Instructor's Manual to Accompany

THE LANGUAGE OF MACHINES:

An Introduction to Computability and Formal Languages

Richard Beigel

Instructor's Manual to accompany

The Language of Machines An Introduction to Computability and Formal Languages

Richard Beigel
Yale University

Computer Science Press

An imprint of W. H. Freeman and Company

New York

Richard Beigel can be contacted at the following email address:
beigel@cs.yale.edu

ISBN 0-7167-8291-X

Printed in the United States of America

Computer Science Press

An imprint of W. H. Freeman and Company

41 Madison Avenue, New York, NY 10010
20 Beaumont Street, Oxford OX1 2NQ, England

1 2 3 4 5 6 7 8 9 0 RRD 9 9 8 7 6 5 4

Contents

Preface

The manual is organized into ten topics. Topics 1–9 correspond to chapters 1–9. Topic 10, logic, is based on material from chapters 7 and 8.

Each topic includes a list of exercises that I think are appropriate (perhaps too many), as well as recommended reading. Each topic is then broken down into 50-minute lectures. I have not tried to make each topic fit into three lectures. Many lectures are optional, and the total number of lectures is between 33 and 46.

I have recommended reading for each topic and, within topics, for each lecture. If a particular section is recommended for a topic, but not for any of its lectures, the students can read it at their convenience, but really shouldn't skip it.

In addition, I have included solutions to many exercises, some new exercises, and, finally, a take-home exam (with solutions) that consists mostly of new questions.

I hope that this manual will help you teach.

Topic 1

Introduction to Machines

Reading: Sections 0.3 and 0.6. Chapter 1.

Homework: 0.3-6, 0.3-11, 0.6-12, 1.1-3, 1.3-3, 1.5-1, 1.9-4, 1.9-6

Teaching tip: Never use the word "nondeterministic" in class to mean "not deterministic." Some students have enough trouble as it is appreciating that determinism is a special case of nondeterminism rather than its negation.

Lecture 1

Reading: Section 1.1

Concepts: palindromes, programs, input operations, stack operations, nondeterminism.

Lecture:

Don't expect to get a lot done on day one. Students show up eager to learn four things: How much does the final count? Will there be a midterm? When is homework due? And can we turn it in late? For this reason, I start the first lecture by handing out a syllabus that covers grading policies, office hours, and an outline of the course. (I think most students don't actually read this, but it makes me look more professional.) Then I ask for questions, and they usually ask: How much does the final count? Will there be a midterm? When is homework due? And can we turn it in late?

Second, I tell them what the course is about. I try to draw on the typical student's programming experience. They are familiar with time as a limited computational resource. I point out that space (or memory) can be a more critical resource (this is clear to users of X Windows). When you run out of time, it's easy to get more, but when you run out of space, you're stuck. In this course we will

consider *qualitative* restrictions on memory rather than quantitative. As examples of qualitative restrictions, one may organize the memory as a stack, a queue, or a counter. We will see that different computational problems can be solved with different kinds of memory — but not with others. Impossibility proofs require carefully defined models of computation. We will see that some problems cannot be solved with any computer yet known or imagined.

Third, I introduce programs via a DSA for palindromes with central marker. It is important to define the language, describe the input and stack operations, and explain how operations on separate devices work together in an instruction.

Fourth, I bring up general palindromes (without central marker). I ask the students how they would determine if a string is a palindrome. After enough discussion for the students to see the difficulty in solving the problem on a stack machine, I modify the DSA on the board to guess the middle of the string (replace SCAN# by NOOP, SCANa, and SCANb). I tell them that this sort of guessing is called "nondeterminism," because the program's behavior is not strictly determined at all times. There will be questions. Point out that nondeterminism is an important theoretical construct, but not necessarily a practical one. Be sure to state the convention that the program accepts the input if there is any choice at all that leads to acceptance, even though other choices might not lead to acceptance. (Don't expect everyone to get it the first time. You will explain nondeterminism a few more times in this course.)

Lecture 2

Reading: Section 1.1

Concepts: acceptors vs. recognizers, finite machines, signed counters.

Lecture:

I re-draw the DSA for palindromes with central marker. Then I point out that we often want programs, in addition to accepting all strings that are in a language, to reject all strings that are not in the language. Then I add a rejecting state and error checks, and point out that most of the effort in the new program is error handling. I also make the program exhaust the input and say that this is just a convention — our programs should finish reading the input before giving an answer. Such a program is called a recognizer. Recognizers give an answer — either yes (accept) or no (reject) — for all inputs. Acceptors can only say yes (accept). Recognizers must also be deterministic (meaning they never have a choice of which instruction to perform), whereas acceptors can be either deterministic or not deterministic. Deterministic acceptors are useful to theorists, because they take less work to program, and in many important cases there are automatic techniques to convert them to recognizers.

Second, I present signed counter operations and we do the monadic arithmetic example from the book (Example 1.2). Then we discuss briefly how to solve the problem with an unsigned counter and a finite memory.

Finally, we present a DFA for pattern matching (Example 1.3).

Note: A pattern-matching DFA for the pattern aabaabab makes a nice example. You might want to assign it as homework or present it in section.

Lecture 3

Reading: Sections 1.2, 1.3, and 1.6

Concepts: The control, the control as finite memory, the control can hold two things. Techniques for programming with a counter.

Lecture:

First, I point out that there is really a memory device keeping track of where we are in the program. It is called a control or a finite memory, and it can store a bounded amount of information. For example, we consider decimal numbers that are even (remember whether the digit last scanned is even). Then we consider decimal numbers that are 0 mod 3 (remember the residue mod 3 of the number you have seen so far).

A finite memory can hold a pair of objects of bounded size. For example, we consider the set of strings over $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ that have an even number of a's and odd length.

Then we present DCAs for strings of the form $\mathtt{a}^n\mathtt{b}^n$ and $\mathtt{a}^n\mathtt{b}^{2n}$, and 2-DCAs for strings of the form $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$, and $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n\mathtt{d}^n$. It is good to involve the students in these examples, so that they become comfortable programming these machines. The second 2-DCA example is nice because you can tell them that a 2-DCA can solve any problem we know how to solve at all, although we aren't ready to prove it quite yet (and in any case the program can be rather inefficient).

Optional: Before presenting the DCA for it, use the pigeonhole principle to prove that there is no DFA for the set of strings of the form $\mathtt{a}^n\mathtt{b}^n$.

Lecture 4

Reading: Sections 1.3, 1.5, and 1.9

Concepts: balanced parentheses, balanced parentheses and brackets, uses of nondeterminism.

Lecture:

We further illustrate the power of counters and stacks via the language of balanced parentheses (DCA language) and balanced parentheses and brackets (DSA language). It is good to involve the students in these examples, so that they become comfortable programming these machines.

Optional: Note the similarity between Figure 1.11 and Figure 1.12. The kind of counter used makes a big difference. Think about designing an SCM program for balanced parentheses.

Then we present paradigms of nondeterministic programming by working Example 1.19 (an NCA that accepts $\{\mathtt{a}^m\mathtt{b}^n : m = n \text{ or } m = 2n\}$) and Example 1.20 (an NCA that accepts $\{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k : \neg(i = j = k)\}$). Encourage the students to offer several solutions to each problem. Their incorrect solutions will help you to debug their concepts of nondeterministic acceptance.

Remind the students of the following if their nondeterministic programs accept too many strings: to be correct, a nondeterministic program to accept L must have a way to accept every string in L, and no way to accept any string not in L.

Topic 2

Devices, Machines, and Programs

Reading: Sections 0.2.6, 0.2.7. Chapter 2.

Homework: 0.2-9, 0.2-14, 0.2-20, 0.6-24, 2.2-3, 2.4-2, 2.6-8(b), 2.7-1, 2.9-4, 2.10-3(a)

Note: For Exercise 2.7-1, it is necessary to tell the students how to number the states in Figure 1.10. See the exercise solution.

Lecture 1

Reading: Sections 0.2.6, 0.2.7, 2.2, 2.3, 2.4

Concepts: relations, devices, operations, machines, configurations, instructions

Lecture:

Review relations. A relation is a set of ordered pairs. To denote that x and y are related by the relation ρ we may write $(x, y) \in \rho$ (following the definition); more often we write $x \ \rho \ y$ (e.g., $x < y$) or $x \overset{\rho}{\mapsto} y$ (when ρ performs some kind of action on x). Define composition and illustrate with is-mother $\circ$ is-parent = is-grandmother. Define converse and illustrate with is-parent^{-1} = is-child. Define reflexive transitive closure and illustrate with is-parent* = is-ancestor.

Define partial functions, for example is-mother^{-1} is a partial function but is-mother is not. Describe postfix notation: $x\rho = y$.

Now we are going to formalize everything about Example 1.1, accepting palindromes with central marker.

The *realm* of a device is the set of states it can be in. The *repertory* of a device is the set of operations that it can perform. In this example,

control with control set $\{1, 2, 3\}$

realm $\{1, 2, 3\}$

repertory $\{1 \to 1, 1 \to 2, 1 \to 3, 2 \to 1, 2 \to 2, 2 \to 3, 3 \to 1, 3 \to 2, 3 \to 3\}$.

input with alphabet $\{\mathtt{a}, \mathtt{b}, \#\}$

realm $\{\mathtt{a}, \mathtt{b}, \#\}^*$

repertory $\{\mathrm{SCAN}\mathtt{a}, \mathrm{SCAN}\mathtt{b}, \mathrm{SCAN}\#, \mathrm{EOF}\}$

stack with alphabet $\{\mathtt{a}, \mathtt{b}\}$

realm $\{\mathtt{a}, \mathtt{b}\}^*$

repertory $\{\mathrm{PUSH}\mathtt{a}, \mathrm{PUSH}\mathtt{b}, \mathrm{POP}\mathtt{a}, \mathrm{POP}\mathtt{b}, \mathrm{EMPTY}\}$

An *operation* is a partial function from the device's realm to the device's realm. If it is defined, it maps the device's state to another state. Input and stack operations are defined as follows.

SCANa	$\mathtt{a}x \to x$
SCANb	$\mathtt{b}x \to x$
SCAN#	$\#x \to x$
EOF	$\Lambda \to \Lambda$
PUSHa	$x \to x\mathtt{a}$
PUSHb	$x \to x\mathtt{b}$
POPa	$x\mathtt{a} \to x$
POPb	$x\mathtt{b} \to x$
EMPTY	$\Lambda \to \Lambda$

To clarify, $c x \mathrm{SCAN}\mathtt{a} = x$ if $c = \mathtt{a}$ but is undefined otherwise, $c x \mathrm{SCAN}\mathtt{b} = x$ if $c = \mathtt{b}$ but is undefined otherwise, and $x\mathrm{EOF} = x$ if $x = \Lambda$ but is undefined otherwise.

The operations EOF and EMPTY are called tests. A *test* is a subset of the identity function.

A machine is a tuple of devices. In this example, the machine is (control, input, stack). Our notation: machine [control, input, stack].

A *configuration* is to a machine as a state is to a device. A configuration of a machine is a tuple denoting the state of each device (analogous to a core dump). Consider input ab#ba. The initial configuration is $(1, \mathtt{ab\#ba}, \Lambda)$.

An instruction is a tuple of operations, one for each device. To apply an instruction to a configuration, we apply each operation to the corresponding state. The result

is defined if and only if the results of all operations are defined. For example, let $\pi_1 = (1 \rightarrow 1, \text{SCANa}, \text{PUSHa})$ and $\pi_2 = (1 \rightarrow 1, \text{SCANb}, \text{PUSHb})$. Then

$$(1, \text{ab\#ba}, \Lambda)\pi_1 = (1, \text{b\#ba}, \text{a})$$

and

$$(1, \text{b\#ba}, \text{a})\pi_2 = (1, \text{\#ba}, \text{ab})$$

but $(1, \text{ab\#ba}, \Lambda)\pi_2$ is undefined because the SCANb operation can't be performed.

Lecture 2

Reading: Sections 2.5, 2.6, 2.7

Concepts: initializers, terminators, programs, running a program; (partial) computations, histories, and traces

Lecture:

Programs need to initialize their devices to reasonable values before doing anything else. Typically, counters start at 0, stacks and outputs start empty, the input holds the program's argument, and the control starts in a unique specified initial state. The *initializer* of a device maps the argument to an initial state of the device. The initializer of a machine maps the argument to an initial configuration of the machine.

Continuing yesterday's example, the initializer α is defined as follows:

$$x\alpha = (1, x, \Lambda).$$

The initial control state is 1, the argument x is placed in the input device, and the stack is empty.

A program halts only when all devices are in final states. Typically, counters finish at 0, stacks and inputs finish empty, the output holds the program's result, and the control finishes in one of several specified final states. In a sense, a result requires a unanimous vote of all devices. But sometimes we want only some of the devices to have a vote; in that case we make all states of the other devices final.

Define halting: halting or termination is synonymous with giving a result. Programs that don't halt either run forever or block (get stuck in a configuration with no applicable instruction).

In our example, the terminator ω could be defined as follows:

$$(3, \Lambda, \Lambda)\omega = \text{ACCEPT}.$$

3 is the final state of the control, Λ is the final state of the input, and Λ is the final state of the stack.

Alternatively, ω could also be defined as follows:

$$(3, \Lambda, s)\omega = \text{ACCEPT} \qquad \text{for all strings } s.$$

3 is the final state of the control, Λ is the final state of the input, and every string s is a final state of the stack.

Could we let every string be a final state of the input? In principle, yes. But by convention, no. We *always* require that the entire input be read before termination (i.e., before the program gives a result).

A *program* consists of an initializer α, a terminator ω, and an instruction set $\mathcal{I}$. Unlike programs you are used to writing, there is no order to instructions. At each step, the program executes any instruction that it can. (This is how microcode works.) Continuing our example, the instruction set is

$$\left\{ \begin{array}{llll} (\ 1 \to 1\ , & \texttt{SCANa}\ , & \texttt{PUSHa}\ ), \\ (\ 1 \to 1\ , & \texttt{SCANb}\ , & \texttt{PUSHb}\ ), \\ (\ 1 \to 2\ , & \texttt{SCAN\#}\ , & \texttt{NOOP}\ ), \\ (\ 2 \to 2\ , & \texttt{SCANa}\ , & \texttt{POPa}\ ), \\ (\ 2 \to 2\ , & \texttt{SCANb}\ , & \texttt{POPb}\ ), \\ (\ 2 \to 3\ , & \texttt{EOF}\ , & \texttt{EMPTY}\ ) \end{array} \right\}.$$

Now run this program, step by step, on input `ab#ba`. Construct the history as you go along. Note that because the program is deterministic, there is only one choice of instruction at each step; if the program weren't deterministic there might be more than one way to run it. Define partial and complete histories. Define traces (the sequence of configurations) and computations (the sequence of instructions) and extract them from the history.

Lecture 3

Reading: Sections 2.8 and 2.9.

Concepts: determinism, blocking, transfer relation, transfer function, acceptors, recognizers, transducers

Lecture:

(There is no good word for the absence of determinism. Nondeterminism is overloaded. How about undeterminism? not-determinism?)

You can illustrate not-determinism and blocking with Figure 1.19.

A program is deterministic if there is no configuration (not even one that is never entered) to which two instructions are applicable; a deterministic program never

has a choice of which instruction to perform. This is a syntactic property, not a semantic property (Figure 2.9). Because determinism is a syntactic property we can test for it, which is important. Remember that determinism is a special case of nondeterminism.

A configuration is blocked if no instruction is applicable to it and it is not final; the program cannot produce a result or proceed from a blocked configuration. A program is nonblocking if no configuration (reachable or not) is blocked. This is a syntactic property.

The *transfer relation* τ of a program is the relation that holds between its arguments and results. $x\ \tau\ y$ iff the program, run with argument x, can produce result y. If the program is deterministic, then there is at most one result for each argument, so τ is a partial function, called the *transfer function.*

A common problem is to test whether a string belongs to a language L. Acceptors solve such problems, and tell you "yes" when the string is in the language. If $x \in L$ then the program has a complete computation with input x and result ACCEPT; if $x \notin L$, then the program blocks or runs forever. $x\tau$ = ACCEPT or $x\tau$ is undefined.

Acceptors are nondeterministic programs, but they may be deterministic. Emphasize that if some partial computations of a nondeterministic acceptor on input x are blocked or infinite, and some accept, then the program accepts x.

Input conventions: the argument is placed in the input device. The initial states of the other devices are independent of the argument.

Example: acceptor for palindromes.

Notation: NFA, NCA, NSA, NTA, DFA, DCA, DSA, DTA.

Recognizers solve the language membership problem and give a "yes" or "no" answer for each input. They are deterministic, nonblocking, and halt on every input. If $x \in L$ then the program has a complete computation with input x and result ACCEPT; if $x \notin L$ then the program has a complete computation with input x and result REJECT. $x\tau$ = ACCEPT or $x\tau$ = REJECT.

Input conventions: same as for acceptors.

Example: recognizer for palindromes with central marker.

Notation: DFR, DCR, DSR, DTR.

Closure under complementation.

Another problem is to compute partial functions, functions, or, more generally, multiple valued functions (relations) on strings. A transducer computes a relation ρ. If $x\ \rho\ y$, then there is a complete computation of the transducer with argument

x and result y. $\tau = \rho$.

Transducers are nondeterministic, but may be deterministic. Nondeterministic transducers may produce zero, one, or several different results for the same input x. Remember that blocked and infinite computations do not produce results. Whatever they leave on the output device is ignored.

Input conventions: same as for acceptors.

Output conventions: the result is equal to the output device's final state. The result does not depend on which final states the other devices end in.

Examples: Figures 2.13 and 2.14.

Topic 3

Simulation

Reading: Chapter 3.

Homework: 0.6-19, 3.2-3(b), 3.3-7, 3.3-9, 3.3-10, 3.4-2, 3.4-8, 3.4-15(b), 3.4-19, 3.5-2

Lecture 1

Reading: Sections 3.1 and 3.2

Concepts: Lockstep simulation. Simulating a counter by a stack. Simulating two controls with one.

Lecture:

Two programs are equivalent if they produce the same result for every input, i.e., if they have the same transfer relation. We will discuss somewhat general techniques called simulation for constructing programs that are equivalent to a given one. Typically the program we construct will be simpler in some way than the program we start with. If we can prove that the simple, but equivalent, program can't solve some particular problem, then we can conclude that the complicated program can't solve it either. In this way, simulation helps us to prove impossibility results.

Present lockstep simulation via the example of simulating a counter by a stack. Describe each of the three conditions of lockstep simulation and prove that they imply $\tau = \tau'$. Then present the simulation of two controls by one.

Lecture 2

Reading: Section 3.3.1

Concepts: Simulation via subprograms. Eliminating the NONZERO test from an unsigned counter.

Lecture: Show how to eliminate the NONZERO test from an unsigned counter. This introduces the notion of simulation via subprograms. (Why eliminate the NONZERO test? Restricting a program's behavior is a good things in general, because it makes subsequent proofs easier.) Then describe the simulation-via-subprograms method in general, but don't prove its validity.

Lecture 3

Reading: Sections 3.3.2 and 3.3.3

Concepts: An unsigned counter simulates a signed counter. Eliminating the EMPTY test from a stack.

Lecture: Present the simulation of a signed counter by an unsigned counter. (Mention that the reverse simulation is nontrivial but will be left as an exercise.) Then show how to eliminate the EMPTY test from a stack (this involves a nontrivial initial subprogram). Why eliminate EMPTY? The more we can restrict the behavior of programs, the easier it is to prove things about them.

Lecture 4

Reading: Section 3.4

Concepts: Factoring programs. Eliminating dead and unreachable states. Eliminating null instructions. Cleaning up and eliminating blocking.

Lecture:

Standardization means converting a program to an equivalent program that has a restricted form. We will present lots of standardizations today. Why standardize? We do some programming now so that proofs will be easier later.

(1) Factoring programs. In a factored program each instruction operates on only one device other than the control. Furthermore, each control state is *dedicated* to a single noncontrol device, meaning that all instructions going from it operate on the same noncontrol device. This standardization is easy to accomplish if you don't care about preserving determinism.

The book describes a general factoring technique that preserves determinism. First you have a tree of tests (so-called partitioning tests) that determine which instructions are applicable; then you perform the appropriate instruction, one noncontrol operation at a time. For every device we use, it is possible to express the partioning

tests in terms of the device's standard operations.

Note: This is overkill, to some extent, but the partioning tests like TOP are useful in some other contexts.

(2) Eliminating dead and unreachable states. A control state is dead if there is no path from it to an accepting control state. A control state is unreachable if there is no path from the start state to it. (Noncontrol operations on the path are ignored.) Simply delete dead and unreachable states and all instructions going to or from them.

(3) Eliminating null instructions. Each path consisting entirely of null instructions is combined with each nonnull instruction following it to form a nonnull instruction. If a path of null instructions leads to a final state, then the start of the path is made final.

(4) Cleaning up and eliminating blocking. To clean up a device, make sure it reverts to its initial state before termination. For example, to clean up an unsigned counter, repeatedly subtract one from it until its value is zero. To clean up a stack, pop characters from it until it is empty. Clean up devices only when the program is about to produce a result.

To eliminate blocking, for each situation in which the program would block, include a transition to a rejecting state. For example, if the program blocks in control state 2 when the counter holds a nonzero value and the top stack character is a `b` then include the instruction $(2 \rightarrow R, \text{DEC}, \text{POP}\texttt{b})$, where R is a rejecting state.

Note: If you run out of time, you can have the students read the end of section 3.4 on their own.

Topic 4

Finite Machines and Regular Languages

Reading: Chapter 4

Homework: 4.2-2, 4.2-4, 4.3-4, 4.4-6, 4.5-1(c), 4.6-1, 4.6-8, 4.7-14, 4.7-18(a), 4.8-2(a), 4.8-10, 4.8-13, 4.9-6(a), 4.10-3(c). Extra credit: 4.5-3(b), 4.10-5.

Teaching tips: Don't cover all the algorithms for computing regular sets of paths. If your students have trouble with pumping, cover Al and Izzy. Don't present a detailed proof of the composition theorem.

Lecture 1

Reading: Sections 4.1, 4.2

Concepts: Standardizing FM programs. Regular languages and expressions.

Lecture:

Why consider finite machine programs? They formalize what you can compute with very little space.

(1) Briefly discuss standardizations for NFAs and DFRs.

What we can already do: eliminate null instructions, dead states, and unreachable states from NFAs and DFRs; eliminate EOF from NFAs.

New standardization for DFRs: eliminate EOF. (Eliminate null instructions and standardize the input; if there is an instruction of the form $(q \rightarrow r, \text{EOF})$ then delete that instruction and make q a final state that gives the same result as r.)

Standard form for DFRs: Eliminate null instructions, EOF, dead states, and un-

reachable states. Then every instruction is of the form $(q \to r, \text{SCAN}c)$ and every state is on a path from the initial state to a final state.

New standardization for DFRs and NFAs: make sure no instruction goes to the initial state.

New standardization for NFAs: unique accepting state.

Standard form for NFAs: Eliminate EOF, and make sure no instruction goes to the initial state and there is a unique accepting state; then eliminate null instructions. Then every instruction is of the form $(q \to r, \text{SCAN}c)$, no instruction goes to the initial state, and the final state is unique.

(2) Present regular languages and expressions. Give the definitions and some examples. Motivation: regular expressions are a convenient way to express patterns. They are used in compilers and programs that do searches, e.g., grep, awk, and most editors.

Lecture 2

Reading: Sections 4.3 and 4.4

Concepts: egrep. Every NFA language is a regular language.

Lecture:

(1) Present egrep. Basically the same as regular expressions, but the symbols are different because of keyboard limitations. Also, the expression r matches $\Sigma^* r \Sigma^*$.

(2) Prove that NFA languages are regular languages (the pencil-and-paper method of Section 4.4.3 is best for classroom presentation).

Note: egrep is an optional topic. If you are pressed for time, you can omit egrep and squeeze the rest into lecture 3, but you'll have to talk fast.

Lecture 3

Reading: Sections 4.5, 4.6.

Concepts: regular language $\equiv$ NFA. NFA $\equiv$ DFA $\equiv$ DFR.

Lecture:

(1) Prove that regular languages are NFA languages.

(2) Prove that NFA languages are DFA languages. By eliminating EOF we convert a DFA to a DFR, so DFA languages are DFR languages.

Thus NFAs, DFAs, DFRs, and regular expressions are equivalent.

Lecture 4

Reading: Section 4.7

Concepts: Minimization algorithm, prefix equivalence, Myhill-Nerode theorem. $\{a^n b^n : n \geq 0\}$ is not regular.

Lecture:

Present the minimization algorithm for DFRs, prefix equivalence, and the Myhill–Nerode theorem. As an application, prove that $\{a^n b^n : n \geq 0\}$ is not regular. Stress the importance of lower bounds.

Do not present the algorithm to determine equivalent states. Just let the students read it.

Lecture 5

Reading: Section 4.8

Concepts:

- Closure properties: union, intersection, complementation, Kleene star, concatenation, quotient by arbitrary languages (nonconstructive)
- Algorithm for testing emptiness of regular language.
- Closure under quotient (constructive)
- Closure under finite transductions (deterministic finite transductions assigned as reading)
- Closure of machine-based language classes under finite transductions.

Lecture:

Today we will present lots of closure properties for regular languages. Closure properties are important them because you can use them in to prove that certain languages are regular and also to prove that certain other languages are not regular.

Present closure of regular languages under union, intersection, complementation, Kleene closure, concatenation, and quotient by arbitrary languages (nonconstructive).

By "constructive," we mean computable. Stress the difference between constructive and nonconstructive techniques. The difference will be important later (because reductions must be computable).

Note: I don't think this is worth mentioning in class, but mathematicians mean something different by "constructive." They mean that existence is demonstrated by a direct proof, rather than by contradiction.

Present an algorithm for testing emptiness of a regular language (minimize the DFR and see if there is only a rejecting state). Then the last proof can be made constructive for quotient by regular languages.

Prove closure under finite transductions. How? Compose a finite transducer for τ with an acceptor for L to show that $L\tau^{-1}$ is regular. (Just give the basic idea of a buffer so you don't have to store the output, but don't provide details). But the class of finite transductions is closed under converse (exchange input and output), so we are done. Note that the proof works as well for NSA or DSR languages, etc.

Construct a finite transducer that maps the set of palindromes over $\{\mathtt{a}, \mathtt{b}\}$ to $\{\mathtt{a}^n\mathtt{b}^n : n \geq 0\}$. Conclude that the set of palindromes over $\{\mathtt{a}, \mathtt{b}\}$ is not regular.

Lecture 6

Reading: Section 4.9

Concepts: Pumping theorem for regular languages. Palindromes not regular (proof via pumping). Primes not regular.

Lecture: Present the pumping theorem for regular languages. Using it, prove that the set of palindromes is not regular. Then use the pumping theorem to prove that $\{\mathtt{a}^n : n \text{ is prime}\}$ is not regular.

Topic 5

Context Free Languages

Reading: Sections 5.1, 5.2 (optional), 5.3, 5.4 (optional), 5.5–5.6, 5.8–5.9, 5.11, 5.12 (optional), 6.1

Homework: 5.1-1(a,b,c), 5.1-2, 5.2-10(d), 5.3-1(c,e,f), 5.6-1, 5.8-3(a,c,g) 5.9-4, 5.11-2(b), 6.1-7(c).

Lecture 1 (optional)

Reading: Sections 5.1 and 5.2

Concepts: Regular equations and containments. Parse trees. Tarski-Knaster theorem.

Lecture: Introduce regular equations as a generalization of regular expressions. The pal example in Section 5.1 implicitly introduces parse trees.

When proving the Tarski-Knaster theorem, use the standard partial order on sets in order to preserve the students' intuitions, but point out in advance that the proof really applies to more general partial orders. Prove that every regular equation in one variable has a least fixed point. Obtain the corollary for systems of equations in many variables by considering them as a single equation in a tuple of variables (redefine the partial order in the natural way).

Proof of the simple version of the Tarski–Knaster theorem: Let f be a monotone function. Call C a *contractor* if C satisfies $f(C) \subseteq C$. Let L be the intersection of all contractors. First we show that L is a contractor.

$$\begin{aligned} L &\subseteq C && \text{for all contractors } C \text{ (property of intersection)} \\ f(L) &\subseteq f(C) && \text{for all contractors } C \text{ (monotonicity)} \\ &\subseteq C && \text{for all contractors } C \text{ (def. of contractor)} \end{aligned}$$

$$\subseteq \; L \qquad \text{(property of intersection)}$$

Since $f(L) \subseteq L$, L is a contractor.

Because L is contained in every contractor (as we noted above), L is a least contractor.

Now we show that the least contractor is unique. If there are two least contractors, then each is contained in the other, so they are equal.

Next we show that $f(L) = L$. We have

$$\begin{aligned} f(L) &\subseteq L && \text{(because } L \text{ is a contractor)} \\ f(f(L)) &\subseteq f(L) && \text{(monotonicity), so } f(L) \text{ is a contractor} \\ L &\subseteq f(L) && \text{(because } L \text{ is a least contractor)} \\ f(L) &= L && \text{(combining the containments).} \end{aligned}$$

In fact, we will now show that L is the unique least solution to $f(X) = X$. We just saw that L is a solution. Every solution to $f(X) = X$ is a solution to $f(X) \subseteq X$; since L is contained in each of the latter, L is contained in each of the former. Thus L is a least solution to $f(X) = X$. If there were two least solutions to $f(X) = X$, then they would be contained in each other, hence equal, so L is the unique least solution to $f(X) = X$.

Lecture 2

Reading: Sections 5.1, 5.3, 5.4 (optional), 5.5

Concepts: Context-free grammars. Chomsky normal form. Derivations. Equivalence of parse trees and derivations (without proof).

Lecture:

(1) If you haven't already done so, introduce regular equations as a generalization of regular expressions; the pal example in Section 5.1 implicitly introduces parse trees.

(2) Describe the conversion to CNF in Section 5.3. The first three standardizations (multiply out by the distributive law, replace $=$ by $\supseteq$, and eliminate unions) produce context-free grammars. The remaining steps (eliminate mixed right sides, eliminate long right sides, eliminate Λ, and eliminate unit productions) produce Chomsky-normal-form context-free grammars.

Note: others also require the elimination of useless variables from CNF. A variable V is useless if V is not reachable from the start variable or if $L(V) = \emptyset$. The former is a simple graph accessibility problem. The latter is solved in Exercise 5.3-4.

(3) Define parse trees formally if you wish. Define partial derivations and derivations along with the notation $\Rightarrow$ and $\stackrel{*}{\Rightarrow}$. Define leftmost derivations as well. State without proof that $x \in L(A)$ iff there is a derivation $A \stackrel{*}{\Rightarrow} x$ iff there is a parse tree whose root is A and yield is x. State without proof that there is a one-one correspondence between leftmost derivations and parse trees.

Lecture 3

Reading: Section 5.6.

Concepts: CFL = NSA language.

Lecture: Show how to convert a CFL to an equivalent NSA. Present the unnumbered example given in Figures 5.8 and 5.9. Show how to convert an NSA to an equivalent CFL. Present Example 5.19.

Lecture 4

Reading: Section 5.8

Concepts: First pumping theorem for CFLs. Ogden's lemma and theorem (without proof).

Lecture:

(1) State and carefully prove the first pumping theorem for CFLs. Carefully present Example 5.22.

(2) Recall the two pumping theorems for regular languages; the second lets us specify where in a string pumping occurs. The same is desirable for CFLs. Define the notion of marked characters, and give an example, as on page 361. State Ogden's derivation-pumping lemma for CFGs, plus the conclusion $uv^iwx^iy \in L$ for the sake of motivation. Point out the analogy to the first pumping theorem for CFLs. Ask students if they believe the lemma (this can be a good way to clear up misunderstandings like thinking that v and x consist entirely of marked characters). If time remains, present Example 5.27.

Lecture 5

Reading: Section 5.9

Concepts: Ambiguity. Inherent ambiguity. The existence of an inherently ambiguous CFG.

Lecture:

(1) Define ambiguity, and illustrate it with ALGOL 60's "dangling else," which can be disambiguated.

(2) Unfortunately some ambiguous grammars cannot be disambiguated. Define inherent ambiguity and prove, using Ogden's derivation-pumping lemma, that $\{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k : i = j \text{ or } j = k\}$ is inherently ambiguous.

Lecture 6

Reading: Sections 5.11 and 6.1

Lecture:

(1) Present the CYK dynamic-programming algorithm for testing membership in a CFL. Mention that dynamic programming is a general technique; they will see it again in their algorithms class if they haven't seen it already.

(2) Present closure properties for CFLs: regular operations (union, concatenation, Kleene-closure), reversal, finite transductions, intersection with a regular language, quotient by a regular language

Note: when proving closure under reversal, it is interesting to talk about reversing time (running a program backwards). If we reversed time for every device, then the program would still accept the same language. If we reversed time for the input device but no other device, then the argument would be input in reverse order so the program would accept the reverse of the language. But it is not feasible to reverse time for the input.

A feasible approach is to reverse time for everything but the input. This is equivalent to reversing time for the input device and then reversing time for every device, so the program would accept the reverse of the language. What's more, reversing time for a device corresponds to replacing each operation on the device by its converse, which we can do for all common devices except input and output.

(3) Present closure properties for DCFLs: complementation, converse deterministic finite transductions, intersection with a regular language, union with a regular language.

Encourage the students to ask about other possible closure properties.

Note: In Section 6.4 we will prove that the class of DCFLs is closed under quotient by regular languages and concatenation on the right by regular languages.

Lecture 7 (optional)

Reading: Section 5.12

Concepts: Earley's algorithm

Lecture: Discuss the advantages and disadvantages of Earley's algorithm vs. the CYK algorithm. Earley is faster in important special cases: unambiguous grammars and especially DCFLs. CYK is a little easier to implement.

Present the algorithm.

Topic 6

Stack and Counter Machines

Reading: Sections 6.2, 6.3, 6.4 (optional), 6.5, 6.6

Note: Section 6.1 was covered under topic 5.

Homework: 6.2-1, 6.4-2(b), 6.5-1

Lecture 1

Reading: Section 6.2

Concepts: DSA $\equiv$ DSR. All-inputs halting problem for DSM programs. Not every CFL is a DCFL.

Lecture:

Today we will show how to convert a DSA to an equivalent DSR, so DCFLs, DSA languages, and DSR languages are all the same thing. Why is this important? It means that the class of DCFLs is closed under complementation. Since the class of CFLs is not, it means we can prove that some CFLs are not DCFLs; this is a case where nondeterministic programs are provably more powerful than deterministic programs for the same kind of machine. We will also be able to test whether a DSM program halts on all inputs.

(1) Eliminating PUSH–POP pairs. We consider *deterministic* stack acceptors and recognizers exclusively in this lecture. A PUSH–POP pair is a PUSH*c* followed by a POP*c*, with no intervening nonnull instructions (there can be no intervening SCAN, for example, in a PUSH–POP pair). In a deterministic program, anything that can happen must happen, so the PUSH*c* must inevitably be cancelled by the POP*c*. Therefore the particular instruction performing the PUSH*c* (but not the POP*c*, which may have other uses) can be removed from the program and replaced by a null instruction that simulates the PUSH–POP pair. Repeat this process (which may

in fact create PUSH–POP pairs) until there are no PUSH–POP pairs left; termination is guaranteed because the number of PUSH operations decreases with each iteration.

In a program without PUSH–POP pairs every PUSH is followed by an input operation, EMPTY, or another PUSH.

(2) DSA $\equiv$ DSR. Start with a DSA P. The hard part of the standardization is to guarantee termination. Factor P so that each instruction operates on at most one device. Eliminate dead states. Standardize the input so that EOF is performed at most once. Standardize the stack so that two consecutive EMPTY tests are not performed. Eliminate PUSH–POP pairs. Finally eliminate null instructions.

(Why didn't we eliminate EMPTY entirely as in Section 3.3.3? Because we want the same technique to work for counters.)

Because the input is standardized, any partial computation of P contains at most $|x|+1$ input operations, which divide the partial computation into $|x|+2$ fragments. In each fragment, there is a sequence of POP operations, possibly one EMPTY test, and then a sequence of PUSH operations. If there were $|Q|$ consecutive PUSH operations the program would repeat a configuration, so it would be in an infinitely repeating loop; that loop cannot contain an accepting state because the program is deterministic; therefore it consists of dead states, but dead states were eliminated. Therefore there can be at most $|Q|-1$ consecutive PUSH operations. In total, the partial computation can contain at most $(|x|+2)(|Q|-1)$ PUSH operations. Therefore it contains at most $(|x|+2)(|Q|-1)$ POP operations. It contains at most $|x|+2$ EMPTY tests. Thus the total number of operations in the partial computation is bounded, so there are no infinite computations.

Now eliminate blocking to produce an equivalent recognizer.

Theorem: the class of DCFLs is equal to the class of DSR languages.

Corollary: the class of DCFLs is closed under complementation.

Proof: interchange accepting and rejecting states.

Corollary: there is an algorithm to test whether a DCFL is equal to Σ^*.

Proof: Test the complement (a CFL) for emptiness.

Corollary: there is an algorithm to determine whether a DSM program halts on all inputs.

Proof: By removing the output device (if any) and making all final states accepting, we produce a DSA for the set of strings that P halts on. See whether the DCFL it accepts is Σ^*.

Theorem: Not every CFL is a DCFL.

Proof: Let

$$L = \{a^i b^j c^k : i \neq j \text{ or } j \neq k\}.$$

L is a CFL. Assume, for the sake of contradiction, that L is a DCFL. Then

$$\overline{L} \cap a^* b^* c^* = \{a^i b^j c^k : i = j = k\}$$

is a DCFL by closure under complementation and under intersection with regular languages. But that language isn't even a CFL. This contradiction proves that L is not a DCFL.

Remember that NFAs are equivalent to DFAs. But we have just seen that NSAs are more powerful than DSAs. Thus nondeterministic programs can be more powerful than deterministic ones for the same machine type. This is not just an intuition, because we have proved it.

Lecture 2 (optional)

Reading: Section 6.4

Concepts: On-line recognition. DCFLs closed under Double-Duty($\cdot$). Palindromes are not a DCFL.

Lecture: Today we will prove that palindromes (our favorite CFL) are not a DCFL.

(1) On-line recognition. After scanning the entire input, a DSR might spend a lot of time playing with the stack before deciding whether to accept or reject. We want to standardize DSRs so they decide immediately after finishing the input.

The following is substantially simpler than what is in the book.

Start with a DSR. Factor it. Standardize the input device so that EOF is performed at most once. Standardize the stack so that EMPTY is not performed twice in a row. Clean up the stack before the program accepts or rejects. Eliminate PUSH–POP pairs. Eliminate null instructions. Call the standardized program P. Then P performs no PUSH operations after EOF (because those PUSH's could not be followed by POP's).

Let F be the set of all partial functions from Q to Q. Define a partial function $p_c \in F$ by $qp_c = r$ if $(q \to r, \text{NOOP}, \text{POP}c)$ is an instruction of P; qp_c is undefined if there is no such instruction. Let $qp_\Lambda = r$ if $(q \to r, \text{NOOP}, \text{EMPTY})$ is an instruction of P; qp_Λ is undefined if there is no such instruction. (p_c and p_Λ are partial functions because P is deterministic.)

Let P' have stack alphabet $\Gamma \times F$. When P would push a character c on the stack, let P' determine its top stack character, say (d, f), and push (c, fp_c). If the

stack is empty, P' pushes (c, p_c). When P would pop a character c, P' pops (c, f) regardless of the value of f. Otherwise, P' does exactly what P would do. As soon as the input is exhausted, P' accepts if qf or qfp_Λ is an accepting state, where q is its control state and (d, f) is the top stack character (qf is the state that P would be in immediately after emptying its stack). Note: P' really keeps the so-called top stack character in a buffer, so that it is accessible without performing additional instructions.

(2) The following closure property is useful for proving negative results.

$$\text{Double-Duty}(L) = \{x\#y : x \in L \text{ and } xy \in L\}.$$

The class of DCFLs is closed under Double-Duty$(\cdot)$ because you can run an on-line recognizer P one step at a time on input $x\#y$, see if x is accepted, have P ignore the #, and continue running P to see if xy is accepted.

(3) If L is the language of palindromes over $\{\mathtt{a}, \mathtt{b}\}$ then

$$\text{Double-Duty}(L) \cap (\mathtt{a}^*\mathtt{b}\mathtt{a}^*\#\mathtt{b}\mathtt{a}^*) = \{\mathtt{a}^i\mathtt{b}\mathtt{a}^i\#\mathtt{b}\mathtt{a}^i : i \geq 0\},$$

which is not even a CFL. So the set of palindromes over $\{\mathtt{a}, \mathtt{b}\}$ is not a DCFL.

Lecture 3

Reading: Sections 6.5 and 6.6

Concepts: Simulating a stack by 2 counters. Simulating many stacks by 2 counters.

