta b \gamma$. Here B is a nonterminal, b is a terminal, and β and γ are some fixed sequence of terminals and nonterminals. The net effect of this production rule is to replace one nonterminal B by a terminal b, but only if B is surrounded by β and γ, i.e., if B is in the context of β and γ.

3.4 Type 0 Grammar (recursive)

A Type 0 grammar has no restriction on the types of production rules it is allowed to use.

3.5 Membership and Parsing

Given a grammar, the following two questions are essential for designing compilers (or foreign language translators):

- **(lexical analysis)** Given a string, does it belong to the language generated by the grammar?
- **(parsing)** If so, how can it be derived from the production rules?

The *lexical analysis problem* is important if we wish to know whether or not an English sentence or C program is syntactically correct. Compilers use this for two reasons, "tokenizing" and identifying syntax errors. When you read an English sentence, you implicitly decompose it into individual words, each of which imparts some meaning. Then you implicitly need to determine whether the concatenation of words leads to a grammatically correct English sentence. Compilers explicitly do the same thing. First, the compiler breaks up each statement into *tokens*. Tokens are keywords, variable names, and operators, e.g., `int, for, while, main(), +, ->, count.` Second, the compiler determines whether these statements correspond to a well-defined C program, e.g., matching parentheses, correct syntax of a `for` loop, etc.

Once we have a grammatically correct English sentence or C statement, the *parsing problem* is to understand its meaning. The derivation brings out the implicit "meaning" of the string. For example, the parse tree corresponding to the preorder expression `* * + 1 2 * 3 4 + 5 6` is shown in Figure 1. The expression evaluates to `((1 + 2) * (3 * 4)) * (5 + 6) = 396`.

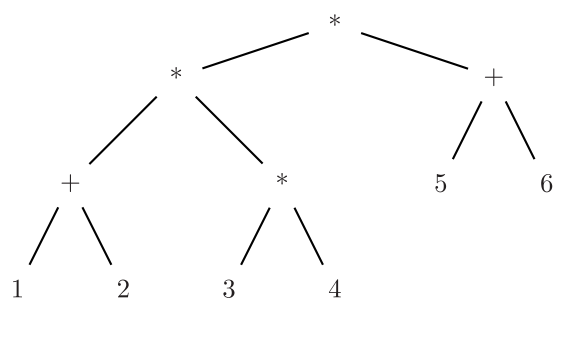


Figure 1: Parse tree for preorder expression `* * + 1 2 * 3 4 + 5 6`.

Our ability to solve these two problems crucially depends on the type of grammar. For Type III grammar, there is a linear time algorithm. For Type II, there is a polynomial-time algorithm. For Type I, there is an exponential-time algorithm. For Type 0, the problem is unsolvable. Context-free grammar (Type II) plays a fundamental role in describing programming languages, in large part because the parsing and lexical analysis problems can be solved efficiently by computers.

3.6 Programming Languages

Computers do exactly what we program them to do, so we need rigid rules to unambiguously specify our commands. A machine language program (e.g., TOY) is a simple and unambiguous way to tell a computer what to do. Yet, it is quite cumbersome to write complex programs in this language. It would be much easier to express algorithmic ideas in English sentences. Unfortunately, the English language has words and sentences with multiple meanings. So how would the computer know which one we meant? Moreover, how would we get the computer to translate from English into its internal machine language?

The C programming language (a context-free grammar specified in Kernighan and Ritchie, Appendix A13) provides a medium between the English language and machine language. It allows us great flexibility in expressing high-level algorithmic ideas (e.g., variables, loops, functions). Yet, its syntax rules are still rigid enough so that each C statements has only one meaning. Furthermore, since it is a Type II grammar, it is possible to write a compiler to quickly translate a C program into a machine language program. This is no accident – modern programming languages were developed only after the development of formal language theory.

3.7 Other Types of Grammar

There are some types of grammar that fall in between levels in the Chomsky hierarchy. An example is *Lindenmayer systems*, where production rules are applied *simultaneously*. We won't go into the details, but it was originally developed by biologists as a mathematical framework for analyzing the development of multicellular organisms. It is also useful for modeling branching and flowering patterns in plant life, or to create synthetic plants, as in Lecture T3.

4 Abstract Machines

Modern computers (PC, Macintosh, TOY) are capable of performing a wide variety of computations. An *abstract machine* reads in some input (typically a sequence of bits) and outputs either `yes` or `no`, depending on the input. (Actually, there is a third possibility: the machine goes into an infinite loop and never returns anything.)

We say that a machine *recognizes* a particular language, if it outputs `yes` for any input string in the language and `no` otherwise. Virtually all computational problems can be recast as language recognition problems. E.g., suppose we want to know whether a given input integer is prime. We can think of this question as deciding whether or not the integer is in the language consisting of all prime numbers.

We would like to be able to compare different classes of abstract machines, (e.g., finite state automata vs. pushdown automata) formally to answer questions like *Can my Macintosh do more things than your PC?* We say that machine A is at least as *powerful* as machine B if machine A can be "programmed" to recognize all of the languages that B can. Machine A is more powerful than B, if in addition, it can be programmed to recognize at least one additional language. Two machines are *equivalent* if they can be programmed to recognize precisely the same set of languages. Using this definition of power, we will classify several fundamental machines. Naturally, we are interested in designing the most powerful computer, i.e., the one that can solve the widest range of language recognition problems. Note that our notion of "power" does not say anything about how fast a computation can be done. Instead, it reflects a more fundamental notion of whether or not it is even possible to perform some computation in a finite number of steps.

4.1 Finite State Automata (FSA)

A *finite-state automata* is, perhaps, the simplest type of machine that is still interesting to study. Many of its important properties carry over to more complicated machines. So, before we hope to understand these more complicated machines, we first study FSA's.

An FSA captures the basic elements of a machine: it reads in a fixed sequence of characters (0 or 1 over the binary alphabet), and depending on the input and the way the machine was built, it returns `yes` (accept) or `no` (reject). An FSA is always is one of N *states*, which we name $0, 1, \ldots N-1$. As the input characters are read in one at a time, the FSA changes from one state to another. The new state is completely determined by the current state and the character just read in. By default, the FSA starts in state 0. There are also one or more *accept states*. If after all the input characters are read, the FSA is in one of these accept states, it accepts the input. Otherwise it rejects the input. The set of all strings for which the machine will end up in an accept state is called the *language recognized* by the machine.

FSA's are ubiquitous in everyday life. Most consumer electronics products (e.g., dishwashers, CD players, stop lights, remote controls, alarm clocks) contain these primitive machines, mainly because they can perform simple computations and are very inexpensive to build. As an illustrative example, consider your alarm clock. If the alarm clock is in the "wake up state" then it will make a deafening noise. If the next input bit is the "Snooze character," then the alarm clock will transition to a "quiet state." Each minute without user activity will transition the alarm clock one state closer to the "wake up state." FSA's are also widely used implicitly in user interfaces, e.g., Windows pull-down menus. Depending on what menu (state) you are in, various user inputs (e.g., mouse clicks, pressing the Ctrl-key) will cause the program to function differently.