Lecture:

(1) Two counters can simulate a stack. For simplicity assume the stack alphabet is $\{1, 2\}$. The first counter holds the dyadic value of the stack, and the second counter is used for temporary storage. To test for emptiness, see if the first counter holds 0. To push c, multiply the first counter value by 2 and add c. To pop c, divide the first counter by 2 and check that the remainder is c. (Multiplication is accomplished by counting down on the first counter while counting up twice as fast on the second, then moving the value from the second counter back to the first. Division is similar.)

(2) Two counters can simulate three counters. Call the simulated counters $K1$, $K2$, and $K3$. The first simulating counter holds $2^{K1}3^{K2}5^{K3}$, and the second is used for temporary storage. To test whether $K1$ is zero, divide by 2 and check that the remainder is nonzero. To increment $K1$, multiply by 2. To decrement $K1$ divide by 2 and check that the remainder is zero. The other counters are handled similarly.

(3) Thus k counters can simulate $k+1$ counters, if $k \geq 2$. By transitivity, 2 counters can simulate any number of counters, so 2 counters can simulate any number of stacks.

Topic 7

Computability

Reading: Sections 7.1–7.4, 7.6–7.9, 7.12 (optional), 7.13, 7.14 (optional)

Homework: 7.1-7, 7.6-1, 7.9-2(a,c,e,g,i), 7.9-3, 7.13-1(b,g)

Lecture 1

Reading: Sections 7.1 and 7.3

Concepts: Quick review of tape operations. 2 stacks simulate 1 tape. 1 tape simulates k tapes. k tapes simulate a RAM.

Lecture:

(1) Review the tape operations: SEEc, PRINTc, MOVEL, MOVER, ATHOME. Also the shorthand [SEEc, PRINTd, MOVED].

(2) Two stacks simulate a tape. A control stores the character under the tape head; one stack stores the contents of the tape left of the head; and the other stores the contents of the tape right of the head, but in reverse. Because the characters next to the tape head are the top stack characters they can be easily accessed as the tape head moves.

(3) One tape simulates k tapes. One square of the simulating tape stores the contents of the corresponding square of all simulated tapes. To simulate an operation on one of the simulated tapes, you first have to find which square its tape head is on, then you can update its contents, and finally move the simulating tape head back to the home square.

(4) k tapes simulate a RAM. Use one tape for each register, one tape for the RAM's memory array (stored as a sequence of ordered pairs), and one for temporary storage. When simulating a register operation, use the temporary storage to perform

any arithmetic. To simulate memory access, find the appropriate ordered pair, update its value (which may now be longer or shorter than before), and recopy the remaining ordered pairs (which may have to be moved to make room for the new value).

(5) By transitivity, one tape can simulate a RAM.

Note: The simulations we have presented today are not very inefficient. This will be important when we get to Chapter 9.

Lecture 2

Reading: Sections 7.4 and 7.6

Concepts: Universal TM programs. Recursive functions, partial recursive functions, recursive languages, r.e. languages. Computations of a nondeterministic program can be checked deterministically using the same kind of machine. NTA $\equiv$ DTA.

Lecture:

(1) A universal Turing machine program is an interpreter for Turing machine programs — itself running on a Turing machine. The basic idea is pretty easy: you write a simple RAM program that can emulate Turing machine programs and then argue that the RAM program can itself be simulated by a Turing machine program.

Details of a RAM program that takes as input a pair (P, x) and determines the result of running the 1-tape deterministic TM program P on input x: Read and store the instructions of P in the first m memory locations. Read and store the characters of x in memory locations $m+1, \ldots, n$. Store $m+1$ in register 1. Register 1 will be a pointer into x. Store n in register 2, to permanently mark the end of x. Memory locations $n+1$ and higher will hold the contents of P's tape. Register 3 is a pointer to P's tape head. Register 4 holds P's control state. To emulate an instruction of P, the RAM program looks through the list of instructions to find one that is applicable and then updates its registers and memory accordingly. (It is clear that the RAM program has access to all the information it needs for that purpose.) The RAM emulates instructions of P until P halts.

Note: coding a large alphabet by a small one is not much of an issue in this construction, because we use a RAM, which can store arbitrarily large numbers in a register or memory location. The coding is taken care of when we simulate the RAM by a TM.

Note: Our weaker machine models (finite machine, counter machine, stack machine) do not have universal programs.

(2) Note: By DTM program, the textbook usually means deterministic Turing transducer (DTT). By NTM program, the textbook usually means nondeterministic Turing transducer (NTT).

Definitions: The *recursive functions* are the total functions computed by DTTs. The *partial recursive functions* are the partial functions computed by DTTs. The recursive languages are the languages recognized by DTRs. The r.e. languages are the languages accepted by NTAs.

(3) Computations of a nondeterministic program can be checked deterministically using the same kind of machine — just execute the sequence of instructions, if possible, and see if the resulting configuration is in fact final. In particular, if P is an NTA, the set of computations of P is a recursive language.

A sequence of instructions can be represented as a number (use base $|\mathcal{I}|$). A DTA can simulate an NTA by trying every possible sequence of instructions, in order, to see if it is a computation of the NTA on input x. This gives another characterization of the r.e. languages: a language is r.e. iff it is accepted by a DTA.

Lecture 3

Reading: Section 7.6

Concepts: The r.e. languages are projections of the recursive languages. L is recursive iff L is r.e. and co-r.e. A nonempty set L is r.e. (recursive) iff L is the range of a total recursive (and nondecreasing) function.

Lecture:

(1) L is r.e. iff there exists a recursive set R such that

$$x \in L \iff (\exists y)[(x,y) \in R].$$

Proof: If L is r.e. then let P be an NTA that accepts L and let $R = \{(x,C) : C \text{ is an accepting computation of } P \text{ on input } x\}$. For the converse, an NTA can guess a y and check that $(x,y) \in R$.

Note: L is the geometrical projection of R onto the x-axis.

(2) L is recursive iff L is r.e. and co-r.e. Proof: If L is recursive then $\overline{L}$ is recursive and therefore L and $\overline{L}$ are r.e. For the converse, timeshare DTAs for L and $\overline{L}$ (alternately running each for a single step); one of them must eventually accept.

(3) Let L be nonempty. L is r.e. iff L is the range of a total recursive function. Proof: Since L is nonempty let a be an element of L. Let P be an NTA that accepts L. Let $f(C) = x$ if C is an accepting computation of P on input x; let

$f(C) = a$ if C is not an accepting computation of P on any input. Then L is the range of the total recursive function f.

For the converse, note that the range of f is $\{y : (\exists x)[y = f(x)]\}$, which is r.e. by (1) (with $R = \{(y, x) : y = f(x)\}$).

(4) Let L be nonempty. L is recursive iff L is the range of a total recursive, nondecreasing function. Proof: Let L be nonempty and recursive. Let a be the least element of L. Define a total recursive, nondecreasing function f as follows:

$$f(x) = \begin{cases} a & \text{if } x = 0 \\ x & \text{if } x > 0 \text{ and } x \in L \\ f(x-1) & \text{if } x > 0 \text{ and } x \notin L. \end{cases}$$

Then L is the range of f.

For the converse, if the range of f is finite then it is recursive. If the range of f is infinite, then we can test whether x belongs to the range of f by computing $f(0), f(1), \ldots$ until we find the first i such that $f(i) = x$ or $f(i) > x$.

Lecture 4

Reading: Sections 7.7, 7.8

Concepts: The halting problem, Russell's barber paradox, undecidability of the halting problem. Diagonalization. Distinguishability problems, the existence of recursively inseparable r.e. sets.

Lecture:

(1) The halting problem is to determine, for P and x, whether DTM program P halts on input x.

$$K_{xy} = \{(P, x) : \text{DTM program } P \text{ halts on input } x\}.$$

(2) Russell's barber has a sign that says, "I shave those men, and only those, who do not shave themselves." In other words, for every man x,

$$\text{the barber shaves } x \iff \neg(x \text{ shaves } x),$$

which implies that the barber is not equal to x. Since that is true for every man x, the barber is not a man. Any questions? We will apply the same idea to Turing machine programs.

(3) There is no algorithm to solve the halting problem.

Proof: Suppose there was. Then we could use it as a subroutine in a DTA P that behaves as follows:

```
input a DTA x;
if program x halts on input x then
    loop forever
else
    halt;
```

Observe that, for every DTA x,

$$P \text{ halts on input } x \iff \neg(x \text{ halts on input } x),$$

which implies that P is not equal to x. Since that is true for every DTA x, P is not a DTA. This contradiction implies that there is no algorithm that solves the halting problem. Any questions?

(4) Russell made his barber different from every man. He did it by making the barber behave differently from each man x in one little way.

We made our program P different from every DTA. We did it by making P behave different differently from each DTA x on just one input.

Could we have made P differ from two programs x and x' on the same input? We'd have to find an input where both x and x' halt or neither halt. It's easier to make P differ from x and x' on distinct inputs. Which inputs? $x \neq x'$. A simple choice is to make P differ from program x on input x and differ from program x' on input x'. That's what we did. That's essentially what Russell did (with men and beards rather than programs and inputs). The general technique is called "diagonalization."

(5) Let S_1 and S_2 be disjoint languages. A recognizer *distinguishes* S_1 from S_2 if it accepts all strings in S_1 and rejects all strings in S_2 (it may accept or reject strings in $\overline{S_1 \cup S_2}$ but it must give a result on all inputs). For example, a program might help an investor by distinguishing excellent stocks from terrible stocks, even though fairly good stocks might be hard to distinguish from fairly bad stocks.

(6) A pair of disjoint languages are called *recursively separable* if there is a DTR that distinguishes them; they are called *recursively inseparable* otherwise. For example, S is recursive iff S and $\overline{S}$ are recursively separable.

Example of nonrecursive, but recursively separable, sets:

$$\begin{aligned} S_1 &= \{P : P \text{ has an odd number of control states and } P \in K_{inatt}\}, \\ S_2 &= \{P : P \text{ has an even number of control states and } P \in K_{inatt}\}. \end{aligned}$$

Example of r.e., but recursively inseparable, sets:

$$K_{\text{REJECT}} = \{x : x \text{ is a DTM program and } \tau_x(x) = \text{REJECT}\},$$

$$K_{\text{ACCEPT}} = \{x : x \text{ is a DTM program and } \tau_x(x) = \text{ACCEPT}\};$$

i.e., K_{ACCEPT} is the set of programs that accept themselves and K_{REJECT} is the set of programs that reject themselves.

Why are they recursively inseparable? Suppose that P distinguishes K_{ACCEPT} from K_{REJECT}. Then, for all DTM programs x,

$$\begin{array}{rcl} P \text{ accepts } x & \Rightarrow & x \notin K_{\text{ACCEPT}} \Rightarrow x \text{ does not accept } x, \\ P \text{ rejects } x & \Rightarrow & x \notin K_{\text{REJECT}} \Rightarrow x \text{ does not reject } x. \end{array}$$

Thus, for every deterministic Turing *recognizer* x,

$$P \text{ accepts } x \iff x \text{ does not accept } x,$$

so $P \neq x$. Thus P is not a DTR.

Lecture 5

Reading: Section 7.9

Concepts: many-one reductions. Their use in proving undecidability and non-r.e.-ness. All versions of K are equivalent. K is complete.

Lecture:

We won't usually prove undecidability results from scratch. Instead we use one result to prove others. Today we will see how this is done.

(1) A problem A is *reducible* to a problem B if there is an algorithm that solves problem A using the ability to solve problem B as a subroutine. A language A is *reducible* to a language B if there is an algorithm that tests membership in A using the ability to test membership in B as a subroutine.

If A is reducible to B and B is recursive then A is recursive. Contrapositive: If A is reducible to B and A is nonrecursive then B is nonrecursive. Thus reductions may sometimes be used to prove a language is recursive and may sometimes be used to prove a language is nonrecursive.

A special kind of reduction tests membership in B at one well-chosen point and outputs the answer to that membership test. A is m-reducible to B (denoted $A \leq_m B$) if there exists a total recursive function f such that, for all x,

$$x \in A \iff f(x) \in B.$$

Think of A as at least as easy as B, and B at least as hard as A.

- If A is recursive what does that say about B? Nothing.
- If B is recursive, A is recursive.
- If A is nonrecursive, B is nonrecursive.
- If B is r.e., A is r.e.
- If A is non-r.e., B is non-r.e.

(2)

- $K_{xy} = \{(x,y) : \text{DTM program } x \text{ halts on input } y\}$
- $K_{diag} = \{x : \text{DTM program } x \text{ halts on input } x\}$
- $K_{inatt} = \{x : \text{inattentive DTM program } x \text{ halts}\}$
- $K_{pos} = \{x : \text{DTM program } x \text{ halts on at least one input}\}$

$x \in K_{diag} \iff (x,x) \in K_{xy}$, so $K_{diag} \leq_m K_{xy}$.

Let $f(x,y)$ = "Run DTM program x on input y." (The value of $f(x,y)$ is an inattentive DTM program, expressed in a high-level language, not the result of running that program!). Then $(x,y) \in K_{xy} \iff f(x,y) \in K_{inatt}$, so $K_{xy} \leq_m K_{inatt}$.

Let $f(x)$ = "Input z; run inattentive DTM program x." Then $x \in K_{inatt} \iff f(x) \in K_{pos}$, so $K_{inatt} \leq_m K_{pos}$.

Let $f(x)$ be a program that ignores its input and searches for a complete computation of program x. Then $x \in K_{pos} \iff f(x) \in K_{diag}$, so $K_{pos} \leq_m K_{diag}$.

Therefore, K_{diag}, K_{xy}, K_{inatt}, and K_{pos} are all equally difficult. Since K_{xy} is nonrecursive, all of them are nonrecursive. They are all non-co-r.e. as well.

(3) The halting problem for 2-counter machine programs is undecidable. If x is a TM program, let $f(x)$ be an equivalent 2-CM program. $x \in K_{xy}$ iff $f(x) \in K_{2\text{-}CM}$. Because we can effectively construct the equivalent 2-CM program, f is a recursive function. Similarly, the halting problem for 2-stack machine programs is undecidable.

(4) In fact, every r.e. set is m-reducible to K_{xy}. Why? Let A be an r.e. set, so A is accepted by a DTA P. An acceptor halts iff it accepts, so

$$x \in A \iff (P,x) \in K_{xy}.$$

Thus K_{xy} is a hardest r.e. set. So are K_{diag}, K_{inatt}, and K_{pos}. Such sets are called *complete*. (The name K comes from the German word for complete.)

(5) Practice with reductions (Exercise 7.9-2(b)). Let

$$L = \{x : x \text{ is a DTA that accepts a regular language}\}.$$

Let $f(x)$ be a program that behaves as follows:

```
input z;
if z has the form a^n b^n c^n then accept;
if x halts on input x then accept;
```

If $x \in K$ then $f(x)$ accepts Σ^*; otherwise $f(x)$ accepts $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n : n \geq 0\}$. Thus $x \in K \iff f(x) \in L$. Since K is not co-r.e., L is not co-r.e.

Let $g(x)$ be a program that behaves as follows:

```
input z;
if z has the form a^n b^n c^n then
    if x halts on input x then accept;
```

If $x \in K$ then $g(x)$ accepts $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n : n \geq 0\}$; otherwise $g(x)$ accepts $\emptyset$. Thus $x \in \overline{K} \iff f(x) \in L$. Since $\overline{K}$ is not r.e., L is not r.e.

Thus, L is neither r.e. nor co-r.e. A fortiori, L is not recursive.

Lecture 6

Reading: Section 7.13

Concepts: Valid and invalid computations.

Undecidability results for CFLs:

- is the intersection of 2 DCFLs empty?
- is a CFL equal to Σ^*?
- are 2 CFLs equal to each other?
- are a CFL and a regular language equal to each other?
- is the intersection of 2 DCFLs a CFL? a DCFL? a regular language? a finite language?
- is a CFL a co-CFL? a DCFL? a regular language? a co-finite language?

- is a CFG ambiguous?
- is a CFL inherently ambiguous?

Lecture: We will prove lots of undecidability results for CFLs today. These results build on the undecidability of the halting problem and on each other.

(1) Let $\mathcal{C}(P)$ denote the set of computations of the program P. Observe that P halts iff $\mathcal{C}(P) \neq \emptyset$. Since the halting problem for 2-CM programs is undecidable, there is no algorithm that takes a 2-CM program P as input and determines whether $\mathcal{C}(P)$ is empty.

If P is a 2-counter machine program then the set of computations of P is the intersection of two deterministic 1-counter languages (let L_1 consist of all sequences of instructions of P that are executable on the first counter; let L_2 consist of all sequences of instructions of P that are executable on the second counter, the control, and the input). Therefore there is no algorithm to determine whether the intersection of two DCR languages (or two NCA languages, or two DCFLs, or two CFLs) is empty.

$L_1 \cap L_2 = \emptyset$ iff $\overline{L}_1 \cup \overline{L}_2 = \Sigma^*$. Therefore there is no algorithm to determine whether an NCA language (or a CFL) is equal to Σ^*. Consequently, there is no algorithm to determine whether a CFL and a regular language (or two CFLs) are equal to each other.

(2) Let D_1 be a DSR for L_1 and let D_2 be a DSR for L_2. Let N be an NSA that chooses nondeterministically whether to run D_1 or D_2. On inputs belonging $L_1 - L_2$ or $L_2 - L_1$, N has exactly one accepting computation. On inputs belonging to $\overline{L_1 \cup L_2}$, N has zero accepting computations. On inputs belonging to $L_1 \cap L_2$, N has exactly two accepting computations. Convert N to a CFG G in the standard way. G has two parse trees for strings belonging to $L_1 \cap L_2$, and one or fewer for other strings. Thus G is ambiguous iff $L_1 \cap L_2 \neq \emptyset$, so there is no algorithm to determine whether a CFG is ambiguous.

(3) Now let L be the following CFL:

$$\{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k\#x : x \in L_1 \text{ and } i = j\} \cup \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k\#x : x \in L_2 \text{ and } j = k\}.$$

If $L_1 \cap L_2 = \emptyset$ then the grammar constructed for L in the standard way (using closure of NSA languages under concatenation and union, then converting the NSA to a CFG) is unambiguous.

If $L_1 \cap L_2 \neq \emptyset$, then let s be any element of $L_1 \cap L_2$. Any CFG for L must have two parse trees for $\mathtt{a}^{n+n!}\mathtt{b}^{n+n!}\mathtt{c}^{n+n!}\#s$ for all sufficiently large n (as in the proof of Theorem 5.31). (Alternatively, use closure of unambiguous CFLs under converses of deterministic finite transductions. Define $x\tau = x\#s$. Then $L\tau$ is the language proved inherently ambiguous in Theorem 5.31.) L is inherently ambiguous iff

$L_1 \cap L_2 \neq \emptyset$, so there is no algorithm to determine whether a CFL is inherently ambiguous.

(How do you state the inherent ambiguity problem formally? The problem instance should be a string. Encode the CFL by a CFG or NSA.)

(4) Now let

$$\begin{aligned} L_1' &= \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k\#x : x \in L_1 \text{ and } i = j\} \\ L_2' &= \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k\#x : x \in L_2 \text{ and } j = k\}. \\ L_1' \cap L_2' &= \{\mathtt{a}^i\mathtt{b}^i\mathtt{c}^i\#x : x \in L_1 \cap L_2\}. \end{aligned}$$

If $L_1 \cap L_2$ is empty, then $L_1' \cap L_2'$ is empty (hence a CFL, a DCFL, a regular language, and a finite language). If $L_1 \cap L_2$ is nonempty, then $L_1' \cap L_2'$ is not a CFL (apply a finite transduction that removes the # and everything following it) (hence not a DCFL, a regular language, or a finite language). Thus there is no algorithm to determine whether the intersection of two DCR languages (or two CFLs) is any of the following: a CFL, a DCFL, a regular language, or a finite language.

$\overline{L_1' \cap L_2'} = \overline{L_1'} \cup \overline{L_2'}$, which is an NCA language. Therefore there is no algorithm to determine whether an NCA language (or a CFL) is any of the following: a co-CFL, a DCFL, a regular language, or a co-finite language.

Lecture 7 (optional)

Reading: Section 7.12

Concepts: The set of theorems in first-order logic is nonrecursive.

Lecture:

First-order logic (FOL) consists of statements involving constants, variables, functions, and Boolean predicates, as well as the usual logical symbols $\wedge$, $\vee$, $\neg$, $\exists$, $\forall$. Additional operations like $\Rightarrow$ can be defined in terms of the usual logical symbols. A statement in FOL is a theorem if it can be proved using the usual logical rules of inference. We will reduce the halting program for inattentive 2-counter machines to the set of theorems in FOL. We use the fact that FOL is sound, meaning that its theorems are true in every model (interpretation).

Note: The set of theorems in FOL is r.e. because we can guess a proof and check each step.

Note: The proof does not go through for FOL without quantifiers.

Note: Although we have only stated the halting problem for deterministic programs, the reduction works for nondeterministic programs as well.

Fix an inattentive 2-counter machine program P. Standardize P so that it halts iff

it reaches the configuration $(q_{\text{accept}}, 0, 0)$. For each control state q we have a two-place predicate $f_q(\cdot,\cdot)$. We intend $f_q(i,j)$ to mean that the configuration (q,i,j) is reached in some partial computation of P. We have a one-place function $s(\cdot)$. We intend $s(i)$ to mean $i+1$. We have a constant $\mathbf{z}$. We intend $\mathbf{z}$ to mean 0.

Corresponding to P's initial configuration, we have the statement $f_{q_{\text{start}}}(\mathbf{z},\mathbf{z})$.

Corresponding to each instruction π we define a statement ψ_π as in the table. Let H (standing for hypotheses) be the conjunction of all the statements given so far.

Instruction π	Statement ψ_π
($q \to r$, INC , NOOP)	$(\forall i,j)[f_q(i,j) \Rightarrow f_r(s(i),j)]$
($q \to r$, DEC , NOOP)	$(\forall i,j)[f_q(s(i),j) \Rightarrow f_r(i,j)]$
($q \to r$, ZERO , NOOP)	$(\forall j)[f_q(\mathbf{z},j) \Rightarrow f_r(\mathbf{z},j)]$
($q \to r$, NOOP , INC)	$(\forall i,j)[f_q(i,j) \Rightarrow f_r(i,s(j))]$
($q \to r$, NOOP , DEC)	$(\forall i,j)[f_q(i,s(j)) \Rightarrow f_r(i,j)]$
($q \to r$, NOOP , ZERO)	$(\forall i)[f_q(i,\mathbf{z}) \Rightarrow f_r(i,\mathbf{z})]$

Corresponding to P's final relation we have the statment

$$f_{q_{\text{accept}}}(\mathbf{z},\mathbf{z}).$$

We assert that P halts iff the statement $H \Rightarrow f_{q_{\text{accept}}}$ is a theorem, i.e., iff there is a proof of the following statement:

$$\left(f_{q_{\text{start}}}(\mathbf{z},\mathbf{z}) \wedge \bigwedge_{\pi\in\mathcal{I}} \psi_\pi\right) \Rightarrow f_{q_{\text{accept}}}(\mathbf{z},\mathbf{z}).$$

Prove by induction that if $(q_{\text{start}},0,0) \overset{\Pi^k}{\mapsto} (q,i,j)$ then there is a proof that $H \Rightarrow f_q(s^{(i)}(\mathbf{z}), s^{(j)}(\mathbf{z}))$ (the exponent denotes iterated composition of s with itself). You might like to go into detail for one particular instruction π, but I wouldn't recommend going over all six types of instruction. The special case where $(q,i,j) = (q_{\text{accept}},0,0)$ completes half of the proof.

For the converse, if there is a proof of the statement, then interpret the symbols in the desired way, i.e., $\mathbf{z} = 0$, s is the successor function, $f_q(i,j)$ denotes that configuration (q,i,j) is reached by some partial computation of P. H is true in this interpretation, so $f_{q_{accept}}(\mathbf{z},\mathbf{z})$ is true, i.e., the configuration $(q_{\text{accept}},0,0)$ is reached, so P halts.

Epilogue: We have just seen that a problem outside computer science can be undecidable. Today we coded computation into logic in order to prove a logic

problem undecidable. We can also code computation into polynomials and prove that there is no algorithm to find an integral solution to a polynomial equation in several variables.

Lectures 8 and 9 (optional)

Reading: Section 7.14

Concepts: Matijasevič's theorem.

Note: the name is pronounced Matty-yasevich

Topic 8

Recursion Theory (optional)

Reading: Sections 8.1, 8.2, 8.3 (optional), 8.4, 8.5

Homework: 8.2-1, 8.2-4, 8.4-1(c), 8.3-4, 8.4-5(a), 8.5-2, 8.5-4, 8.5-6, 8.5-7(a,k)

Note: Compare Exercises 8.5-2(e) and 8.5-6.

Lecture 1

Reading: Sections 8.1 and 8.2

Concepts: Rice's theorem. The fixed-point and recursion theorems.

Note: You are probably wondering why Rice's theorem is not included in topic 7 (computability). I have found that if students learn Rice's theorem early, they learn only Rice's theorem and no other method for proving undecidability. Students who learn only Rice's theorem think it applies to every problem, so they don't really even learn Rice's theorem.

Lecture:

(1) Rice's theorem says that every nontrivial property of a TM program's transfer relation is undecidable. What's a property of the transfer relation? For example, does the program produce a result on input abab, does it compute a one-one function? What's not a property of the transfer relation: For example, does the program have at least 5 control states, does it have a computation of length 17 or greater?

Warning: Rice's theorem is a beautiful general result, but there are lots of problems that Rice's theorem doesn't tell us anything about.

Optional warning: In particular, Rice's theorem will not help you on the final.

Let us formalize the concept "property of a TM program's transfer relation." A set S of NTTs (nondeterministic Turing transducers) is an *index set* if for all programs P and P'

$$\tau_P = \tau_{P'} \iff \chi_S(P) = \chi_S(P').$$

In other words, equivalent programs either both belong to S or neither belongs to S. Thus, membership in S depends only on the program's transfer relation, i.e., membership in S is a property of the program's transfer relation. Give examples. (You could translate the properties discussed above into index sets and non–index sets. You could also point out that K_{pos} and K_{inatt} are index sets but K_{xy} and K_{diag} are not.)

An index set is *trivial* if it is the empty set or if it contains all NTTs. A trivial index set corresponds to a property that is false for all NTTs or true for all NTTs.

Rice's theorem: An index set is recursive iff it is trivial.

Proof: Clearly trivial index sets are recursive.

Let S be a nontrivial index set. Let D be a program that runs forever on every input.

Case 1: $D \notin S$. We will reduce K. Let A be any element of S (remember S is nontrivial). Let $f(x)$ be the following program:

> run program x on input x, without producing any output;
> input z;
> run A on input z;

If $x \in K$, then $f(x)$ is equivalent to A, so $f(x) \in S$. If $x \notin K$, then $f(x)$ is equivalent to D, so $f(x) \notin S$. Thus $K \leq_m S$, so K is nonrecursive.

Case 2: $D \in S$. $\overline{S}$ is a nontrivial index set, and $D \notin \overline{S}$. By case 1, $\overline{S}$ is nonrecursive, so S is nonrecursive.

In either case, S is nonrecursive.

Note: you can prove the same kind of theorem for DTAs, NTAs, or DTTs, as well.

(2)

Let f be a total recursive function mapping DTTs to DTTs. We will try to find a program P such that $f(P) = P$. Ask students who have seen this before to hold their questions until the end of the construction. We guess that P is obtained by running a program with itself as input, i.e., that $P = \tau_V(V)$ for some V yet to be determined. Define $D(x) = \tau_x(x)$, so $P = D(V)$. We want $f(P) = P$, so we want a V such that

$$f(D(V)) = D(V) = \tau_V(V).$$

Thus it suffices to take $\tau_V = f \circ D$, i.e., let V be a program that computes the partial function $f \circ D$. Then $D(V)$ is the desired fixed point of f. But wait a second. The function $f(x) = x + 1$ has no fixed point, so we must have done something wrong. Well, f is total recursive and D is partial recursive, so $f \circ D$ is partial recursive, implying that there really exists a program V for $f \circ D$, but program D might not converge on input V, so P might not exist.

Let's try a gain with a more reasonable goal. We will try to find a program P such that $f(P) \equiv P$ (equivalent programs), i.e., such that $f(P)$ and P compute the same partial recursive function. Let's be careful to define a total recursive function d. Let $d(x)$ be the following program:

> let y be the result of running x on input x;
> input z;
> run y on input z

Then $d(x) \equiv \tau_x(x)$ if $\tau_x(x)$ is defined ($d(x)$ is a program that never halts otherwise). But the program $d(x)$ always exists.

Let v be a program that computes $f \circ d$. Since τ_v is total, $d(v) \equiv \tau_v(v)$. Let $P = d(v)$. Then

$$P = d(v) \equiv \tau_v(v) = (f \circ d)(v) = f(d(v)) = f(P),$$

so P and $f(P)$ are equivalent programs.

Fixed-point theorem: If f is a total recursive function then there exists a program P such that $f(P) \equiv P$.

Remark 1: the proof is constructive. Remark 2: the proof works for any machine with the same power as a TM, e.g., a RAM or a 2-counter machine.

Example: An inattentive DTM program P with output is called *self-actualized* if it writes itself, i.e., if the output of P is P. We will construct a self-actualized program. Let $f(x)$ be a DTM program that behaves as follows:

> output x;

By the fixed-point theorem, we can construct a program P such that $P \equiv f(P)$. Since $f(P)$ outputs P, P outputs P, so P is self-actualized.

(3) Suppose you want a program to refer to itself while it is running. A program on a modern computer could do this by opening the file that contains its source code. What what if its source was deleted? The recursion theorem allows us to write general-purpose TM programs that refer to (or know) themselves.

Recursion Theorem: Let $h(\cdot,\cdot)$ be a two-place partial recursive function. Then there exists a DTT P that computes the one-place partial function $h(P,\cdot)$, i.e., on input x the program P outputs $h(P,x)$.

Proof: Let $f(e)$ be the following program

```
input x;
output h(e, x);
```

That is, on input x, the program outputs $h(e,x)$. By the fixed-point theorem, we can construct P such that $P \equiv f(P) \equiv h(P,\cdot)$.

Remark 1: the proof is constructive. Remark 2: the proof works for any machine with the same power as a TM.

Example: A DTR P is called *self-aware* if it recognizes itself, i.e., if P recognizes the set $\{P\}$. We will construct a self-aware program. Let $h(P,x) =$ ACCEPT if $P = x$, REJECT otherwise, so $h(P,\cdot)$ recognizes $\{P\}$. By the recursion theorem we can construct a program P such that $P \equiv h(P,\cdot)$. This program P recognizes $\{P\}$.

Remark: In practice, we skip the step of explicitly constructing h; instead we write a self-referential P directly.

Example: A new proof that the halting problem is undecidable. Suppose that we had an algorithm to solve the halting problem for inattentive DTM programs. Then, using the recursion theorem, we could construct an inattentive DTM program P that does the following:

```
if P halts then (* self-reference justified by recursion theorem *)
    go into an infinite loop
else
    halt;
```

P halts iff P doesn't halt. This contradiction proves that there is no algorithm that solves the halting problem for inattentive DTM programs.

Lecture 2 (optional)

Reading: Section 8.3

Concepts: Gödel's incompleteness theorem. Rosser's version.

Lecture:

(1) Mathematicians used to think that every true statement about numbers could be proved and every false statement about numbers could be disproved. All that changed in 1931, when Kurt Gödel (pronounced like girdle), constructed a true statement about numbers that cannot be proved or disproved, unless the axioms of arithmetic are inconsistent.

Gödel's statement expresses the idea "there is no proof that this statement is true." Because the idea is circular, it is not obvious how to state it in terms of arithmetic. Gödel invented a version of the recursion theorem that allowed him to make such a statement. (The recursion theorem itself was discovered later by Kleene (pronounced "CLAY-nee").)

Some definitions:

- A *theory* is a set of axioms (for arithmetic). (Today, we consider only theories of arithmetic.)
- A *proof* is a sequence of statements, each of which is an axiom or else follows from the preceding statements by logical rules.
- A *theorem* is the last statement in a proof.
- A theory is *consistent* if there is no statement ψ such that ψ and $\neg\psi$ (not ψ) are both theorems.
- A theory is *sound* if all theorems are true in the standard model of arithmetic. (This is not the same definition as in Section 7.12. Logicians define soundness two different ways, depending on whether they are talking about arithmetic.)
- A theorem is *complete* if, for every ψ, ψ is a theorem or $\neg\psi$ is a theorem.

Pick your favorite set of axioms for arithmetic. It's recursive (maybe even finite). The set of logical rules is finite. Therefore, we can check proofs mechanically.

(2) Gödel's incompleteness theorem: Assume that the axioms of arithmetic are sound. Then there is an arithmetic statement ψ such that

- ψ is true,
- ψ is not a theorem, and
- $\neg\psi$ is not a theorem.

Proof: By the recursion theorem for 2-CM programs, define a 2-CM program that behaves as follows:

```
for i := 0 to ∞ do
    if i is a proof that P does not halt, then halt;
```

Let ψ be the statement "P does not halt." We have seen how to express the halting problem for 2-CM programs in first-order logic; we use the same method to express ψ in arithmetic (if you skipped Section 7.12, then you prove that computability theory is either incomplete or inconsistent).

If ψ is false, then P halts, but the only way P halts is if it finds a proof that P does not halt, i.e., a proof of ψ. This violates the soundness assumption.

Therefore ψ is true. By the soundness assumption, there is no proof of $\neg\psi$. There can be no proof of ψ, because that would be a proof that P does not halt, so P would halt, and ψ would be false, a contradiction. Those are the three things we set out to prove.

(3) **Rosser's version of the incompleteness theorem:** Assume that the axioms of arithmetic are consistent. Then there is an arithmetic statement ψ such that

- ψ is true,
- ψ is not a theorem, and
- $\neg\psi$ is not a theorem.

In particular, whatever axioms you choose for arithmetic, arithmetic is incomplete or inconsistent.

Proof: Using the 2-CM recursion theorem, construct a 2-CM program P that behaves as follows:

```
for i := 1 to ∞ do begin
    if i is a proof that P halts, then loop forever;
    if i is a proof that P does not halt, then halt;
end.
```

Let ψ be the statement "P does not halt." Assume for the sake of contradiction that there is a proof of ψ or a proof of $\neg\psi$. Let i be the minimum such proof. There are two cases:

Case 1: i is a proof of ψ, i.e., a proof that P does not halt. But then P halts the ith time through the for loop. If a program halts, we can prove that it halts, by checking its computation. Therefore there is a proof that P halts. Since there is also a proof that P does not halt, this violates consistency.

Case 2: i is a proof of $\neg\psi$, i.e., a proof that P halts. Then P goes into an infinite loop. We can prove that P goes into an infinite loop by checking its behavior the first i times through the for loop. Therefore there is a proof that P does not halt. This violates consistency.

In either case we have a contradiction. Therefore there is no proof of ψ and there is no proof of $\neg\psi$. Therefore the program goes through the for loop forever, so it never halts, i.e., ψ is true.

(4) The statement ψ constructed above is rather unnatural. Let *Con* be the statement that the axioms of arithmetic are consistent (later we will see how to phrase *Con* as a statement in arithmetic). This is a statement of real interest. We show that *Con* is true iff it cannot be proved.

First, let's prove that *Con* implies that *Con* is not a theorem. Assume *Con* is true. *Con* is the only assumption in Rosser's theorem. Because *Con* is true, all the conclusions are true, in particular the second, so ψ is not a theorem. For the sake of contradiction, assume that *Con* is a theorem. Because *Con* is a theorem, all the conclusions are theorems, in particular the first, so ψ is a theorem. This is a contradiction.

Conversely, let's prove that $\neg Con$ implies that *Con* is a theorem. This is easy. If arithmetic is inconsistent then every statement is a theorem.

How do you express *Con* as a statement in arithmetic? Write a program that searches for a statement s and proofs of s and $\neg s$. Convert the program to a deterministic 2-counter machine program. Express that program's not halting in arithmetic.

Epilogue: This was much easier for us than it was for Gödel. Computability theory hadn't been invented when he did this work, so he had to do everything directly in terms of numbers. We also had the advantage of knowing the result was true.

Lecture 3

Reading: Section 8.4

Concepts: Oracle Turing machines, relativization, and jumps

Lecture:

(1) An oracle is a very powerful device, named after the mythological oracle at Delphi. An oracle for a language B allows us to test whether an arbitrary string belongs to B. (Why don't we allow a universal oracle that can answer questions about any set? Because we can only phrase questions as strings, we have no way of expressing which set we want to ask about.)

Example: Let M be a Turing machine with oracle for K_{pos}. We write a program for M that determines whether a program halts on exactly 1 input. Let $K_{\geq 2}$ be the set of DTM programs that halt on at least 2 inputs. $K_{\geq 2}$ is r.e., because we can guess two strings and check that the program halts on them. Therefore

$K_{\geq 2} \leq_m K_{pos}$; call the reduction f.

input P;
if $P \notin K_{pos}$ then reject; (* P halts on 0 inputs *)
$y := f(P)$;
if $y \in K_{pos}$ then reject; (* P halts on 2 or more inputs *)
accept.

A language A is Turing-reducible to a language B, written $A \leq_T B$, if A is recognized by DTR P with oracle B. (P is called a Turing reduction from A to B.)

By the example, the set of DTM programs that halt on exactly 1 input is Turing-reducible to K_{pos}. Also, for any set A, $\overline{A} \leq_T A$.

(2) Most of the proofs we have given for TMs work equally well for TMs with oracle B. Most theorems about TMs have true analogues about TMs with oracle B.

Example: Consider the following statement about TMs:

(1) Every language accepted by an NTA is accepted by a DTA.

The analogous statement about TMs with oracle B is

(1^B) Every language accepted by an NTA with oracle B is accepted by a DTA with oracle B.

Transforming statements in this way is called relativization. Often we can prove a relativized statement, by relativizing the proof of the original statement.

Let's review the proof of statement (1): Let L be accepted by an NTA P. The following DTA also accepts L:

input x;
for $i := 1$ to ∞ do
    if i is an accepting computation of P on input x, then accept;

Note that a DTA can check a computation of an NTA step by step.

Corresponding proof of statement (1^B): Let L be accepted by an NTA P with oracle B. The following DTA with oracle B also accepts L:

input x;
for $i := 1$ to ∞ do

if i is an accepting computation of P on input x, then accept;

Note that a DTA with oracle B can check a computation of an NTA with oracle B step by step.

We say that A is recursive in B if A is recognized by a DTR with oracle B. A is r.e. in B if A is accepted by an NTA with oracle B. f is recursive in B if f is computed by a DTT with oracle B.

Examples of relativized theorems:

- L is recursive in B iff L and $\overline{L}$ are r.e. in B.
- If $S \leq_m T$ and T is recursive in B then S is recursive in B.
- If $S \leq_m T$ and T is r.e. in B then S is r.e. in B.

Warning: It is not always possible to relativize problems unless the problems are about Turing machines. For example, there is no obvious way to relativize context-free grammars. One should be careful to avoid incompletely relativized statements. There is no DTR that determines whether a CFG generates Σ^*; however there is a DTR with oracle K that determines whether a CFG generates Σ^*.

(3) Given any language we can always find a harder one. The halting problem for DTM programs with oracle B (denoted K^B or B') is harder than B. By easy relativizations we can show:

- A is r.e. in B iff $A \leq_m K^B$.
- K^B is not recursive in B.

K^B is called the *jump* of B because its difficulty is higher than B's. Using the jump operator we can define a sequence of problems, each of which is harder than the one before

$$K, K^K, K^{K^K}, K^{K^{K^K}}, \ldots.$$

Notation:

- $A^{(0)} = A$
- $A^{(1)} = A' = K^A$
- $A^{(2)} = A'' = K^{K^A}$
- $A^{(3)} = A''' = K^{K^{K^A}}$

- $A^{(i+1)} = \left(A^{(i)}\right)' = K^{A^{(i)}}$

Example: $\emptyset^{(3)} = K^{K^{K^{\emptyset}}}$.

Lectures 4 and 5

Reading: Section 8.5

Concepts: Arithmetical hierarchy

Lecture:

(1) Not all undecidable problems are equivalent in difficulty (for example, K and K'). We define a hierarchy of undecidable problems. Many natural problems can be classified by where they fit in this hierarchy.

Recall that the r.e. languages are obtained by applying an existential quantifier to the recursive languages. The co-r.e. languages can be obtained by applying a universal quantifier to the recursive language. We generalize these ideas by applying a finite sequence of quantifiers.

Definition: Let C be a class of languages.

- $L \in \Sigma C$ iff there exists $R \in C$ such that

$$x \in L \iff (\exists y)[(x, y) \in R].$$

- $L \in \Pi C$ iff there exists $R \in C$ such that

$$x \in L \iff (\forall y)[(x, y) \in R].$$

(Represent ordered pairs, tuples, and sequences as strings.)

For example, if C is the class of recursive languages, then ΣC is the class of r.e. languages, and ΠC is the class of co-r.e. languages.

Definition (Arithmetical Hierarchy):

- $\Sigma_0 = \Pi_0 =$ the class of recursive languages
- $\Pi_{i+1} = \Pi\Sigma_i$
- $\Sigma_{i+1} = \Sigma\Pi_i$
- $\Delta_i = \Sigma_i \cap \Pi_i$

Σ_1 is the class of r.e. languages. Π_1 is the class of co-r.e. languages.

Example: $L \in \Sigma_2$ iff there exists a recursive language R such that

$$x \in L \iff (\exists y)(\forall z)[((x,y),z) \in R].$$

By unrolling quantifiers we obtain the following normal forms:

- $L \in \Sigma_i$ iff there exists a recursive language R such that

  $$x \in L \iff (\exists y_1)(\forall y_2)\cdots(Qy_i)[(x,y_1,\ldots,y_i) \in R]$$

  where Q is $\forall$ or $\exists$ depending on whether i is even.

- $L \in \Pi_i$ iff there exists a recursive language R such that

  $$x \in L \iff (\forall y_1)(\exists y_2)\cdots(Qy_i)[(x,y_1,\ldots,y_i) \in R]$$

  where Q is $\exists$ or $\forall$ depending on whether i is even.

Since

$$\neg(\exists y_1)(\forall y_2)\cdots(Qy_i)[(x,y_1,\ldots,y_i) \in R]$$

is the same as

$$(\forall y_1)(\exists y_2)\cdots(Q'y_i)[(x,y_1,\ldots,y_i) \in \overline{R}],$$

co-$\Sigma_i = \Pi$, and therefore co-$\Pi_i = \Sigma_i$.

Recall that a predicate is a function whose result is Boolean. A predicate $P(\cdot)$ is a recursive predicate if P is a recursive function. Recursive predicates and recursive sets are virtually interchangeable.

- $L \in \Sigma_i$ iff there exists a recursive predicate R such that

  $$x \in L \iff (\exists y_1)(\forall y_2)\cdots(Qy_i)[R(x,y_1,\ldots,y_i)].$$

- $L \in \Pi_i$ iff there exists a recursive predicate R such that

  $$x \in L \iff (\forall y_1)(\exists y_2)\cdots(Qy_i)[R(x,y_1,\ldots,y_i)].$$

Corollary:

- If $A \leq_m B$ and $B \in \Sigma_i$ then $A \in \Sigma_i$
- If $A \leq_m B$ and $B \in \Pi_i$ then $A \in \Pi_i$

We prove only the second part. The first part is similar.

$$x \in B \iff (\forall y_1)(\exists y_2)\cdots(Qy_i)[R(x, y_1, \ldots, y_i)].$$

Let $x \in A$ iff $f(x) \in B$. Then

$$x \in A \iff (\forall y_1)(\exists y_2)\cdots(Qy_i)[R(f(x), y_1, \ldots, y_i)].$$

Since $R(f(x), y_1, \ldots, y_i)$ is a recursive predicate, $A \in \Pi_i$.

The following containments follow pretty easily from the normal form:

- $\Sigma_i \subseteq \Sigma_{i+1}$
- $\Sigma_i \subseteq \Pi_{i+1}$
- $\Pi_i \subseteq \Pi_{i+1}$
- $\Pi_i \subseteq \Sigma_{i+1}$
- $\Sigma_i \subseteq \Sigma\Sigma_i$
- $\Pi_i \subseteq \Pi\Pi_i$

If like quantifiers are adjacent, we can combine them.

Lemma (Quantifier Contraction): $\Sigma\Sigma_i = \Sigma_i$ and $\Pi\Pi_i = \Pi_i$.

Proof: We prove only the second equality. The containment in one direction was discussed above. Now we show $\Pi\Pi_i \subseteq \Pi_i$. Let $A \in \Pi\Pi_i$.

$$\begin{aligned} x \in A &\iff (\forall y)[(x, y) \in B] \qquad \text{where } B \in \Pi_i \\ &\iff (\forall y)(\forall y_1)(\exists y_2)\cdots(Qy_i)[R((x, y), y_1, y_2, ..., y_i)] \qquad \text{where } R \text{ is recursive} \\ &\iff (\forall (y, y_1))(\exists y_2)\cdots(Qy_i)[R((x, y), y_1, y_2, ..., y_i)]. \end{aligned}$$

Therefore $A \in \Pi_i$.

Corollary: Σ_i and Π_i are closed under union and intersection.

Have the students read the proof on their own.

Quantifiers whose argument ranges over a finite set can be eliminated.

Lemma (Bounded Quantification):

- Let $A \in \Sigma_i$ and $B = \{x : (\forall k \leq f(x))[(x, k) \in A]\}$, where f is a total recursive function. Then $B \in \Sigma_i$.

- Let $A \in \Pi_i$ and $B = \{x : (\exists k \leq f(x))[(x,k) \in A]\}$, where f is a total recursive function. Then $B \in \Pi_i$.

Proof: We prove only the second part. If $i = 0$ then B is recursive, so A is recursive. Now let $i \geq 1$.

$$\begin{aligned}
x \in B &\iff (\exists k \leq f(x))[(x,k) \in A] \\
&\iff (\exists k \leq f(x))(\forall y)[((x,k),y) \in R] \qquad \text{where } R \in \Sigma_{i-1} \\
&\iff (\forall y)[((x,0),y) \in R] \vee \cdots \vee (\forall y)[((x,f(x)),y) \in R] \\
&\iff (\forall y_0)[((x,0),y_0) \in R] \vee \cdots \vee (\forall y_{f(x)})[((x,f(x)),y_{f(x)}) \in R] \\
&\iff (\forall \langle\langle y_0,\ldots,y_{f(x)}\rangle\rangle)[((x,0),y_0) \in R \vee \cdots \vee ((x,f(x)),y_{f(x)}) \in R] \\
&\iff (\forall \langle\langle y_0,\ldots,y_{f(x)}\rangle\rangle)(\exists k \leq f(x))[((x,k),y_k) \in R]
\end{aligned}$$

If $i - 1 = 0$ then R is recursive, so the predicate $(\exists k \leq f(x))[((x,k),y_k) \in R]$ is recursive, so $B \in \Pi_1$. If $i-1 \geq 1$, then apply quantifier contraction to $(\exists k \leq f(x))$ and the first quantifier of R, to obtain $B \in \Pi\Sigma_{i-1} = \Pi_i$.

(2) Next we will look at the hardest languages in each portion of the arithmetic hierarchy. L is *hard* for a class if every language in the class is m-reducible to L. L is *complete* for a class if L is in the class and hard for the class. Recall the complete languages defined in Chapter 7; by definition, a language is complete iff it is complete for the class of r.e. languages.

$\emptyset'$ (K) is complete for Σ_1 (r.e.). By iterating the jump operator we obtain complete languages for each Σ_i.

Theorem: For all $i \geq 1$, $\emptyset^{(i)}$ is complete for Σ_i.

Have the students read the proof on their own.

The next result is analogous to the fact that a set is r.e. and co-r.e. iff it is recursive.

Theorem: $A \in \Delta_{i+1} \iff (\exists B \in \Sigma_i)[A \leq_T B] \iff (\exists B \in \Pi_i)[A \leq_T B]$.

Proof: The last two conditions are equivalent because $A \leq_T B$ iff $A \leq_T \overline{B}$. We will prove the first two equivalent.

Let $A \in \Delta_{i+1} = \Sigma_{i+1} \cap \Pi_{i+1} = \Sigma_{i+1} \cap \text{co-}\Sigma_{i+1}$. By the preceding theorem, A and $\overline{A}$ are m-reducible to $\emptyset^{(i+1)} = K^{\emptyset^{(i)}}$. By relativizing the fact that K is complete, A and $\overline{A}$ are both r.e. in $\emptyset^{(i)}$. By relativizing the fact that a set is recursive if itself and its complement are r.e., A is recursive in $\emptyset^{(i)}$, which belongs to Σ_i.

Conversely, assume that $A \leq_T B$, where $B \in \Sigma_i$. Because $\emptyset^{(i)}$ is complete for Σ_i, $A \leq_T \emptyset^{(i)}$, so A and $\overline{A}$ are r.e. in $\emptyset^{(i)}$. Therefore A and $\overline{A}$ are m-reducible to $K^{\emptyset^{(i)}} = \emptyset^{(i+1)}$, which is in Σ_{i+1}, so A and $\overline{A}$ belong to Σ_{i+1}. Therefore $A \in \Sigma_{i+1} \cap \Pi_{i+1} = \Delta_{i+1}$

Except for Σ_0, Π_0, and Δ_0 all classes in the arithmetical hierarchy are distinct. We will use $\emptyset^{(i+1)}$ to separate Σ_i from Δ_i.

Corollary: For all $i \geq 1$, $\Sigma_i \neq \Delta_i$.

Proof by contradiction. Assume $\Sigma_i = \Delta_i$. Then every language in Σ_i is recursive in a language in Σ_{i-1}. Therefore $\emptyset^{(i)}$ is recursive in $\emptyset^{(i-1)}$ (because the former is in Σ_i and the latter is complete for Σ_{i-1}), a contradiction.

Corollary: For all $i \geq 1$, $\Pi_i \neq \Delta_i$.

Proof: Since Δ_i is closed under complementation, $\Pi_i = \Delta_i \Rightarrow \Sigma_i = \Delta_i$.

Corollary: For all $i \geq 1$, $\Sigma_i \neq \Pi_i$.

Proof: If $\Sigma_i = \Pi_i$, then $\Sigma_i = \Sigma_i \cap \Pi_i = \Delta_i$.

Corollary: For all $i \geq 0$, $\Sigma_i \neq \Sigma_{i+1}$.

Proof: If $\Sigma_{i+1} = \Sigma_i$, then $\Pi_i \subseteq \Sigma_i$. Since $\Pi_i = \text{co-}\Sigma_i$, this implies $\Pi_i = \Sigma_i$.

Corollary: $\Sigma_i \subset \Delta_{i+1}$.

Proof: Δ_{i+1} contains Π_i.

Corollary: $\Pi_i \subset \Delta_{i+1}$.

Proof: Δ_{i+1} contains Σ_i.

(Actually $\Sigma_i \cup \Pi_i \subset \Delta_{i+1}$. $\emptyset^{(i)} \oplus \overline{\emptyset^{(i)}}$ separates them.)

Draw the diagram from page 594 which shows the proper containments.

Let TOT be the set of DTM programs that halt on all inputs.

(3) Theorem: TOT is complete for Π_2.

Proof: First we prove membership.

$$x \in \text{TOT} \iff (\forall y)[(x,y) \in K_{xy}].$$

Because K_{xy} is r.e., TOT $\in \Pi\Sigma_1 = \Pi_2$.

Let A be any language in Π_2. We m-reduce A to TOT. There exists a recursive predicate R such that

$$x \in A \iff (\forall y)(\exists z)[R(x,y,z)].$$

Let $f(x)$ be a DTM program that behaves as follows:

```
input y;
    for z := 1 to ∞ do
```

if $R(x, y, z)$ then halt;

Then $x \in A \iff f(x)$ halts on all inputs, so $A \leq_m$ TOT.

Topic 9

NP-completeness

Reading: Sections 0.5, 9.1–9.8

Homework: 9.2-2, 9.2-3, 9.4-2, 9.5-4, 9.6-5. Extra Credit: 9.1-9.

Lecture 1

Reading: Sections 9.1, 9.2, 9.4

Concepts: DTIME() and NTIME(). P and NP. Polynomial-time m-reductions. NP-completeness. Canonical NP-complete problem.

Lecture:

(1) In practice the running time of a program is important. Give some examples that you care about, or ask the students to suggest some.

The running time of a program on input x is the length of its longest partial computation. If the program is a recognizer, then the running time on input x is the length of the unique computation. If the program has an infinite computation on input x, then the running time is ∞.

We say that a program runs in time $t(n)$ if it runs for at most $t(|x|)$ steps on every input x. Stress the importance of considering running time as a function of input length, because we expect programs to take more time on longer inputs.

We say that a program runs in time $O(t(n))$ if there is a function $s(n)$ such that $s(n) = O(t(n))$ and the program runs in time $s(n)$. (Have the students review $O(\cdot)$ notation in Section 0.5 if necessary.)

- DTIME($t(n)$) is the class of languages that are recognized by DTRs that run in time $t(n)$.

- NTIME($t(n)$) is the class of languages that are accepted by NTAs that run in time $t(n)$.

A program runs in polynomial time if there exists k such that it runs in time $O(n^k)$.

- P is the class of languages recognized by DTRs that run in polynomial time.
- NP is the class of languages accepted by NTAs that run in polynomial time.

Give some examples of problems in P and NP.

Observe that every problem in NP can be solved in exponential time, by trying all possible nondeterministic choices, so problems in NP are decidable, although perhaps by an impractically slow algorithm.

(2) Give some examples of problems in NP that happen to be NP-complete. Point out that no one knows how to solve any of these problems efficiently, and in fact they are all equally difficult in a formal sense.

Define polynomial-time m-reductions, $\leq_m^p$-hardness, $\leq_m^p$-completeness, NP-hardness, and NP-completeness. Point out analogies to the recursion-theoretic concepts.

Prove that $\leq_m^p$ is transitive. Note that you have to compose the time bounds, so this is not 100% obvious.

Remember that you can't reduce a hard problem to an easy problem. If $A \leq_m^p B$ and A is NP-hard then B is NP-hard. Typically we will reduce an NP-complete problem in order to prove that another problem is hard. (But it is usually a waste of time to reduce *to* an NP-complete problem.)

Soon we will be able to prove that natural problems are NP-complete, but first we just need to prove that an NP-complete problem exists.

(3) It is not hard to define a canonical NP-complete problem, which is essentially a time-bounded version of K_{xy}. Let $K_{1,\mathrm{NP}} = \{P\#x\#0^s : P \text{ is an NTA, } x \in \{0,1\}^* \text{ and } P \text{ has an accepting computation } s \text{ steps or shorter on input } x\}$.

$K_{1,\mathrm{NP}}$ is in NP because we can guess an accepting computation of P having length s or less and check the computation step by step in time that is polynomial in s, the length of x, and the size of the program P. (Actually we are using a universal TM program here, which does in fact run in polynomial time.)

$K_{1,\mathrm{NP}}$ is NP-hard, because if A is in NP then A is accepted by some NTA P that runs in time bounded by a polynomial $p(n)$, so

$$x \in A \iff P\#x\#0^{p(|x|)} \in K_{1,\mathrm{NP}}.$$

Oops. We forgot one little thing. What if the input alphabet of A is not $\{0,1\}$? Then encode the strings in A over $\{0,1\}$ and let B be the language consisting of the so encoded strings. Then $A \leq_m^p B$ by an easy reduction (f just performs the encoding) and $B \leq_m^p K_{1,\mathrm{NP}}$ as shown above, so $A \leq_m^p K_{1,\mathrm{NP}}$ by transitivity.

Epilogue: We don't usually prove NP-completeness results from scratch. Now we have one NP-complete problem. Next time we will use it in proving that another is NP-complete. Other NP-completeness results will build on that one.

Note: Szelepscenyi is pronounced Seleps-cheny

Lecture 2

Reading: Section 9.5

Concepts: SSS is NP-complete.

Lecture:

Note: SSS is a generalization of k-SAT which doesn't involve Booleans. Students who don't understand formal logic can still appreciate SSS. Proving SSS NP-complete is much cleaner than proving SAT NP-complete because the correctness of a tableau can be expressed very simply in terms of constraints on control states and tape characters. In Section 9.6 we will convert constraints to groups of clauses. It is easier to describe the general construction than to actually grind it out for the constraints corresponding to a Turing machine program. It also lets us separate the key ideas from the technical details.

In this lecture we present an important NP-complete problem. We will use it later to prove that other problems are NP-complete.

(1) Present symbol systems initially via an example. Describe constraints, variables, and locality.

In a (k,Γ)-symbol system, every constraint involves at most k variables (locality $= k$), and the variables take values in the finite set Γ.

An assignment is formally mapping from the variable set V to the symbol-system alphabet Γ. An assignment A satisfies a constraint if the constraint evaluates to true when you substitute $A(x)$ for each variable x. An assignment satisfies the symbol system if it satisfies every constraint.

Informally, we represent constraints in whatever terms are most easily understood. On a computer we must represent constraints formally. How to do so? Include a list of the variables in the constraint. Then we have to indicate which assignments satisfy the constraint. Don't list all *assignments* because there could be $2^{|V|}$, which is huge. Instead list satisfying *local assignments* to the variables that actually

appear in the constraint; there are at most 2^k of them. For example, the constraint $z > w$ over alphabet $\{0, 1, 2\}$ is represented as $\langle\langle wz, 01, 02, 12\rangle\rangle$. We represent a symbol system as a list of variables and a list of constraints.

A symbol system is satisfiable if there is an assignment that satisfies it. (k, Γ)-SSS is the set of satisfiable (k, Γ)-symbol systems.

(k, Γ)-SSS is in NP because we can guess and check a satisfying assignment, if there is one, efficiently.

Lemma: If L is in NP then there exist k and Γ such that $L \leq_m^p (k, \Gamma)$-SSS.

Important: This does not say that (k, Γ)-SSS is NP-complete. The reduction gives us a different Γ for every L. This is why we constructed canonical NP-complete problems in the previous lecture. By reducing an L that is NP-complete, we obtain k and Γ such that (k, Γ)-SSS is NP-complete.

Theorem: There exist k and Γ such that (k, Γ)-SSS is NP-complete.

Proof of lemma: Let L be accepted by an NTA with a unique accepting state. Have the input start on the tape. Let the NTA run in time $p(n)$. Modify the program by putting an infinite loop on the accepting state; then if the program accepts in t steps it also accepts in u steps for all $u \geq t$. (Technically the last modification makes the program's running time infinite, which is why we defined $p(n)$ before making this change.)

Define variables that will represent the control state and the first $p(n) + 1$ tape squares at times 0 through $p(n)$. Recall that unreached tape squares are blank. Pad the tape with an additional blank character on the left of the home square.

Initial constraints: Ensure that q_0 is the start state, the input starts on the tape, and the rest of the tape is blank.

Boundary constraints: the leftmost and rightmost tape squares are always blank.

Each constraint so far involves exactly one variable.

Away-from-the-head constraints: Consider any three adjacent squares. If the tape head is not on any of them at time t, then the middle square contains the same character at time $t + 1$. Each such constraint involves exactly six variables.

Near-the-head constraints: Consider any three adjacent squares. If the tape head is on the middle one at time t, then the contents of those squares and the control state at time $t+1$ are obtained by applying some instruction. Each such constraint involves exactly eight variables.

Final constraint: $q_{p(n)}$ must be an accepting state.

Lecture 3

Reading: Section 9.6

Concepts: SAT, 3-SAT, and Clique are NP-complete.

Lecture:

(1) Define Boolean variables, literals, clauses, and CNF formulas. CNF formulas are a special case of symbol systems (the alphabet is {true, false}, there is no locality, and the constraints have a highly restricted form).

SAT is the set of satisfiable CNF formulas. SAT is in NP, because we can guess and check a satisfying assignment, if there is one, in polynomial time.

Theorem: SAT is NP-complete. We reduce SSS. We define variables that represent the condition that x is assigned the value c.

For each variable x, a single clause ensures that x is assigned at least one value.

We could also include clauses that ensure that x is not assigned two values (one clause for each variable and each pair of values), but there is no need to do so.

For each local unsatisfying assignment I (there are at most 2^k per constraint) for each constraint, we include a clause that ensures that at least one variable x is not assigned the value $I(x)$. (See why it was important that we assigned at least one value to x? Otherwise these clauses could be satisfied trivially. But if x is assigned more than one value, it can only make these clauses harder to satisfy.)

(2) A k-CNF formula is a CNF formula in which each clause contains exactly k literals. k-SAT is the set of satisfiable k-CNF formulas. k-SAT is in NP because it is a special case of SAT.

Theorem: 3-SAT is NP-complete.

Note: 2-SAT is in P. See the exercises.

Proof of theorem: First we show how to convert a clause into a collection of clauses each involving 3 variables or fewer. Consider a clause

$$\ell_1 \vee \cdots \vee \ell_k$$

where $k \geq 3$. Create a new variable y and replace the clause by

$$(\ell_1 \vee \ell_2 \vee y) \wedge (\overline{y} \vee \ell_3 \vee \cdots \vee \ell_k)$$

This pair of clauses is satisfiable iff the original one is (consider two cases depending on whether y is assigned true or false). Their sizes are 3 and $k-1$ respectively.

Repeat this process (introducing a different new variable each time), until all clauses contain three variables or fewer. Thus we obtain an equivalent formula

in which each clause has size 3 or less. The original formula is satisfiable iff the new one is. We are almost done.

Introduce three new variables and clauses (involving all three) that guarantee that the first two are assigned false. Add false variables to any clause that contains fewer than three literals, so as to make its size exactly three. The original formula is in SAT iff the new formula is in 3-SAT.

(3) Define a clique. If vertices represent people, and edges connect people who think highly of each other, then a clique is a mutual admiration society.

Define the clique problem.

Prove that clique is NP-complete.

Lecture 4 (optional)

Reading: Section 9.7

Concepts: Hamiltonian path is NP-complete.

Note: This may be covered in your algorithms class instead.

Lecture 5 (optional)

Reading: Section 9.8

Concepts: Set cover and subset sum are NP-complete.

Note: This may be covered in your algorithms class instead.

Topic 10

Logic (optional)

Reading: Sections 7.12, 8.2, and 8.3

Lecture 1

Concepts: The set of theorems in first-order logic is nonrecursive.

See topic 7, lecture 7

Lecture 2

Concepts: The fixed-point and recursion theorems.

See topic 8, lecture 1

Lecture 3

Concepts: Gödel's incompleteness theorem. Rosser's version.

See topic 8, lecture 2

11

Solutions to Selected Exercises

We include solutions to selected exercises. Some solutions are detailed, but others are more like very good hints.

0.2-9 What is the relation is-mother ∘ is-parent ∘ is-parent?

Solution: $\{(x, y) : x$ is a great-grandmother of $y\}$

0.2-14 What is the transitive closure of is-child? the reflexive transitive closure of is-child?

Solution: is-descendant. $I \cup$ is-descendant.

0.2-20 Prove that ρ is a one-one correspondence if and only if ρ^{-1} is a one-one correspondence.

Solution:

A relation ρ is a function iff

for every x there exists exactly one y such that $x\ \rho\ y$ iff

for every x there exists exactly one y such that $y\ \rho^{-1}\ x$ iff

for every y there exists exactly one x such that $x\ \rho^{-1}\ y$ iff

ρ^{-1} is one-one and onto.

Therefore ρ is a one-one onto function iff ρ^{-1} is a one-one onto function. But a one-one correspondence is exactly the same as a one-one onto function.

0.3-6 What is $\{\mathtt{aaa}, \mathtt{aaaaaa}\}^*$? What is $\{\mathtt{aaa}, \mathtt{aaaa}\}^*$?

Solution: $\{\mathtt{aaa}\}^*$. $\{\mathtt{a}^i : i = 3,\ i = 4,$ or $i \geq 6\}$.

0.3-11 When adding two binary numbers in the standard way, you carry either 0 or 1 into the next position. Describe a similar algorithm for adding two dyadic numbers. What values might be carried?

Solution: Let the two numbers be represented by the strings $x_i \cdots x_0$ and $y_j \cdots y_0$ (where i and j are at least -1).

```
carry := 0;
for k := 0 to max(i, j) do begin
    if k > i then x_k := 0;
    if k > j then y_k := 0;
    sum := carry + x_k + y_k;
    carry := 0;
    while sum > 2 do begin
        carry := carry + 1
        sum := sum − 2
    end;
    z_k := sum;
end;
if carry > 0 then
    output carry z_k ··· z_0
else
    output z_k ··· z_0;
```

By inspection of the algorithm, the carry is always at least 0. If the carry is at most 2 at the top of the for loop, then it is at most 2 at the bottom of the for loop (the largest possible new carry is obtained when x_k and y_k are both 2). Thus, by a simple induction, the carry is always at most 2. All three values 0, 1, and 2 are possible as seen from the example $211 + 221$.

0.6-12 Let f be a function on natural numbers satisfying

$$f(n) = \begin{cases} 0 & \text{if } n = 0, \\ 4f(n/2) & \text{if } n \text{ is even}, \\ f(n-1) + 2n - 1 & \text{if } n \text{ is odd}. \end{cases}$$

Prove that $f(n) = n^2$ for all $n \geq 0$.

Solution: The proof is by strong induction.

Inductive hypothesis: $(\forall k < n)[f(k) = k^2]$. We take three cases:

Case 1: $n = 0$. Then $f(n) = 0 = n^2$, as desired.

Case 2: $n > 0$ and n is even. Then $f(n) = 4f(n/2) = 4(n/2)^2$ (by the inductive hypothesis) $= n^2$, as desired.

Case 3: n is odd. Then $f(n) = f(n-1) + 2n - 1 = (n-1)^2 + 2n - 1$ (by the inductive hypothesis) $= n^2$, as desired.

0.6-19 Suppose that we are given 100 numbers $x_1, \ldots, x_{100}$ between 1 and 100,000,000,000. Prove that for two of those numbers x_i and x_j, the sum of the base-10 digits of x_i is equal to the sum of the base-10 digits of x_j.

Solution: Let S be the set consisting of the given 100 numbers. For each $x \in S$, let $f(x)$ be the sum of the base-10 digits of x. Then $1 \leq f(x) \leq f(99999999999) = 99$. Since f maps from a set of size 100 to a set of size 99, f must not be one-one; i.e., there exist distinct x_i and x_j such that $f(x_i) = f(x_j)$. That is, the sum of the base-10 digits of x_i is equal to the sum of the base-10 digits of x_j.

0.6-25 For the purposes of this exercise, a tree is called *prolific* if every internal node has at least two children.

(a) Give a recursive definition of prolific trees.

(b) Prove that the number of leaves in a prolific tree is greater than the number of internal nodes.

Solution:

(a) **Rule 1:** If N is a node, then there is a prolific tree whose only node is N. N is the prolific tree's root. N has no parent and no children.

Rule 2: Let N be a node. Let $k \geq 2$. Let $T_1, \ldots, T_k$ be prolific trees containing disjoint sets of nodes. Let $N_1, \ldots, N_k$ be the roots of $T_1, \ldots, T_k$, respectively. Then there is a prolific tree whose nodes are N and all the nodes of $T_1, \ldots, T_k$. We call this prolific tree T. T's root is N. N has no parent, and the children of N are $N_1, \ldots, N_k$, in that order. The parent of N_i is N for $i = 1, \ldots, k$. Other than that, the parents and children of nodes in T are the same as in $T_1, \ldots, T_k$, which are called the *subtrees* of T.

(b) The proof is by structural induction.

Case 1: T consists of a single node. Then it contains 1 leaf and 0 internal nodes, so it has more leaves than internal nodes.

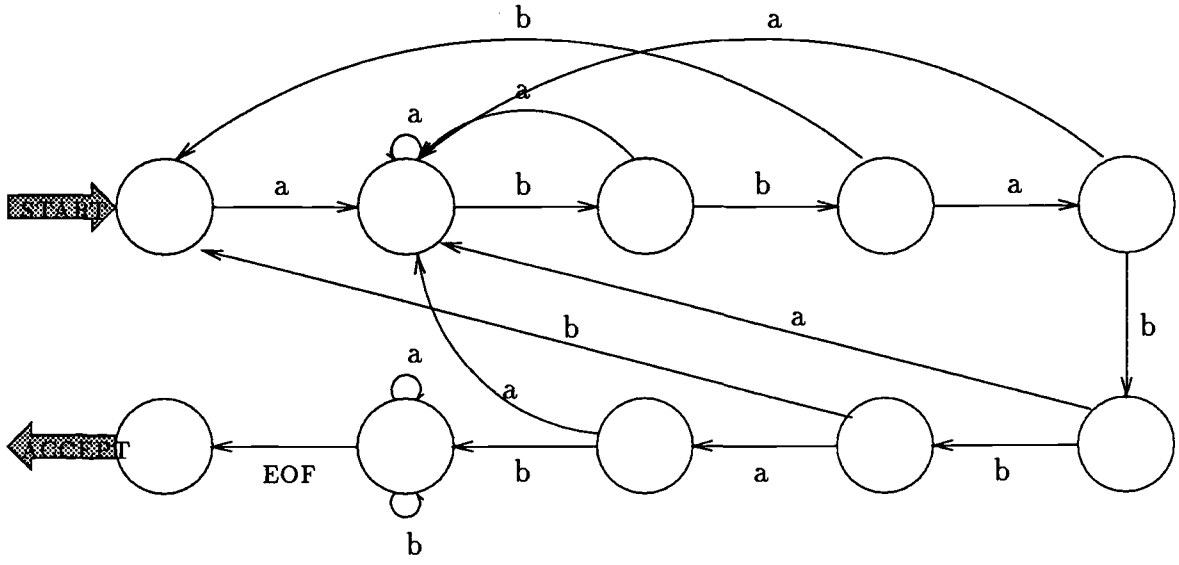
Case 2: T has prolific subtrees $T_1, \ldots, T_k$, where $k \geq 2$. Let each T_i contain l_i leaves and n_i internal nodes. By structural induction, we may assume that $l_i \geq n_i + 1$ for each i. The number of leaves in T is

$$\sum_{1 \leq i \leq k} l_i \geq \sum_{1 \leq i \leq k} (n_i + 1) = \left(\sum_{1 \leq i \leq k} n_i \right) + k > \left(\sum_{1 \leq i \leq k} n_i \right) + 1,$$

which is the number of internal nodes in T. Thus T has more leaves than internal nodes.

1.1-3 Design a program for a machine with only an input device that accepts exactly those strings containing the substring abbabbab. Use only 10 control states.

Solution:



2.2-3 A deque (pronounced "dek") is a double-ended queue that holds a string. The operations on a deque are

left-insert(c): Insert a c at the left end of the deque.

right-insert(c): Insert a c at the right end of the deque.

left-delete(c): Delete a c from the left end of the deque.

right-delete(c): Delete a c from the right end of the deque.

EMPTY: Test whether the deque is empty.

Fix a deque alphabet Γ.

(a) Define the realm of a deque formally.

(b) Express the deque operations informally using the $\rightarrow$ notation.

(c) Define the deque operations formally.

Solution:

(a) Γ^*

(b) left-insert(c): $x \rightarrow cx$
right-insert(c): $x \rightarrow xc$
left-delete(c): $cx \rightarrow x$
right-delete(c): $xc \rightarrow x$
EMPTY: $\Lambda \rightarrow \Lambda$

(c) left-insert(c): $\{(x, cx) : x \in \Gamma^*\}$
right-insert(c): $\{(x, xc) : x \in \Gamma^*\}$
left-delete(c): $\{(cx, x) : x \in \Gamma^*\}$
right-delete(c): $\{(xc, x) : x \in \Gamma^*\}$
EMPTY: $\{(\Lambda, \Lambda)\}$

2.4-2 Suppose we allow the repertory of a control to include all partial functions from Q to Q. Show how to transform a program using this expanded repertory into a program that uses only the standard control operations like $q_i \rightarrow q_j$.

Solution: Each partial function F on Q is a set of ordered pairs of the form (q, r). Replace each instruction of the form $(F, g_1, ..., g_k)$ by the set of all instructions of the form $(q \rightarrow r, g_1, ..., g_k)$ such that $(q, r) \in F$.

2.6-3 Consider the program represented in Figure 1.2.

(a) What are the machine's devices?

(b) What is the program's initializer?

(c) What is the program's instruction set?

(d) What is the program's terminator?

Solution:

(a) control, input, stack

(b) $x\alpha = (1, x, \Lambda)$

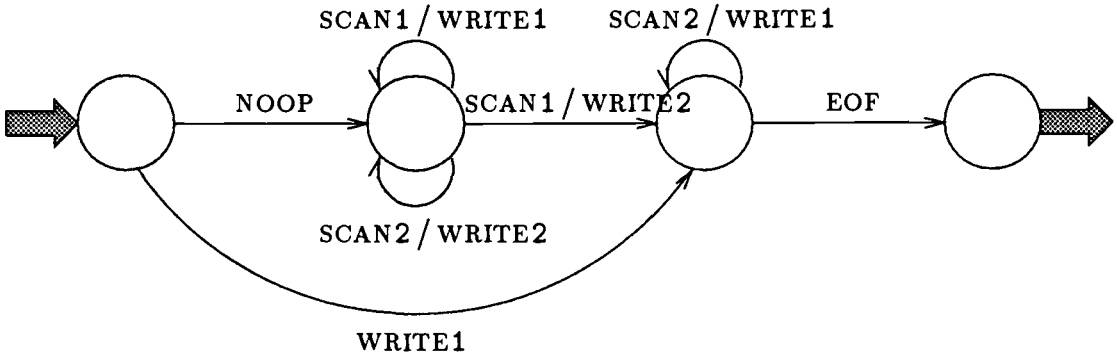
(c)

$$\left\{\begin{array}{llll} (1 \rightarrow 1\ , & \texttt{SCANa}\ , & \texttt{PUSHa}\ ), \\ (1 \rightarrow 1\ , & \texttt{SCANb}\ , & \texttt{PUSHb}\ ), \\ (1 \rightarrow 2\ , & \texttt{SCAN\#}\ , & \texttt{NOOP}\ ), \\ (2 \rightarrow 2\ , & \texttt{SCANa}\ , & \texttt{POPa}\ ), \\ (2 \rightarrow 2\ , & \texttt{SCANb}\ , & \texttt{POPb}\ ), \\ (2 \rightarrow 3\ , & \texttt{EOF}\ , & \texttt{EMPTY}), \end{array}\right\}$$

(d) $(3, \Lambda, y)\omega =$ ACCEPT for all stack strings y (ω is undefined elsewhere)

2.6-8(b) Design an NFM program that reads n as a dyadic numeral (with the digits in the normal order) and writes $n+1$ as a dyadic numeral (with the digits in the normal order).

Solution:



2.7-1 (a) What are the initializer, terminator, and instruction set of the program shown in Figure 1.10.

(b) What are that program's history, computation, and trace on input aabbbb?

Solution:

(a)
- $x\alpha = (1, x, 0)$
- $(4, \Lambda, n)\omega = \text{ACCEPT}$
- $\mathcal{I} = \left\{ \begin{array}{llll} (1 \to 2\ , & \text{SCANa}\ , & \text{INC}\ ), \\ (2 \to 1\ , & \text{NOOP}\ , & \text{INC}\ ), \\ (1 \to 3\ , & \text{SCANb}\ , & \text{DEC}\ ), \\ (3 \to 3\ , & \text{SCANb}\ , & \text{DEC}\ ), \\ (3 \to 4\ , & \text{EOF}\ , & \text{ZERO}) \end{array} \right\}$

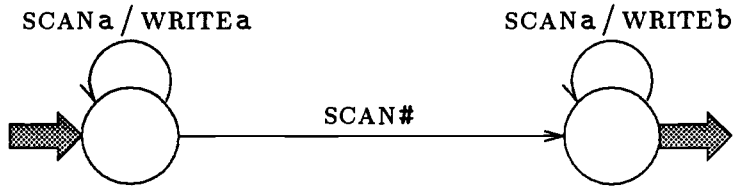
(b)
- The history is

$$\begin{array}{lll} \langle\!\langle\ \texttt{aabbbb} & , & \alpha \\ \quad (1, \texttt{aabbbb}, 0) & , & (1 \to 2, \text{SCANa}, \text{INC}) \\ \quad (2, \texttt{abbbb}, 1) & , & (2 \to 1, \text{NOOP}, \text{INC}) \\ \quad (1, \texttt{abbbb}, 2) & , & (1 \to 2, \text{SCANa}, \text{INC}) \\ \quad (2, \texttt{bbbb}, 3) & , & (2 \to 1, \text{NOOP}, \text{INC}) \\ \quad (1, \texttt{bbbb}, 4) & , & (1 \to 3, \text{SCANb}, \text{DEC}) \\ \quad (3, \texttt{bbb}, 3) & , & (3 \to 3, \text{SCANb}, \text{DEC}) \\ \quad (3, \texttt{bb}, 2) & , & (3 \to 3, \text{SCANb}, \text{DEC}) \\ \quad (3, \texttt{b}, 1) & , & (3 \to 3, \text{SCANb}, \text{DEC}) \\ \quad (3, \Lambda, 0) & , & (3 \to 4, \text{EOF}, \text{ZERO}) \\ \quad (4, \Lambda, 0) & , & \omega \\ \quad \text{ACCEPT} & \rangle\!\rangle. & \end{array}$$

- The computation is $\langle\langle(1 \rightarrow 2, \text{SCANa}, \text{INC}), (2 \rightarrow 1, \text{NOOP}, \text{INC}), (1 \rightarrow 2, \text{SCANa}, \text{INC}), (2 \rightarrow 1, \text{NOOP}, \text{INC}), (1 \rightarrow 3, \text{SCANb}, \text{DEC}), (3 \rightarrow 3, \text{SCANb}, \text{DEC}), (3 \rightarrow 3, \text{SCANb}, \text{DEC}), (3 \rightarrow 4, \text{EOF}, \text{ZERO})\rangle\rangle$.
- The trace is $\langle\langle(1, \text{aabbbb}, 0), (2, \text{abbbb}, 1), (1, \text{abbbb}, 2), (2, \text{bbbb}, 3), (1, \text{bbbb}, 4), (3, \text{bbb}, 3), (3, \text{bb}, 2), (3, \text{b}, 1), (3, \Lambda, 0), (4, \Lambda, 0)\rangle\rangle$.

2.9-4 Design a finite transducer that maps each string of the form $a^i\#a^j$ to a^ib^j.

Solution:



2.9-7 Design a deterministic finite transducer with the following properties:

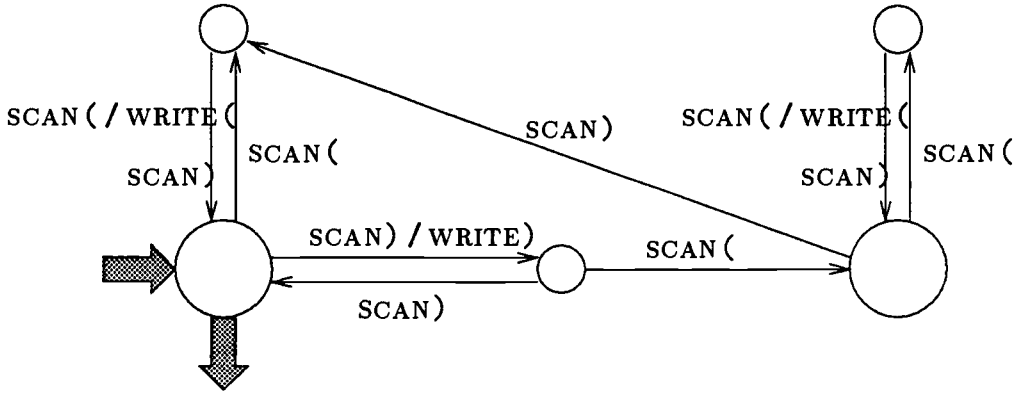
- If the input x is a balanced string of parentheses, then the program outputs a balanced string of parentheses having length $\frac{1}{2}|x|$ or less.
- If the input x is not a balanced string of parentheses, then the program either rejects or outputs an unbalanced string of parentheses having length $\frac{1}{2}|x|$ or less.

Solution: Have the program process the input characters in pairs, replacing ((by (,)) by), and () by Λ.

We would like to replace)(by Λ, but that will create an error when the prefix of the input that precedes it is balanced. Instead, we replace)(by), which causes us the output to have an excess of)s. To compensate for the excess, we treat the next pair that starts with) as though it started with (.

If the input length is odd then we reject.

The program is shown below:

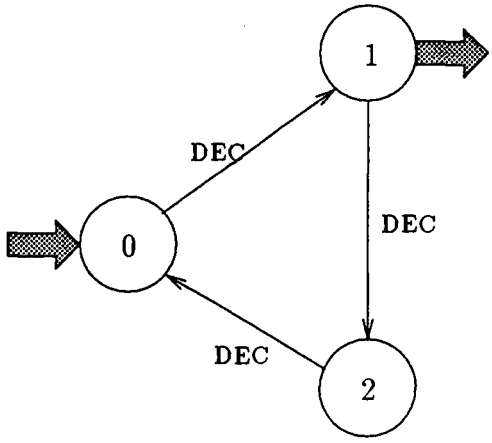


2.10-3(a) Construct a DUCM program that starts with a number x in the counter and determines whether $x \equiv 1 \pmod 3$.

Solution:

- $n\alpha = (0, n)$
- $(1, 0)\omega = \text{ACCEPT}$

Note that the counter accepts only when its state is 0.



2.10-4 **Generators**

(a) Let P be an acceptor for a language L. Assume that P does not use the EOF test. Show how to convert P into a program P' running on the same type of machine, that has no input but writes all strings belonging to L. To be precise, let the argument to P' be called START (since the argument is ignored, its name does not really matter). The transfer relation τ' of P' is given by

$$\text{START} \overset{\tau'}{\mapsto} x \iff x \in L.$$

Such a program is called a *generator* for L, and we say that the program *generates* L.