We can represent an FSA in a table or graphically, as illustrated in Figure 2. Interpret the table as follows. Each row i corresonds to one of the N states. Each column j corresponds to one of the characters in the alphabet. The i-j entry of the table indicates which state the FSA will transition to, assuming it is currently in state i and the next input character read in is j. The final column indicates the accept states. It is also convenient to represent the FSA using a *directed graph*. We include a node (circle) for each state. Accept states are indicated in yellow (or with two concentric circles in the course packet). Each state has directed arcs (arrows) leaving it, labeled with a single character. Each state has *exactly* one outgoing arc for each character. Now, to simulate the behavior of the FSA, you start in state 0. For each input character, you follow the arc labeled with that character and end up in the new state. If after reading all the input characters, the FSA ends up in one of the accept states, then the FSA accepts the input string; otherwise it rejects it.

	0	1	
0	1	2	
1	1	2	accept
2	3	2	
3	4	4	accept
4	4	4	

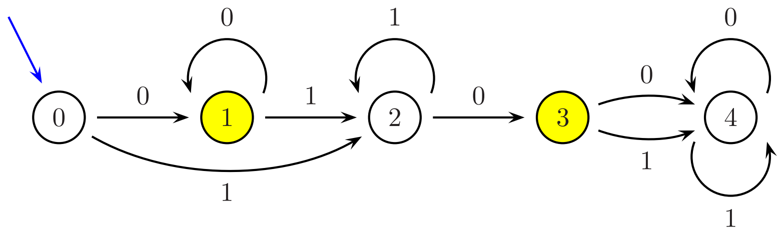


Figure 2: A 4-state FSA.

Consider the FSA in Figure 2. Which of the following bit strings will be accepted?

- 111: `no` – the FSA starts in state 0; the first bit read in is a 1, so it moves to state 2; the next bit is a 1, so it stays in state 2; the final bit is a 1 so it stays in state 2 (a reject state)
- 010: `yes` – the FSA starts in state 0; the first bit read in is a 0, so it moves to state 1; the next bit is a 1, so it moves to state 2; the final bit is a 0 so it moves to state 3 (an accept state)
- 01010: `no` – the FSA goes from state 0 to 1 to 2 to 3 to 4 to 4
- 0001110: `yes` – the FSA goes from state 0 to 1 to 1 to 1 to 2 to 2 to 2 to 2 to 3

Now let's design an FSA that accept all bit strings that have an even number of 1's. First, we need to figure out what the states should be. For FSA's, each state should completely summarize the input read in so far. Suppose that the following bits were read in: 100111. There are four 1's, so we could think of creating a state for each integer k, representing the fact that exactly k 1's have been read in so far. While this integer k does summarize the input (and knowing the state determines whether or not the input has an even or odd number of 1's), this approach would require arbitrarily many states, hence it would not be a *finite* state automata.

Instead, create only two states: state 0 represents that an even number of 1's have been read in so far; state 1 represents an odd number of 1's. Again, knowing the state is sufficient to answer the original language question, but now we have only 2 states. State 0 is the accept state; state 1 is the reject state. Now, we figure out the transition table. If the FSA is in state 0 (even) and the next bit is a 0, then it should stay in state 0, since the number of 1's hasn't changed. Otherwise, if the FSA is in state 0 (even) and the next bit is a 1, then the FSA moves to state 1, since now an odd number of 1's have been read in. Similarly, if the FSA is in state 1 (odd) and the next bit is 0, then it stays in state 1; if the next bit is 1, it moves to state 0.

	0	1	
0	0	1	accept
1	1	0	

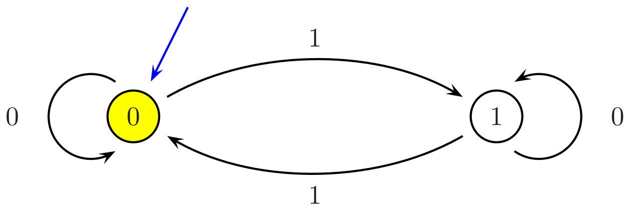


Figure 3: An FSA that recognizes all bit strings with an even number of 1's.

The process of building a FSA to solve a particular language recognition problem is akin to programming, and it is helpful to think of a specific FSA (described by states and transitions) as being a "program." Like C programming, it is an acquired skill, and you become more proficient with practice. When "programming" an FSA, you are very limited since you can only create a finite number of states. Moreover, there is no memory – you cannot store any information except the current state. In addition, the FSA moves deterministically from one state to another, depending only on the next input character and the current state.

4.1.1 FSA's Have Limitations

FSA's are incredibly simple machines, so it is not surprising that they have fundamental limitations. When we speak of the computational power of FSA's, we are really referring to the entire class of FSA's that could be built, i.e., the set of all languages that can be recognized by "FSA programs." In general, it is easier to describe what a machine can do than to identify its limitation. Nevertheless, we will give specific languages for which no corresponding FSA could ever be built.

We now show that no FSA can be built to determine whether an input bit string has the same number of 0's and 1's, i.e., recognize the language $\{01, 10, 0011, 0101, 0110, 1100, \dots\}$. To show this, we will assume that such an FSA could be built, say with N states. Then, we will derive a mathematical contradiction, implying that our assumption could not have been valid, i.e., no such FSA could be built.[5]

Consider the input string consisting of $N + 1$ consecutive 0's followed by $N + 1$ consecutive 1's. For simplicity, we'll take $N = 10$, but the same argument works for any value of N. Here the input bit string is 0000000000011111111111. By our assumption, the FSA will accept this input. Consider all of the states that the FSA visits when reading in the eleven 0's. Since there are only ten states, the FSA must revisit one of the states while reading in the 0's.[6] Once it does this, it will retrace its path over and over again, until the first 1 is read in. The FSA will look something like the one in Figure 4.[7] The FSA makes the following transitions: 0-1-2-3-4-1-2-3-4-1-2-3. Then, it will read in the 1's and do the following transitions: 5-6-7-8-9-7-8-9-7-8-9, and end up in an accept state.

The crucial idea is that by removing the intervening 0's that lead the FSA around the cycle 1-2-3-4, we obtain a different input with fewer 0's and still eleven 1's (00011111111111) that the FSA must also accept.

[5] This mathematical method of proof is called *proof by contradiction*, and is quite useful when trying to prove a "negative."

[6] This is fondly referred to as the *pigeonhole principle*, since it is impossible to stick 11 pigeons into 10 pigeonholes.

[7] For simplicity, only half of the transition arcs are depicted. Note also that the states associated with reading in the 0's and the states associated with reading in the 1's will not, in general, be entirely disjoint.

This input has fewer 0's than 1's, contradicting the assumption that the FSA accepts only bit strings with equal numbers of 0's and 1's.