(b) Show how to convert a generator for L into an acceptor for L running on the same type of machine.

(c) Prove that if a program P generates an infinite language, then P has at least one infinite computation. Hint: Use König's tree lemma (Exercise 0.6-29).

Solution:

(a) Replace the input device by an output device with the usual initializer and terminator. Replace each occurrence of SCAN in P by WRITE in P'.

(b) Replace the output device by an input device with the usual initializer and terminator. Replace each occurrence of WRITE in the generator by SCAN in the acceptor.

(c) Assume that P generates an infinite language. Then P has infinitely many complete computations. A fortiori, P has infinitely many partial computations.

Form an infinite tree T whose nodes are the partial computations of P. A node p is the parent of a node c if there exists an instruction π in P such that $c = p\pi$; in this case label the edge from p to c by the instruction π.

Each node has at most $|\mathcal{I}|$ children, so T is finite-branching. Therefore T has an infinite branch. The sequence of labels on that infinite branch is an infinite computation of P.

3.2-3(b) Prove that a machine [control, input, unsigned counter] can simulate a machine [control, input, stack] where the stack has a one-character alphabet.

Solution: Let M be a machine [control, input, stack] where the stack alphabet is $\{d\}$. Let M' be a machine [control, input, unsigned counter]. We use the relation of representation

$$(q, x, n)\ \rho\ (q, x, d^n).$$

For each instruction in a program P running on M the corresponding instruction in an equivalent program P' running on M' is given in the table below:

Instruction of P	Instruction of P'
$(q \to r\ ,\ f\ ,$ NOOP $)$	$(q \to r\ ,\ f\ ,$ NOOP$)$
$(q \to r\ ,\ f\ ,$ PUSHd $)$	$(q \to r\ ,\ f\ ,$ INC $)$
$(q \to r\ ,\ f\ ,$ POPd $)$	$(q \to r\ ,\ f\ ,$ DEC $)$
$(q \to r\ ,\ f\ ,$ EMPTY$)$	$(q \to r\ ,\ f\ ,$ ZERO$)$

3.2-4 In this exercise we consider extending the definition of simulating a collection of devices to allow a list of more than one "other" device. Say that one collection of devices $d'_1, \ldots, d'_{k'}$ *simulates a collection of devices* $d_1, \ldots, d_k$ if every machine $[d'_1, \ldots, d'_{k'}, e_1, \ldots, e_j]$ simulates a machine $[d_1, \ldots, d_k, e_1, \ldots, e_j]$. Although this definition may seem more restrictive than the one in the text, prove that they are in fact equivalent. Your proof should apply to any kind of simulation, not just lockstep.

Solution: Assume that every machine $[d'_1, \ldots, d'_{k'}, \text{other}]$ simulates a machine $[d_1, \ldots, d_k, \text{other}]$. We wish to show that every machine $[d'_1, \ldots, d'_{k'}, e_1, \ldots, e_j]$ simulates a machine $[d_1, \ldots, d_k, e_1, \ldots, e_j]$. Toward this end, combine

$e_1, \ldots, e_j$ into a single device E. Operations on E act on the devices $e_1, \ldots, e_j$ individually. To be precise, the realm of E is the Cartesian product of the realms of $e_1, \ldots, e_j$. The repertory of E is the same as the set of possible instructions for a machine $[e_1, \ldots, e_j]$.

Clearly, a machine $[d'_1, \ldots, d'_{k'}, e_1, \ldots, e_j]$ simulates a machine $[d'_1, \ldots, d'_{k'}, E]$.
By assumption, a machine $[d'_1, \ldots, d'_{k'}, E]$ simulates a machine $[d_1, \ldots, d_k, E]$.
Clearly, a machine $[d_1, \ldots, d_k, E]$ simulates a machine $[d_1, \ldots, d_k, e_1, \ldots, e_j]$.
By transitivity, a machine $[d'_1, \ldots, d'_{k'}, e_1, \ldots, e_j]$ simulates a machine $[d_1, \ldots, d_k, e_1, \ldots, e_j]$.

3.3-4 Show informally how to simulate a signed counter via an unsigned counter and a control with realm $\{+, -\}$ as in this section but without augmenting Q with any new control states N. Does your simulation preserve determinism?

Solution: We simulate ZERO, POS, and NEG as before in a single step. The new simulations of INC and DEC are shown in Table 11.1.

Instruction of P	Subprograms of P'
$(q_1 \to q_2$, INC , $f)$	$(q_1 \to q_2$, $+ \to +$, INC , f $)$ $(q_1 \to q_2$, $- \to -$, DEC , f $)$ $(q_1 \to q_1$, $- \to +$, ZERO , NOOP$)$
$(q_1 \to q_2$, DEC , $f)$	$(q_1 \to q_2$, $+ \to +$, DEC , f $)$ $(q_1 \to q_2$, $- \to -$, INC , f $)$ $(q_1 \to q_1$, $+ \to -$, ZERO , NOOP$)$

Table 11.1: An unsigned counter simulates a signed counter without using any extra states. This simulation does not preserve determinism.

In general, P' might not be deterministic even if P is, because, for example, the simulation of $(q_1 \to q_2, \text{DEC}, \text{SCAN}\mathtt{a})$ and $(q_1 \to q_2, \text{INC}, \text{SCAN}\mathtt{b})$, which do not have overlapping domains, would involve the instructions $(q_1 \to q_2, - \to -, \text{INC}, \text{SCAN}\mathtt{a})$ and $(q_1 \to q_1, - \to +, \text{ZERO}, \text{NOOP})$, which are both applicable to the configuration $(q_1, -, 0, \mathtt{a})$.

3.3-5 Show informally how to simulate a signed counter via an unsigned counter and a control with realm $\{+, -\}$ as in this section, using subprograms that take at most two steps to simulate an instruction of P. Your simulation should preserve determinism. In your construction, how many subprograms of P' simulate a single instruction of P?

Solution: We simulate ZERO, POS, and NEG as before in a single step. The new simulations of INC and DEC are shown in Table 11.2.

Instruction of P	Subprogram of P'
$(q_1 \to q_2$, INC , $f)$	$(q_1 \to q_2$, $+ \to +$, INC , f) $(q_1 \to q_2$, $- \to -$, DEC , f) $(q_1 \to \nu$, $- \to +$, ZERO , f) $(\nu \to q_2$, $+ \to +$, INC , NOOP)
$(q_1 \to q_2$, DEC , $f)$	$(q_1 \to q_2$, $+ \to +$, DEC , f) $(q_1 \to q_2$, $- \to -$, INC , f) $(q_1 \to \nu$, $+ \to -$, ZERO , f) $(\nu \to q_2$, $- \to -$, INC , NOOP)

Table 11.2: An unsigned counter simulates a signed counter in two steps.

The instructions of P' used in simulating an instruction of P containing the INC operation constitute three subprograms, because only two of them meet at a new state. Similarly, the instructions used in simulating DEC constitute three subprograms.

3.3-6 Design a simulation of a signed counter by an unsigned counter using a relation of representation that is one-one.

Solution: We use an extra control called the "sign," as in this section. However, we translate the representation of negative numbers by 1, to avoid the duplicate representation of 0, i.e., we let

$$(q, +, n, s) \quad \rho \quad (q, n, s),$$
$$(q, -, n, s) \quad \rho \quad (q, -1 - n, s).$$

The simulation is shown in Table 11.3.

3.3-7 Refer to the simulation of a signed counter by an unsigned counter and a control called "sign," as explained in this section.

(a) Define α' and ω' appropriately. What do we need to prove about those two partial functions? Prove it.

(b) Construct the commutative diagram corresponding to the NEG operation.

(c) Draw the subprogram corresponding to the DEC operation.

(d) Construct the commutative diagram corresponding to the DEC operation.

Solution:

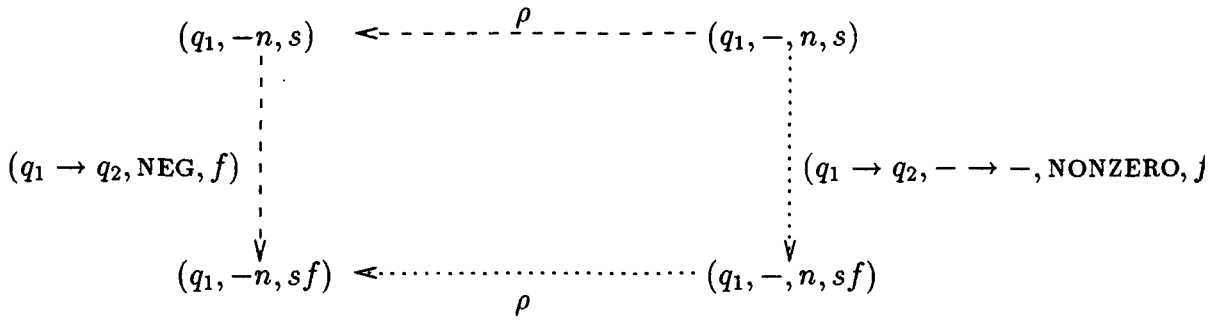
Instruction of P	Subprogram of P'
$(q_1 \to q_2$, INC , $f)$	$(q_1 \to q_2$, $+ \to +$, INC , $f)$ $(q_1 \to q_2$, $- \to -$, DEC , $f)$ $(q_1 \to q_2$, $- \to +$, ZERO , $f)$
$(q_1 \to q_2$, DEC , $f)$	$(q_1 \to q_2$, $+ \to +$, DEC , $f)$ $(q_1 \to q_2$, $- \to -$, INC , $f)$ $(q_1 \to q_2$, $+ \to -$, ZERO , $f)$
$(q_1 \to q_2$, ZERO , $f)$	$(q_1 \to q_2$, $+ \to +$, ZERO , $f)$
$(q_1 \to q_2$, POS , $f)$	$(q_1 \to q_2$, $+ \to +$, NONZERO , $f)$
$(q_1 \to q_2$, NEG , $f)$	$(q_1 \to q_2$, $- \to -$, NOOP , $f)$

Table 11.3: A signed counter simulates an unsigned counter by subprograms with a one-one relation of representation. A generic instruction of the SCM is shown in the first column. A subprogram that simulates it is shown in the second column.

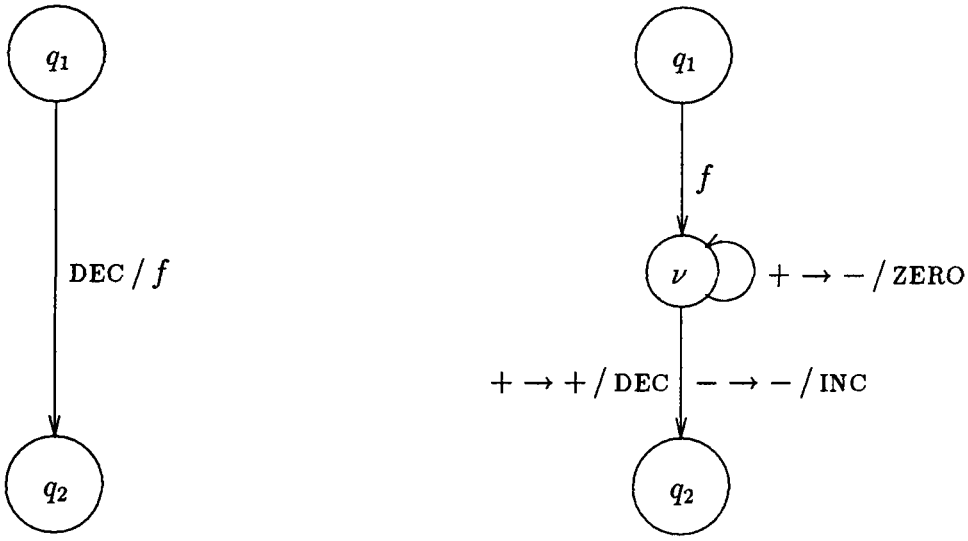
(a)
- Let $\alpha'_{\text{control}} = \alpha_{\text{control}}$, $\alpha'_{\text{other}} = \alpha_{\text{other}}$, $\alpha'_{\text{sign}} = X \times \{+\}$, and $\alpha'_{\text{counter}} = X \times 0$.
  Since the initial subprogram is trivial, we must prove that $\alpha = \alpha' \circ \rho$. For all arguments x, we have
  $$x\alpha' = (x\alpha_{\text{control}}, +, 0, x\alpha_{\text{other}}),$$
  so
  $$x\alpha'\rho = (x\alpha_{\text{control}}, 0, x\alpha_{\text{other}}) = x\alpha.$$
  Thus $\alpha'\rho = \alpha$, as desired.
- Let $\omega'_{\text{control}} = \omega_{\text{control}}$, $\omega'_{\text{other}} = \omega_{\text{other}}$, and $\omega'_{\text{sign}} = \{+, -\} \times Y$. $\omega'_{\text{counter}} = 0 \times Y$ if $\omega_{\text{counter}} = 0 \times Y$; $\omega'_{\text{counter}} = \mathrm{N} \times Y$ if $\omega_{\text{counter}} = \mathrm{Z} \times Y$. Since the final subprogram is trivial, we must prove that $\rho \circ \omega = \omega'$. For any configuration $C = (q, \text{sign}, n, s)$ of P', we have $C\rho = (q, \text{sign}\, n, s)$, so $C\ \rho\omega\ y$ iff $q\ \omega_{\text{control}}\ y$, sign is arbitrary, n is 0 if 0 is the only accepting state of P's counter (n is arbitrary otherwise),

and $s\ \omega_{\text{other}}\ y$, i.e., iff $C\ \omega'\ y$. Thus $\rho\omega = \omega'$.

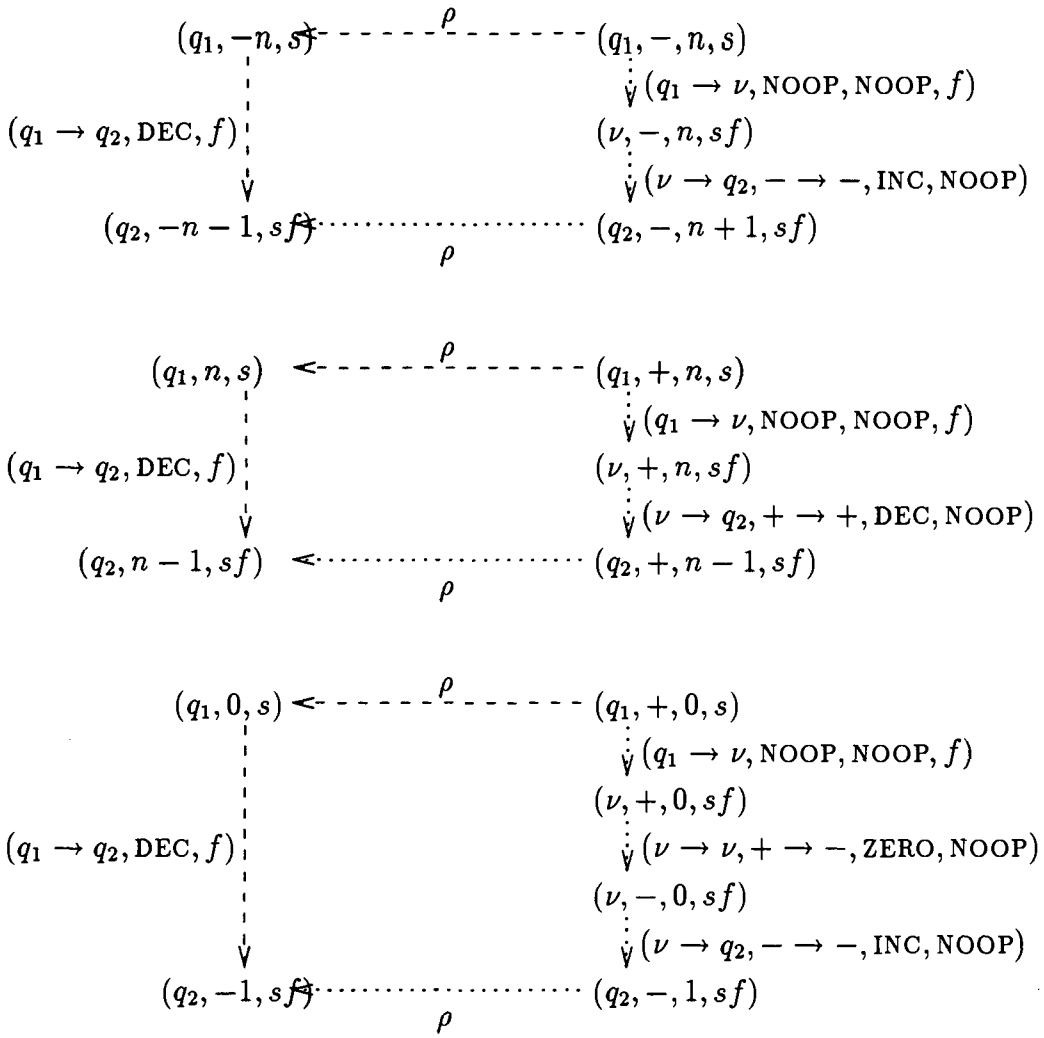
(b)



(c)



(d)



3.3-9 Show how to simulate an unsigned counter by a signed counter.

Solution: Let M be a machine [control, unsigned counter, other]. Let M' be a machine [control, signed counter, other]. We use the relation of representation

$$(q, n, s)\ \rho\ (q, n, s).$$

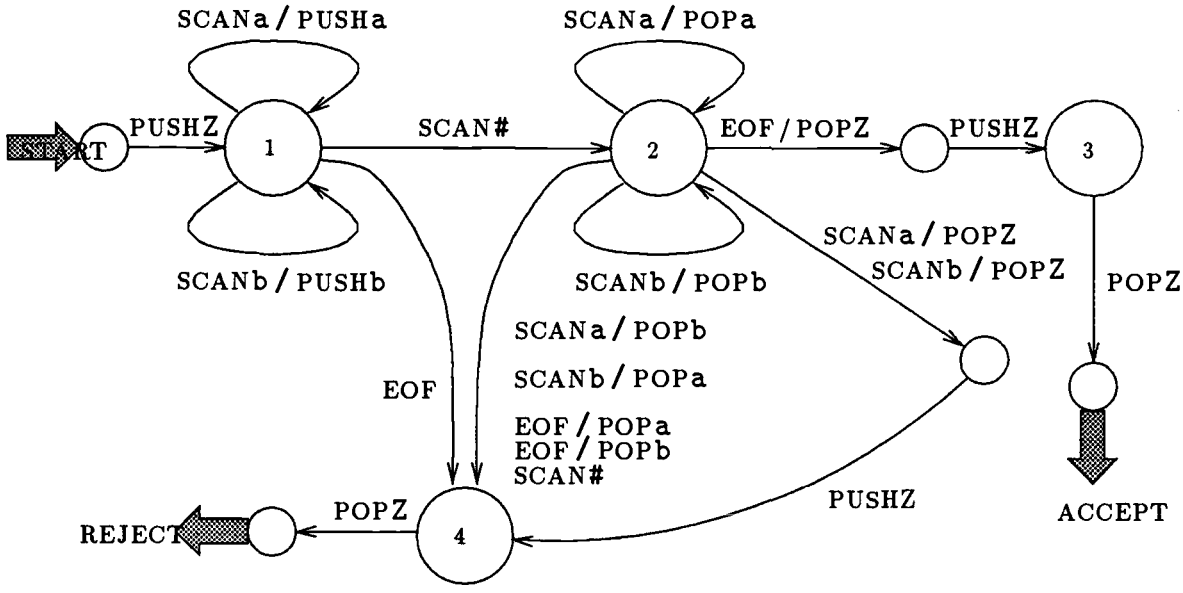
For each instruction in a program P running on M the corresponding instruction in an equivalent program P' running on M' is given in the table

below (only the simulation of DEC is nontrivial):

Instruction of P	Instruction of P'
$(q \rightarrow r$, NOOP , $f)$	$(q \rightarrow r$, NOOP , f $)$
$(q \rightarrow r$, INC , $f)$	$(q \rightarrow r$, INC , f $)$
$(q \rightarrow r$, ZERO , $f)$	$(q \rightarrow r$, ZERO , f $)$
$(q \rightarrow r$, DEC , $f)$	$(q \rightarrow \nu$, POS , f $)$
	$(\nu \rightarrow r$, DEC , NOOP$)$

3.3-10 Eliminate the EMPTY test from the program in Figure 1.1.

Solution:



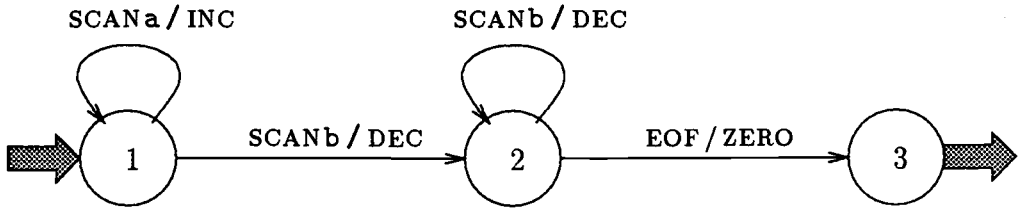
3.4-2 In order to standardize a stack, we must eliminate the TOP operations and also ensure that any two occurrences of EMPTY in a partial computation are separated by a PUSH. Show how to perform this standardization. Be sure to preserve factored form.

Solution: As when we standardized the input, we store the top stack character in a buffer. Initialize the buffer to hold z, which denotes an empty stack. To simulate TOPc, test that the buffer holds c. To simulate POPc, replace the buffer's contents by the top character, which you pop; if the stack is empty, then put a z in the buffer. To simulate PUSHc, push the buffer's contents onto the stack (unless the buffer holds z), and store a c in the buffer. To simulate EMPTY, test that the buffer holds z. This is a lockstep simulation.

An alternative solution uses simulation by subprograms, without a buffer. Initialize the stack to hold z. To simulate TOP*c*, pop a *c* and then push a *c*. To simulate POP*c*, just pop a *c*. To simulate PUSH*c*, just push a *c*. To simulate EMPTY, pop a z and then push a z.

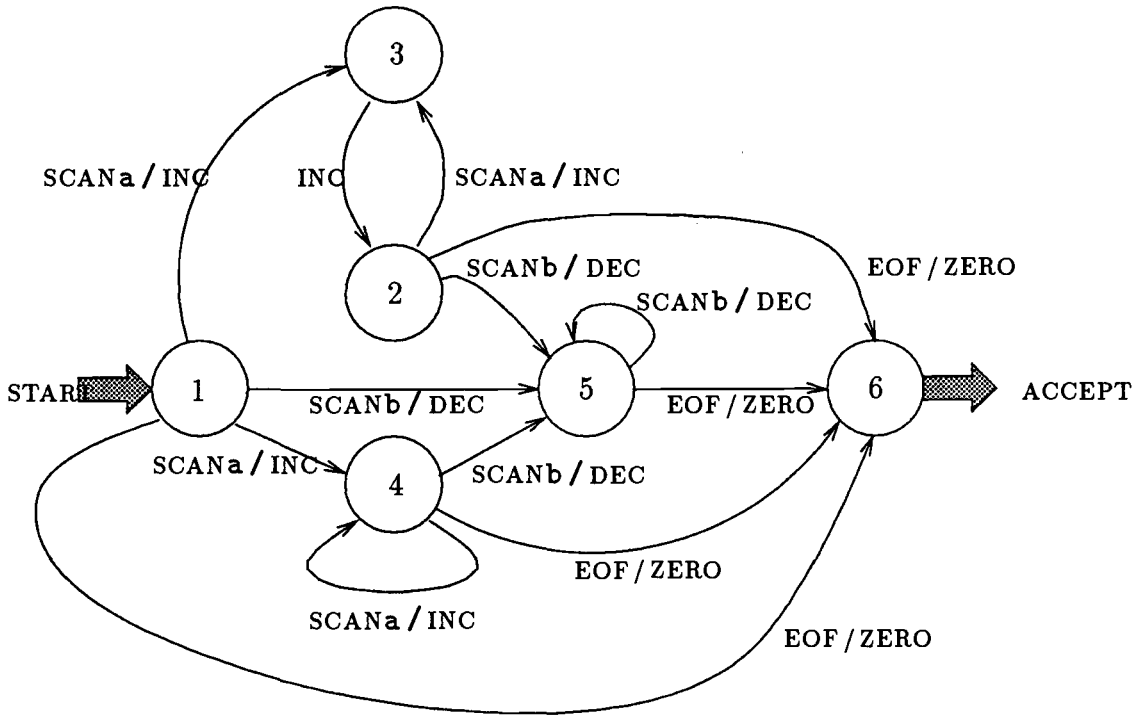
3.4-8 Eliminate dead states from the program depicted in Figure 3.42.

Solution:



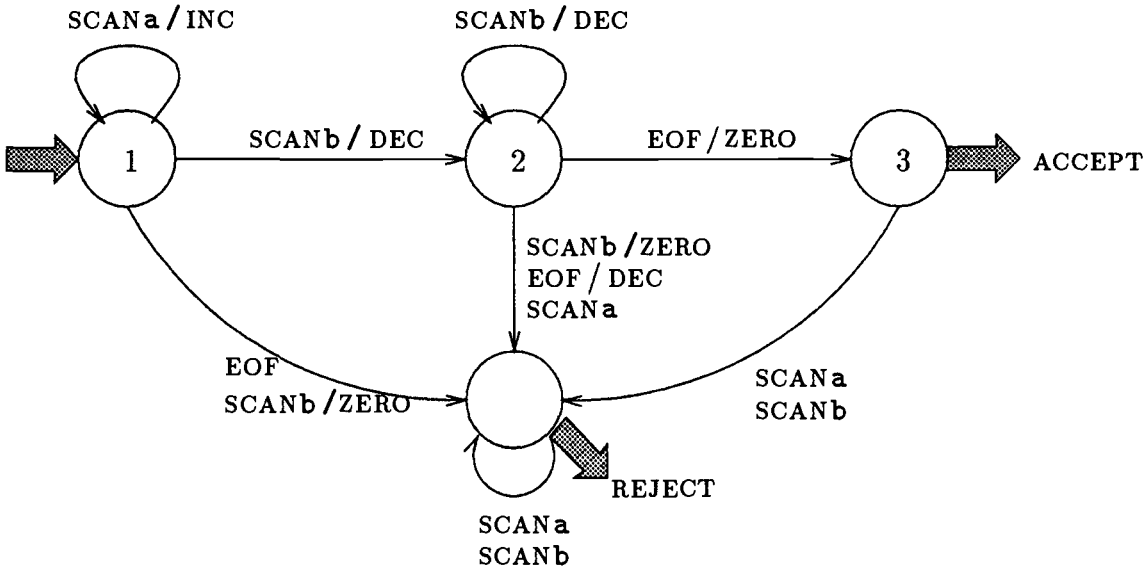
3.4-15(b) Eliminate the null instructions from the program in figure Figure 1.21.

Solution:



3.4-19 Eliminate blocking from your program for Exercise 3.4-8.

Solution:



3.5-1 Chaos Computing is designing a new line of computers that have no control device. Because of your expertise in simulation, you have been hired to consult on the project. Assume that the control to be simulated has realm Q. Be careful not to use any controls in your simulation.

(a) Show how to simulate the control using a tape.

(b) Show how to simulate the control using two stacks.

(c) Show how to simulate the control using $|Q|$ unsigned counters. You may use the NONZERO instruction on the counters.

(d) Show how to simulate the control using unsigned counters without the NONZERO test.

(e) What effects do your simulations in parts (a–d) have on programs that are in factored form?

(f) Show how to simulate a machine [control, stack, stack] by a machine [stack, stack, stack].

Solution:

(a) We use a tape with alphabet Q. The control state q will be represented by a tape with q on its first square. The operation $i \to j$ is simulated by [SEEi, PRINTj].

(b) We use two stacks with alphabet Q. The control state q will be represented by having one stack (not necessarily the first) hold q and the other hold Λ. The instruction $(i \to j, f)$ is simulated by a pair of instructions (because we don't know which stack holds q): (POPi, PUSHj, f) and (PUSHj, POPi, f).

Alternative solution: Let the first stack hold q and the second be empty. Simulate $(i \to j, f)$ by the pair of instructions (POPi, PUSHj, f) and (PUSHj, POPj, NOOP).

(c) We use $|Q|$ unsigned counters. The control state q is represented by having the qth counter hold 1 and the other counters hold 0. The operation $i \to j$ is simulated by DEC_i / INC_j, if $i \neq j$. The operation $i \to i$ is simulated by NONZERO_i.

(d) The simplest solution is to eliminate self-loops by replacing $i \to i$ with a pair of instructions $i \to i'$, $i' \to i$. Then proceed as in part (c). The standardization doubles the size of the control, so $2|Q|$ counters are needed.

The simulation above is not lockstep. An alternative solution uses $2|Q|$ counters directly. For each control state q, there will be two counters called q and q'. The control state q is represented by having either q or q' hold 1 (but not both) and the other counters hold 0. The operation $i \to j$ is simulated by DEC_i / $\text{INC}_{j'}$ and $\text{DEC}_{i'}$ / INC_j.

(e) Assume that the simulated program is factored. In part (a), each instruction of the simulating program will operate on the tape and at most one other device. In part (b), each instruction of the simulating program will operate on both stacks and at most one other device. In part (c), each instruction of the simulating program will operate on one or two of the unsigned counters and at most one other device. In part (d), each instruction of the simulating program will operate on exactly two of the unsigned counters and at most one other device.

(f) Let P be a program for a machine [control, stack, stack]. First, factor P so that each instruction operates on at most one of the stacks. Without loss of generality assume that P's control set is disjoint from its stack alphabets. In P', let the first stack hold the control state. Let op be a stack operation. The instruction $(i \rightarrow j, op, \text{NOOP})$ is simulated by the pair of instructions $(\text{POP}i, op, \text{PUSH}j)$ and $(\text{PUSH}j, \text{NOOP}, \text{POP}j)$. The instruction $(i \rightarrow j, \text{NOOP}, op)$ is simulated by the pair of instructions $(\text{POP}i, \text{PUSH}j, op)$ and $(\text{PUSH}j, \text{POP}j, \text{NOOP})$.

3.5-2 Devices 'R Us (TM) is marketing a new-and-improved counter. Their new device has all the ordinary operations, and two additional operations: INC2, which adds 2 to the counter's state, and DEC2, which subtracts 2 from the counter's state. Their ads say that their new device will speed up counter machine programs by a factor of 2.

Their competitors, the Itty Bitty Machine Corporation, have countered by providing a free control with every purchase of an ordinary counter. They have hired you — an independent researcher — to prove that an ordinary counter together with a control is just as good and just as fast as a new-and-improved counter.

Show how to simulate a new-and-improved counter, in lockstep, using an ordinary counter and a control.

Solution: The value n in the new-and-improved counter is represented by $\lfloor n/2 \rfloor$ in an ordinary counter and $n \bmod 2$ in a control. Formally, let M be a machine [new-and-improved counter, other] and let M' be a machine [counter, control, other]. The relation of representation is

$$(n, b, s)\ \rho\ (2n + b, s).$$

Instructions are simulated as follows:

Instruction of P	Instruction of P'
(INC2 , f)	(INC , NOOP , f)
(DEC2 , f)	(DEC , NOOP , f)
(INC , f)	(INC , $1 \rightarrow 0$, f)
	(NOOP , $0 \rightarrow 1$, f)
(DEC , f)	(DEC , $0 \rightarrow 1$, f)
	(NOOP , $1 \rightarrow 0$, f)
(NOOP , f)	(NOOP , NOOP , f)
(ZERO , f)	(ZERO , $0 \rightarrow 0$, f)

4.2-2 Write regular expressions that generate the following languages:

(a) the set of all strings over $\{\mathtt{a}, \mathtt{b}\}$ that contain exactly two a's

(b) the set of all strings over $\{\mathtt{a}, \mathtt{b}\}$ that contain at least two a's

(c) the set of all strings over $\{\mathtt{a}, \mathtt{b}\}$ whose length is divisible by 3

Solution:

(a) $\mathtt{b}^*\mathtt{a}\mathtt{b}^*\mathtt{a}\mathtt{b}^*$

(b) $(\mathtt{a} \cup \mathtt{b})^*\mathtt{a}(\mathtt{a} \cup \mathtt{b})^*\mathtt{a}(\mathtt{a} \cup \mathtt{b})^*$

(c) $((\mathtt{a} \cup \mathtt{b})(\mathtt{a} \cup \mathtt{b})(\mathtt{a} \cup \mathtt{b}))^*$

4.2-4 Recursively define a function f from regular expressions to natural numbers as follows:

$$\begin{aligned} f(\mathtt{a}) &= 1 \\ f(\mathtt{b}) &= 1 \\ f(\Lambda) &= 0 \\ f(r_1 \cup r_2) &= \min(f(r_1), f(r_2)) \\ f(r_1 r_2) &= f(r_1) + f(r_2) \\ f(r^*) &= 0. \end{aligned}$$

Prove that $f(r)$ is equal to the length of the shortest string generated by r, i.e.,

$$f(r) = \min_{x \in L(r)} |x|.$$

Solution: The proof is by structural induction on the regular expression r. Assume that the claim is true for all subexpressions of r.

Case 1: $r = \emptyset$. The shortest string generated by r is undefined, as is $f(r)$.

Cases 2–4: r is a, b, or Λ. The claim is immediate from the definition of f.

Case 5: $r = r_1 \cup r_2$. Let s_i be the shortest string generated by r_i, for $i = 1, 2$. By the inductive hypothesis, $f(r_i)$ is the length of s_i. The shortest string generated by $r_1 \cup r_2$ is the shorter of s_1 or s_2, so its length is $\min(f(r_1), f(r_2))$, which is $f(r)$ by definition.

Case 6: $r = r_1 r_2$. Let s_i be the shortest string generated by r_i, for $i = 1, 2$. By the inductive hypothesis, $f(r_i)$ is the length of s_i. The shortest string generated by $r_1 r_2$ is $s_1 s_2$, so its length is $f(r_1) + f(r_2)$, which is $f(r)$ by definition.

Case 6: $r = r_1^*$. The shortest string generated by r_1^* is Λ, whose length is 0, which is $f(r)$ by definition.

In every case, the claim is true for r, completing the induction.

4.3-3 Write a command using egrep that searches myfile.txt for all lines that contain at least two y's or at least two z's.

Solution: `egrep 'y.*y|z.*z' myfile.txt`

4.3-4 Write a command using egrep that searches myfile.txt for all lines that contain the string Bob and do not start with %.

Solution: Each of the following commands works correctly:

```
egrep '^[^%].*Bob|^Bob' myfile.txt
egrep '^([^%].*)?Bob' myfile.txt
egrep '^([^%].*)*Bob' myfile.txt
```

4.4-1 Let P be any NFA. Prove that the set of computations of P is a regular language.

Solution: The first thing to note is that P might not be in standard form, and certain standardizations (like eliminating null instructions) would affect the set of computations. At least two solutions are possible:

Solution 1. Each computation of P is a finite sequence of instructions, i.e., a string over the alphabet $\mathcal{I}$. (You could use another alphabet like the set of characters on a typewriter, but that would entail more work.) Let $\mathcal{C}$ denote the set of accepting computations of P. We will construct an NFA P' with input alphabet $\mathcal{I}$ that accepts the language $\mathcal{C}$. Because all NFA languages are regular, this suffices to prove that $\mathcal{C}$ is regular.

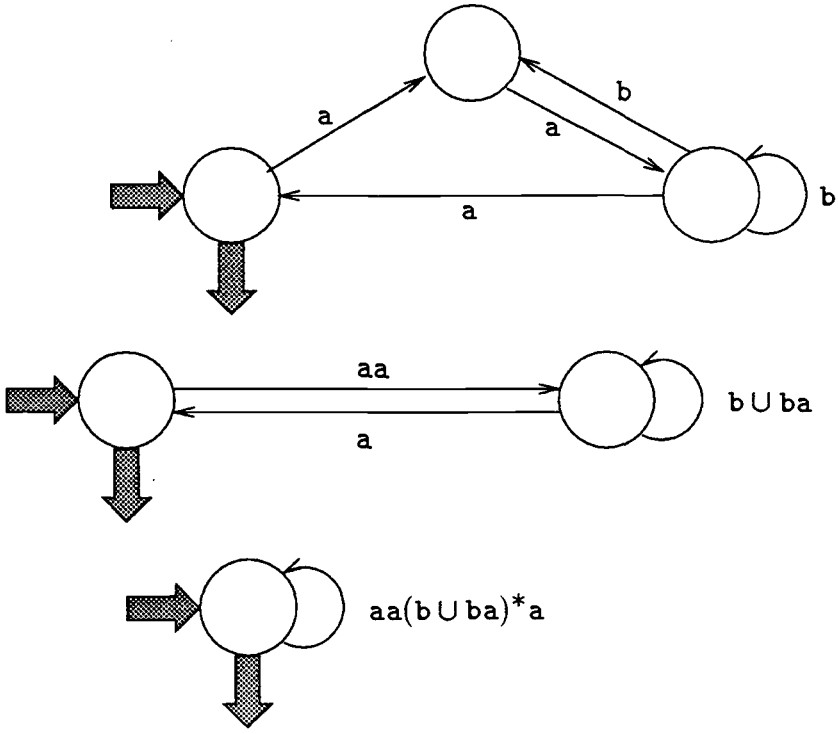
P' will have the same initial and final states as P. For each instruction π in P, replace the input operation by SCANπ, and let P' contain the new instruction. Then P' accepts the language $\mathcal{C}$.

Solution 2. In P's digraph, label each edge by the instruction performed, rather than just the input operation. By Kleene's theorem, the set of paths

from the initial state to a final state in this graph is a regular set; call that set R. R consists of every sequence of instructions in P that leads from the initial state to a final state. Such a sequence of instructions is a computation iff it is possible to execute the sequence of instructions on some input iff no EOF test is followed by a SCAN. Let A be the set of instructions in P that do not perform an EOF test, and let B be the set of instructions in P that do not perform a SCAN. Then the set of computations of P is $R \cap A^*B^*$, which is regular, because the class of regular languages is closed under intersection, concatenation, and Kleene star.

4.4-6 Use our pencil-and-paper algorithm to determine the regular language accepted by the NFA in Figure 4.17. Show your work.

Solution: We will save a little work by not standardizing the initial and final states.

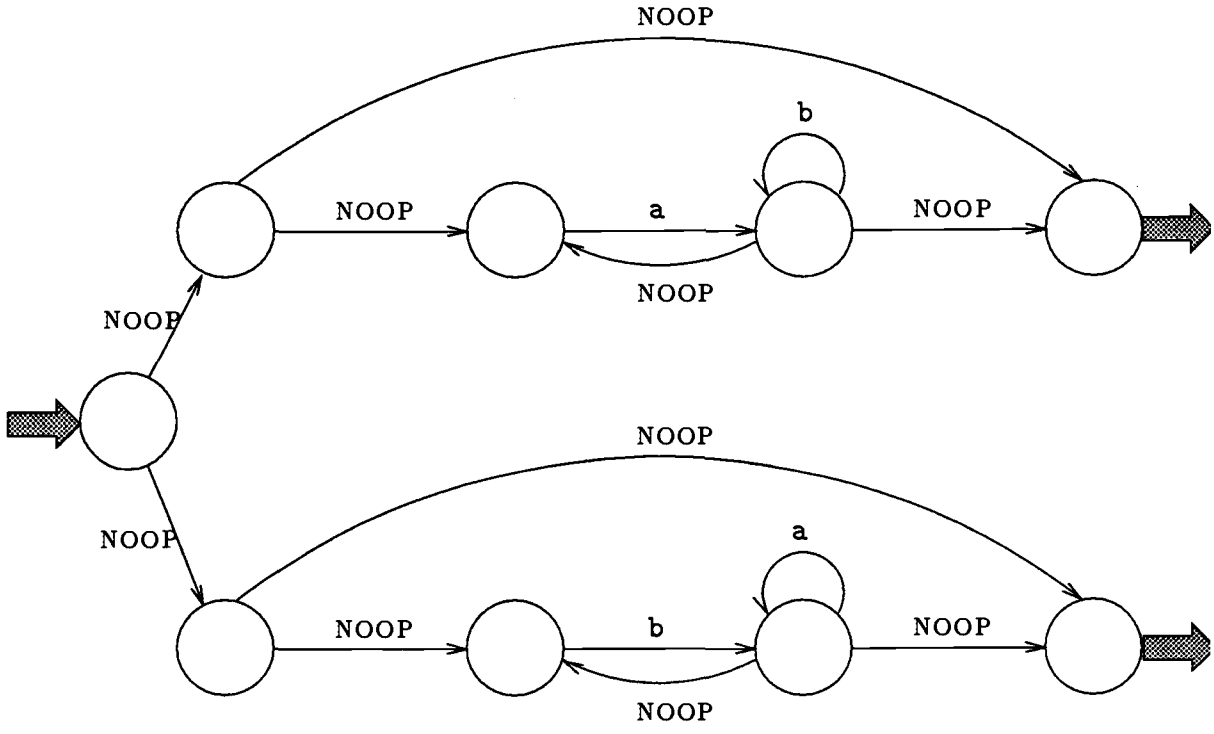


Thus the language accepted is $(\mathtt{aa}(\mathtt{b} \cup \mathtt{ba})^*\mathtt{a})^*$.

4.5-1 Use Lemma 4.16 to construct NFAs that accept the following languages:

(c) $(\mathtt{ab}^*)^* \cup (\mathtt{ba}^*)^*$

Solution:



4.5-2 In Figure 4.20, we introduced four new edges in order to construct an NFA for L_1^* with start state q_- and accepting state q_+. Show how to do the construction with only three new edges, still ensuring that no instructions go to the start state or from the accepting state. If we drop that condition, show that one new edge suffices.

Solution: For the first part, replace $(q_- \rightarrow q_+, \text{NOOP})$ and $(q_{1+} \rightarrow q_+, \text{NOOP})$ by $(q_{1-} \rightarrow q_+, \text{NOOP})$. For the second part, instead of adding a new start and a new final state, it suffices to just add the null instruction going from q_{1+} to q_{1-}.

4.5-3(b) A program is called *planar* if its digraph can be drawn in the plane without having any edges cross.

Construct a regular language that is not accepted by any planar DFA. Hint: Every undirected planar graph contains a vertex with fewer than six neighbors.

Solution:

Let L = the set of strings over $\{1,2,3,4,5,6,7\}$ such that some character occurs in two consecutive positions, i.e., at least one of 11, ..., 77 occurs). We will show that every DFA accepting L must be non-planar.

In our proof, we need a theorem from graph theory: if an undirected graph G (without self-loops or parallel edges) is planar, then G must contain a vertex

of degree 5 or less.[1]

Suppose there is a planar DFA that accepts L. Eliminate null instructions; because the program is deterministic this preserves planarity. Eliminate EOF as follows: make q an accepting state if there is a path from q to an accepting state in which each edge (possibly zero in all) is labeled EOF; then delete each edge labeled EOF. Delete unreachable states. Call the resulting program P.

Let q be any nonaccepting state in P. Because P has no unreachable states and does not use the EOF test, q is reached on some input x. For each character c that is different from the last character of x (or for $c = 1, ..., 6$ in the case $x = \Lambda$), let q_c be the state reached on input xc. Because xc is not in L, q_c is a nonaccepting state. For $c \neq d$, we have $q_c \neq q_d$, because $xcc \in L$ and $xcd \notin L$. In addition we have $q_c \neq q$, because $xcc \in L$ and $xc \notin L$.

Thus from each nonaccepting state q, there are edges to 6 distinct nonaccepting states other than q. Delete all accepting states from P's digraph, and delete all edges going to or from accepting states. Delete self-loops. Make each edge undirected and merge parallel edges. The resulting graph is planar but each vertex has degree at least 6. This contradiction proves that L is not accepted by a planar DFA.

4.6-1 (a) Can every nondeterministic finite transducer be simulated by a deterministic transducer running on some type of machine? Hint: Deterministic transducers compute partial functions.

(b) Suppose that we replace the input device by an output device in this section's proof that an NFA without null instructions can be simulated by a DFA. Where does this introduce an error in the proof?

Solution:

(a) No. A nondeterministic finite transducer can compute a relation that is not a partial function (Figure 2.14 for example). But deterministic transducers compute only partial functions.

(b) In proving determinism we used the fact that if the input operations SCANc and SCANd are both applicable, then $c = d$. The analogous statement is not true for WRITEc and WRITEd.

4.6-8 (a) In this problem we define a notion of acceptance that is dual to nondeterministic acceptance. An ∀FA (pronounced "all-eff-ay") is like an NFA except that it accepts an input x iff all computations on input x are accepting and there are no infinite computations or blocked partial computations on input x. Show how to convert an ∀FA to an equivalent DFR.

Solution: Eliminate blocking. Then use the subset construction. Each set of states is accepting iff it contains only accepting states.

[1]See, for example, Bondy and Murty, *Graph Theory with Applications*, p. 144.

(b) Lucy and Charlie are sitting at a table. On the table is a tray with 4 glasses arranged in a square. Charlie's goal is to turn all the glasses either right-side up or upside down. However, Charlie is blindfolded and he is wearing mittens. He does not know the initial state of the glasses. If they are initially all turned the same way, then Charlie automatically wins. In his turn, Charlie may grab one or two glasses and turn them over; however, because of the blindfold and mittens he cannot see or feel whether the glasses he grabbed are right-side up or upside down. He can, however, choose whether to grab adjacent glasses or diagonally opposite glasses. If the glasses are all turned the same direction, Lucy announces that Charlie has won. Otherwise, Lucy rotates the tray, just to make Charlie's goal harder.

Find the shortest sequence of actions by Charlie that is guaranteed to win the game, no matter how Lucy plays. Prove that your solution is correct and is the shortest possible.

Solution: By symmetry, there are four inequivalent states of the glasses. State oooo: all glasses turned the same way. State xooo: one glass turned one way and three turned the other way. State xxoo: two adjacent glasses turned up and the other two turned down. State xoxo: two opposite glasses turned up and the other two turned down.

Charlie has three possible actions in his turn. Action `1`: turn one glass over. Action `A`: turn two adjacent glasses over. Action `O`: turn two opposite glasses over.

The problem may be modeled as an ∀FA with 4 states corresponding to the arrangement of the glasses. The input is the sequence of actions performed by Charlie, which can be represented as a string over a 3-character alphabet. State oooo is the unique accepting state. All arcs from state oooo go back to state oooo, because the game is won as soon as state oooo is reached. Make an arc from state i to state j labeled SCANc if action c might take the glasses from state i to state j. This ∀FA is shown in Figure 11.1.

A sequence is guaranteed to win if and only if it is accepted by this ∀FA no matter which state it is started in. We convert the ∀FA to an equivalent DFR, as shown in Figure 11.2. A sequence is guaranteed to win if and only if it is accepted by this DFR when started in state {oooo, xooo, xxoo, xoxo}. To find the shortest such sequence, we find the shortest path from {oooo, xooo, xxoo, xoxo} to {oooo} in the DFR's digraph. The sequence of characters scanned on this path is `OA01OAO`, which is therefore the shortest sequence of actions by Charlie that is guaranteed to win.

4.7-1 Repeat Example 4.25 for substrings `abbaba` and `bbaaba`.

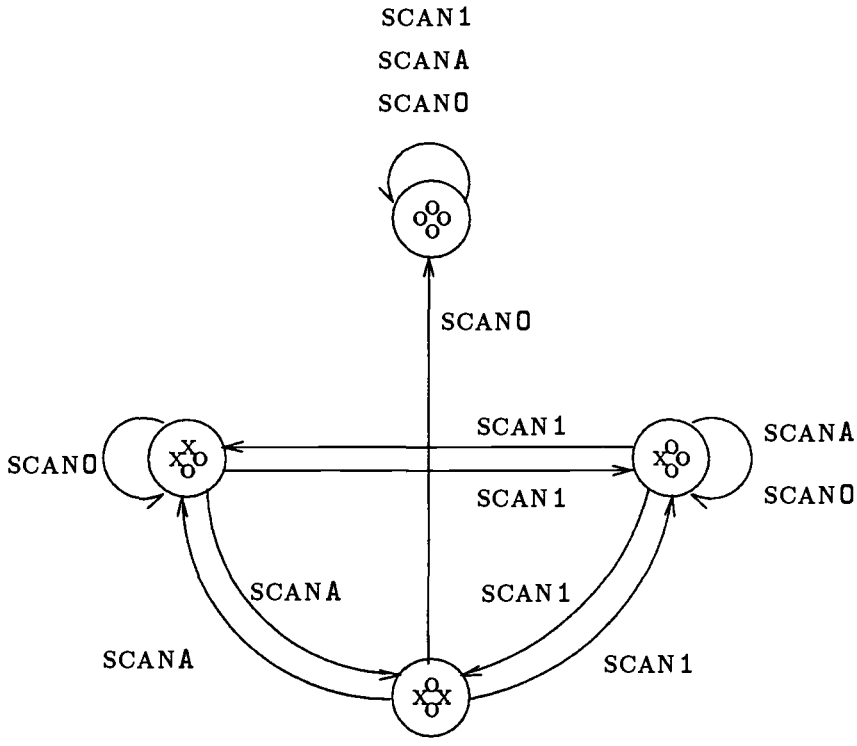


Figure 11.1: An ∀FA for the glasses game. Each state name is written cyclically because that is suggestive of glasses arranged in a square. The input character `1` means turn one glass over, `A` means turn two adjacent glasses over, and `0` means turn two opposite glasses over.

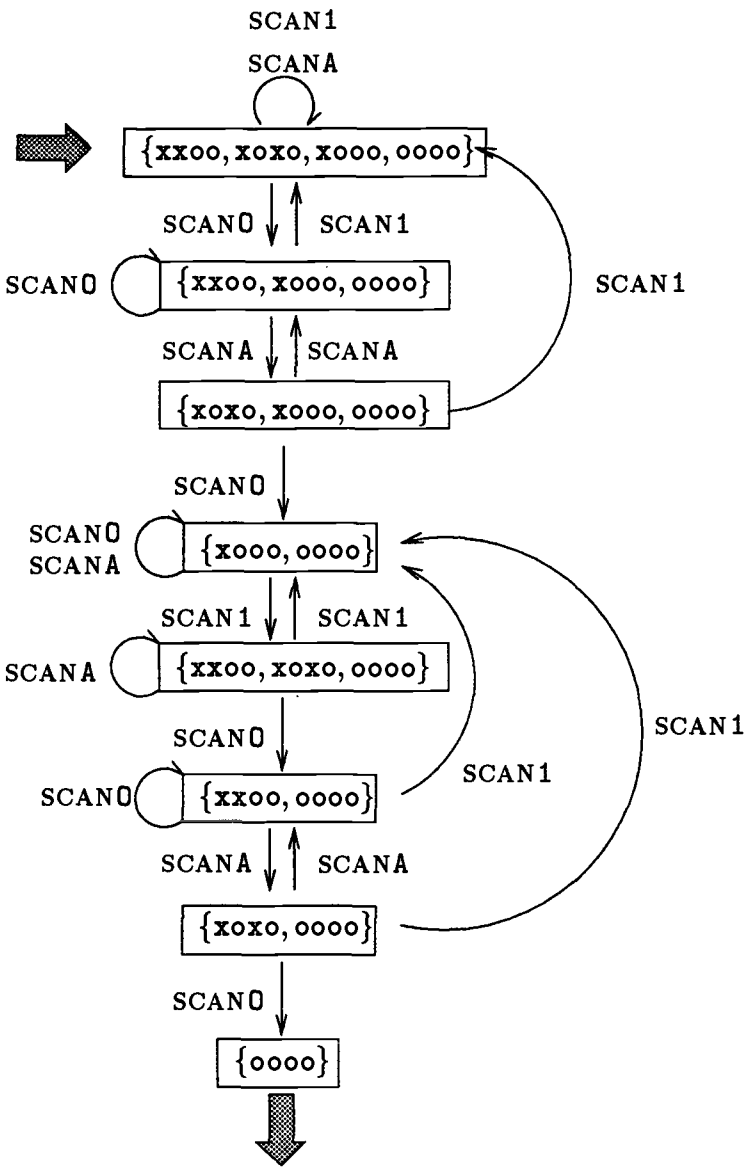


Figure 11.2: A DFR for the glasses game.

Solution: The program is presented in Figure 11.3. The rightmost state is accepting and all others are rejecting. Let $p_i(x)$ denote the prefix of x having length i, and let $s_i(x)$ denote the suffix of x obtained by deleting $p_i(x)$ from the beginning. Note that all but the two rightmost states have been labeled with the string p_i(abbaba) or p_i(bbaaba) for some $i \leq 4$. The states abba and bbaa are distinguished by the string aba. The states abb and bba are distinguished by the string aaba. Every other pair of states is distinguished by a shortest string that takes one of them to an accepting state. Therefore this DFR has the minimum possible number of states.

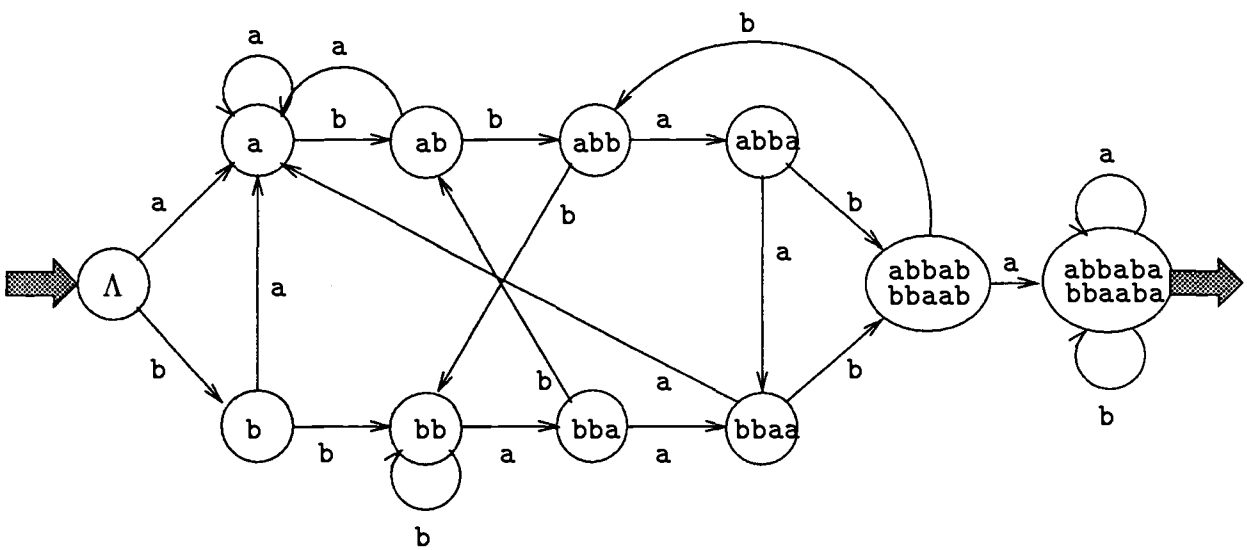


Figure 11.3: A program that recognizes $\{a,b\}^*$(abbaba $\cup$ bbaaba)$\{a,b\}^*$. All states are rejecting unless marked accepting.

4.7-4 Prove that the minimal equivalent DFR P' constructed by the techniques of this section recognizes the same language as P.

Solution: P' simulates P in lockstep with the relation of representation defined by $([q], x)\ \rho\ (q, x)$.

4.7-7 Let L be recognized by a DFR P that has no null instructions, EOF tests, or unreachable states. Let q_{start} be the initial state of P.

(a) Prove that if $q_{\text{start}}t_x = q_{\text{start}}t_y$, then $x \sim_L y$.

(b) Give a counterexample to show that $x \sim_L y$ does not imply $q_{\text{start}}t_x = q_{\text{start}}t_y$.

(c) Define the *infix equivalence relation* for L (denoted $\approx_L$) by $x \approx_L y$ if and only if

$$(\forall u)(\forall v)[uxv \in L \iff uyv \in L].$$

Prove that if $t_x = t_y$ then $x \approx_L y$.

(d) Give a counterexample to show that $x \approx_L y$ does not imply $t_x = t_y$.

(e) Now assume that P is minimal. Prove that

$$q_{\text{start}}t_x = q_{\text{start}}t_y \iff x \sim_L y$$

and

$$t_x = t_y \iff x \approx_L y.$$

Solution:

(a) Assume that $q_{\text{start}}t_x = q_{\text{start}}t_y$. Let u be any string. Then $xu \in L \iff q_{\text{start}}t_{xu}\omega_{\text{control}} = \text{ACCEPT}$. But $q_{\text{start}}t_{xu} = q_{\text{start}}t_xt_u = q_{\text{start}}t_yt_u = q_{\text{start}}t_{yu}$, so

$$xu \in L \iff q_{\text{start}}t_{yu}\omega_{\text{control}} = \text{ACCEPT} \iff yu \in L.$$

Therefore, $x \sim_L y$.

(b) For this problem it is sufficient to consider any nonminimal DFR and choose x and y to be strings that lead to distinct but equivalent states. Since a concrete counterexample was requested, let $L = \mathtt{a}^*$. Let P be a DFR with control set $\{0, 1\}$, initial state 0, accepting states 0 and 1, and instruction set $\{(0 \to 1, \text{SCAN}\,\mathtt{a}), (1 \to 0, \text{SCAN}\,\mathtt{a})\}$. Observe that P accepts L. Let $x = \Lambda$ and $y = \mathtt{a}$, which are prefix-equivalent, but $q_{\text{start}}t_x = 0$ and $q_{\text{start}}t_y = 1$.

(c) Assume that $t_x = t_y$. Let u and v be any strings. Then $uxv \in L \iff q_{\text{start}}t_{uxv}\omega_{\text{control}} = \text{ACCEPT}$. But $t_{uxv} = t_ut_xt_v = t_ut_yt_v = t_{uyv}$, so $uxv \in L \iff q_{\text{start}}t_{uyv}\omega_{\text{control}} = \text{ACCEPT} \iff uyv \in L$. Therefore, $x \approx_L y$.

(d) We use the same concrete counterexample as in part (b). The strings Λ and $\mathtt{a}$ are infix-equivalent (in fact all strings over $\{\mathtt{a}\}$ are), but $t_x \neq t_y$ because $q_{\text{start}}t_x \neq q_{\text{start}}t_y$.

(e) The forward implications were proved in parts (a) and (b), so it suffices to prove the reverse implications. Assume that P is minimal.

To show $x \sim_L y \Rightarrow q_{\text{start}}t_x = q_{\text{start}}t_y$. Assume that $x \sim_L y$, so $(\forall u)[xu \in L \iff yu \in L]$. Let $q_x = q_{\text{start}}t_x$ and $q_y = q_{\text{start}}t_y$. Then $(\forall u)[q_xt_u\omega_{\text{control}} = q_yt_u\omega_{\text{control}}]$, so q_x and q_y are equivalent states. If $q_x \neq q_y$ then we could merge q_x and q_y into a single state, contradicting P's minimality; therefore $q_x = q_y$.

To show $x \approx_L y \Rightarrow t_x = t_y$. Assume that $x \approx_L y$, so $(\forall u, v)[uxv \in L \iff uyv \in L]$; therefore $(\forall u)[ux \sim_L uy]$. Let q be any state; because P is minimal, all of its states are reachable, so there exists a string u such that $q = q_{\text{start}}t_u$. Because $ux \sim_L uy$, we know $q_{\text{start}}t_{ux} = q_{\text{start}}t_{uy}$ by the preceding paragraph. Therefore $qt_x = qt_y$. Because that statement is true for all q, $t_x = t_y$.

4.7-14 Let $L = \{x : \#_{\mathtt{a}}(x) \equiv 0 \pmod{k}\}$.

(a) Prove that L is recognized by a DFR with k control states but not by any DFR with fewer than k control states.

(b) Prove that L is not recognized by any NFA with fewer than k control states.

Solution:

(a) L is recognized by a DFR with control set $\{0, \ldots, k-1\}$; start state 0; accepting state 0; rejecting states $1, \ldots, k-1$, and instruction set consisting of $(i \to (i+1) \bmod k, \text{SCAN}\mathtt{a})$ and $(i \to i, \text{SCAN}\mathtt{b})$ for $0 \leq i < k$.

If $i \neq j$, then states i and j are distinguished by the string $\mathtt{a}^{k-i}$, so that DFR is minimal.

(b) Suppose there is an NFA that accepts L and has fewer than k states. Consider an accepting computation on input $\mathtt{a}^k$; for $i = 0, \ldots, k-1$, let q_i be the state reached in that computation immediately after $\mathtt{a}^i$ has been scanned. Because P has fewer than k states, by the pigeon hole principle there exist $i \neq j$ such that $q_i = q_j$. But then P accepts $\mathtt{a}^i\mathtt{a}^{k-j}$, which is not in L.

4.7-15 Prove that for every $k \geq 3$ there is a language accepted by an NFA with k states but not recognized by any DFR with fewer than 2^k states. Hint: Consider an NFA N with control set $\{0, \ldots, k-1\}$, initial state 0, accepting state 0, and the following instructions:

$$\begin{array}{ll} (i \to (i+1) \bmod k, \text{SCAN}\mathtt{a}) & \text{for } 0 \leq i \leq k-1 \\ (i \to i, \text{SCAN}\mathtt{b}) & \text{for } 1 \leq i \leq k-1 \\ (0 \to 0, \text{SCAN}\mathtt{a}). & \end{array}$$

(Such an NFA, with $k = 6$, is shown in Figure 4.25.) Construct an equivalent DFR D via the subset construction but without eliminating unreachable states. Then D's control set consists of all subsets of N's control set. If S and S' are distinct control states of D prove that S and S' are not equivalent states. Prove that every subset of $\{0, \ldots, k-1\}$ is a reachable state in D. Therefore D is a minimal DFR, so every equivalent DFR requires at least 2^k states.

Solution: Construct D as in the hint.

For a set $S \subseteq \{0, \ldots, k-1\}$, let us write $S+1$ to denote $\{(x+1) \bmod k : x \in S\}$. For every S, we have $St_{\mathtt{b}} = S - \{0\}$ and

$$St_{\mathtt{a}} = \begin{cases} S+1 & \text{if } 0 \notin S \\ (S+1) \cup \{0\} & \text{if } 0 \in S \end{cases}$$

If S and T are distinct subsets of $\{0,\ldots,k-1\}$, let i be any element of $(S-T)\cup(T-S)$. Then S and T are distinguished by the input $\mathtt{a}^{k-i-1}\mathtt{ba}$. Thus all states in D are inequivalent.

The set $\{0,\ldots,k-1\}$ is reached on input $\mathtt{a}^{k-1}$, so an arbitrary set S is reached on input $\mathtt{a}^{k-1}\mathtt{b}^{\chi_S(k-1)}\mathtt{ab}^{\chi_S(k-2)}\mathtt{a}\cdots\mathtt{ab}^{\chi_S(0)}$.

4.7-16 Let R and S be regular languages recognized by DFRs with r control states and s control states, respectively. By using the pairing construction, we can produce DFRs with rs control states that recognize the languages $R\cap S$ and $R\cup S$. Prove that these constructions are optimal, i.e., for every r and s, there exist regular languages R and S recognized by DFRs with r control states and s control states, respectively, such that

(a) every DFR that recognizes $R\cap S$ has at least rs control states.

(b) every DFR that recognizes $R\cup S$ has at least rs control states.

Solution: Let

$$\begin{aligned} R &= \{x\in\{\mathtt{a},\mathtt{b}\}^* : \#_{\mathtt{a}}(x)\equiv 0 \pmod r\},\\ S &= \{x\in\{\mathtt{a},\mathtt{b}\}^* : \#_{\mathtt{b}}(x)\equiv 0 \pmod s\}. \end{aligned}$$

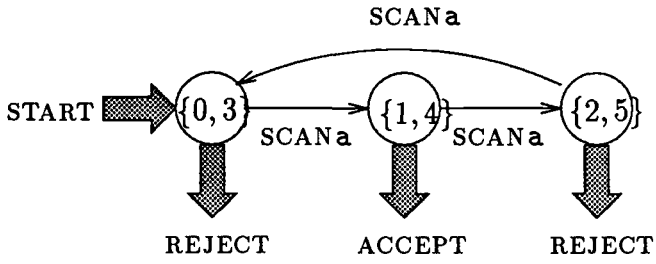
4.7-17 (b) Is Exercise 4.7-16(b) true for NFAs?

Solution: No. $r+s+1$ states suffice.

4.7-18 Construct a minimal DFR equivalent to the following:

(a) the DFR in Figure 4.32

Solution:



4.7-24 (a) Let P_1 and P_2 be two minimal DFRs that do not use the EOF test and recognize the same language. Prove that P_1 and P_2 are identical except for the renaming of control states.

Solution: All states in P_1 and P_2 must be reachable, for otherwise a smaller DFR would recognize L. Combine P_1 and P_2 as in the algorithm for determining whether two DFRs are equivalent, and construct E for the combined program. Because P_1 and P_2 recognize the same language,

their initial states must be equivalent. Call these states q_1 and q_2 respectively. Because P_1 and P_2 produce results for all inputs, the states q_1t_x and q_2t_x must exist for all strings x. For $i = 1, 2$ the instructions of P_i must be all instructions of the form $(q_it_x \rightarrow q_it_{xc}, \text{SCAN}c)$. Because q_1 and q_2 are equivalent, the two states q_1t_x and q_2t_x must be equivalent for each string x. In particular q_1t_x is an accepting (rejecting) state if and only if q_2t_x is an accepting (rejecting) state. Rename each state q_1t_x of P_1 to be q_2t_x, thus obtaining a recognizer with the same initial state, control set, instruction set, accepting states, and rejecting states as P_2. That is, this renaming produces the program P_2.

4.8-2 One can emulate the behavior of two FM programs by using two controls. These controls can be merged into one using the pairing construction described in Section 3.2.1. Use pairing constructions to prove the following theorems directly:

(a) The intersection of two DFR languages is a DFR language.

Solution: For $i = 1, 2$ let P_i be a standardized DFR with control set Q_i, start state s_i, accepting states A_i, and rejecting states $Q_i - A_i$. We construct a DFR P' that emulates P_1 and P_2 by remembering both of their control states in its own control and recognizes $L(P_1) \cap L(P_2)$. The control set of P' is $Q_1 \times Q_2$. Its start state is (s_1, s_2). Its accepting states are $A_1 \times A_2$, and its rejecting states are all others. For each instruction $(q_1 \rightarrow r_1, \text{SCAN}c)$ in P_1 and each instruction $(q_2 \rightarrow r_2, \text{SCAN}c)$ in P_2, let P' contain the instruction $((q_1, q_2) \rightarrow (r_1, r_2), \text{SCAN}c)$.

4.8-10 For strings x and y of equal length, define the *perfect shuffle* of x and y (denoted $x \spadesuit y$) to be the set of all strings formed by interleaving the characters of x and y one at a time. Formally,

$$x \spadesuit y = x_1y_1 \cdots x_my_m$$

where $x_1 \cdots x_m = x$, $y_1 \cdots y_m = y$, and $x_1, \ldots, x_m, y_1, \ldots, y_m$ denote single characters. We extend the perfect shuffle operation to languages in the standard way:

$$L \spadesuit R = \{x \spadesuit y : x \in L,\ y \in R, \text{ and } |x| = |y|\}.$$

If R is a regular language, construct a finite transduction τ such that $L\tau = L \spadesuit R$ for all languages L. Prove that the set of regular languages is closed under perfect shuffle.

Solution: Because the class of regular languages is closed under finite transductions, it suffices to construct a finite transducer that maps L to $L \spadesuit R$. Our program will copy the input string to the output while perfect-shuffling it with an arbitrary string in R. At each odd-numbered step the program

copies a character from its input to its output; at each even-numbered step it emulates P for one step without scanning anything, writing a character that P might scan. It uses an extra control to remember whether it is performing an odd-numbered or even-numbered step.

4.8-11 Let PREFIX(L) be the set of all prefixes of strings belonging to L. Prove that if L is regular then PREFIX(L) is regular. Your proof should be constructive.

Solution: PREFIX(L) = L/Σ^*, so this follows from the constructive proof that the class of regular languages is closed under quotient by a regular language.

Alternative solution: Start with an NFA that accepts L. Eliminate the EOF test. Then make each live state accepting. The resulting NFA accepts PREFIX(L).

4.8-13 A *permutation* of a set S is a one-one function from S to S. We define the PERM operation on strings and languages. If $x = x_1 \cdots x_n$ is a string of n characters, then PERM(x) = $\{x_{\sigma(1)} \cdots x_{\sigma(n)} : \sigma \text{ is a permutation of } \{1, \ldots, n\}\}$. That is, PERM($x$) is the set of all anagrams of the string x. If L is a language, then PERM(L) = $\bigcup_{x \in L}$ PERM(x). For example,

$$\text{PERM}(\{\mathtt{aba}, \mathtt{aa}\}) = \{\mathtt{aab}, \mathtt{aba}, \mathtt{baa}, \mathtt{aa}\}.$$

Is the class of regular languages closed under PERM?

Solution: No. PERM($(\mathtt{ab})^*$) is equal to the set of all strings over $\{\mathtt{a}, \mathtt{b}\}$ with equal numbers of a's and b's, which is not regular.

4.8-14 Recall from Exercise 4.8-11 that if L is regular then PREFIX(L) is regular. Prove that if PREFIX(L) is regular then L is not necessarily regular.

Solution: A counterexample is required. Let $L = \{\mathtt{a}^p : p \text{ is prime}\}$. Then PREFIX($L$) = $\mathtt{a}^*$, which is regular, but L is not regular.

4.9-4 (a) Let L be a regular language. Prove that there is a positive integer N (depending on L) such that if $z_1, \ldots, z_N$ are nonempty strings with the concatenation $z_1 \cdots z_N \in L$, then there exist i and j with $0 \le i < j \le N$ satisfying

$$z_1 \cdots z_i (z_{i+1} \cdots z_j)^* z_{j+1} \cdots z_N \subseteq L.$$

(b) Use part (a) to prove Theorem 4.47.

Solution:

(a) Since L is regular, there is a DFA P that accepts L. Let N be the number of control states of P. Let $q_k = q_{\text{start}} t_{z_1} \ldots t_{z_k}$, for $0 \le k \le N$. By the pigeonhole principle, two of these states must be the same. That is, we have $q_i = q_j$ for some i and j with $0 \le i < j \le N$. Let

$q = q_i = q_j$. Let $u = z_1 \cdots z_i$, $v = z_{i+1} \cdots z_j$, and $w = z_{j+1} \cdots z_{N+1}$. Then $q_{\text{start}} t_u = q$, $qt_v = q$, and $qt_w \omega_{\text{control}} = \text{ACCEPT}$. Since $qt_v = q$, we have $qt_{v^m} = q$ for every $m \geq 0$. Therefore, $q_{\text{start}} t_{uv^m w} = \text{ACCEPT}$ for every $m \geq 0$. In other words, $uv^m w \in L$ for every $m \geq 0$, so $uv^* w \subseteq L$.

(b) Let $z_1, \ldots, z_N$ be one-character strings representing each of the first N characters of z, and let z_{N+1} be the remaining suffix of z. Apply part (a), and let $u = z_1 \cdots z_i$, $v = z_{i+1} \cdots z_j$, and $w = z_{j+1} \cdots z_{N+1}$.

4.9-5 Use the first pumping theorem to prove that the following languages are not regular:

(a) $\{\mathtt{a}^n : n \text{ is composite}\}$

Solution: Let $z = \mathtt{a}^{p-N!}$ where p is a prime and $p \geq N! + N$.

4.9-6 Use closure properties and/or the pumping theorems to prove that the following languages are not regular:

(a) $\{w : w \text{ is the decimal representation of an integer squared}\}$. Hint: Intersect with $\mathtt{1(00)^*2(00)^*1}$.

Solution: Let $L = \{w : w \text{ is the decimal representation of an integer squared}\}$. Assume for the sake of contradiction that L is regular. Let $L' = L \cap \mathtt{1(00)^*2(00)^*1}$. Then L' is regular as well because the class of regular languages is closed under intersection. Let N be as in the first pumping theorem for regular languages, applied to L'. Let $z = \mathtt{10}^{2N}\mathtt{20}^{2N}\mathtt{1}$, which is the square of $\mathtt{10}^{2N}\mathtt{1}$, and hence in L'. By the first pumping theorem for regular languages there exist u, v, w such that $z = uvw$, $|uv| \leq N$, $v \neq \Lambda$, and $(\forall i \geq 0)[uv^i w \in L']$. Let u, v, w be such strings.

v cannot contain a 1 because then $uw \notin \mathtt{1(00)^*2(00)^*1}$, and therefore not in L'. Therefore $v = \mathtt{0}^k$ where $1 \leq k \leq N$. k must be even for otherwise $uw \notin \mathtt{1(00)^*2(00)^*1}$, and therefore not in L'. Let $k = 2j$.

Consider $uv^2w = \mathtt{10}^{2N+2j}\mathtt{20}^{2N}\mathtt{1}$. This number is between $\mathtt{10}^{4N+2j+2}$ and $\mathtt{10}^{2N+j}\mathtt{20}^{2N+j}\mathtt{1}$, which are two consecutive squares. Therefore, uv^2w is not a square, so it is not in L'. This contradiction proves that L is not regular.

4.9-8 (b) Let $L = \{x_1 y_1 \cdots x_n y_n : n \text{ is composite and } (\forall i \leq n)[(x_i \in \mathtt{a}^+) \text{ and } (y_i \in \mathtt{b}^+)]\}$. (Less formally, L is the set of all strings of the form $(\mathtt{a}^+\mathtt{b}^+)^n$ such that n is composite). Prove that L is not regular, but that strings in L can be pumped in the middle. That is, show that L has a pumping number N as in the second pumping theorem. Do not forget the case $i = 0$, which permits pumping down. Conclude that the converse of the second pumping theorem is not true for general L.

Solution: To see that L is not regular, perform a finite transduction that maps L to $\{\mathtt{0}^n : n \text{ is composite}\}$, which is not regular.

A pumping number for L is 14. Assume we are given $z = z_1z_2z_3$ where $|z_2| \geq 14$. If z_2 contains two consecutive a's, then we can pump one of those a's; if z_2 contains two consecutive b's, then we can pump one of those b's. Otherwise, z_2 contains as a substring $(\mathtt{ab})^6$, so we write $z_2 = s(\mathtt{ab})^6t$. In particular $n \geq 6$ in what follows.

Case 1: z is of the form $(\mathtt{a}^+\mathtt{b}^+)^n$ where n is odd. Write $z_2 = uvw$ where $u = s\mathtt{a}$, $v = \mathtt{b}$, and $w = (\mathtt{ab})^5t$. For $i \geq 1$, $z_1uv^iwz_3$ has the form $(\mathtt{a}^+\mathtt{b}^+)^n$, so $z_1uv^iwz_3$ belongs to L. z_1uwz_3 has the form $(\mathtt{a}^+\mathtt{b}^+)^{n-1}$. Because $n-1$ is even and at least 6, all strings of the second form belong to L as well. Thus, for all $i \geq 0$, $z_1uv^iwz_3 \in L$.

Case 2: z is of the form $(\mathtt{a}^+\mathtt{b}^+)^n$ where n is even. Write $z_2 = uvw$ where $u = s\mathtt{a}$, $v = \mathtt{baba}$, and $w = \mathtt{bababab}t$. For all $i \geq 0$, $z_1uv^iwz_3$ has the form $(\mathtt{a}^+\mathtt{b}^+)^{n-2+2i}$. Since $n-2+2i$ is even and at least 4, all strings of that form belong to L.

Note: It is not hard to show that 9 is in fact a pumping number for L.

4.9-9 Call a language an n-state language if it is recognized by an n-state DFR. Find functions f and g such that the following are true:

(a) An n-state language is empty if and only if it contains no strings of length $\leq f(n)$. Make f as small as possible.

(b) An n-state language is infinite if and only if it contains a string of length $\geq g(n)$. Make g as small as possible.

Solution:

(a) If L is nonempty, let k be the length of the shortest string in L. If $k \geq n$, then apply the pumping theorem and pump down to find a shorter string in L. Therefore $k \leq n-1$. Thus if L is nonempty, then L contains a string of length $\leq n-1$, so we can take $f(n) = n-1$. This is the best possible, because there is an n-state DFR that recognizes $\{\mathtt{a}^{n-1}\}$, which is nonempty.

Conversely, if L is empty, then L contains no strings of any length.

(b) If L contains a string of length $\geq n$, then apply the pumping theorem to pump that string up an arbitrary number of times, showing that L is infinite. Thus we can take $g(n) = n$. This is the best possible because there is an n-state DFR that recognizes $\{\mathtt{a}^{n-1}\}$, which is finite.

Conversely, if L is infinite, then L contains arbitrarily long strings.

4.10-2 Let p be a fixed string of length n. Let P be a minimal DFR that recognizes $\{x : p \text{ is a substring of } x\}$. How many states does P have?

Solution: Let partialmatch(x) be the longest suffix of x that is a prefix of p. Let x and y be two strings that do not contain p as a substring. We assert

that

$$x \sim_L y \iff \text{partialmatch}(x) = \text{partialmatch}(y).$$

Proof of assertion: First, assume that partialmatch(x) = partialmatch(y) = m, $x = x'm$, and $y = y'm$. If the pattern p is a substring of $x'mz$, then p cannot overlap x'; otherwise p would be a substring of $x'm$ or m would not be the longest suffix of $x'm$ that is a prefix of p. Similarly, if the pattern p is a substring of $y'mz$, then p cannot overlap y'. Then p is a substring of xz iff p is a substring of mz if p is a substring of yz. Therefore $x \sim_L y$.

Conversely, suppose that partialmatch$(x) = m$ and partialmatch$(y) = m'$ where $m \neq m'$. Without loss of generality assume that $|m| > |m'|$. Let $ms = p$. Then $xs = x'ms \in L$ but $ys = y'm's \notin L$, so $x \not\sim_L y$. ∎

In addition, if p is a substring of both x and y, then xz and yz belong to L for all z, so $x \sim_L y$. If p is a substring of exactly one of x and y, then $x\Lambda \in L$ if and only if $y\Lambda \notin L$, so $x \not\sim_L y$.

Therefore $x \sim_L y$ iff (1) partialmatch(x) = partialmatch(y) and p is not a substring of x or y or (2) p is a substring of both x and y. Thus the prefix equivalence classes of L are $\{x :$ partialmatch$(x) = m$ and x does not contain p as a substring$\}$, where m is any prefix of p other than p itself; and $\{x : p$ is a substring of $x\}$.

Therefore, the relation $\sim_L$ has exactly $n + 1$ equivalence classes, so L's minimal DFR has exactly $n + 1$ control states.

4.10-3 (c) We define MIDDLE-THIRD(x) as follows: If x is a string whose length is a multiple of 3, let $x = uvw$ where $|u| = |v| = |w|$, and let MIDDLE-THIRD$(x) = v$; if the length of x is not a multiple of 3 then MIDDLE-THIRD(x) is undefined. Extend MIDDLE-THIRD() to languages in the ordinary way. Prove that if R is regular, then MIDDLE-THIRD(R) is regular.

Solution: Let R be a regular language and let D be a standardized DFR (so that each instruction scans a character) that recognizes R. We construct an NFA P that tests whether $v \in$ MIDDLE-THIRD(R) as follows:

Step 1: Nondeterministically guess two states q_u and q_{uv} of the program D.

Step 2: Perform (A) through (C) simultaneously:

(A) emulate D started in state q_{start}, with input a nondeterministically generated string of length $|v|$, accepting iff D finishes in state q_u;

(B) emulate D started in state q_u, with input v, accepting iff D finishes in state q_{uv};

(C) emulate D started in state q_{uv}, with input a nondeterministically generated string of length $|v|$, accepting iff D finishes in an accepting state.

Step 3: Accept iff (A) through (C) all accept.

Note that (B) is accomplished simply by emulating D with the actual input, but with different start and accepting states. To accomplish (A), allow D to take one step for each step of (B), but without scanning anything. Similarly for (C).

Formally, P is constructed as follows. Let Q, q_{start}, Q_{accept} respectively be D's control set, start state, and set of accepting states. P's control set will be $Q^5 \cup \{0\}$. P's start state will be 0. P's set of accepting states will be $\{(q_u, q_u, q_{uv}, q_{uv}, q) : q_u \in Q, q_{uv} \in Q, q \in Q_{\text{accept}}\}$. P will contain the instructions $(0 \rightarrow (q_{\text{start}}, q_u, q_u, q_{uv}, q_{uv}), \text{NOOP})$ for each $q_u \in Q$ and $q_{uv} \in Q$. In addition for each triple of instructions $(p \rightarrow p', \text{SCAN}c)$ $(r \rightarrow r', \text{SCAN}d)$ $(t \rightarrow t', \text{SCAN}e)$ in D, P will contain the instruction $((p, q, r, s, t) \rightarrow (p', q, r', s, t'), \text{SCAN}d)$.

4.10-5 Let L be any language and let $A = \text{SUBSEQ}(L)$. In this exercise you will prove that A is regular.

(a) Let us write $x \leq y$ if x is a subsequence of y. We say that a string x is a minimal element of a set S if (1) $x \in S$ and (2) for all $y \in S$, if $y \leq x$ then $y = x$. Let M be the set of all minimal elements of $\overline{A}$. Prove that $y \in \overline{A}$ if and only if there exists $x \in M$ such that x is a subsequence of y.

(b) If x is any string, prove that $\{y : x \leq y\}$ is regular.