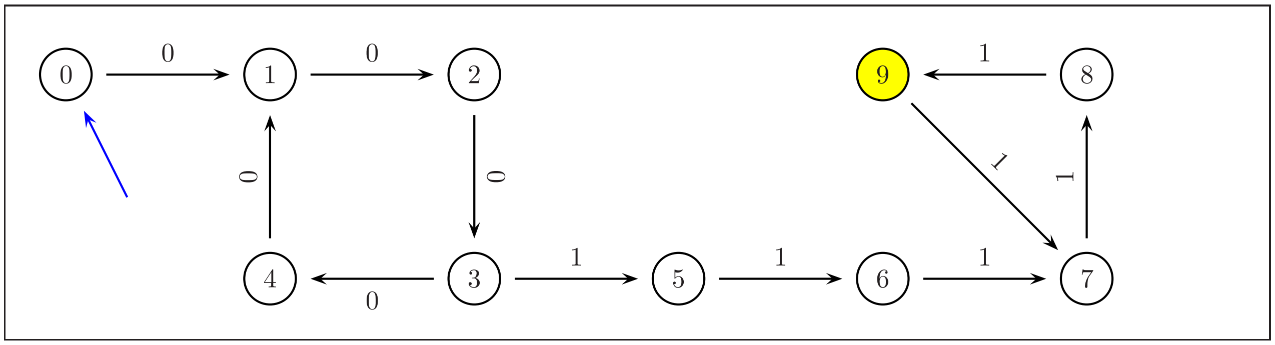


Figure 4: Part of a hypothetical FSA that purports to recognize all bit strings with the same number of 0's and 1's. The string 00011111111111 is accepted so it does not live up to its claim.

4.1.2 Nondeterministic Finite State Automata (n-FSA)

An FSA moves from one state to another, depending on the current state and input character. For each current state and input character, there is *exactly one* state to transition to. A *nondeterministic finite state automata* is exactly the same, except that given a current state and input character, there can be *zero, one, or several* possible states to transition to. Graphically, there can be zero or more arcs leaving a state corresponding to the same input character. This affects what it means for the machine to accept a string. For FSA's, there is a unique outcome given an input string. For n-FSA's, there may be zero, one or more final states that can be realized, depending on the "choices" made when moving from one state to another. An n-FSA *accepts* an input string if there exists *any* possible selection of transitions for which the n-FSA ends up in an accept state.

	0	1	
0	0, 1	0	
1	2	-	
2	-	-	accept

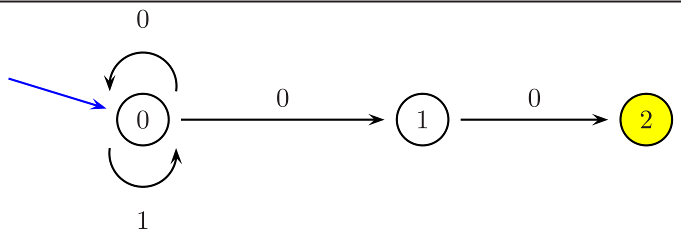


Figure 5: A n-FSA that accepts all bit strings ending with two 0s.

Consider the n-FSA in Figure 5. Which of the following bit strings will be accepted?

- 100: `yes` – the n-FSA starts in state 0, the first bit read in is a 1, so it stays in state 0; the next bit is a 0, so it could choose to transition to state 0 or 1 (let's suppose state 1); the final bit is a 0, so the n-FSA transitions to state 2 (an accept state)
- 101000: `yes` – the n-FSA could choose to stay in state 0 after reading in the bits 1, 0, 1, 0; then, upon reading the third 0, it could choose to transition to state 1 next; the final bit is 0, so it transitions to state 2
- 110001: `no` – there is no way to choose transitions so that the n-FSA ends up in state 2 - it could arrive in state 2 after reading the first four bits, but then there would be no place for the n-FSA to transition upon reading in the fifth bit
- 01110100011: `no` – the n-FSA will accept precisely those bit strings ending with two 0's

It is straightforward to determine whether or not a particular input string is accepted by an FSA. It is a little more difficult to do the same with a n-FSA, since you must consider all possible transitions, before you conclude that a string is not in the language.

You can think of the machine as "guessing" which transition to make (when multiple transition choices are possible). If any possible way of guessing the transitions leads to an accept state, the machine accepts the input string. This is where the term "nondeterministic" comes from.

4.1.3 Deterministic and Nondeterministic FSA's are Equally Powerful

Recall, the power of machines is measured by how many languages they can recognize. Nondeterministic FSA's are at least as powerful as deterministic ones, since FSA's are special nondeterministic FSA's that don't have multiple transitions from a state, corresponding to the same input character. At first, it would seem that n-FSA's are more powerful than FSA's, since they allow the extra possibility to select from several choices when moving from one state to another. Surprisingly, they are no more powerful. If some language can be recognized by a n-FSA, then it is always possible to build a FSA that recognizes the same language. Such a recipe is indicated in the course packet. Figure 6 is a deterministic FSA that recognizes the same language as the nondeterministic FSA in Figure 5.

Unfortunately, if the original n-FSA has N states, then, in general, the equivalent FSA constructed may have 2^N states. However, our definition of power only relies on the ability to recognize a language – it does not differentiate between how long it takes.

	0	1	
0	1	0	
1	2	0	
2	2	0	accept

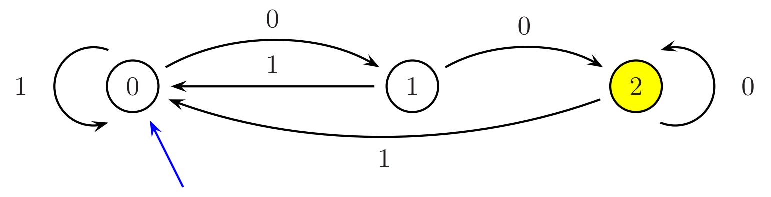


Figure 6: A FSA that recognizes the same language as the n-FSA in Figure 5.

4.2 Pushdown Automata (PDA)

FSA's are very simple machines, but they cannot "remember" all of the characters that they've seen, since they have only finitely many states. We now consider a more powerful machine called a *pushdown automata*. Just like a FSA, it reads in an input string. It also has finitely many states, named $0, 1, \ldots N$-1. It starts in state 0 and moves from one state to another, as it reads in the input. The main difference between a PDA and an FSA is that the PDA has a single *pushdown stack* on which it can store 0's and 1's (assuming we are working over the binary alphabet). Initially the stack is empty. Each time a bit is read in, the PDA is also allowed to push or pop a single bit. (Recall, a stack can store an arbitrary number of bits, but it is only possible to access the topmost element, either to read, push, or pop.)

The PDA reads in a new bit and moves from one state to another, but now the transition can depend on both the input bit as well as the contents of the topmost element on the stack. Thus, for each state, there may be 6 possible transitions depending on the input and stack: the input bit can be 0 or 1, and the top stack element can be 0, 1, or empty. Much like a nondeterministic FSA, it is not necessary to have arcs for each of the 6 possibilities. However, at most one arc is permitted to represent each of the 6 possibilities. Thus, given an input bit string, the sequence of state transitions is uniquely determined.