(c) Call two strings x and y *incomparable* if $x \not\leq y$ and $y \not\leq x$. Prove that if x and y are distinct strings in M, then x and y are incomparable.

(d) Let S be a language over $\{\mathtt{a}, \mathtt{b}\}$. Prove that if all strings in S are incomparable, then S is finite. Hint: Let x be any string and let $\ell = |x|$. Prove that $x \leq (\mathtt{ba})^\ell$. Next, prove that if x and y are incomparable, then $(\mathtt{ba})^\ell \not\leq y$. Let $L_{2k} = (\mathtt{a}^*\mathtt{b}^*)^k$ and $L_{2k+1} = L_{2k}\mathtt{a}^*$. Prove that if x and y are incomparable, then $y \in L_{2\ell}$. Finally, prove, by induction on k, that if $z \in L_k$, $S \subseteq L_k$, and all strings in S are incomparable, then S contains only finitely many strings y such that $z \not\leq y$.

(e) Let S be any language. Prove that if all strings in S are incomparable, then S is finite.

(f) Prove that M is finite.

(g) Prove that $\overline{A}$ is regular.

(h) Prove that A is regular.

Solution: (Thanks to Nick Reingold.)

(a) Assume that $y \in \overline{A}$ and let x be a shortest subsequence of y that is in $\overline{A}$; then $x \in M$.

Conversely, assume that there exists $x \in M$ such that x is a subsequence of y. For the sake of contradiction, assume that $y \in A$. Then y is a subsequence of some element of L. But then x is also a subsequence of some element of L, contradicting the fact that $x \in \overline{A}$. Therefore $y \in \overline{A}$.

(b) Fix x. $\{y : x \leq y\}$ is accepted by a DFA with control set PREFIX(x), start state Λ, accepting state x, and instructions $(p \rightarrow pc, \text{SCAN}c)$ for all p and c such that $pc \in \text{PREFIX}(x)$ and $(p \rightarrow p, \text{SCAN}c)$ for all p and c such that $p \in \text{PREFIX}(x)$ and $pc \notin \text{PREFIX}(x)$.

(c) Let x and y be distinct strings in M. If $x \leq y$ then y is not minimal; if $y \leq x$ then x is not minimal. Thus $x \not\leq y$ and $y \not\leq x$.

(d) Let x be any string in S, and let $\ell = |x|$. Because the ith character of x is a subsequence of ba for each i, the string x is a subsequence of $(\mathtt{ba})^\ell$. If $x \not\leq y$, then $(\mathtt{ba})^\ell \not\leq y$, so $y \in (\mathtt{a}^*\mathtt{b}^*)^\ell$.

Let $T = S - \{x\}$, so $T \subseteq (\mathtt{a}^*\mathtt{b}^*)^\ell$. A *chain* is an infinite sequence $\langle\langle s_1, s_2, \ldots \rangle\rangle$ such that $s_1 \leq s_2 \leq s_3 \leq \cdots$. A *subsequence* of a sequence is obtained by deleting some elements from the sequence. We will prove that every infinite sequence of strings in $(\mathtt{a}^*\mathtt{b}^*)^\ell$ contains an infinite chain as a subsequence. In particular, if T is infinite, then T contains a pair of comparable strings.

Consider an infinite sequence U of strings of the form $\mathtt{a}^{e_1}\mathtt{b}^{e_2} \cdots \mathtt{a}^{e_{2\ell-1}}\mathtt{b}^{e_{2\ell}}$. Fix an i. Then U contains an infinite subsequence in which the e_i's are in nondecreasing order. Why? Choose the first string in which e_i is minimum. Then choose the next string in which e_i is at least as large, and the next string in which e_i is at least as large as that, and so on.

Let $U[i]$ denote the sequence obtained from U as above, and let $V = U[1][2]\cdots[2\ell]$. Then V is an infinite sequence in which all the e_i's are in nondecreasing order, so V is a chain.

This completes the proof that if S is infinite then S contains a pair of comparable strings.

(e) Call a set S of strings *good* if every infinite sequence of strings in S contains a chain as a subsequence. It suffices to prove that Σ^* is good for every alphabet Σ. We prove that by induction on the size of Σ. Base cases: The assertion is trivial if $|\Sigma| = 0$ and obvious if $|\Sigma| = 1$.

Assume that Δ^* is good whenever $|\Delta| = k$. Let $|\Sigma| = k + 1$; we will prove that Σ^* is good.

Lemma 1: If A and B are good then $A \cup B$ is good. **Proof:** Consider any infinite sequence of elements of $A \cup B$. It must contain an infinite subsequence consisting entirely of elements of A or an infinite subsequence consisting entirely of elements of B. In either case that infinite subsequence contains a chain. ∎

Lemma 2: If A and B are good then AB is good. **Proof:** Let $\langle\langle a_1b_1, a_2b_2, \ldots\rangle\rangle$ be an infinite sequence such that each $a_i \in A$ and each $b_i \in B$. Since A is good $\langle\langle a_1, a_2, \ldots\rangle\rangle$ contains an infinite chain $\langle\langle a_{i_1}, a_{i_2}, \ldots\rangle\rangle$. Since B is good, $\langle\langle b_{i_1}, b_{i_2}, \ldots\rangle\rangle$ contains an infinite chain $\langle\langle b_{i_{j_1}}, b_{i_{j_2}}, \ldots\rangle\rangle$. Thus the original sequence contains the chain $\langle\langle a_{i_{j_1}}b_{i_{j_1}}, a_{i_{j_2}}b_{i_{j_2}}, \ldots\rangle\rangle$. ∎

For each $c \in \Sigma$, let $D_c = (\Sigma - \{c\})^*$. By the inductive hypothesis D_c is good. Let $D = \cup_{c\in\Sigma} D_c$. By repeated application of Lemma 1, D is good. By repeated application of Lemma 2, D^n is good for every n.

Lemma 3: Let x and y be strings such that $|x| = n$ and $y \notin D^{2n-1}$. Then $x \leq y$. **Proof:** Let $x = c_1c_2\cdots c_n$. Assume $x \not\leq y$. Choose the least i such that $c_1\cdots c_i \not\leq y$. Then

$$\begin{aligned} y &\in (\Sigma - c_1)^*c_1(\Sigma - c_2)^*c_2\cdots(\Sigma - c_i)^* \\ &\subseteq D^{2i-1} \\ &\subseteq D^{2n-1}. \end{aligned}$$ ∎

Let S be a sequence of strings in Σ^*. We are now prepared to show that S contains a chain. We consider two cases.

Case 1: There exists m such that every string in S belongs to D^m. Then, since D^m is good, S contains a chain.

Case 2: For every m, there exists a string in S that does not belong to D^m. We find a chain in S as follows:

Step 1: Let $i = 1$. Let x_1 be the first element in the sequence S.

Step 2: Let $m = 2|x_i| - 1$. Let x_{i+1} be the first string after x_i in the sequence S that does not belong to D^m. Let $i = i + 1$.

Step 3: Go to Step 2.

By Lemma 3, we have $x_i \leq x_{i+1}$ for each i, so $\langle\langle x_1, x_2, \ldots\rangle\rangle$ is a chain in S.

(f) Because M consists of incomparable strings, M is finite by (e).

(g) By (a), (b), and (f), $\overline{A}$ is the union of finitely many regular languages. Because the class of regular languages is closed under union, $\overline{A}$ is regular.

(h) Because the class of regular languages is closed under complementation, A is regular.

4.10-6 Define

$$\text{HALF-SUBSEQ}(x) = \{y : y \text{ is a subsequence of } x \text{ and } |y| = \tfrac{1}{2}|x|\},$$

i.e., y is obtained by deleting exactly half of the characters from x, and

$$\text{HALF-SUBSEQ}(L) = \bigcup_{x\in L} \text{HALF-SUBSEQ}(x).$$

Is the class of the regular languages closed under HALF-SUBSEQ()?

Solution: No. Let $L = (\mathtt{abbb})^*$. Let $L' = \text{HALF-SUBSEQ}(L)$. Because every string in L contains 3 times as many b's as a's, every string in L' contains at least as many b's as a's. Suppose that L' is regular. Then L' has a pumping number n as in the first pumping theorem.

Let $x = (\mathtt{abbb})^{2n}$. By deleting the first $3n$ b's and the last n a's from x, we obtain the string $y = \mathtt{a}^n\mathtt{b}^{3n}$, which is in L'. We will show that L' contains a string of the form $\mathtt{a}^m\mathtt{b}^{3n}$ where $m > 3n$, a contradiction.

By the first pumping theorem, there exist u, v, w such that $y = uvw$, $|uv| \leq n$, $|v| \geq 1$, and for all $i \geq 0$, $uv^iw \in L'$. Since $|uv| \leq n$, $v = \mathtt{a}^k$ for some k. Thus $uv^iw = \mathtt{a}^{n+(i-1)k}\mathtt{b}^{3n}$. Since $|v| \geq 1$, $k \geq 1$. Let $i = 2n + 2$, so $uv^iw = \mathtt{a}^{n+(2n+1)k}\mathtt{b}^n$. But $n + (2n + 1)k \geq n + 2n + 1 = 3n + 1 > 3n$.

4.10-7 Prove that the following languages are regular:

(a) the set of all strings representing decimal numbers that neither contain the digit 7 nor are divisible by the number 7

Solution: We construct a DFA that accepts the language. Its control set is $\{0, 1, 2, 3, 4, 5, 6\}$. Its start state is 0, and all states except 0 are accepting. Its instruction set consists of all instructions of the form $(i \rightarrow (10i + j) \bmod 7, \text{SCAN}j)$ where $0 \leq i \leq 6$, $0 \leq j \leq 9$, and $j \neq 7$.

4.10-8 If x is a string, then a *rotation* of x is a string zy such that $yz = x$. Let rotation(x) be the set consisting of all rotations of x and let rotation(L) be the set consisting of all rotations of strings in L, i.e.,

$$\text{rotation}(L) = \{zy : (\exists x \in L)[yz = x]\}.$$

For example,

$$\text{rotation}(\{\mathtt{abc}, \mathtt{abab}\}) = \{\mathtt{abc}, \mathtt{bca}, \mathtt{cab}, \mathtt{abab}, \mathtt{baba}\}.$$

Is the class of regular languages closed under rotation()?

Solution: Yes. Assume that L is accepted by an NFA P. The language rotation(L) is accepted by a nondeterministic program P' that guesses what state q the program P would be in after scanning y. Then P' simulates the behavior of P starting from state q while z is read. P' guesses when z is finished and checks that P is in an accepting state. Finally P' simulates the behavior of P while y is read and checks that P finishes in state q, as guessed. The program P' just described uses one control to store q and one control to simulate P; these can be merged into a single control.

5.1-1 In English, quotations are surrounded by double quotation marks, as in

I say "hello."

Nested quotations are surrounded by single quotation marks, as in

The Beatles sang, "You say 'goodbye,' and I say 'hello.' "

When quotations are nested more deeply than that, single and double quotation marks alternate, as in

My textbook says, "The Beatles sang, 'You say "goodbye," and I say "hello." ' "

In parts (a–c) below, assume that opening and closing quotation marks are represented by distinct characters, as in the examples above, so there are four distinct characters in all. For simplicity, assume that apostrophes are represented by yet another character.

(a) Let L be the set of all sequences of single quotation and double quotation marks in correctly punctuated English sentences. Prove that L is a DCA language.

(b) Prove that L is not a regular language.

(c) Prove that the set of English sentences is not a regular language in this representation.

In parts (d–f), assume that opening and closing quotation marks are represented by the same character, as on a typewriter, e.g.,

```
The Beatles sang, "You say 'goodbye,' and I say 'hello.'"
```

Still assume that apostrophes are represented by yet another character.

(d) Let L' be the set of all sequences of single and double quotation marks in correctly punctuated English sentences, where opening and closing quotation marks are represented by the same character. Prove that L' is a DCA language.

(e) Prove that L' is not a regular language.

(f) Prove that the set of English sentences is not a regular language in this representation.

Solution:

(a) Let B be the language of balanced parentheses. Recall that B is a DCA language by Example 1.9. Call the input $x_1 \cdots x_n$ and say that x_i is odd-numbered if i is odd, and even-numbered if i is even. In a string belonging to L, " and ' must occur at odd-numbered positions and " and ‘ must occur at even-numbered positions. Construct a deterministic finite transducer that replaces " and ‘ by (and " and ' by) if they occur at positions with the proper parity (the control remembers i mod 2); it blocks otherwise. Call the transduction computed by that finite transducer τ. Then $B\tau^{-1} = L$, so L is a DCA language by Theorem 4.38(ii).

(b) Construct a finite transducer τ that replaces " and ‘ by a while in state 1, nondeterministically moves to state 2, and then replaces " and ' by b until the input is exhausted; call its transfer relation τ. Then $L\tau = \{\mathtt{a}^n\mathtt{b}^n : n \geq 0\}$, which is not regular. Since the class of regular languages is closed under finite transductions, L must not be regular.

(c) Compose the set of English sentences with a finite transduction that copies single and double quotes but deletes all other characters. If the set of English sentences were regular, then L would be regular, which is not true by part (b).

(d) This is just like part (a) except that the DCA uses its control to determine whether to interpret each input character as opening or closing, rather than to check the correctness of each input character.

(e) Let τ be a finite transduction that replaces "' by a, replaces '" by b, and blocks if it scans "" or ''. Then $L'\tau = \{\mathtt{a}^n\mathtt{b}^n : n \geq 0\}$, which is not regular. Therefore L' is not regular.

(f) Compose the set of English sentences with a finite transduction that copies single and double quotes but deletes all other characters. If the set of English sentences were regular, then L' would be regular, which is not true by part (d).

5.1-2 Pascal does not permit a semicolon before the word "else" in if-then-else statements. Modify the defining equations in Figure 5.2in order to permit an optional semicolon before the word "else" in if-then-else statements.

Solution:

$$\begin{aligned}
\langle \textit{Statement} \rangle &= \Lambda \cup \langle \textit{Simple-Statement} \rangle \cup \langle \textit{Compound-Statement} \rangle \cup \langle \textit{If-Statement} \rangle \\
&\quad \cup \langle \textit{While-Statement} \rangle \cup \langle \textit{Repeat-Statement} \rangle \cup \langle \textit{For-Statement} \rangle \\
\langle \textit{Simple-Statement} \rangle &= \langle \textit{Procedure-Call} \rangle \cup \langle \textit{Assignment-Statement} \rangle \\
\langle \textit{Compound-Statement} \rangle &= \texttt{begin}\ \langle \textit{Statement-List} \rangle\ \texttt{end} \\
\langle \textit{Statement-List} \rangle &= \langle \textit{Statement} \rangle\ (\Lambda \cup ; \langle \textit{Statement-List} \rangle) \\
\langle \textit{If-Statement} \rangle &= \texttt{if}\ \langle \textit{Condition} \rangle\ \texttt{then}\ \langle \textit{Statement} \rangle
\end{aligned}$$

$$\begin{aligned} & (\Lambda \cup \texttt{else}\ \langle \textit{Statement} \rangle \cup \texttt{;else}\ \langle \textit{Statement} \rangle) \\ \langle \textit{While-Statement} \rangle &= \texttt{while}\ \langle \textit{Condition} \rangle\ \texttt{do}\ \langle \textit{Statement} \rangle \\ \langle \textit{Repeat-Statement} \rangle &= \texttt{repeat}\ \langle \textit{Statement-List} \rangle\ \texttt{until}\ \langle \textit{Condition} \rangle \\ \langle \textit{For-Statement} \rangle &= \texttt{for}\ \langle \textit{Assignment-Statement} \rangle\ \texttt{to}\ \langle \textit{Expression} \rangle\ \texttt{do}\ \langle \textit{Statemen} \end{aligned}$$

5.2-10 Recall that a partial order is a relation satisfying Proposition 5.2(i, ii). A *complete lower semilattice* $(S, \prec, \bigwedge)$ is a set S together with a partial order $\prec$ and a meet operation $\bigwedge$ satisfying Proposition 5.3. A partial order is called *complete* if it satisfies Proposition 5.2(iii) as well. A complete lower semilattice with a complete partial order is called a *TK-semilattice.*

(b) Let $\succ$ denote the converse of $\prec$; i.e., $x \succ y$ iff $y \prec x$. We say that $(S, \prec, \bigwedge, \bigvee)$ is a *complete lattice* if $(S, \prec, \bigwedge)$ and $(S, \succ, \bigvee)$ are complete lower semilattices. (The $\bigvee$ operation is called *join* or *least upper bound.*) If $(S, \prec, \bigwedge, \bigvee)$ is a complete lattice, prove that $(S, \prec, \bigwedge)$ is a TK-semilattice.

Solution: Let $\mathcal{U} = \bigvee S$. The other TK-semilattice properties follow directly from the corresponding properties of complete lattices.

(d) We say that a function on the reals is *monotone* if $x \leq y \Rightarrow f(x) \leq f(y)$. Let f be a monotone function on $[0, 1]$. Prove that f has a fixed point.

Solution: $([a, b], \leq, \min)$ is a TK-semilattice.

(f) Construct a monotone function f on $(0, 1]$ that does not have a fixed point. Which property of TK-semilattices is violated?

Solution: Let $f(x) = \frac{1}{2}x$. Completeness is violated because $\bigwedge (0, 1] = 0 \notin (0, 1]$.

5.3-1 Construct context-free grammars for the languages in Exercise 5.2-1.

(c) $\{\mathtt{a}^i\mathtt{b}^j : j = i \text{ or } j = 2i\}$
Solution:

$$\begin{aligned} S &\rightarrow U \\ S &\rightarrow V \\ U &\rightarrow \Lambda \\ U &\rightarrow \mathtt{a}U\mathtt{b} \\ V &\rightarrow \Lambda \\ V &\rightarrow \mathtt{a}V\mathtt{bb} \end{aligned}$$

(e) $\{\mathtt{a}^i\mathtt{b}^j : j < i \text{ or } j > 2i\}$

Solution:

$$\begin{aligned} S &\rightarrow U \\ S &\rightarrow V \\ U &\rightarrow \mathtt{a} \\ U &\rightarrow \mathtt{a}U \\ U &\rightarrow \mathtt{a}U\mathtt{b} \\ V &\rightarrow \mathtt{b} \\ V &\rightarrow V\mathtt{b} \\ V &\rightarrow \mathtt{a}V\mathtt{bb} \end{aligned}$$

(f) $\{\mathtt{a}^i\mathtt{b}^j : i \leq j \leq 2i\}$

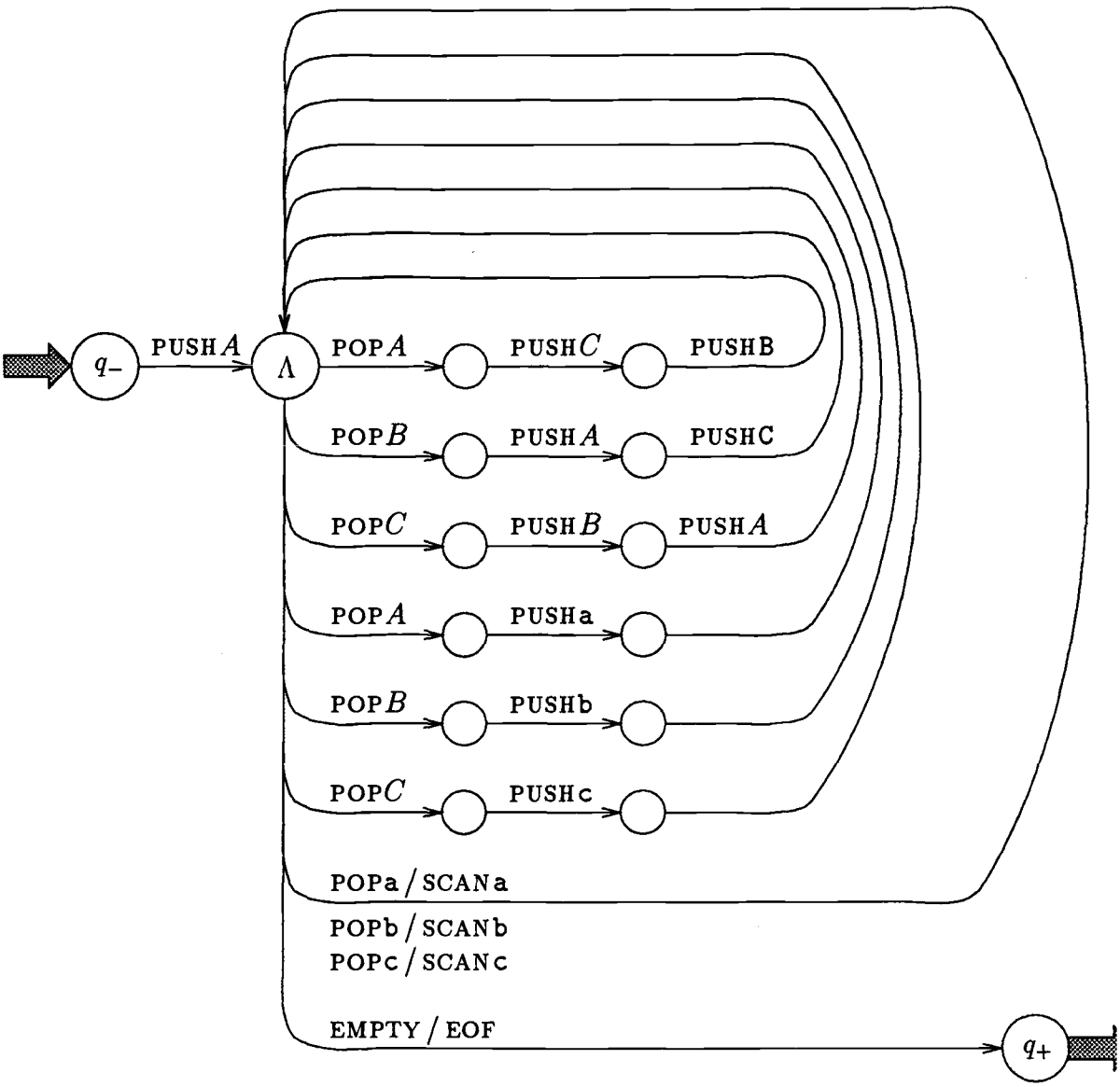
Solution:

$$\begin{aligned} S &\rightarrow \Lambda \\ S &\rightarrow \mathtt{a}S\mathtt{b} \\ S &\rightarrow \mathtt{a}S\mathtt{bb} \end{aligned}$$

5.6-1 Construct an NSA that is equivalent to the following grammar:

$$\begin{aligned} A &\rightarrow BC \\ B &\rightarrow CA \\ C &\rightarrow AB \\ A &\rightarrow \mathtt{a} \\ B &\rightarrow \mathtt{b} \\ C &\rightarrow \mathtt{c}. \end{aligned}$$

Solution:



5.8-1 (a) Prove that the first pumping theorem for CFLs remains true if we require that v be nonempty instead of just requiring that v or x be nonempty.

Solution: Apply the first pumping theorem to obtain $z = uvwxy$. If v is nonempty, then we are done; if v is empty, then $z = uwxy$ and x is nonempty, so we can let $u' = uw$, $v' = x$, $w' = x' = \Lambda$, and $y' = y$.

(b) Does Ogden's pumping theorem for CFLs remain true if we require that v contain a marked character instead of just requiring that v or x contain one?

Solution: No. Let $L = \{\mathtt{a}^n\mathtt{b}^n : n \geq 0\}$ and mark all the b's for pumping. When Ogden's theorem is applied, v must consist of a positive number of a's and x must consist of the same number of b's. In particular, v contains no b's.

5.8-3 Prove that the following are not CFLs:

(a) $\{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k : i < j < k\}$

(c) $\{\mathtt{a}^{i^2} : i \geq 0\}$

(g) $\{w : w$ is the decimal representation of a prime number$\}$

(h) $\{w : w$ is the decimal representation of a composite number$\}$

Solution:

(a) Call the language above L and suppose, for the sake of contradiction, that L is a CFL. Let N be the number from Corollary 5.28, and let $z = \mathtt{a}^N\mathtt{b}^{N+1}\mathtt{c}^{N+2}$. Then $z = z_1z_2z_3$ where $z_1 = \Lambda$, $z_2 = \mathtt{a}^N$, and $z_3 = \mathtt{b}^{N+1}\mathtt{c}^{N+2}$. By Corollary 5.28 there exist strings u, v, w, x, y such that $z = uvwxy$, v or x is a nonempty string of a's, and $uv^iwx^iy \in L$ for all $i \geq 0$. If vx contains an a, a b, and a c, then $uv^wwx^wy \notin \mathtt{a}^*\mathtt{b}^*\mathtt{c}^*$, a contradiction. If vx contains no bs, then uv^2wx^2y contains at least as many as as bs, a contradiction. If vx contains no cs, then uv^3wx^3y contains at least as many as as cs, a contradiction. In every possible case, we have obtained a contradiction, so L is not a CFL.

(c) Call the language above L and suppose, for the sake of contradiction, that L is a CFL. Let N be the number from Theorem 5.21 (applied to L), and let $z = \mathtt{a}^{N^2}$. By Theorem 5.21 there exist strings u, v, w, x, y such that $z = uvwxy$, v or x is nonempty, $|uvx| \leq N$ and $uv^iwx^iy \in L$ for all $i \geq 0$. Thus $vx = \mathtt{a}^k$ where $1 \leq k \leq N$. Then $uv^2wx^2y = \mathtt{a}^{N^2+k}$. But $N^2 < N^2 + k \leq N^2 + N < (N+1)^2$, so $N^2 + k$ is not a square and $uv^2wx^2y \notin L$, a contradiction. Therefore L is not a CFL.

(g) We will prove this for primes written in b-ary for any $b \geq 2$. The exercise is the special case $b = 10$. Let $L = \{w :$ w is the b-ary representation of a prime number$\}$ and suppose, for the sake of contradiction, that L is a CFL. We will treat strings as equal to the b-ary number they represent.

Let N be the number from Theorem 5.21 (applied to L). Let z be the b-ary representation, without leading zeroes, of a prime p greater than b such that z has at least N digits (this is possible because there are infinitely many primes). By Theorem 5.21 there exist u, v, w, x, y such that $uvwxy = z$, v or x is nonempty, and, for all $i \geq 0$, $uv^iwx^iy \in L$.

We will show that $uv^{p!+1}wx^{p!+1}y$ is an integer multiple of p. Since $uv^{p!+1}wx^{p!+1}y > p$, $uv^{p!+1}wx^{p!+1}y$ is not prime. This contradiction (taking $i = p!$) proves that L is not a CFL.

Now we prove that $uv^{p!+1}wx^{p!+1}y$ is a multiple of p. As in the solution to Exercise 4.9-6(b), let $f_y(z) = zy \bmod p$ and let $f^{(m)}$ denote the m-fold composition of f with itself. In that solution, we showed for all u and v that $f_v^{(p!+1)}(uv) \equiv uv \pmod p$ and for all z_1, z_2, w that $z_1 \equiv z_2 \pmod p \Rightarrow z_1 w \equiv z_2 w \pmod p$. Therefore

$$\begin{aligned} uv^{p!+1} &\equiv uv \bmod p, \\ uv^{p!+1}w &\equiv uvw \bmod p, \\ uv^{p!+1}wx^{p!+1} &\equiv uvwx \bmod p, \\ uv^{p!+1}wx^{p!+1}y &\equiv uvwxy \bmod p. \end{aligned}$$

Since $uvwxy = z = p$, $uv^{p!+1}wx^{p!+1}y$ is divisible by p, as promised.

(h) This is an open question as far as I know. An affirmative answer would imply that primality is in P, that there are finitely man primes of the form 1^*, and that there are finitely many primes of the form 10^*1. (If the set of composite numbers in binary is context-free then there are finitely many Mersenne primes and finitely many Fermat primes.)

5.8-4 Let L be the set of syntactically correct Pascal programs (if you do not know Pascal, consider some other block-structured programming language in which variables must be declared before they are used). Prove that L is not a CFL.

Solution: Let τ be a finite transduction that maps strings of the form

```
program whynot; var x: integer; begin for y := 1 to 2 do end.
```

where x and y are strings over $\{\mathtt{a}, \mathtt{b}\}$, to xy and rejects all other strings. Then $L\tau = \{xx : x \in \{\mathtt{a}, \mathtt{b}\}^*\}$, which is not a CFL, so L is not a CFL.

5.9-4 Prove that $\{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k : i = j \text{ or } i = k\}$ is inherently ambiguous.

Solution: Let G be a grammar that generates $\{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k : i = j \text{ or } i = k\}$, and let N be the corresponding constant in Ogden's lemma. Apply Ogden's lemma to $\mathtt{a}^N\mathtt{b}^N\mathtt{c}^{N+N!}$, marking all the a's. As in the proof of Theorem 5.31, there is a variable A satisfying the three conditions,

$$\begin{aligned} S &\stackrel{*}{\Rightarrow} uAy \\ A &\stackrel{*}{\Rightarrow} vAx \\ A &\stackrel{*}{\Rightarrow} w, \end{aligned}$$

where $uvwxy = \mathtt{a}^N\mathtt{b}^N\mathtt{c}^{N+N!}$, $v = \mathtt{a}^k$, $x = \mathtt{b}^k$, and $1 \le k \le N$. Thus there is a derivation:

$$\begin{aligned} S &\stackrel{*}{\Rightarrow} uAy \\ &\stackrel{*}{\Rightarrow} uv^{N!/k+1}Ax^{N!/k+1}y \\ &\stackrel{*}{\Rightarrow} uv^{N!/k+1}wx^{N!/k+1}y \\ &= u\mathtt{a}^{N!+k}w\mathtt{b}^{N!+k}y \\ &= \mathtt{a}^{N!+N}\mathtt{b}^{N!+N}\mathtt{c}^{N!+N}. \end{aligned}$$

$\mathtt{a}^{N!+k}w\mathtt{b}^{N!+k}$ is a phrase in that derivation.

Next apply Ogden's lemma to $\mathtt{a}^N\mathtt{b}^{N+N!}\mathtt{c}^N$, marking all the a's. Then there is a variable $\hat{A}$ satisfying the three conditions,

$$\begin{aligned} S &\stackrel{*}{\Rightarrow} \hat{u}\hat{A}\hat{y} \\ A &\stackrel{*}{\Rightarrow} \hat{v}\hat{A}\hat{x} \\ A &\stackrel{*}{\Rightarrow} \hat{w}, \end{aligned}$$

where $\hat{u}\hat{v}\hat{w}\hat{x}\hat{y} = \mathtt{a}^N\mathtt{b}^{N+N!}\mathtt{c}^N$, $\hat{v} = \mathtt{a}^{\hat{k}}$, $\hat{x} = \mathtt{c}^{\hat{k}}$, and $1 \le \hat{k} \le N$. Thus there is a derivation

$$\begin{aligned} S &\stackrel{*}{\Rightarrow} \hat{u}\hat{A}\hat{y} \\ &\stackrel{*}{\Rightarrow} \hat{u}\hat{v}^{N!/\hat{k}}\hat{A}\hat{x}^{N!/\hat{k}}\hat{y} \\ &\stackrel{*}{\Rightarrow} \hat{u}\hat{v}^{N!/\hat{k}}\hat{v}\hat{A}\hat{x}\hat{x}^{N!/\hat{k}}\hat{y} \\ &\stackrel{*}{\Rightarrow} \hat{u}\hat{v}^{N!/\hat{k}}\hat{v}\hat{w}\hat{x}\hat{x}^{N!/\hat{k}}\hat{y} \\ &= \hat{u}\mathtt{a}^{N!+\hat{k}}\hat{w}\mathtt{c}^{N!+\hat{k}}\hat{y} \\ &= \mathtt{a}^{N!+N}\mathtt{b}^{N!+N}\mathtt{c}^{N!+N}, \end{aligned}$$

in which $\hat{v}\hat{w}\hat{x} = \mathtt{a}^{\hat{k}}\hat{w}\mathtt{c}^{\hat{k}}$ is a phrase.

The phrase $\mathtt{a}^{N!+k}w\mathtt{b}^{N!+k}$ contains at least $N!+1$ a's and no c's, but the phrase $\mathtt{a}^{\hat{k}}\hat{w}\mathtt{c}^{\hat{k}}$ contains at most $N < N!+1$ a's and at least one c. Therefore neither phrase is a substring of the other, so if they belong to the same parse tree, then the phrases do not overlap. But then the yield of that parse tree cannot be of the form $\mathtt{a}^*\mathtt{b}^*\mathtt{c}^*$, so the phrases must belong to different parse trees. Consequently, there are at least two parse trees yielding $\mathtt{a}^{N!+N}\mathtt{b}^{N!+N}\mathtt{c}^{N!+N}$, so G is ambiguous. Because this is true for all grammars G, the language is inherently ambiguous.

5.9-5 A CFG G is *unboundedly ambiguous* if for every m there is a string that can be parsed at least m different ways in G. A CFL L is *inherently* unboundedly ambiguous if every CFG for L is unboundedly ambiguous. Let L be the set of all strings of the form $\mathtt{a}^i\mathtt{c}\{\mathtt{a},\mathtt{b}\}^*\mathtt{b}\mathtt{a}^i\mathtt{b}\{\mathtt{a},\mathtt{b}\}^*$, i.e.,

$$L = \{\mathtt{a}^i\mathtt{c}x\mathtt{b}\mathtt{a}^i\mathtt{b}y : x, y \in \{\mathtt{a},\mathtt{b}\}^* \text{ and } i \ge 0\}.$$

Prove that L is inherently unboundedly ambiguous.

Solution: Let G be a grammar for L, and let N be the constant for G given by Ogden's lemma. Fix a natural number m. Let

$$z_j = \mathtt{a}^N\mathtt{c}(\mathtt{a}^{N!+N}\mathtt{b})^{2j}\mathtt{a}^N\mathtt{b}(\mathtt{a}^{N!+N}\mathtt{b})^{2m-2j}.$$

Mark the first N a's for pumping. Apply Ogden's lemma and observe that the c cannot be pumped. Therefore there exist A_j, u_j, v_j, w_j, x_j, y_j, and k_j satisfying the following conditions:

- $S \stackrel{*}{\Rightarrow} u_j A_j y_j$.
- $A_j \stackrel{*}{\Rightarrow} v_j A_j x_j$.
- $A_j \stackrel{*}{\Rightarrow} w_j$.
- $z_j = u_j v_j w_j x_j y_j$.
- $v_j = x_j = \mathtt{a}^{k_j}$.
- $1 \leq k_j \leq N$.

Then there is a derivation that we call D_j —

$$\begin{aligned} S &\stackrel{*}{\Rightarrow} u_j A_j y_j \\ &\stackrel{*}{\Rightarrow} u_j v_j^{N!/k_j} A_j x_j^{N!/k_j} y_j \\ &\stackrel{*}{\Rightarrow} u_j v_j^{N!/k_j} v_j A_j x_j x_j^{N!/k_j} y_j \\ &\stackrel{*}{\Rightarrow} u_j v_j^{N!/k_j} v_j w_j x_j x_j^{N!/k_j} y_j \\ &= u_j \mathtt{a}^{N!+k_j} w_j \mathtt{a}^{N!+k_j} y_j \\ &= \mathtt{a}^{N!+N} \mathtt{c} (\mathtt{a}^{N!+N} \mathtt{b})^{2m+1} \end{aligned}$$

— in which $\mathtt{a}^{N!+k_j} w_j \mathtt{a}^{N!+k_j}$ and $\mathtt{a}^{k_j} w_j \mathtt{a}^{k_j}$ are both phrases.

We assert that if $1 \leq i < j \leq m$, then D_i and D_j correspond to distinct parse trees yielding $\mathtt{a}^{N!+N} \mathtt{c} (\mathtt{a}^{N!+N} \mathtt{b})^{2m+1}$. Proof: D_i contains the phrase $\mathtt{a}^{N!+k_i} w_i \mathtt{a}^{N!+k_i}$, which starts with at least $N!+1$ a's and then a c, and is at most $(N!+N+1)(2i+2)$ characters long. D_j contains the phrase $\mathtt{a}^{k_j} w_j \mathtt{a}^{k_j}$, which starts with at most N a's and then a c, and is at least $(N!+N+1)(2j)+1$ characters long. Because $N < N!+1$ and $(N!+N+1)(2i+2) < (N!+N+1)(2j)+1$, neither phrase can be a substring of the other. Therefore, if they occur as phrases in the same parse tree, then they do not overlap. But then the string yielded by that parse tree would contain two c's, so the phrases must belong to distinct parse trees.

Thus there are at least m parse trees yielding the string $\mathtt{a}^{N!+N} \mathtt{c} (\mathtt{a}^{N!+N} \mathtt{b})^{2m+1}$ in the grammar G, so G is unboundedly ambiguous. Because this is true for every grammar G, L is inherently unboundedly ambiguous.

5.11-2(b) Modify the CYK algorithm so that it produces a parse tree for x if $x \in L(G)$.

Solution: Let MakeLeaf be a function that takes a character and makes a leaf labeled by that character. Let MakeTree be a function that takes two trees and makes a new tree that has them as them as its children.

```
n := |s|; (* initialization *)
for every variable X do begin
    for i := 1 to n do
        for k := i to n do
            T[i, k, X] := nil;
```

```
        for i := 1 to n do
            if X → s_ii is a production then
                T[i, i, X] := MakeLeaf(s_ii);
end;

for k := 2 to n do
    for i := k − 1 down to 1 do
        for all productions of the form X → YZ do
            for j := i to k − 1 do
                if T[i, j, Y] ≠ nil and T[j + 1, k, Z] ≠ nil then
                    T[i, k, X] := MakeTree(T[i, j, Y], T[j + 1, k, Z]);
```

6.1-7 Use closure properties to prove that the following languages are not CFLs:

(c) the set of all strings x with $\#_{\mathtt{a}}(x) = \#_{\mathtt{b}}(x)$ and $\#_{\mathtt{c}}(x) = \#_{\mathtt{d}}(x)$. (Recall that $\#_e(x)$ is the number of e's in the string x.)

Solution: Let L be the given language. Assume, for the sake of contradiction, that L is a CFL. Let R be the regular language $\mathtt{a}^*\mathtt{c}^*\mathtt{b}^*\mathtt{d}^*$. Then $L \cap R$ is a CFL; call it L'. But $L' = \{\mathtt{a}^i\mathtt{c}^j\mathtt{b}^i\mathtt{d}^j : i, j \geq 0\}$. Let τ be the finite transduction that replaces b's by c's and c's by b's. Then $L'\tau$ is a CFL. But $L'\tau = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^i\mathtt{d}^j : i, j \geq 0\}$, which is not a CFL by Example 5.23.

6.1-8 See Exercise 4.8-13 for the definition of PERM$(\cdot)$.

(b) If L is a regular language over a 2-character alphabet, prove that PERM(L) is an NCA language and therefore a CFL.

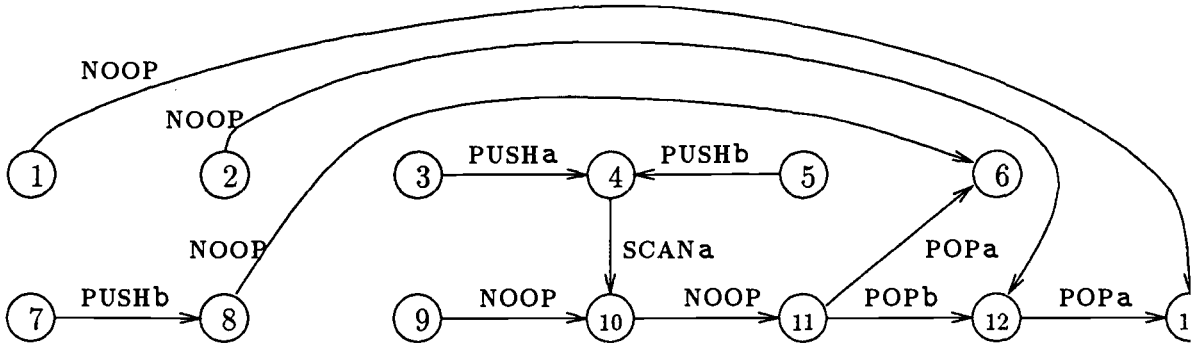
Solution: Without loss of generality, assume that L is a language over $\{\mathtt{a}, \mathtt{b}\}$. Let P be an NFA that accepts L. Standardize P by eliminating null instructions and EOF tests. We will construct an NCA P', with a signed counter, that accepts PERM(L). On input x, P' guesses a string with the same number of a's and b's as x (i.e., a permutation of x) and checks whether that string is in L.

By "processing c," we mean emulating one step of P that scans a c, but without actually scanning anything. We set aside one control in P' for emulating P. P' processes a's as soon as they are read but guesses when to process b's, using the signed counter to keep track of the number of b's processed minus the number of b's actually scanned. This allows P' to process any sequence of characters that has the same number of a's as the input string. We require that the signed counter finish at 0, so that the sequence processed by P' has the same number of b's as the input string. Thus P' processes any permutation of the input string. P' accepts if P accepts the processed string. Therefore P' accepts PERM(L). Because every NCA language is a CFL, PERM(L) is a CFL. The construction is given explicitly in Table 11.4.

Instruction of P (control , input)	Instruction(s) of P' (control , input , signed counter)	Remark
	$(q \to q$, SCANb , DEC $)$	for all control states q
$(q \to r$, SCANb$)$	$(q \to r$, NOOP , INC $)$	(process b)
$(q \to r$, SCANa$)$	$(q \to r$, SCANa , NOOP $)$	(process a)

Table 11.4: Converting an NFA for a language over $\{\mathtt{a}, \mathtt{b}\}$ to an NCA for PERM(L).

6.2-1 **Solution:**



6.2-3 Define INC–DEC pairs by analogy to PUSH–POP pairs. Show how to eliminate INC–DEC pairs from a DCA. (Assume that the counter is unsigned.)

Solution: The textbook's standardization can be modified by replacing PUSHc by INC and POPc by DEC everywhere.

Alternatively, convert the DCA to a DSA with a 1-character stack alphabet. Eliminate PUSH–POP pairs from the DSA and then convert the standardized program back into a DCA.

6.3-3 Let L be the set of all strings of the form $\mathtt{a}^i\mathtt{b}\{\mathtt{a},\mathtt{b}\}^*\mathtt{b}\mathtt{a}^i\mathtt{b}\{\mathtt{a},\mathtt{b}\}^*$, i.e.,

$$L = \{\mathtt{a}^i\mathtt{b}x\mathtt{b}\mathtt{a}^i\mathtt{b}y : x, y \in \{\mathtt{a},\mathtt{b}\}^* \text{ and } i \geq 0\}.$$

Prove that L is inherently unboundedly ambiguous (defined in Exercise 5.9-5). Hint: You may use the result of that exercise.

Solution: Construct a deterministic finite transducer T that takes strings in $\{\mathtt{a},\mathtt{b}\}^*$ and replaces the first b by a c. Call its transfer relation τ. Then τ is one-one and $L\tau$ is equal to the language proved inherently unboundedly ambiguous in Exercise 5.9-5.

Let P be a generator for L. Let P' be the program obtained by composing P and T as in the composition theorem. Recall that T is deterministic and computes a one-one partial function. By a careful analysis of the construction in the proof of the composition theorem, we see that there are at least as many computations of P that generate x as computations of P' that generate $x\tau$.

Let m be any natural number. Because $L\tau$ is inherently unboundedly ambiguous, there must be a string $x\tau$ that is generated by at least m distinct computations of P'. Therefore x is generated by at least m distinct computations of P. Because such an x exists for every m, P is unboundedly ambiguous. Because that is true for every P that generates L, L is inherently unboundedly ambiguous.

6.4-2 Prove that the following are not DCFLs:

(b) $\{\mathtt{a}^i\mathtt{b}^j : 2i = 3j \text{ or } 3i = 2j\}$

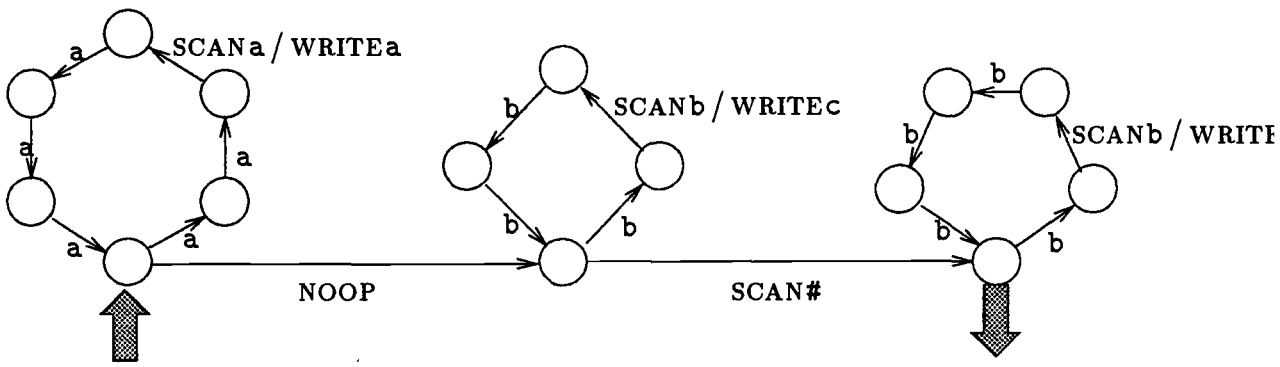
Solution: Call the language in question L. Assume, for the sake of contradiction, that L is a DCFL. Let

$$L' = \text{Double-Duty}(L) \cap \left(\mathtt{a}^+\mathtt{b}^+\#\mathtt{b}^+\right),$$

which must be a DCFL because the class of DCFLs is closed under Double-Duty$(\cdot)$ and under intersection with regular languages. If i, j, and j' are greater than 0, then $\mathtt{a}^i\mathtt{b}^j\#\mathtt{b}^{j'}$ belongs to L' if and only if $j = 2i/3$ and $j + j' = 3i/2$. Therefore

$$L' = \{\mathtt{a}^{6k}\mathtt{b}^{4k}\#\mathtt{b}^{5k} : k \geq 1\}.$$

Construct a finite transducer that replaces groups of 6 a's by a single a, then replaces groups of 4 b's by a single b until a # is scanned (the # is consumed), and finally replaces groups of 5 b's by a single c (see diagram below). Then $L'\tau = \{\mathtt{a}^k\mathtt{b}^k\mathtt{c}^k : k \geq 1\}$, which must be a CFL because the class of CFLs is closed under finite transductions. Then $L'\tau \cup \{\Lambda\}$ must be a CFL as well because the class of CFLs is closed under union, but that language is $\{\mathtt{a}^k\mathtt{b}^k\mathtt{c}^k : k \geq 0\}$, which is not a CFL by Example 5.22. This contradiction proves that L is not a DCFL.



6.4-4 Find a regular language R and a DCR language L such that RL is not a DSR language. Conclude that the class of DCR languages and the class of DSR languages are not closed under concatenation.

Solution: Let $R = \{\Lambda, \mathtt{a}\}$. Let $L = \{a^i b^j : i \text{ is even and } j = 2i, \text{ or } i \text{ is odd and } j = i\}$.

6.4-5 Prove that the class of DCR languages and the class of DSR languages are not closed under reversal.

Solution: Let $L = \{a^i b^j : i \text{ is odd and } j = 2i, \text{ or } i \text{ is even and } j = i\}$.

6.4-6 Prove that the class of DCR languages and the class of DSR languages are not closed under Kleene-closure.

Solution: Let

$$L = \{\mathtt{a}\mathtt{b}^i\mathtt{c}^i : i \geq 0\} \cup \{\mathtt{b}^i\mathtt{c}^{2i} : i \geq 0\}.$$

Then L is a DCR language and hence a DSR language as well. We assert that L^* is not a DSR language and hence not a DCR language either. Suppose, for the sake of contradiction, that L^* is a DCFL. Let $R = \mathtt{a}\mathtt{b}^*\mathtt{c}^*$, which is regular. Then $L^* \cap R$ must be a DCFL. But

$$L^* \cap R = \{\mathtt{a}\mathtt{b}^i\mathtt{c}^i : i \geq 0\} \cup \{\mathtt{a}\mathtt{b}^i\mathtt{c}^{2i} : i \geq 0\}.$$

Let $L' = L^* \cap R$. Let τ be the deterministic finite transduction that maps x to $\mathtt{a}x$. Then $L'\tau^{-1}$ must be a DCFL. But

$$L'\tau^{-1} = \{\mathtt{b}^i\mathtt{c}^i : i \geq 0\} \cup \{\mathtt{b}^i\mathtt{c}^{2i} : i \geq 0\},$$

which is not a DCFL by Example 6.22 (renaming characters). This contradiction proves that L^* is not a DCFL, as desired.

6.4-7 Let S be a DCFL and let R be a regular language.

(b) Prove that S/R is a DCFL; i.e., the class of DCFLs is closed under quotient with a regular language. Hint: Use Lemma 6.16.

Solution: Let P be a DSA that recognizes S. Let L be the set of strings s such that the program P, when started with s on the stack, accepts at least one input string y that belongs to R. We assert that L is regular. Proof: L is accepted by an NSM program that starts with its input string s on the stack; nondeterministically guesses y character by character, checking that y belongs to R; and simulates P as though y were read. By Lemma 6.16, L is regular.

S/R is accepted by a DSA P' that simulates P on input x, simultaneously keeping track of whether P's stack string belongs to L, as in the proof of Theorem 6.18. P' accepts x iff the final stack contents belong to L.

6.4-8 Fix an alphabet Σ. Let S be a DCFL. Let F be a finite set of strings. Let R be a regular language.

(b) Let $\text{MIN}(L)$ be the set of strings x in L such that no proper prefix of x is also in L, i.e.,

$$\text{MIN}(L) = L - (L\Sigma^+).$$

For example,

$$\text{MIN}(\{\texttt{a}, \texttt{ab}, \texttt{bb}\}) = \{\texttt{a}, \texttt{bb}\}.$$

Prove that $\text{MIN}(S)$ is a DCFL.

Solution: Let P be an on-line DSR that recognizes S. Modify P to remember in an extra control whether a proper prefix of the input belongs to S. If none of them does but the entire string belongs to S, then accept; otherwise reject.

(c) Prove that SF is a DCFL.

Solution: Assume that all elements of F have length less than k. S is recognized by an on-line DSR. Modify the DSR to remember which of the last k prefixes read belong to S. Using its finite control, it can also remember the last k characters read. When the end of the string is reached, the DSR can check for $i = 0, 1, \ldots, k-1$ whether the last i characters scanned form a string in F and whether the corresponding prefix is an element of S.

(d) Prove that SR is a DCFL, i.e., the class of DCFLs is closed under concatenation on the right with a regular language.

Solution: S is recognized by an on-line DSR. R is accepted by an NFA P. Modify the on-line DSR so that each time it finishes scanning a prefix of the input that belongs to S, it begins simulating P on the remainder of the input. It continues checking whether other prefixes belong to S, and it may begin additional simulations of P. All this may be done

simultaneously using just one extra control which stores the set H of states that P might possibly be in (as in the proof of Theorem 4.18). H is initialized to be empty. Each time that a character c is scanned, replace each state in H by its successors, i.e., $H := Ht_c$. If the prefix just scanned belongs to S, then insert the start state of P into H. When the input is exhausted, accept if H contains an accepting state of P.

6.5-1 Informally show how two counters can simulate a stack with a k-character alphabet.

Solution: Let P be a program for a machine [stack,other], where the stack has a k-character alphabet. By renaming characters, we may assume that the stack alphabet is $\{1, \ldots, k\}$. We will simulate P by a program P' for a machine [$K1$,$K2$,other] where $K1$ and $K2$ are two counters. The relation of representation is

$$(k\text{-adic}(s), 0, x)\ \rho\ (s, x).$$

The operation PUSHj is simulated by multiplying the first counter by k and adding j. The operation POPj is simulated by subtracting j from the first counter and dividing by k (there must be no remainder). The operation EMPTY is simulated ZERO$_1$.

We can multiply, divide, augment, or diminish the first counter by a constant as in Sections 6.5 and 6.6.

6.8-2 Queues were defined in Exercise 1.1-1. Show informally how a machine [control, stack, stack, input] can simulate a machine [control, queue, input].

Solution: Several different solutions are possible.

Solution 1. We store the contents of the queue on the first stack and use the relation of representation

$$(q, z, \Lambda, x)\ \rho\ (q, z, x).$$

To test whether the queue is empty, we test whether the first stack is empty. The operation ENQUEUEc is simulated by pushing a c on the first stack. The operation DEQUEUEc is simulated by copying the contents of the first stack to the second stack (reversing it in the process), popping a c from the second stack, and then copying its contents back to the first stack (unreversing it in the process).

Solution 2. We use both stacks to represent the queue's contents. The relation of representation is

$$(q, z, w, x)\ \rho\ (q, z^R w, x).$$

To test whether the queue is empty, we test whether both stacks are empty. The operation ENQUEUEc is simulated by pushing a c on the second stack.

The operation DEQUEUEc is simulated as follows: if the first stack is empty then copy the contents of the second stack to the first stack (reversing it in the process); pop a c from the first stack.

Solution 3. We use both stacks to represent the queue's contents. The relation of representation is

$$\begin{array}{lll}(q,w,\Lambda,x) & \rho & (q,w,x)\\ (q,\Lambda,w,x) & \rho & (q,w^R,x)\end{array}$$

To test whether the queue is empty, we test whether both stacks are empty. The operation ENQUEUEc is simulated as follows: copy the contents of the second stack to the first stack (reversing it in the process); push a c on the first stack. The operation DEQUEUEc is simulated as follows: copy the contents of the first stack to the second stack (reversing it in the process); pop a c from the second stack.

7.1-6 A 2-*dimensional* tape (2-D tape) is a rectangular grid of tape squares extending infinitely far up and right. The operations SEEc and PRINTc are defined as usual. The ATHOME operation checks that the tape head is in the lower-left corner of the tape. The MOVEL, MOVER, MOVEU, and MOVED operations move the tape head left, right, up, and down, respectively — except that it is not possible to move left or down past the edge of the tape.

(a) Formally define the realm and repertory of a 2-D tape. What is a reasonable initial state for a 2-D tape?

(b) Informally prove that a tape can simulate a 2-D tape.

Solution:

(a) Fix an alphabet Γ containing $\sqcup$. Informally, think of the squares of the 2-D tape as having coordinates (i,j) where i and j are natural numbers; the home square is $(0,0)$. A function C from $\mathbf{N}\times\mathbf{N}$ to $\Gamma\cup\boxed{\Gamma}$ indicates which character is on each square and whether the tape head is present (each square starts with a blank character, so C is total). C is called "good" if there is exactly one pair (i,j) such that $C(i,j)\in\boxed{\Gamma}$. The realm of a 2-D tape with alphabet Γ is the set of all good functions from $\mathbf{N}\times\mathbf{N}$ to $\Gamma\cup\boxed{\Gamma}$. Tape operations follow:

- ATHOME $=\{(C,C) : C(0,0)\in\boxed{\Gamma}\}$.
- SEEa $=\{(C,C) : (\exists i,j)[C(i,j)=\boxed{a}]\}$.
- PRINTa $=\{(C,C') : (\exists i,j)\ [(C(i,j)\in\boxed{\Gamma}),\ (C'(i,j)=\boxed{a}),$ and $((k,\ell)\neq(i,j)\Rightarrow C(k,\ell)=C'(k,\ell))]\}$.

- MOVEL $= \{(C, C') : (\exists i, j)\,[(C(i,j) = \boxed{C'(i,j)}), (C'(i-1,j) = \boxed{C(i-1,j)})$ and $((k,\ell) \notin \{(i,j),(i-1,j)\} \Rightarrow C(k,\ell) = C'(k,\ell))]\}$.
- MOVER $= \{(C, C') : (\exists i, j)\,[(C(i,j) = \boxed{C'(i,j)}), (C'(i+1,j) = \boxed{C(i+1,j)})$ and $((k,\ell) \notin \{(i,j),(i+1,j)\} \Rightarrow C(k,\ell) = C'(k,\ell))]\}$.
- MOVED $= \{(C, C') : (\exists i, j)\,[(C(i,j) = \boxed{C'(i,j)}), (C'(i,j-1) = \boxed{C(i,j-1)})$ and $((k,\ell) \notin \{(i,j),(i,j-1)\} \Rightarrow C(k,\ell) = C'(k,\ell))]\}$.
- MOVEU $= \{(C, C') : (\exists i, j)\,[(C(i,j) = \boxed{C'(i,j)}), (C'(i,j+1) = \boxed{C(i,j+1)})$ and $((k,\ell) \notin \{(i,j),(i,j+1)\} \Rightarrow C(k,\ell) = C'(k,\ell))]\}$.

A reasonable initial state for a 2-D tape is C, where $C(0,0) = \boxed{\sqcup}$ and $(i,j) \neq (0,0) \Rightarrow C(i,j) = \sqcup$.

(b) Because $C(i,j) = \sqcup$ for all but finitely many pairs (i,j), the function $C(i,j)$ can be represented on an ordinary tape as the list of all triples (i,j,a) such that $C(i,j) = a$ and $a \neq \sqcup$. The operation SEEa is implemented by looking for a triple of the form $(i,j,\boxed{a})$. The operation PRINTa is implemented by looking for a triple of the form $(i,j,\boxed{b})$ and replacing it by $(i,j,\boxed{a})$.

The operation MOVEL is implemented by looking for a triple of the form $(i,j,\boxed{a})$, replacing it by (i,j,a), then looking for $(i-1,j,b)$ and replacing it by $(i-1,j,\boxed{b})$. Special cases: If $i = 0$, then the operation is not performed; if $a = \sqcup$, then (i,j,a) is deleted from the list; if there is no triple $(i-1,j,b)$, then $(i-1,j,\boxed{\sqcup})$ is inserted into the list.

The other MOVE operations are similar.

It remains to show how to add or subtract 1 and how to implement the list of triples on an ordinary tape. This is described in Section 7.3.

7.1-7 Let P be a nondeterministic 1-TM program with no input or output device. Construct informally an NCA that accepts all strings that are not traces of P.

Solution: The NCA nondeterministically guesses which step in the input computation to invalidate. It scans and ignores the first $s-1$ configurations, stores the control state of configuration s in its finite memory, and then nondeterministically counts to i while ignoring the first i tape characters of configuration s. It stores tape characters $i+1$, $i+2$, and $i+3$ of configuration s in its finite memory, and it ignores the rest of configuration s. Next it stores the control state of configuration $s+1$ in its finite memory. Then it counts down while ignoring the first i tape characters of configuration $s+1$. After that it stores tape characters $i+1$, $i+2$, and $i+3$ of configuration $s+1$ in its finite memory. If the computation proceeds improperly at location $i+2$, this can be detected and the NCA will accept. An error at either end of the tape can be caught similarly.

7.1-8 Let P be a nondeterministic 1-TM program with no input or output device. Prove that the set of noncomputations of P is an NCA language.

Solution:

We will design an NCA with a signed counter. First the NCA guesses whether to check the left end of the tape, the control, or a single tape square. To check the left end of the tape, the NCA counts the number of MOVER's minus the number of MOVEL's seen so far and accepts if the number ever becomes negative. To check the control, the NCA accepts if any instruction goes from a state other than the state the preceding instruction went to. To check a single tape square, the NCA chooses the ith square nondeterministically by counting up to i before looking at the input computation; as it reads the computation, the NCA subtracts 1 for each MOVER and adds one for each MOVEL; whenever the counter holds 0, that means that the tape head is on the ith square; the NCA keeps track of what is on the ith square at all times, accepting if an instruction sees a character other than the one that is there.

7.2-4 Let P be a DTR (1-tape recognizer). Show how to construct an equivalent DTR P' that does not use the EOF test.

Solution: First convert P to an equivalent DTR that takes its input from the tape. The program P' will have an extra tape and an extra control for simulating P. P' keeps track of a prefix w of the input on its tape. Each time P' scans a character, it appends that character to w, copies w to the extra tape, and simulates P on input w using the extra tape and extra control. P' accepts if and only if the input is empty and the extra control is in an accepting state. P' rejects if and only if the input is empty and the extra control is in a rejecting state. Otherwise, P' scans another character. Although its terminator requires the input to be empty, P' makes no explicit EOF tests.

7.5-4 Let the constants false and true denote two distinct natural numbers. Write HG programs for the following functions:

(a) greater(x, y) = true if $x > y$, false otherwise.
Solution:

$$\begin{aligned} \text{greater}(x, x+y) &= \text{false} \\ \text{greater}(x+y+1, x) &= \text{true.} \end{aligned}$$

(b) prime(x) = true if x is a prime number, false otherwise.
Solution:

$$\begin{aligned} \text{not(true)} &= \text{false} \\ \text{not(false)} &= \text{true} \end{aligned}$$

$$
\begin{aligned}
\text{and}(\text{true}, \text{true}) &= \text{true} \\
\text{and}(\text{false}, \text{true}) &= \text{false} \\
\text{and}(\text{true}, \text{false}) &= \text{false} \\
\text{and}(\text{false}, \text{false}) &= \text{false} \\
\text{divisible}(x+1, x+1) &= \text{true} \\
\text{divisible}(x+y, y) &= \text{divisible}(x, y) \\
\text{divisible}(x, x+y+1) &= \text{false} \\
\text{divisible}(x, 0) &= \text{false} \\
\text{primetest}(x, k+2) &= \text{and}(\text{not}(\text{divisible}(x, k+2)), \text{primetest}(x, k+1)) \\
\text{primetest}(x, 1) &= \text{true} \\
\text{prime}(x+2) &= \text{primetest}(x+2, x+1) \\
\text{prime}(1) &= \text{false} \\
\text{prime}(0) &= \text{false.}
\end{aligned}
$$

Note that primetest$(x, x-1)$ tries $2, \ldots, x-1$ as potential divisors of x.

7.6-1 Let P be a nondeterministic program for a machine [control, input, output, tape]. If P computes a partial function, prove that P is simulated by a deterministic program for a machine [control, input, output, tape].

Solution: Let P' behave as follows

```
for i := 0 to ∞ do
    if i encodes a computation of P then
        output the result of computation i and halt
```

7.6-8 (a) Give a constructive proof of the forward direction of Theorem 7.20(i); i.e., present an algorithm that solves the following problem:

Instance: a DTA P that accepts a nonempty language

Answer: a DTM program P' such that $\tau_{P'}$ is a total function and $L(P) = \text{Range}(\tau_{P'})$

The algorithm may give an incorrect answer or fail to halt if P accepts the empty language.

(b) Give a constructive proof of the forward direction of Theorem 7.20(ii); i.e., present an algorithm that solves the following problem:

Instance: a DTR P that recognizes a nonempty language

Answer: a DTM program P' such that $\tau_{P'}$ is a nondecreasing function and $L(P) = \text{Range}(\tau_{P'})$

The algorithm may give an incorrect answer or fail to halt if P is not a DTR or if P accepts the empty language.

Solution:

(a) Let the algorithm print the following program P':

```
input C;
if C is a computation of P then
    output the string scanned during C
else
    for i = 1 to ∞ do
        if i encodes a computation of P then
            output the string scanned during that computation;
```

The for-loop must terminate if P accepts a nonempty language. Thus $\tau_{P'}$ is a total function; clearly, $L(P) = \text{Range}(\tau_{P'})$.

(b) The proof in the textbook is constructive except for determining the least element of L. Since L is recursive and nonempty, the least element may be determined by a simple for-loop:

```
for i := 0 to ∞ do
    if i ∈ L then
        print i and halt.
```

7.6-10 Professor Cindy Simd has received a huge grant to design a computer that can run a single program on infinitely many inputs. You will solve her problem with a mere Turing machine, although less efficiently than she would like.

(a) Design a Turing machine program that will take a DTM program P and produce the results of running P on all inputs. Your program's output should be a (possibly infinite) sequence consisting of all pairs $(x, \tau_P(x))$ such that P halts on input x. Hint: Generalize the time-sharing/dovetailing idea.

(b) Modify your program from part (a) so that its output is sorted according to the number of steps that P runs on input x. Hint: You may assume that P scans its entire input, so P runs for at least s steps on inputs of length s.

Solution: We violate convention by constructing a program that produces output without halting. This is unavoidable, because the program may have to output an infinite sequence.

```
input P;
```

```
for s := 1 to ∞ do
    for t := 0 to s do
        for x := each string of length t do begin
            emulate P on input x for s steps;
            if the emulation finishes in exactly s steps then
                print (x, τ_P(x));
        end;
```

7.8-6 A function is monotone if $x \leq y \Rightarrow f(x) \leq f(y)$. Prove that there is no monotone one-one correspondence from the set of rational numbers to N.

Solution: Let f be a monotone one-one correspondence from the rational numbers to N. Because f is monotone and one-one, we have $x < y \Rightarrow f(x) < f(y)$. Therefore $f(1/1), f(1/2), f(1/3), \ldots$ is an infinite decreasing sequence of natural numbers, which is impossible, because every set of natural numbers contains a least element (Exercise 0.6-23).

7.8-12 Prove that S_1 and S_2 are recursively separable if and only if S_2 and S_1 are recursively separable.

Solution: Let P be as in the definition of recursive separation. Interchange accepting and rejecting states in P.

7.8-14 In contrast to Theorem 7.26, prove that if S_1 and S_2 are disjoint *co-r.e.* languages, then S_1 and S_2 are recursively separable.

Solution: Let A and B be disjoint co-r.e. sets. Dovetail acceptors for $\overline{A}$ and $\overline{B}$, and let R consist of the elements that are accepted into $\overline{B}$ before $\overline{A}$. Then R is r.e. But $\overline{R}$ is the set of elements that are accepted into $\overline{A}$ before $\overline{B}$, because $\overline{A} \cup \overline{B} = \Sigma^*$, so $\overline{R}$ is also r.e. Therefore R is recursive. Clearly R separates A from B.

7.9-2 Which of the following languages are recursive? Which are r.e.?

(a) $\{P : P$ is a DTA that accepts a finite language$\}$

Solution: Call the language in question L. Let $f(x)$ be a program that behaves as follows:

```
input z;
if x halts on input x then accept;
```

If $x \in K$ then $f(x)$ accepts Σ^*; otherwise $f(x)$ accepts $\emptyset$. Thus $x \in \overline{K} \iff f(x) \in L$. Since $\overline{K}$ is not r.e., L is not r.e. (and therefore not recursive).

(b) $\{P : P$ is a DTA that accepts a regular language$\}$

Solution: Call the language in question L. Let $f(x)$ be a program that behaves as follows:

```
input z;
if z has the form a^n b^n c^n then
if x halts on input x then accept;
```

If $x \in K$ then $f(x)$ accepts $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n : n \geq 0\}$; otherwise $f(x)$ accepts $\emptyset$. Thus $x \in \overline{K} \iff f(x) \in L$. Since $\overline{K}$ is not r.e., L is not r.e. (and therefore not recursive).

(c) $\{P : P$ is a DTA that accepts an r.e. language$\}$

Solution: This language is recursive because every DTA accepts an r.e. language and there is an algorithm to test whether a program is deterministic.

(d) $\{P : P$ is a DTA that accepts a co-r.e. language$\}$

Solution: Call the language in question L. Let $f(x)$ be a program that behaves as follows:

```
input z;
if x halts on input x then
    if z halts on input z then accept;
```

If $x \in K$ then $f(x)$ accepts K, which is not co-r.e.; otherwise $f(x)$ accepts $\emptyset$, which is co-r.e. Thus $x \in \overline{K} \iff f(x) \in L$. Since $\overline{K}$ is not r.e., L is not r.e. (and therefore not recursive).

(e) $\{P : P$ is a DTA that has at least 17 control states$\}$

Solution: This language is recursive because a DTR can easily check whether P is deterministic and count the number of control states.

(f) $\{P : P$ has at least 17 control states and $P \in K_{inatt}\}$

Solution: Let L be the language in question. Let $f(x)$ be constructed from x by adding 17 unreachable control states. Then $x \in K_{inatt} \iff f(x) \in L$. Since K_{inatt} is nonrecursive, L is nonrecursive. On the other hand L is accepted by the following DTA, so L is r.e.:

```
input P;
if P has at least 17 control states then
    if P halts then accept;
```

(g) $\{P : P$ is a DTA that has a computation of length 17 or less$\}$

Solution: This language is recursive (and therefore r.e.) because we can test each sequence of 17 or fewer instructions to see if it is a computation of P and accept if it is.

(h) $\{P : P$ is a DTM program that halts on fewer than 17 inputs$\}$

Solution: Call the language in question L. Let $f(x)$ be a program that behaves as follows:

input z;
if x halts on input x then accept;

If $x \in K$ then $f(x)$ accepts Σ^*; otherwise $f(x)$ accepts $\emptyset$. Thus $x \in \overline{K} \iff f(x) \in L$. Since $\overline{K}$ is not r.e., L is not r.e. (and therefore not recursive).

(i) $\{P : P$ is a DTM program that halts on fewer inputs than P has control states$\}$

Solution: Call the language in question L. Let $f(x)$ be a program that behaves as follows:

input z;
if x halts on input x then accept;

If $x \in K$ then $f(x)$ accepts Σ^*; otherwise $f(x)$ accepts $\emptyset$. Thus $x \in \overline{K} \iff f(x) \in L$. Since $\overline{K}$ is not r.e., L is not r.e. (and therefore not recursive).

7.9-3 (a) Prove that $\{(P, Q) : P$ and Q are equivalent DTAs$\}$ is not r.e.

(b) Prove that there is no DTM program D that will take a pair of equivalent DTAs P and Q and produce a proof that they are equivalent. Make no assumption about D's behavior when its input consists of a pair of inequivalent DTAs.

Solution:

(a) Call the language in question L. Let $f(x) = (P, Q)$, where P is some DTA with no accepting states and Q behaves as follows:

input z;
if x halts on input x then accept;

If $x \in K$ then Q accepts Σ^*, so P and Q are inequivalent; otherwise Q accepts $\emptyset$ so P and Q are equivalent. Thus $x \in \overline{K} \iff f(x) \in L$. Since $\overline{K}$ is not r.e., L is not r.e. (and therefore not recursive).

(b) For the sake of contradiction suppose that such a DTA D exists. Then the following DTA accepts L (contradicting part (a)):

input (P, Q);
let y be the result (if any) of running D on input (P, Q);
if y is a proof that P and Q are equivalent then accept;

7.10-5**(a) Prove that the word problem is undecidable for Thuë systems in which the right sides of all equalities are Λ.

**(b) Prove that there is no algorithm to decide whether $x = \Lambda$ in a Thuë system G, even if the right sides of all equalities are Λ. Hint: Use part (a).

Solution: See Zhang's paper in J. Symbolic Comput. 14(1992), 359–370.

7.10-8 So far we have seen that regular expressions are equivalent to NFAs, CFGs are equivalent to NSAs, and rewriting systems are equivalent to NTAs. Design a class of grammars that are equivalent to NCAs. (Your class of grammars need not be as elegant and natural as regular expressions, CFGs, and rewriting systems.)

Solution: Consider grammars G with terminal alphabet Σ and nonterminal alphabet N containing three special nonterminal characters A, B, C. Require that all productions have one of the following forms:

- $CV \to V'$,
- $V \to CV'$,
- $Vc \to V'$,

where c can be any terminal character, and V and V' can be any nonterminal characters other than C. Define

$$L(G) = \{x : x \text{ is a terminal string and } Ax \stackrel{*}{\Rightarrow} B\}.$$

First we show that $L(G)$ is accepted by a nondeterministic program for a machine [input, control, unsigned counter]. Observe that all strings of terminals and nonterminals derivable from Ax have the form C^iVx' where x' is a suffix of x and V is a nonterminal. This string will be represented by the configuration (x', V, i) in the NCA. The control set is $N - \{C\}$, the initial control state is A, and the unique accepting control state is B. The counter's initial and accepting states are 0. The production $CV \to V'$ is simulated by the instruction (NOOP, $V \to V'$, DEC). The production $V \to CV'$ is simulated by the instruction (NOOP, $V \to V'$, INC). The production $Vc \to V'$ is simulated by the instruction (SCANc, $V \to V'$, NOOP).

Now we show how to simulate an NCA program by such a grammar. Assume the program runs on a machine [input, control, unsigned counter]. Standardize the program so that $(x, A, 0)$ is the initial configuration on input x, and $(\Lambda, B, 0)$ is the unique accepting configuration. Rename characters so that the input alphabet is disjoint from the control set. The instruction (NOOP, $q \to q'$, DEC) is simulated by the production $Cq \to q'$; the instruction (NOOP, $q \to q'$, INC) is simulated by the production $q \to Cq'$; and the instruction (SCANc, $q \to q'$, NOOP) is simulated by the production $qc \to q'$.

7.11-2 Consider the following modified version of PCP.

Problem name: semimodified Post Correspondence Problem (SMPCP)

Instance: a "starting" pair of strings $(x_{\text{start}}, y_{\text{start}})$ and a set S of "other" pairs of strings

Question: Does there exist a sequence $\langle\langle(x_1, y_1), \ldots, (x_k, y_k)\rangle\rangle$ of pairs in S such that

$$x_{\text{start}} x_1 \cdots x_k = y_{\text{start}} y_1 \cdots y_k?$$

Prove that SMPCP is undecidable.

Solution: Reduce MPCP to SMPCP. This is an easy modification of the textbook's reduction from MPCP to PCP.

7.11-3 In parts (a) and (b) we will consider points $(x, y) \in \mathsf{N} \times \mathsf{N}$. An *affine transformation* on $\mathsf{N} \times \mathsf{N}$ is a function t such that

$$(x, y)\, t = (ax + by + c, dx + ey + f)$$

for some fixed natural numbers $a, b, \mathrm{c}, d, e, f$.

(b) Prove that the following problem is undecidable:

Instance: a point $(x_0, y_0) \in \mathsf{N} \times \mathsf{N}$ and a finite set S of affine transformations on $\mathsf{N} \times \mathsf{N}$

Question: Is there a sequence of transformations in S that maps (x_0, y_0) to a point on the line $x = y$, i.e., do there exist a natural number x and a sequence of transformations $t_1, \ldots, t_k$ in S such that

$$(x_0, y_0) t_1 t_2 \cdots t_k = (x, x)?$$

Hint: Reduce SMPCP (Exercise 7.11-2). Represent each string by a k-adic numeral. Observe that $k\text{-adic}(ru) = k^{|u|} k\text{-adic}(r) + k\text{-adic}(u)$.

Solution: Consider an SMPCP instance over a k-character alphabet. Let

$$(x_0, y_0) = (k\text{-adic}(x_{\text{start}}), k\text{-adic}(y_{\text{start}})).$$

For each internal pair $\frac{u}{v}$ in the MPCP instance, let S contain the transformation

$$(x, y)\, t = (k^{|u|} x + k\text{-adic}(u), k^{|v|} y + k\text{-adic}(v)).$$

(c) Prove that the following problem is undecidable:

Instance: a point $(x_0, y_0, z_0) \in \mathsf{N}^3$ and a finite set S of 3×3 matrices over N

Question: Are there a sequence of matrices $M_1, \ldots, M_k$ in S and a point $(x, x, z) \in \mathsf{N}^3$ such that

$$(x_0\ y_0\ z_0) M_1 M_2 \cdots M_k = (x\ x\ z)?$$

Solution: We reduce part (b). Let $z_0 = 1$. For each affine transformation of the form

$$(x, y)\,t = (ax + by + c, dx + ey + f),$$

construct the matrix

$$\begin{pmatrix} a & d & 0 \\ b & e & 0 \\ c & f & 1 \end{pmatrix}.$$

Observe that

$$\begin{pmatrix} x & y & 1 \end{pmatrix} \begin{pmatrix} a & d & 0 \\ b & e & 0 \\ c & f & 1 \end{pmatrix} = \begin{pmatrix} ax + by + c & dx + ey + f & 1 \end{pmatrix}.$$

(d) Prove that the following problem is undecidable:

Instance: a finite set S of 3×3 matrices over N

Question: Is there a sequence of matrices $M_1, \ldots, M_k$ in S such that the product $M_1 M_2 \cdots M_k$ contains equal numbers in row 3, column 1 and in row 3, column 2?

Solution: This is just like part (c) only you reduce PCP instead of SMPCP.

(e) Prove that the following problem is undecidable:

Problem name: mortality

Instance: a finite set S of 3×3 matrices over N and a pair of natural numbers i, j between 1 and 3

Question: Is there a sequence of matrices $M_1, \ldots, M_k$ in S such that the product $M_1 M_2 \cdots M_k$ contains a 0 in row i, column j?

Solution: We reduce part (d). Let

$$T = \begin{pmatrix} 1 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

T is the matrix for the transformation that subtracts column 1 from column 2. T^{-1} is the matrix for the transformation that adds row 2 to row 1. Thus M contains equal numbers in row 3, column 1 and in row 3, column 2 if and only if $T^{-1}MT$ contains a 0 in row 3, column 2. Replace each matrix M in part (d)'s problem by the matrix $M' = T^{-1}MT$. (This is called a conjugate of M.) Since

$$M'_1 M'_2 \cdots M'_k = T^{-1} M_1 M_2 \cdots M_k T,$$

the product $M'_1 M'_2 \cdots M'_k$ contains equal numbers in row 3, column 1 and in row 3, column 2 if and only if the product $M'_1 M'_2 \cdots M'_k$ contains a 0 in row 3, column 2.

7.13-1 Which of the following problems are undecidable?

(b) **Instance:** a CFG G and variables X and Y
Question: Is $L(X) = L(Y)$?
Solution: Undecidable. We will reduce the problem of determining whether $L(G_1) = L(G_2)$ for CFGs G_1 and G_2. Without loss of generality, assume that G_1 and G_2 have disjoint nonterminal alphabets, X is the start variable of G_1, and Y is the start variable of G_2. Let G be the union of the grammars G_1 and G_2. Then $L(G_1) = L(X)$ and $L(G_2) = L(Y)$, so $L(G_1) = L(G_2)$ if and only if $L(X) = L(Y)$.

(g) **Instance:** a CFG G
Question: Is $|L(G)| = 1$; i.e., does G generate exactly one string?
Solution: Decidable. Let S denote $\{G : |L(G)| = 1\}$. S is r.e. because we can guess x and check that $L(G) = \{x\}$ as in part (f). $\overline{S}$ is r.e. because we can check $L(G) = \emptyset$ or guess two strings x, y and check that they belong to $L(G)$ using the CYK algorithm or Earley's algorithm. By Theorem 7.19, S is recursive.

(j) **Instance:** a CFG G
Question: Is $L(G) = \mathtt{a}^*$?
Solution: Decidable. First see if $L(G) \subseteq \mathtt{a}^*$. If not then reject. If so, then let N be a pumping number for G. We assert that $L(G) = \mathtt{a}^*$ iff $L(G)$ contains the strings $\mathtt{a}^0, \mathtt{a}^1, \ldots, \mathtt{a}^{N+N!}$. One direction is obvious. Conversely, suppose that $L(G)$ contains $\mathtt{a}^0, \mathtt{a}, \ldots, \mathtt{a}^{N+N!}$. Every other string in $\mathtt{a}^*$ is of the form $\mathtt{a}^{j+kN!}$ where $N \leq j \leq N + N!$ and $k \geq 1$. Then $\mathtt{a}^j$ is pumpable. Apply the pumping theorem to $\mathtt{a}^j$ to obtain u, v, w, x, y. Let $vx = \mathtt{a}^\ell$. Pumping $kN!/\ell$ times, we find that $L(G)$ contains $\mathtt{a}^{j+kN!}$ as desired. Thus L contains $\mathtt{a}^*$.

7.15-1 A language L is *prefix-free* if, for all x and y in L, x is not a prefix of y. Throughout this problem, let P denote a DTR whose input device does not use the EOF test and whose final relation does not depend on the input device (i.e., the input need not be empty for acceptance to occur).

(c) Let L denote the language accepted by P, and let $\gamma(P) = \sum_{x \in L} \Pr(x)$. Let r denote a rational number. Prove that $\{(P, r) : \gamma(P) > r\}$ is r.e.
Solution: Guess a finite subset $F \subseteq L$ such that $\sum_{x \in F} \Pr(x) > r$.

(d) Is $\{(P, r) : \gamma(P) \geq r\}$ r.e?
Solution: No. Let $S = \{(P, r) : \gamma(P) \geq r\}$. We reduce $\overline{K}_{inatt}$ to S. Given an inattentive DTM program x, design a program $f(x)$ that accepts $\mathtt{0}^s\mathtt{1}$ if and only if x runs for at least s steps. Then

$$x \in \overline{K}_{inatt} \iff \gamma(f(x)) = 1 \iff (f(x), 1) \in S.$$

Since $\overline{K}_{inatt}$ is not r.e., S is not r.e.

(e) Let P be a particular program. Assume that $\{r : \gamma(P) > r\}$ is recursive. Prove that P accepts a recursive language.

Solution: Let L be the language accepted by P. Then L is r.e. It suffices to show that $\overline{L}$ is r.e. The algorithm in Figure 11.4 accepts $\overline{L}$ by approximating $\gamma(P)$ and then finding enough strings to account for almost all of the probability.

Input x;
Guess r;
Verify that $r < \gamma(P) < r + \Pr(x)$;
Guess distinct strings $x_1, \ldots, x_k$;
Verify that $x \neq x_1, \ldots, x \neq x_k$;
If $x_1 \in L, \ldots, x_k \in L$ and $\Pr(x_1) + \cdots + \Pr(x_k) > r$ then
    accept.

Figure 11.4: An algorithm that accepts $\overline{L}$. Note that if the algorithm accepts x and $x \in L$, then $\gamma(P) > r + \Pr(x)$, which is impossible.