As with FSA's, it is convenient to represent a PDA using a graph. We include a node for each state and an arc for each transition possibility. Now, each arc is labeled with 3 values $i/j/k$, where i is the input bit, j is the topmost element on the stack, and k is either push or pop, where push means push the input bit i onto the stack. For example, the arc below from state 0 to 1 with label 1/empty/push means that if the PDA is in state 0 and the next bit is a 1 and the stack is empty, then it will transition to state 1 and push the input bit 1 onto the stack. We label the transition arc between two states with all of the possible scenarios that cause the PDA to move between those two states. For example, the arc from state 0 to 0 includes the three labels 1/0/pop, 0/0/push, and 0/empty/push. Thus, if either (i) the input is a 1 and the topmost stack element is 0, or (ii) the input is a 0 and the topmost stack element is 0, or (iii) the input is a 0 and the stack is empty, then the PDA will to transition to state 0.

The accept/reject mechanism of a PDA differs from an FSA.[8] In fact, there are no accept or reject states. Instead, the PDA accepts an input string only if the stack is complelely empty at the very end of the computation. It rejects the input for one of the following three reasons: (i) if the stack is not empty, (ii) if an attempt is made to read or pop an element off an empty stack at any point during the computation, or (iii) if at some point during the computation, it ends up in a state, and there is no arc leaving it corresponding to the current input bit and stack contents. Reason (iii) can occur since there might not be an arc for each of the 6 possible combinations of input bit (0 or 1) and stack contents (0, 1, or empty).

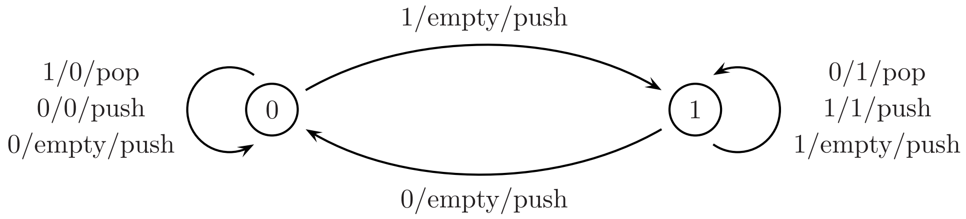


Figure 7: A PDA that recognizes the language of all bit strings with an equal number of 0's and 1's.

4.2.1 PDA's Have Limitations

While PDA's can distinguish between bit strings that have the same number of 0's and 1's and those that don't, they still suffer from fundamental limitations. For example, they can't recognize the language that consists of all palindromes. A *palindrome* is a string that looks the same forwards and backwards, e.g., 10011001, 1110111, etc.

4.2.2 Nondeterministic Pushdown Automata (n-PDA)

A *nondeterministic pushdown automata* is exactly the same as a PDA, except that there can be zero, one, or several possible transition choices from a particular state corresponding to the same input bit and the stack contents. Also, there may be multiple choices in deciding whether or not to push and pop elements from the stack.

To see the power of nondeterminism, consider the n-PDA is Figure 8 that accepts all even length palindromes. It is nondeterministic because, for example, there are two arcs leaving state 0, corresponding to the next input bit being 0 and when the stack is empty. Actually, if you are in state 0, the n-PDA always has the "choice" to either stay in state 0 or move to state 1.

It is natural to wonder whether or not adding nondeterminism to a PDA would make the machine more powerful. In contrast with FSA's, the answer is yes, since it turns out to be impossible to construct a deterministic PDA that recognizes the language of even length palindromes. We note that Figure 8 is a n-PDA corresponding to the language generated by the context-free grammar in Section 3.2.

4.3 Linear Bounded Automata (LBA)

A *linear bounded automata* is just like a Turing machine (see Section 4.4), except that the size of the tape is limited to being no bigger than a constant times the number of characters in the input string.

[8]This new mechanism for accepting strings can be shown to be equivalent to the method for FSA's, but it is more convenient to use the new mechanism.

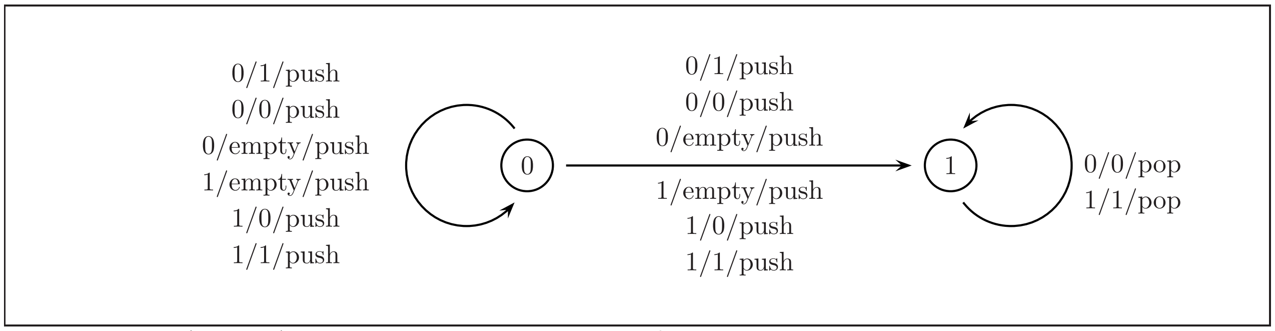


Figure 8: A n-PDA that recognizes the language of all bit strings that are even length palindromes.

LBA's are more powerful than pushdown automata, but less powerful than Turing machines. This seems intuitive since LBA's can access any element on the tape, while PDA's can only access the topmost element of the stack. However, the stack can have an arbitrarily large size, while the tape size is limited to be no bigger than the size of the input. Nevertheless, it turns out that LBA's are more powerful than PDA's.

4.3.1 Nondeterministic LBA

Interestingly, it is not known whether or not a nondeterministic LBA is more powerful than a deterministic one. Recall, nondeterminism makes pushdown automata more powerful, but not FSA's.

4.4 Turing Machine (TM)

The *Turing machine* is the foundation of modern theoretical computer science. It is a simple model of computation that is general enough to embody any computer program. Because of its simple description and behavior, it is amenable to mathematical analysis. This analysis has led to deep insights into the nature of computation.

A Turing machine is just like a PDA, except that instead of a stack, the Turing machine has a *tape* where it can store characters. Think of a tape as an array, so that the Turing machine can read and write characters to different elements of the array. The size of the tape can be arbitrarily large in size. It is the extra ability to *write* to the array that gives the Turing machine its power compared with the previous machines. (Recall, a PDA can only access the topmost element of its stack.) A Turing machine consists of a finite number of states, just like an FSA. Similarly, there is one start state, and one or more accept states. The input to the TM is entered on the tape before the computation begins. The TM also maintains a *cursor* which points to one position on the tape. The TM can only read or write to this position on the tape. As the computation progresses, the cursor position changes. At some point during a computation, the TM is completely described by its current state, the contents of the tape, and the cursor. Given this information, the TM transitions from its current state to another, depending on the current state and the character on the tape read in. The TM can also write to the part of the tape that the cursor currently points to. After writing, the TM moves the cursor either one position to the left or one position to the right. The TM accepts the input if it ever reaches an accept state. It rejects if the computation *does not terminate.*

The TM in Figure 9 is defined over the alphabet consisting of the four characters: `a, b, x,` and `#`. Here, the character `#` is used to delimit the boundaries of the real input to the problem.

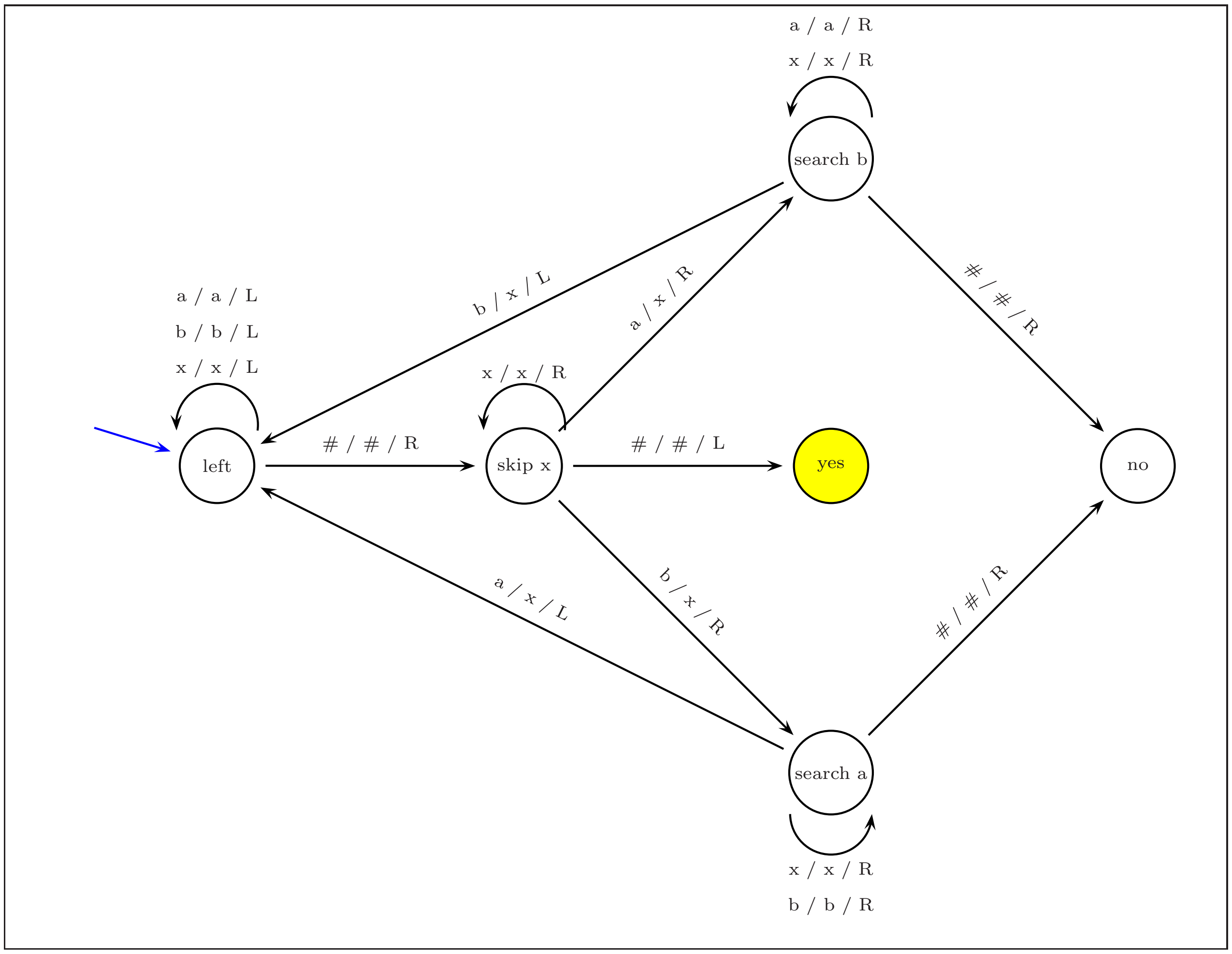
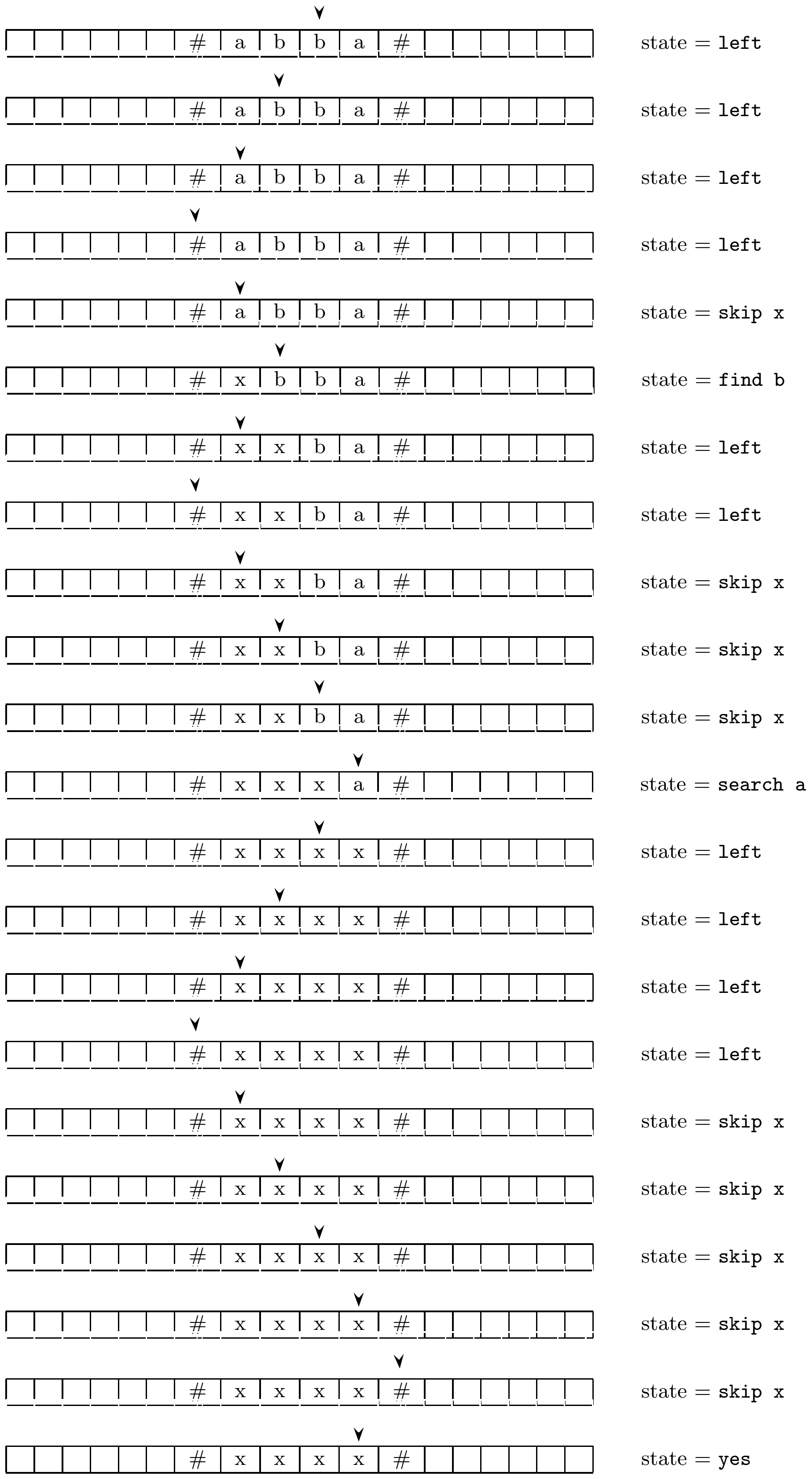


Figure 9: A Turing machine that accepts all inputs with an equal number of `a`'s and `b`'s.

The TM will accept all strings with an equal number of a's and b's. Figure 10 gives a step-by-step simulation of the Turing machine when run with input `#abba#`. To do this, the TM first moves to the left end of the tape. If the first character (other than `#`) is an `a`, the TM "deletes" it by overwriting it with an `x`. Then the TM will search for the leftmost `b`. If the TM finds a `b`, it "deletes" it by overwriting it with an `x`. Otherwise, the TM concludes that there are more `a`'s than `b`'s and proceeds to state `no`. (The process is analogous if the first character is a `b`. Then it deletes the `b` and searches for an `a` to delete.) This whole procedure is repeated, until the string contains no more `a`'s or `b`'s. In this case, the TM will go to state `yes`. The TM works as claimed because every time it "deletes" an `a`, it also "deletes" a corresponding `b`, thereby reducing the size of the problem. The new smaller input has an equal number of `a`'s and `b`'s if and only if the original input did. Here's a description of each state.

- `yes`: the accept state
- `no`: the reject state
- `left`: This state repeatedly moves the cursor one position to the left, unless the cursor already points to `#`. That is, `left` moves the cursor to the left `#`. If instead, the cursor is already pointing to `#`, then it transitions to `skip x` and moves the cursor one position to the right, i.e., to the beginning of the interesting portion of the tape.
- `skip x`: This state repeatedly moves the cursor one position to the right, until it reaches the first character that is not `x`. If the first such character is `#`, then it accepts the string; if it is `a`, the TM



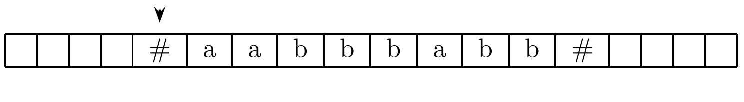
# a b b a # state = left
# a b b a # state = left
# a b b a # state = left
# a b b a # state = left
# a b b a # state = skip x
# x b b a # state = find b
# x x b a # state = left
# x x b a # state = left
# x x b a # state = skip x
# x x b a # state = skip x
# x x b a # state = skip x
# x x x a # state = search a
# x x x x # state = left
# x x x x # state = left
# x x x x # state = left
# x x x x # state = left
# x x x x # state = skip x
# x x x x # state = skip x
# x x x x # state = skip x
# x x x x # state = skip x
# x x x x # state = skip x
# x x x x # state = yes

Figure 10: The contents of TM tape when started with input `#abba#`.

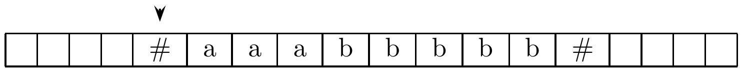
goes to state `search b` to find a `b` to "delete"; if it is `b`, the TM goes to state `search a` to find an `a` to delete.

- `search b`: This state skips over all `a`'s and `x's`. If it finds a `b`, it "deletes" it by overwriting it with an `x` and goes back to state `left`. If it doesn't find a `b` (it reaches the right `#`) then in concludes there were more `a`'s than `b`'s, and rejects the input.
- `search a`: analogous to `search b`

In addition to answering `yes-no` questions, Turing machines can also be used to do other tasks. For example, we may want to read in a list of characters and output a sorted list. The Turing machine tape can be used both for input and for output. The TM in Figure 11 is defined over the alphabet `a, b, #`. It reads in a sequence of `a`'s and `b`'s delimited by `#`'s. Upon halting, the TM leaves on its tape the sorted sequence of `a`'s and `b`'s. For example, if the input is:



then the output will be



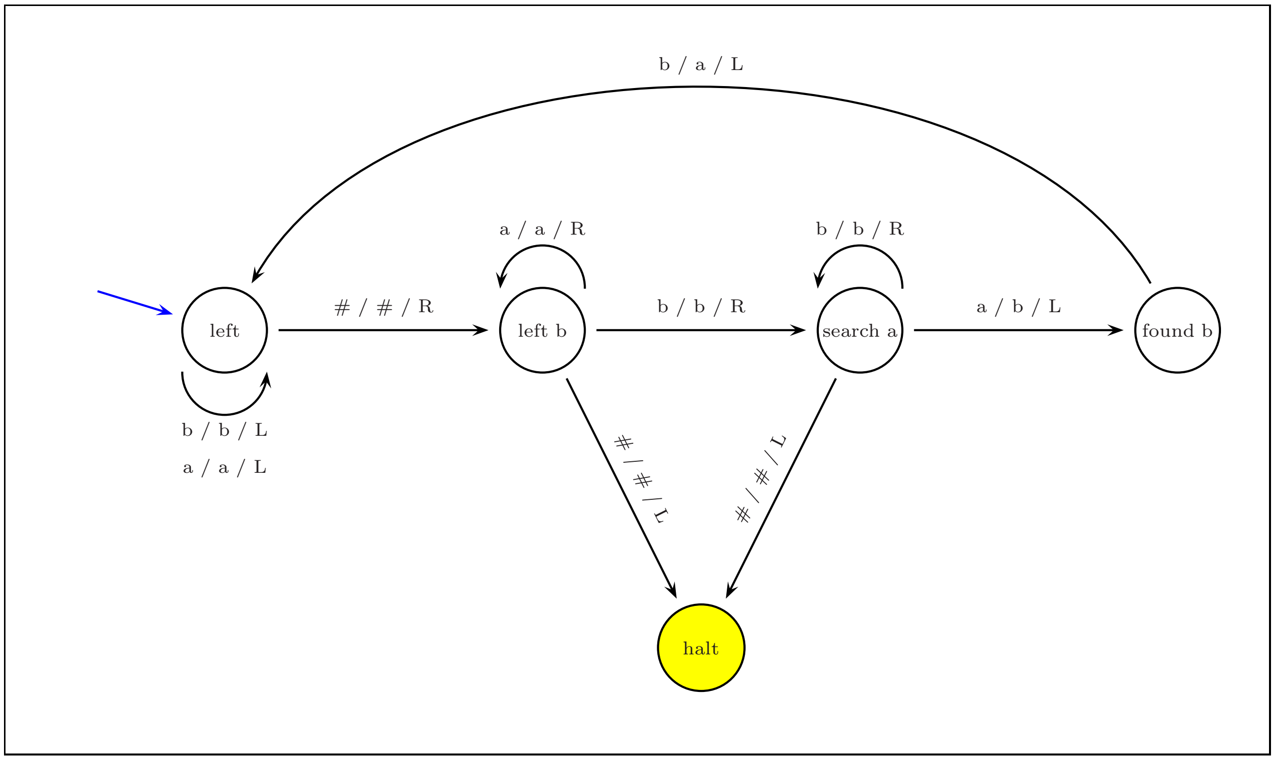


Figure 11: A Turing machine that sorts inputs consisting of `a`'s and `b`'s.

To understand how the TM in Figure 11 sorts, here's a description of each state:

- `halt`: the TM halts
- `left`: find the left `#` which signifies the beginning of the input
- `left b`: find the leftmost `b` in the input by skipping over all of the `a`'s
- `search a`: search for the first occurrence of `ba` in the input by looking for the first `a` to the right of the leftmost `b`; if you find such an `a`, swap the two adjacent characters by going to state `swap`

- `swap`: complete the swapping of `b` and `a` and start the process over again

4.4.1 Nondeterministic Turing Machine

A nondeterministic TM is no more powerful than a deterministic TM. That is, for any language accepted by a nondeterministic TM, we can construct a deterministic TM that recognizes the same language.

However, much like the situation with FSA's, the only known way to convert from a nondeterministic TM to a deterministic one suffers from an exponential blowup in the number of states. You can simulate a nondeterministic TM using exponentially many deterministic TM's running in parallel. So while nondeterminism does not help us recognize a broader class of languages, it may help us recognize the same languages faster. This is the widely believed $P \neq NP$ conjecture. See the document *Complexity, Unsolvability and Intractability* for more details. The solution to this conjecture would have profound influence on our ability to efficiently solve a huge number interesting problems, including the travelling salesperson problem and factoring integers.

4.4.2 Universal Turing Machine (general purpose computers)

A Turing machine is specifically designed to solve *one* particular problem (e.g., recognize whether or not some integer is prime). It corresponds with a single computer program. A *universal Turing machine (UTM)* is a single machine that can simulate the behavior of *any* Turing machine. It corresponds with a single piece of hardware on which different programs can run. It is hard to imagine a world without a general purpose computer, where we can perform new tasks by installing new software. However, Alan Turing invented these theoretical devices in the 1930's, a decade before engineers could build such computers. The UTM played a significant role in the development of early stored-program computers, and continue to play a central role in the development of new speculative models of computation.

The key idea for creating a UTM is that any particular Turing machine can be characterized by its state transition table, its initial tape contents, and its initial cursor location; this information can be systematically encoded, e.g., in binary. The job of the UTM is to decode the systematic description of the Turing machine, and simulating the behavior of the Turing machine. This decoding and simulating process is itself a mechanical process; thus we can build a single general purpose computer, i.e., a UTM.

4.4.3 Turing Machines Have Limitations

Turing machines are most powerful type of machine we have considered. Modern computers (PC, Macintosh, TOY) are all equivalent to a UTM in terms of power (assuming they are given an unbounded amount of memory). Thus, personal computers are capable of performing exactly the same types of computations as each other. Several new speculative computational models (DNA, quantum, and soliton) are also equivalent to Turing machines in terms of power.

Nevertheless, Turing machines suffer from fundamental limitations. Just as with FSA's and PDA's, there are some languages that even the mighty TM cannot recognize. The most famous of these is the *halting problem.* Here, the goal is to design a TM that recognizes the language of all programs (e.g., in C) that do not go into an infinite loop. See the document *Complexity, Unsolvability, and Intractability* for more details.

4.5 Other Models of Computation

Over the past few decades, several other interesting models of computation have been proposed. We consider one such intriguing model called *The Game of Life.* It was invented by Princeton mathematician John Conway. The game simulates the life and death process of an $N \times N$ community of "electronic organisms" using very simple rules: (i) an organism dies if it has 0 or 1 neighbors (loneliness), (ii) an organism dies if it has 4 or more neighbors (overpopulation), (iii) an organism survives if it has 2 or 3 neighbors, and (iv) a new organism is created if it has 3 neighbors (birth). Check out `http://www.bitstorm.org/gameoflife/` for a Java simulation. The game produces patterns of enormous complexity. You have likely seen the "screen saver" version of this game. You probably did not know that if you carefully choose where to locate the initial organisms in the community (analogous to writing a program), and watch the game as it progresses,

Grammar Type	Language	Rules	Machine
III	regular	$A \rightarrow b$, $C \rightarrow De$	FSA
II	context-free	$A \rightarrow \alpha$	nondeterministic PDA
I	context-sensitive	$A \rightarrow \alpha$, $\beta B \gamma \rightarrow \beta c \gamma$	linear bounded automata
0	recursive	arbitrary	Turing machine

Figure 12: The Chomsky Hierarchy.

you can perform any computation that a Turing machine can. That is, Conway's Game of Life is as powerful as a Turing machine.

5 Two Beautiful Theories

In this section we consider two core ideas that help explain what machines can do, and what they cannot do.

5.1 Chomsky Hierarchy

There is a striking connection between the four types of grammar (regular, context-free, context-sensitive, recursive) and the four classical abstract machines (finite state automata, nondeterministic pushdown automata, linear bounded automata, Turing machine). Given an input string, an abstract machine *decides* whether or not that string is in the language, e.g., to decide whether or a given integer is prime. Grammar plays a dual role. By applying the production rules in some order, a grammar *generates* all strings in the language, e.g., to enumerate all prime numbers. The Chomsky hierarchy orders grammar according to the types of languages they can generate, and machines according to the types of languages they can recognize. There is a striking one-to-one relationship between each of the four grammars and four machines: each grammar type generates *precisely* the same type of languages as its corresponding machine. (See Figure 12.) As we go down the Chomsky hierarchy, each grammar is strictly more expressive than the previous; each machine is strictly more powerful than the previous. Figure 13 illustrates the hierarchy of languages.

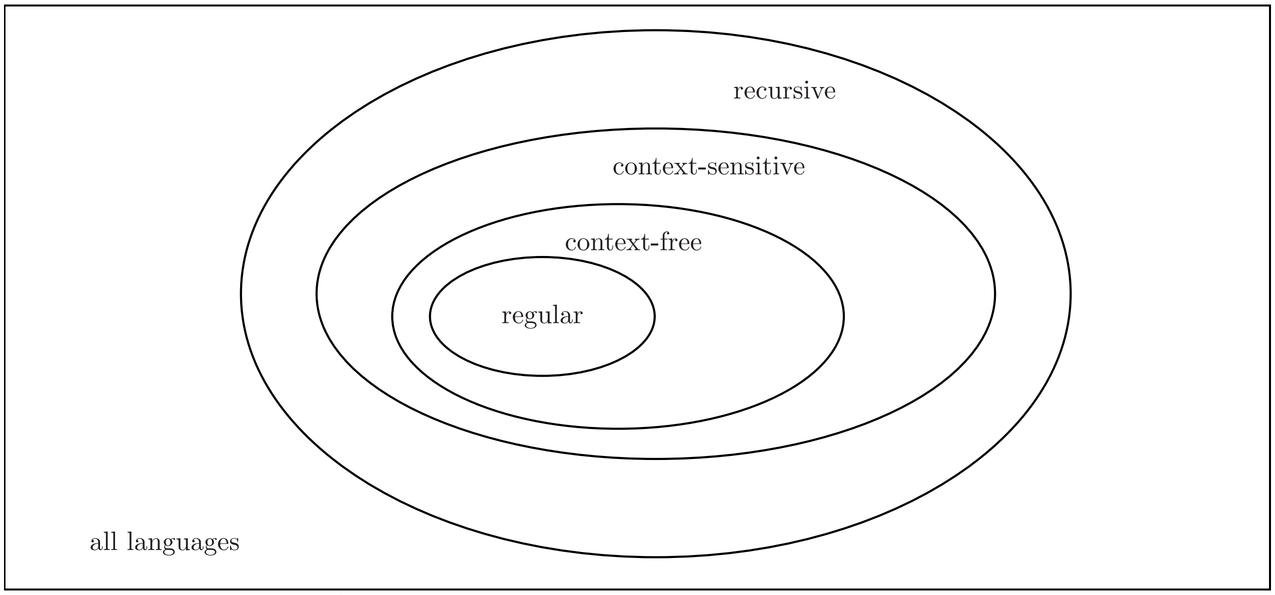


Figure 13: Four classes of languages: each class contains strictly more languages than the previous one.

We will not dwell on establishing the correspondence between each of the four types of grammar and four types of machines. Instead, we indicate how to construct a Type III grammar to correspond to a given FSA. The input characters to the FSA correspond to the terminals in the grammar; states correspond to nonterminals; accept states correspond to start symbols; state transitions correspond to production rules. As an example, we construct a Type III grammar for the FSA in Figure 2. The FSA uses the binary alphabet, so the terminals are 0 and 1. The FSA has 5 states labeled 0, 1, 2, 3, and 4. We will use the nonterminals A, B, C, D, and E to correspond to these states, respectively. The accept states for the FSA are 2 and 4, so we will have two start symbols B and D. The FSA has a transition from state 0 (A) to 2 (C) if the next bit is 1; we include a production rule $C \to A1$. The start state is 0 (A); to model this we include a production rule $A \to \epsilon$. The resulting grammar is:

terminals	$0, 1$
nonterminals	A, B, C, D, E
start	B, D

Rules	Rules
$B \to A0$	$E \to D0$
$B \to B0$	$E \to D1$
$C \to A1$	$E \to E0$
$C \to B1$	$E \to E1$
$C \to C1$	$A \to \epsilon$
$D \to C0$	

5.2 Church-Turing Thesis

The Church-Turing thesis states that Turing machines are the most powerful types of machines: any language that cannot be recognized by a Turing machine cannot be recognized by *any* physical machine. In other words, all physical computing devices can be simulated by a Turing machine. This simplifies the study of computation. Now, we only need to concern ourselves with Turing machines, rather than an infinite number of potential computing devices. Note that the Church-Turing thesis is not a mathematical statement, and is not subject to rigorous proof. The reason is that we can't precisely define what we mean by a computing device or "solving a problem."

The Church-Turing thesis implies a "universality" among models of computation. There is ample support suggesting this universality. Adding new capabilities (e.g., nondeterminism, more tapes, more read/write heads, multidimensional tapes) to a Turing machine does not make it more powerful. Computer scientists and mathematicians have considered numerous other models of computation. Grammar is one such example. The Chomsky hierarchy indicates that Type 0 grammar corresponds exactly with Turing machines. Church proposed a different model of computation based on the λ-calculus. It was later shown that this model of computation was equivalent to the Turing machine model. Other models of computation include: μ-recursive functions, counter machines, cellular automata, Conway's game of life, random access machines, tree-rewriting systems, high level programming languages, combinator calculus, Post machine (FSA + queue), 2-stack machine (FDA + 2 stacks), quantum computing, and DNA computing. These have all been shown to be equivalent in power to the Turing machine.

6 Examples

We give several examples of regular, context-free, non-context-free languages, and undecidable languages.

6.1 Regular Languages

(a) 0 or 11 or 101

(b) only 0's

(c) all strings

(d) all strings except empty string

(e) begins with 1, ends with 1

(f) ends with 00

(g) contains at least three 1's

(h) contains at least three consecutive 1's

(i) contains the substring 110

(j) doesn't contain the substring 110

(k) contains the substring 1101100

(l) has at least 3 characters, and the third character is 0

(m) number of 0's is a multiple of 3

(n) starts and ends with the same character

(o) odd length

(p) starts with 0 and has odd length, or starts with 1 and has even length

(q) contains an even number of 0's or exactly two 1's

(r) length is at least 1 and at most 3

(s) any string except 11 or 111

(t) every odd symbol is a 1

(u) contains at least two 0's and at most one 1

(v) no consecutive 1's

(w) bit string interpreted as binary number is divisible by 3

(x) bit string interpreted as binary number is divisible by 123

(y) equal number of occurrences of 01 and 10 as substrings, e.g., 01110, 01010, 111100100001

6.2 Context-Free Languages

(a) valid C programs

(b) equal number of 0's and 1's, e.g., 010101, 111000

(c) contains one more 1 than 0's, e.g., 110, 10101, 0011110

(d) contains twice as many 1's and 0's, e.g., 101, 001111

(e) palindromes, e.g., 00, 111, 10001, 10011011001

(f) 0^n1^n, e.g., 01, 0011, 000111

(g) 0^n1^n or 1^n0^n, e.g., 10, 01, 000111, 1100, 11110000

(h) odd length and middle symbol is 0, e.g., 101, 001, 11111000000

(i) 0^m1^n such that $m \leq 2n \leq 4m$, e.g., 00111, 00000111, 000001111111

6.3 Not Context-Free Languages

(a) $0^n1^n2^n$, e.g., 012, 001122, 000111222

(b) strings with equal number of 0's, 1's, and 2's

(c) $0^n1^n0^n1^n$, e.g., 0101, 00110011, 000111000111

(d) ww, where w is some bit string, e.g., 1010, 11001100, 100010100010

(e) power of two 1's, e.g, 1, 11, 1111, 11111111

(f) prime number of 1's, e.g., 11, 111, 11111

(g) $1^i2^j3^{i+j}$ (unary addition), e.g., 1233, 1122233333, 112222333333

(h) $1^i2^j3^{i*j}$ (unary multiplication), e.g., 123, 11222333333, 11222233333333

(i) 0^i1^j such that i and j are relatively prime, e.g., 00111, 00011, 0000111111111

6.4 Undecidable Languages

(a) Halting problem - does a program terminate?

(b) Post Correspondence problem

(c) Hilbert's 10th problem - does a multivariate polynomial have an integral root? For example the polynomial $6x^3yz^2 + 3xy^2 - x^3 - 10$ has the integral root $(x, y, z) = (5, 3, 0)$. By Fermat's Last Theorem, the polynomial $x^n + y^n - z^n$ does not have an integral root with $n \geq 3$ and x, y, and z positive.

Acknowledgments.

This document was originally based on lecture notes by Robert Sedgewick. Some of the material and examples were adapted from the textbook of Sipser [1]. The author thanks Eric Shrader-Frechette for providing many useful comments on a very preliminary version of this document.

References

[1] M. Sipser. *Introduction to the Theory of Computation.* PWS Publishing Company, Boston, MA, 1997.