7.15-3 (a) Given two finite sets of strings $U = \{u_1, \ldots, u_m\}$ and $V = \{v_1, \ldots, v_n\}$, construct an NCA that solves the following problem:

Instance: a string x

Question: Do there exist sequences of positive integers $i_1, \ldots, i_k$ and $j_1, \ldots, j_k$ such that $x = u_{i_1} \ldots u_{i_k} = v_{j_1} \cdots v_{j_k}$?

(b) Prove that the following problem is decidable:

Instance: two finite sets of strings $U = \{u_1, \ldots, u_m\}$ and $V = \{v_1, \ldots, v_n\}$

Question: Do there exist sequences of positive integers $i_1, \ldots, i_k$ and $j_1, \ldots, j_k$ such that $u_{i_1} \ldots u_{i_k} = v_{j_1} \cdots v_{j_k}$?

Solution:

(a) Let ℓ be the length of the longest string in $U \cup V$. Let Σ be the set of characters appearing in strings in $U \cup V$. We design a nondeterministic program for a machine [input, control, control, signed counter] where both control sets are $(\Sigma \cup \{\Lambda\})^k$.

The program will nondeterministically partition the input string x into a sequence of strings belonging to U and simultaneously partition it into a sequence of strings belonging to V. The counter will keep track of the number of strings in the former partition minus the number of strings in the latter partition.

The initializer is $x\alpha = (x, \Lambda, \Lambda, 0)$. The instructions are

- (SCAN$c, u \to uc, v \to vc$, NOOP) for all characters c and all strings u and v having length ℓ,

- (NOOP, $u_i \rightarrow \Lambda$, NOOP, INC) for all $u_i \in U$, and
- (NOOP, NOOP, $v_i \rightarrow \Lambda$, DEC) for all $v_i \in V$.

The terminator is $(\Lambda, \Lambda, \Lambda, 0)\omega$ = ACCEPT.

(b) This problem asks us to determine whether the NCA constructed in part (a) accepts a nonempty language. The emptiness problem for NCA languages is m-reducible to the emptiness problem for NSA languages, which is m-reducible to the emptiness problem for CFLs, which is decidable.

7.15-6 Is there an algorithm to determine whether an NSM program halts on all inputs, i.e., has at least one complete computation on each input?

Solution: No. An NSA halts on all inputs if and only if it accepts Σ^*, but there is no algorithm to determine whether a CFL is equal to Σ^*.

8.2-2 Dr. Lychenko purports to have written a DTM program P that solves the halting problem for inattentive DTM programs. Construct an input x on which P diverges or gives the wrong answer.

Solution: Using the recursion theorem, write the following program x:

Step 1: if P accepts x then go to step 1;

Step 2: halt.

If P accepts x, then x does not halt. If P rejects x, then x halts. So P gives either the wrong answer on input x or no answer.

8.2-6 Use the recursion theorem to prove the fixed-point theorem.

Solution: Let f be a total recursive function. Let $h(e, x) = \varphi_{f(e)}(x)$. By the recursion theorem there exists P such that

$$\varphi_P = h(P, x) = \varphi_{f(P)}(x).$$

8.3-3 A theory is *ω-consistent* if there is no predicate $Q(\cdot)$ such that there exists a proof of $(\exists x)[Q(x)]$ as well as proofs of $\neg Q(c)$ for each constant c. (For example, a theory of arithmetic is ω-consistent iff there is no predicate $Q(\cdot)$ such that there exists a proof of $(\exists x)[Q(x)]$ as well as proofs of $\neg Q(0), \neg Q(1), \neg Q(2), \ldots$)

(b) Prove that if a theory is ω-consistent, then it is consistent.

Solution: If the theory is ω-consistent, then at least one statement is unprovable, so the theory is consistent by Exercise 8.3-2.

8.3-4 Let ψ be as in Theorem 8.12. Assume that arithmetic is ω-consistent and prove the following: There is no proof in arithmetic of $\neg\psi$.

Solution: Assume for the sake of contradiction that there is a proof of $\neg\psi$; i.e., there is a proof that P halts. Let $Q(x)$ be the statement "x is a computation of P." P halts if and only if $(\exists x)[Q(x)]$. Since there is a proof that P halts, there is a proof that $(\exists x)[Q(x)]$. By ω-consistency there exists c for which there is no proof of $\neg Q(c)$; i.e., there is no proof that c is not a computation of P.

Assertion: c is a computation of P.

Proof of assertion: For the sake of contradiction, assume that c is not a computation of P. Because computations can be checked mechanically, there must then be a proof that c is not a computation of P. This contradiction proves that c is a computation of P.

The assertion implies that P halts, but we already proved that P does not halt, assuming only consistency. This contradiction implies that there is no proof of $\neg\psi$.

9.1-9 **Pratt's theorem:** PRIMES $\in$ NP. Let Z_p^* denote the group $\{1,\ldots,p-1\}$ under multiplication modulo p. An element a has order m in a group if $a^m = 1$ and for every j such that $0 < j < m$, $a^j \neq 1$.

(a) Prove that p is prime iff Z_p^* contains an element whose order is $p-1$.

(b) Prove that a has order m in Z_p^* iff $a^m = 1$ and for every j such that $0 < j < m$ and j is a divisor of m, $a^j \neq 1$.

(c) Prove that a has order m in Z_p^* iff $a^m = 1$ and for every prime q such that q divides m, $a^{m/q} \neq 1$.

(d) Prove that PRIMES $\in$ NP. Hint: Recursively determine the prime factorization of $p-1$.

Solution:

(a–c) See Section 2.9 of Niven and Zuckerman's *Introduction to The Theory of Numbers.*

(d) We first guess a number a which we hope has order $p-1$ in Z_p^*. In order to check our guess, we check that a^{p-1} is 1 in Z_p^*, we guess a factorization of $p-1$ into primes d, check that $a^{(p-1)/d} \neq 1$ for each such d, and check recursively that each d is in fact prime.

9.2-2 Prove that the following three statements are equivalent:

- P = NP
- at least one NP-complete language is in P

- every NP-complete language is in P

Solution: If P = NP then every language in NP is in P, so the first statement implies the third. The third logically implies the second.

If one NP-complete language C is in P, then consider any language L in NP. L is polynomial-time m-reducible to C, so L is in P. Thus P = NP. We have shown that the second statement implies the first.

Thus all three statements are equivalent.

9.2-3 (a) If L is recognized by a DTR that runs in time bounded by $t(n)$, prove that there are infinitely many DTRs that recognize L and run in time bounded by $t(n)$.

Solution:

(a) Assume that L is recognized by a DTR P that runs in time $t(n)$. For each $i \geq 0$, obtain P_i by adding i unreachable controls states and no instructions to P. Then each P_i is a different DTR that recognizes L in time $t(n)$.

9.4-2 Construct a PSPACE-complete language. (See Exercise 9.1-6 for a definition of PSPACE.)

Solution: Let

$$L = \{P\#x\#0^s : P \text{ is a binary encoding of a DTA } P,\ x \in \{0,1\}^*, \text{ and } P \text{ has an accepting trace in which each configuration uses at most } s \text{ tape squares}\}.$$

9.5-4 Give a polynomial-time m-reduction from (k,Γ)-SSS to $(2,\Gamma^k)$-SSS. Conclude that there exists Γ such that $(2,\Gamma)$-SSS is NP-complete.

Solution: Consider a symbol system S with locality k, alphabet Γ, variable set V, and constraint set C. We construct a new symbol system S' with locality 2, alphabet Γ^k, variable set V^k, and constraints defined as follows: (1) for each pair of variables in X' and Y' in V^k, we enforce consistency by requiring that components with the same name have the same value; (2) for each constraint in C involving variables $X_1, \ldots, X_k$, we include a single equivalent constraint on the single variable $X_1 \cdots X_k$.

We know that there exist k and Γ such that (k,Γ)-SSS is NP-complete. For those values of k and Γ, $(2,\Gamma^k)$-SSS is NP-complete.

9.6-3 A reduction between two problems is called *parsimonious* if it preserves the number of solutions.

(a) Find a parsimonious $\leq_m^p$-reduction from $(3, \{\text{true}, \text{false}\})$-SSS to 3-SAT.
Solution: First we modify the symbol system so that every constraint affects exactly three variables. Towards this end, introduce three new variables x, y, z and one new constraint,

$$x = \text{false} \ \wedge \ y = \text{false} \ \wedge \ z = \text{false}.$$

If a constraint C involves only one variable, replace C by $C \vee x \vee y$. If a constraint C involves only two variables, replace C by $C \vee x$.
Notation: Let x^b denote x if $b = \text{true}$ and $\overline{x}$ if $b = \text{false}$. As in the proof of Theorem 9.15, replace every constraint C by the conjunction

$$\bigwedge_{I \in \text{invalid}(C)} \left(\bigvee_{x \in V(C)} x^{\overline{I(x)}} \right).$$

9.6-5 A clause is called *monotone* if it consists entirely of variables or entirely of the negations of variables. For example, $x \vee y \vee z$ is monotone and $\overline{x} \vee \overline{y} \vee \overline{z}$ is monotone, but $x \vee \overline{y} \vee z$ is not monotone.

(a) MONOTONE-SAT is the set of satisfiable CNF formulas in which every clause is monotone. Prove that MONOTONE-SAT is NP-complete. Hint: Look carefully at the reduction from SAT to 3-SAT. Alternative hint: Replace $\overline{x}$ by x'. Introduce two new clauses $(x \vee x') \wedge (\overline{x} \vee \overline{x'})$.

(b) MONOTONE-3-SAT is the set of satisfiable 3-CNF formulas in which every clause is monotone. Prove that MONOTONE-3-SAT is NP-complete.

Solution:

(a) MONOTONE-SAT is in NP because we can guess an assignment and check that it satisfies the formula in polynomial time. You could use the hints to reduce SAT to MONOTONE-SAT. But the formula constructed in the proof of the Cook–Levin Theorem is monotone, so we already have a polynomial-time m-reduction from SSS to MONOTONE-SAT. Thus MONOTONE-SAT is NP-complete.

(b) MONOTONE-3-SAT is in NP because we can guess an assignment and check that it satisfies the formula in polynomial time. Each non-monotone clause in a 3-CNF formula has the form $(a \vee b \vee \overline{c})$ or $(\overline{a} \vee \overline{b} \vee c)$. For each clause of the form $(a \vee b \vee \overline{c})$ introduce new variables y and z, and replace the clause by

$$(a \vee b \vee y) \wedge (a \vee b \vee z) \wedge (\overline{y} \vee \overline{z} \vee \overline{c}).$$

For each clause of the form $(\overline{a} \vee \overline{b} \vee c)$ introduce new variables y and z, and replace the clause by

$$(\overline{a} \vee \overline{b} \vee \overline{y}) \wedge (\overline{a} \vee \overline{b} \vee \overline{z}) \wedge (y \vee z \vee c).$$

Thus 3-SAT is $\leq_m^p$-reducible to MONOTONE-3-SAT.

9.6-8 (a) Prove that if Γ is an 8-character alphabet, then $(2,\Gamma)$-SSS is NP-complete. Hint: Reduce 3-SAT, letting each variable in the symbol system represent three variables of the 3-CNF formula.

Solution: First note that it does not matter which 8-character alphabet Γ is. We reduce 3-SAT. Let V be the variable set for a 3-SAT formula. We define a symbol system with variable set $V' = V^3$ and alphabet $\{0,1\}^3$. Think of 1 as true and 0 as false. For each pair of variables $x_1x_2x_3$ and $y_1y_2y_3$ in V' we introduce a constraint which enforces that if x_i and y_j are the same variable in V, then the assignments to $x_1x_2x_3$ and $y_1y_2y_3$ have the same value in the corresponding position. Each clause of a 3-SAT formula can then be encoded as a constraint involving a single variable: Encode the clause $x \vee y \vee z$ as $xyz \neq 000$, encode the clause $\neg x \vee y \vee z$ as $xyz \neq 100$, etc.

(b) Prove that if Γ is a 4-character alphabet, then $(2,\Gamma)$-SSS is NP-complete.

Solution: We modify the preceding solution. Define a symbol system with variable set V^2. For each pair of variables x_1x_2, y_1y_2 introduce constraints to ensure consistency of assignments as above. Encode the clause $x \vee y \vee z$ as $xy \neq 00 \vee yz \neq 00$, encode the clause $\neg x \vee y \vee z$ as $xy \neq 10 \vee yz \neq 00$, etc.

*(c) Prove that if Γ is a 3-character alphabet, then $(2,\Gamma)$-SSS is NP-complete.

Solution: We modify the solution of part (a). Let F be a Boolean formula. We define a symbol system whose variables consist of α, β, and $[\alpha,\beta{>}\alpha]$ for each pair of literals α,β. The symbol-system alphabet is $\{00,01,10\}$. When making assignments to α or β, we identify 0 with 00 and 1 with 01 (this does not enlarge the alphabet). We ensure consistency of assignments via constraints of the form

$$\begin{aligned}&(x \neq 10) \wedge (\overline{x} \neq 10) \wedge (x \neq \overline{x})\\ &\alpha = 1 \iff [\alpha,\beta{>}\alpha] = 10\\ &\beta = 1 \Rightarrow [\alpha,\beta{>}\alpha] \neq 00.\end{aligned}$$

The clause $\alpha \vee \beta \vee \gamma$ is represented by the constraint

$$([\alpha,\beta{>}\alpha] \neq 00) \vee ([\beta,\gamma{>}\beta] \neq 00).$$

9.7-3 Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. G_1 is *isomorphic* to G_2 if there exists a one-one, onto mapping g from V_1 to V_2 such that $(u, v) \in E_1$ if and only if $(g(u), g(v)) \in E_2$. Such a function g is called an *isomorphism* from G_1 to G_2.

(b) A graph $G_1 = (V_1, E_1)$ is a *subgraph* of a graph $G_2 = (V_2, E_2)$ if $V_1 \subseteq V_2$ and $E_1 \subseteq E_2$. Prove that the following problem is NP-complete.

Problem name: subgraph isomorphism

Instance: undirected graphs G and H

Question: Does G contain a subgraph isomorphic to H?

Solution: The problem is in NP because we may guess a subgraph and an isomorphism, then verify them deterministically in polynomial time. We reduce clique. If (G, k) is an instance of the clique problem, let H be a complete graph on k vertices. This also shows that the induced subgraph isomorphism problem is NP-complete.

9.7-4 A weighted graph satisfies the *triangle inequality* if for every three vertices u, v, w the weight of the edge (u, w) is less than or equal to the weight of (u, v) plus the weight of (v, w). Prove that the following problem is NP-complete:

Problem name: TSP with triangle inequality

Instance: a weighted graph G and a natural number c

Question: Does G satisfy the triangle inequality and also have a Hamiltonian cycle whose weighted path length is c or less?

Solution: This is in NP because we can check all triples in time $O(n^3)$, then guess a Hamiltonian cycle and check its cost in polynomial time. To show that the problem is NP-hard, we reduce the ordinary traveling salesman problem. Let (G, c) be an instance of TSP. Determine the largest edge weight w and the number n of vertices in the weighted graph G. Obtain G' by adding w to each of the edge weights in G. Let $c' = c + nw$. Then G' satisfies the triangle inequality because every edge weight is between w and $2w$. Furthermore, the cost of any Hamiltonian cycle is higher by nw in G' than in G. Thus G has a Hamiltonian cycle whose cost is c or less if and only if G' (satisfies the triangle inequality and) has a Hamiltonian cycle whose cost is c' or less.

Note that if we apply this reduction to the graph in the proof of Corollary 9.21, we obtain a graph whose edge weights are all either 1 or 2.

9.7-5 An *independent set* in an undirected graph is a set of vertices, no two of which are connected by an edge. In other words, I is an independent set in $G = (V, E)$ if $I \subseteq V$ and $(I \times I) \cap E = \emptyset$. Prove that the following problem is NP-complete.

Problem name: independent set

Instance: an undirected graph G and a positive integer k

Question: Does there exist an independent set in G consisting of k vertices?

Solution: The independent-set problem is in NP because we can guess k vertices and check that they form an independent set in polynomial time. We reduce clique. Let (G, k) be an instance of the clique problem where $G = (V, E)$. Let E' be the set of nonedges in G, i.e., $E' = \overline{E}$. Let $k' = k$, $V' = V$, and $G' = (V', E')$. Then G contains a clique of size k if and only if G' contains an independent set of size k'. The reduction can be performed in polynomial time, so the independent-set problem is NP-complete.

9.7-7 A *dominating set* in an undirected graph is a set D of vertices such that each of the graph's vertices belongs to D or is adjacent to a vertex in D. In other words, D is a dominating set for $G = (V, E)$ if $D \subseteq V$ and, for all v in V,

$$v \in C \text{ or } (\exists u \in D)[(u, v) \in E].$$

(a) Prove that the following problem is NP-complete:

Problem name: dominating set

Instance: an undirected graph G and a positive integer k

Question: Does there exist a dominating set for G consisting of k vertices?

Hint: Reduce SAT or vertex cover.

Solution: Throughout this problem, it will be implicit that if we say an undirected graph contains an edge (a, b) then it will also contain (b, a).

The dominating-set problem is in NP, because we can guess a set of k vertices and check that it is a dominating set in polynomial time. We reduce SAT. Let F be a Boolean formula with m variables. We construct an instance of dominating set. For each variable x in F, let V contain x, $\overline{x}$, and $[x]$, and let E contains edges connecting those three vertices. Connect each clause to each literal that it contains; i.e., for each clause C and each literal ℓ in C, let E contain the edge (C, ℓ). Let $G = (V, E)$. We assert that F is satisfiable if and only if G has a dominating set of size m. The reduction can be performed in polynomial time, so it remains only for us to prove the assertion.

Suppose that A is a satisfying assignment to F. Let D consist of the m literals x of F such that $A(x) = \text{true}$. Since every clause C is satisfied, it is adjacent to a literal that belongs to D. Furthermore, either x or $\overline{x}$ is in D. If x is in D, then $\overline{x}$ and $[x]$ are adjacent to D; if $\overline{x}$ is in D, then x and $[x]$ are adjacent to D. Thus D is a dominating set in G.

Conversely, suppose that G contains a dominating set D of size m. For each x, D must contain either $[x]$ or an adjacent vertex; therefore D contains $[x]$, x, or $\overline{x}$. Since there are m variables and $|D| = m$, D cannot contain both x and $\overline{x}$. Neither can D contain any clauses. Define $A(x) =$ true if $x \in D$ and false otherwise. If C is a clause, then since C does not belong to D, one of C's literals ℓ must belong to D. Then $A(\ell) =$ true, so C is satisfied by A. Because that is true for every clause, F is satisfied by A.

(c) Prove that the following problem is NP-complete.

Problem name: bipartite dominating set

Instance: an undirected bipartite graph G and a positive integer k

Question: Does there exist a dominating set for G consisting of k vertices?

Solution: We modify the solution to part (a). Replace the triangle $(x, \overline{x}, [x])$ by the hexagon $(x, x^{\mathrm{a}}, \overline{x}, x^{\mathrm{b}}, x^{\mathrm{c}}, x^{\mathrm{d}})$ in the construction of G. Then all simple cycles in G have even length, so G is bipartite. We assert that f is satisfiable if and only if g has a dominating set of size $2m$. Except for noting that a dominating set of size $2m$ must contain exactly two diametrically opposite points from each hexagon, instead of exactly one from each triangle, the proof is the same as in part (a).

Alternate solutions to (a) and (c): We reduce vertex cover. Let (G, k) be an instance of vertex cover where $G = (V, E)$. Let V' consist of three kinds of vertices, which will be on opposite sides of the bipartition:

Type 1: v, for each $v \in V$

Type 2: e, for each $e \in E$

Type 3: a pair of dummy vertices d_1 and d_2

Let E' contain the following edges:

- (v, e) for every $v \in V$ and $e \in E$ such that v is an endpoint of e
- (d_2, v) for every $v \in V$
- (d_1, d_2)

We assert that G has a vertex cover of size k if and only if G' has a dominating set of size $k + 1$.

Suppose that G has a vertex cover C of size k. Then $C \cup \{d_2\}$ is a dominating set in G'.

Conversely, suppose that G' has a dominating set D of size $k + 1$. D must contain either d_1 or d_2 because d_1 has no other neighbor. If D contains d_1, we may replace d_1 by d_2 without increasing the size of the dominating set. Henceforth we assume that D contains d_2 but not d_1. We assert that G' contains a dominating set of size $k+1$ or less that is a

subset of $V \cup \{d_2\}$. Suppose instead that D contains an element $e \in E$. Let $e = (u, v)$. We can replace e by either u or v without increasing the size of the dominating set. This process can be repeated to obtain a dominating set of size $k+1$ or less that is contained in V; call it D'. Then $D' - \{d_2\}$ is a vertex cover of size k or less for G.

9.8-2 **Problem name:** bin packing

Instance: a bag of positive integers S (the sizes of the items to be packed), a positive integer B (a bin size), and a positive integer k (a number of bins)

Question: Do there exist bags $S_1, \ldots, S_k$ such that $S_1 \uplus \cdots \uplus S_k = S$ and $\sum_{x \in S_i} x \leq B$ for $i = 1, \ldots, k$? (Is it possible to pack all the items whose sizes are given by S into k bins of size B?)

Prove that bin packing is NP-complete.

Solution: Equipartition is a special case of bin packing with $k = 2$ and $B = \frac{1}{2} \sum_{x \in S} x$.

9.8-3 **Problem name:** knapsack

Instance: a bag S of ordered pairs (v_i, w_i) of positive integers (interpreted as the value and weight, respectively, of the ith object), a positive integer c (the capacity of the knapsack), and a positive integer g (the goal)

Question: Does there exist $T \subseteq S$ such that the sum of the weights of the objects in T is at most c and the sum of the values of the objects in T is at least g?

Solution: Obviously the knapsack problem is in NP. We reduce subset sum. Let (S, g) be an instance of subset sum. For each $w \in S$, let S' contain (w, w); i.e., each object has a value equal to its weight. Let $c' = g' = g$ so that the problem becomes one of filling the knapsack exactly to capacity g. It is clear that the instance (S', c', g') of knapsack has a solution if and only if the instance (S, g) of subset sum has a solution.

9.8-4 Reduce the vector sum problem for vectors over $\mathbb{N}$ to the subset sum problem. (Do not use the fact that subset sum is NP-complete.)

Solution: Let B be greater than $|V|$ times the maximum component of any vector in V. Proceed as in the proof of Theorem 9.26.

9.8-6 (b) Prove that the following problem is NP-complete.

Instance: a bag of strings B and a goal string g

Question: Does there exist an ordering $\langle\langle s_1, \ldots, s_m \rangle\rangle$ such that $\{s_1, \ldots, s_m\} = B$ and $s_1 \cdots s_m = g$?

Solution: Clearly the problem is in NP. We reduce bin packing with bin and object sizes represented in monadic notation. Consider an instance of the bin-packing problem with b bins having size B and a bag of objects having sizes $x_1, \ldots, x_j$. Let $X = x_1 + \cdots + x_j$.
Let $g = (\#1^B)^b\#$. Let $s_i = 1^{x_i}$ for $i = 1, \ldots, j$, i.e., s_i is the monadic representation of x_i. Let $t_i = 1$ for $i = 1$ to $bB - X$. Let $u_i = \#$ for $i = 1$ to $b + 1$. The goal string g and the bag of strings $\{s_1, \ldots, s_j, t_1, \ldots, t_{bB-X}, u_1, \ldots, u_{b+1}\}$ form an instance of the string concatenation problem that has a solution if and only if it is possible to pack the objects of sizes $x_1, \ldots, x_j$ into b bins having size B.

9.10-1 In each of the exercises below, state the problem formally in terms of languages and determine if the problem is in P, is NP-complete, is co-NP-complete, or is PSPACE-complete.

(b) Determine for DFRs P_1 and P_2 whether $L(P_1) \subset L(P_2)$.
Solution: Let

$$L = \{(P_1, P_2) : P_1 \text{ and } P_2 \text{ are DFRs such that } L(P_1) \subset L(P_2)\}$$

L is in P. In polynomial time, we can determine whether $L(P_1) \subseteq L(P_2)$ as in part (a) and whether $L(P_1) = L(P_2)$.

(e) Determine for regular expressions r_1 and r_2 whether $L(r_1) \subset L(r_2)$.
Solution: Let

$$L = \{(r_1, r_2) : r_1 \text{ and } r_2 \text{ are regular expressions such that } L(r_1) \subset L(r_2)\}.$$

L is PSPACE-complete. L is in PSPACE by the following algorithm: Determine whether $L(P_1) \subseteq L(P_2)$ as in part (d) and whether $L(P_1) = L(P_2)$. L is PSPACE-hard because the special case with $r_2 = \Sigma^*$ is PSPACE-hard.

9.10-3 Say that control states q_1 and q_2 are *equivalent* in an FM program P if P accepts the same strings when started in state q_1 as when stated in state q_2. How hard is it to determine for an FM program P and control states q_1, q_2 whether q_1 and q_2 are equivalent in P?

(a) Assume that P is a DFR.
Solution: This problem is in P. q_1 and q_2 are equivalent if and only if they belong to the same equivalence class of the relation E defined as part of the minimization algorithm in Section 5.7.

(b) Assume that P is an NFA.
Solution: This problem is PSPACE-complete. We reduce from the problem of testing whether two NFAs P_1 and P_2 accept the same language. Let P_j have control set Q_j, initial control state q_j, and instruction

set $\mathcal{I}_j$ for $j = 1, 2$. By renaming control states, we may assume that $Q_1 \cap Q_2 = \emptyset$. We construct P with control set $Q_1 \cup Q_2$ and instruction set $\mathcal{I}_1 \cup \mathcal{I}_2$. The control states q_1 and q_2 are equivalent if and only if P_1 and P_2 accept the same language.

9.10-4 We say that an NFA is an *acyclic* NFA if its state graph is acyclic (contains no cycles).

(b) How hard is it to determine whether two acyclic NFAs accept different languages?

Solution: NP-complete. Note that an acyclic NFA with n states accepts only strings having length n or less. A nondeterministic algorithm may guess a string x of length n or less, deterministically see which of the two programs accept x, and verify that exactly one of them accepts x. Therefore the problem is in NP. To prove NP-hardness, reduce the problem of determining whether two star-free regular expressions generate different languages.

9.10-5 (a) Is there a deterministic polynomial-time algorithm for determining whether a string x is generated by a CNF grammar G?

Solution: Yes. The CYK algorithm and Earley's algorithm both run in time $O(|x|^3|G|)$ on grammars in CNF.

(c) Using part (b), present a deterministic polynomial-time algorithm for determining whether a string x is generated by a regular expression r. Do not use the algorithm presented in this section.

Solution: In polynomial time we can convert a regular expression to an NFA. Then we apply part (b).

9.10-8 (b) Let P be a space-bounded Turing machine program. Prove that the set of valid traces of P is a regular set.

Solution: The class of regular sets is closed under complementation, so this follows from part (a).

(c) Let P be a time-bounded Turing machine program. Prove that the set of invalid traces of P is a regular set.

Solution: Every program that runs in time $t(n)$ also runs in space $t(n)$, so this follows from part (a).

12

Additional Exercises

We provide some exercises that were not included in the textbook. In parentheses, we indicate which chapter the exercise is appropriate for.

*A.1 (Chapter 0) Prove that every planar graph is 5-colorable.

A.2 (Chapter 1 or 4) Construct a DFR that recognizes the set of strings that contain aabaabab as a substring.

Note: You might request a minimal DFR.

A.3 (Chapter 2) Design a finite transducer that maps strings of the form $\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k$ to $\mathtt{a}^i\mathtt{b}^{j+k}$ and does not accept any other inputs.

A.4 (Chapter 3) A left stack holds a string like an ordinary stack. But you push or pop from the left end, instead of the right.

(a) Define the left-stack operations using the $\rightarrow$ notation.

(b) Define the left-stack operations formally.

(c) Prove that an ordinary stack can simulate a left stack.

(d) Prove that a left stack can simulate an ordinary stack.

A.5 (Chapter 3) In Section 3.4.4 we showed how to construct a program P' by eliminating null instructions from P. Prove that P' simulates P via subprograms.

A.6 (Chapter 4) We define the outer-thirds operator OT($\cdot$) as follows: for every string xyz such that $|x| = |y| = |z|$, OT(xyz) $= xz$. Extend OT($\cdot$) to languages in the standard way. Is the class of regular languages closed under OT($\cdot$)?

13

Take-Home Final Exam

F.1 The derivative is a kind of quotient on the left. More precisely, if R and S are languages, the *derivative* of R by S is $\{x : (\exists y \in S)[yx \in R]\}$. We write $D_S(R)$ to denote the derivate of R by S. Prove that the class of CFLs is closed under derivative by regular languages.

Solution: The class of CFLs is closed under finite transductions. Thus it suffices to construct a finite transducer whose transfer relation τ satisfies $L\tau = D_S(L)$ for all languages L. Let P_S be an NFA for S. The desired transducer behaves like P_S, but when an accepting state of P_S is reached the transducer may either continue like P_S or nondeterministically move to a new state in which it copies the remaining input to the output device and then accepts.

Alternative solution: Let L be a CFL and S a regular language. Then L^R is a CFL and S^R is regular, so L^R/S^R is a CFL; therefore $\left(L^R/S^R\right)^R$ is a CFL. But

$$\begin{aligned} D_S(L) &= \{x : (\exists y \in S)[yx \in L]\} \\ &= \{x : (\exists y \in S)[(yx)^R \in L^R]\} \\ &= \{x : (\exists y \in S)[x^R y^R \in L^R]\} \\ &= \{x : (\exists y \in S^R)[x^R y \in L^R]\} \\ &= \{x^R : (\exists y \in S^R)[xy \in L^R]\} \\ &= \{x : (\exists y \in S^R)[xy \in L^R]\}^R \\ &= \left(L^R/S^R\right)^R. \end{aligned}$$

Thus the class of CFLs is closed under derivative.

F.2 (a) Let $A = \{\mathrm{a}^{2^k} : k \geq 0\}$. Prove that A is not a CFL.

A.18 (various) Recall from Exercise 6.4-8(b) that MIN(L) is the set of strings x in L such that no proper prefix of x is also in L. Which of the following classes are closed under MIN($\cdot$)?

(a) the regular languages

(b) the context-free languages

(c) P

(d) NP

(e) the recursive languages

(f) the r.e. languages

A.7 (Chapter 5) The FIRST-HALF$(\cdot)$ operator was defined in Exercise 4.10-3(a). Is the class of context-free languages closed under FIRST-HALF$(\cdot)$?

A.8 (Chapter 6) Prove that $\{a^i b^j c^k : i = 2j \text{ or } j = 3k\}$ is inherently ambiguous.

Note: You should have the students read Section 6.3 before assigning this.

A.9 (Chapter 7) Let f be a 1-1 HG-computable function. Write an HG program that computes f^{-1}.

A.10 (Chapter 8) Construct a pair of recursively inseparable r.e. index sets.

A.11 (Chapter 8)

A.12 Let S and T be disjoint index sets. Prove that S and T are recursively separable iff S is empty or T is empty.

A.13 (Chapter 9) Is the class of NP-complete languages closed under

(a) intersection?

(b) union?

A.14 (Chapter 9) Is P closed under

(a) union

(b) intersection

(c) complementation

(d) concatenation

(e) Kleene star

A.15 (Chapter 9) Prove that if $k = 1$ and $|\Gamma| = 1$ then (k, Γ)-SSS is in P.

A.16 (Chapter 9) Justify the claims about T_3, T_4, and S_5 made in Section 9.9.

A.17 (various) Which of the following classes are closed under finite transductions?

(a) the regular languages

(b) the CFLs

(c) the DCFLs

(d) P

(e) NP

(f) the recursive languages

(g) the r.e. languages

(b) Let $\text{MON}(L) = \{1^{\text{binary}(x)} : x \in L\}$ (for example, $\text{MON}(\{101, 110\}) = \{11111, 111111\}$).

Is the class of regular languages closed under MON?

(c) Is the class of CFLs closed under MON?

Solution:

(a) Assume for the sake of contradiction that A is a CFL. Let N be the number from the first pumping theorem for CFLs, and let $z = \mathtt{a}^{2^N}$. Since $z \in A$ and $|z| \geq N$, there exist u, v, w, x, y such that

- $z = uvwxy$,
- $v \neq \Lambda$ or $x \neq \Lambda$,
- $|vwx| \leq N$, and
- $(\forall i)[uv^iwx^iy \in L]$.

Then $vx = \mathtt{a}^k$ for some k where $1 \leq k \leq N$. Therefore $uv^2wx^2y = \mathtt{a}^{2^N+k}$. But

$$2^N < 2^N + k \leq 2^N + N < 2^N + 2^N = 2^{N+1},$$

so $2^N + k$ is not a power of 2 and $uv^2wx^2y \notin A$. This contradiction proves that A is not a CFL.

(b–c) Let $B = \{1^{2^k} : k \geq 0\}$ and let τ be a finite transduction that replaces 1's by a's. Then $B\tau = A$. Since A is not a CFL, B is not a CFL either. Let $L = 1^*0$, which is regular and therefore a CFL. But $\text{MON}(L) = B$, which is not a CFL and therefore not regular. Thus the class of regular languages in not closed under $\text{MON}(\cdot)$ and the class of CFLs is not closed under $\text{MON}(\cdot)$.

F.3 Let $\text{MIN}(L)$ be the set of strings x in L such that no proper prefix of x is also in L, i.e., $\text{MIN}(L) = L - (L\Sigma^+)$. For example, $\text{MIN}(\{\mathtt{a}, \mathtt{ab}, \mathtt{bb}\}) = \{\mathtt{a}, \mathtt{bb}\}$. Prove that the class of DCFLs is closed under $\text{MIN}(\cdot)$.

Solution: Let P be an on-line DSR that recognizes S. Modify P to remember in an extra control whether a proper prefix of the input belongs to S. If none of them does but the entire string belongs to S, then accept; otherwise reject.

Note: this is exercise 6.4-8(b).

F.4 (a) Prove that there is no DTR that takes as input a CFG G, accepts if $L(G) = \Sigma^*$, and rejects if $L(G)$ is any other co-finite language (if $L(G)$ is not co-finite, the DTR may accept or reject G; we don't care which).

(b) Prove that there is no DTT that takes as input a CFG G and outputs a regular expression r such that if $L(G)$ is regular then $L(G) = L(r)$.

Solution:

(a) For the sake of contradiction, assume that there is a DTR D that can distinguish between $L(G) = \Sigma^*$ and $L(G)$ being some other co-finite language. We will show how to solve the halting problem for inattentive deterministic 2-counter machines programs. Let P be an inattentive deterministic 2-counter machine. From P, we can construct a grammar G that generates the set of noncomputations of P. If P halts, then, since P is deterministic, P has a unique computation C, so $L(G) = \Sigma^* - \{C\}$. Otherwise, $L(G) = \Sigma^*$. By running D, we can determine which of those two cases holds, and thus determine whether P halts. Since the halting problem for inattentive deterministic 2-counter machine programs is undecidable, this is a contradiction. Thus there is no such DTR.

(b) If such a DTT T exists, then it produces a regular expression equivalent to G in particular when G is co-finite. We use T to solve the problem from part (a) as follows:

input G;
$r := T(G)$;
if $L(r) = \Sigma^*$ then accept else reject;

Because we have an algorithm to test equivalence of regular expressions, there is no problem in implementing the last line above. Because there is no algorithm to solve the problem in part (a), no such DTT can exist.

F.5 Recall that a language L is *complete* if L is r.e. and every r.e. language is m-reducible to L. Is the class of complete languages closed under

(a) intersection

(b) union

Solution:

(a) No. For $c = 0, 1$, let $S_c = \{cx : x \in K\}$. Then S_c is r.e. because we can check that the input is of the form cx and then run program x on input x to see whether it halts. In addition, $K \leq_m S_c$ via the reduction $f(x) = cx$. Thus S_c is complete. But $S_0 \cap S_1 = \emptyset$, which is not complete.

(b) No. For $c = 0, 1$, let

$$S_c = \{cx : x \in K\} \cup \{y : \text{the first character of } y \text{ is not a } c\}.$$

Then S_c is r.e. because we can accept if the input is not of the form cx and otherwise run program x on input x to see whether it halts. In addition, $K \leq_m S_c$ via the reduction $f(x) = cx$. Thus S_c is complete. But $S_0 \cup S_1 = \Sigma^*$, which is not complete.

F.6 Is NP closed under

(a) union

(b) intersection

(c) concatenation

(d) Kleene star

Solution: Let L_1 and L_2 be two languages in NP, accepted in polynomial time by NTA's P_1 and P_2 respectively.

(a) Yes. To test membership in $L_1 \cup L_2$, nondeterministically choose $i \in \{1,2\}$ and run P_i; accept if P_i accepts.

(b) Yes. To test membership in $L_1 \cap L_2$, run P_1 and then run P_2; accept if they both accept.

(c) Yes. To test membership in $L_1 L_2$, nondeterministically guess strings y and z such that $x = yz$, run P_1 on y and P_2 on z, and accept if they both accept.

(d) Yes. To test membership in L_1^*, nondeterministically guess $k \leq |x|$ and strings $x_1, \ldots, x_k$ such that $x = x_1 \cdots x_k$, and then run P_1 on each of those strings x_i; accept if P_1 accepts every x_i.

Alternative solution: In parts (a,c,d), use the construction we used in order to prove the corresponding closure property for regular languages. In part (b), use a pairing construction.

F.7 Let Restricted-SAT be the set of satisfiable CNF formulas in which each variable appears in at most 3 clauses.

(a) Give a polynomial-time m-reduction from SAT to Restricted-SAT.

(b) Give a polynomial-time m-reduction from Restricted-SAT to SAT.

(c) Prove that Restricted-SAT is NP-complete.

Solution:

(a) Let us be given a formula F with k clauses. For each variable x introduce new variables $x_1, \ldots, x_k$, replace the ith the occurrence of x by the variable x_i, and introduce new clauses $(\overline{x} \vee x_1)(\overline{x}_1 \vee x_2) \cdots (\overline{x}_{k-1} \vee x_k)(\overline{x}_k \vee x)$. Call the new formula F'. This reduction can be carried out in polynomial time.

In the formula F', each variable appears 2 or 3 times. If F is satisfied by an assignment $A()$, then let $A'(x) = A'(x_i) = A(x)$, so A' satisfies F'. If F' is satisfied by an assignment A', then the new clauses force $A'(x) = A'(x_i)$; therefore $A(x) = A'(x)$ satisfies F.

(b) Let $f(F) = (x)(\overline{x})$ if a variable appears in more than 3 clauses in F; $f(F) = F$, otherwise. f is the desired reduction.

(c) By (a) Restricted-SAT is NP-hard because SAT is NP-hard. By (b) Restricted-SAT is in NP because SAT is in NP. Therefore Restricted-SAT is NP-complete.

1 Exercise numbering

There were some errors in the exercise numbering so some pairs of exercises are numbered identically. You may want to include the page number if you assign any of them. Exercises will be renumbered as follows in the second printing and in the instructor's manual.

Pages 64–65 The exercises should be numbered 0.6-24 through 0.6-30.

Page 139 The exercises should be numbered 2.7-2 through 2.7-5.

Page 194 The exercises should be numbered 3.3-10 through 3.3-12.

Page 453 The exercises should be numbered 7.1-7 and 7.1-8.

2 Other errata

Page 31 According to Kleene's obituary in the *New York Times*, his name was pronounced CLAY-nee.

Page 70 Figure 1.1 should have SCANb/EMPTY instead of b/EMPTY on the transition $2 \rightarrow 4$.

Pages 70, 145, 146 Figures 1.1, 2.10, and 2.11 should contain a loop on state 4 labeled with SCANa, SCANb, and SCAN# in order to exhaust the input before rejecting.

Page 84 The states in Figure 1.10 should be labeled as in the solution to Exercise 2.7-1.

Page 123 In Table 2.1, we say that the repertory of a control is $Q \times Q$. But the repertory is $\{\{(i,j)\} : i \in Q \text{ and } j \in Q\}$, whereas $Q \times Q$ is $\{(i,j) : i \in Q \text{ and } j \in Q\}$, so they are technically different.

Page 142 In the last line, replace "all computations of the program on x are blocked or infinite" by "the program must block or run forever on input x"

Pages 145, 146 Figures 2.10 and 2.11 are missing SCAN# on the transition $2 \rightarrow 4$.

Page 171 The reference to Figure 3.2.1 should be to Figure 3.15.

Page 174 On the next to last line, $(\nu \rightarrow 3, \text{NOOP}, \text{INC})$ should be $(\nu \rightarrow 2, \text{NOOP}, \text{INC})$

Page 201 In Figure 3.39, the arc labeled EOF going down from state q_1 should be labeled $\mathbf{z} \rightarrow \mathbf{z}$.

Page 203 The reference to Section 5.4.1 should be to Section 4.1.

Page 207 In the figure caption, the reference to Exercise 3.4-9 should be to Exercise 3.4-8.

Page 209 The statement "P' simulates P" should be "P' is equivalent to P."

Page 214 In Exercise 3.4-19, the reference to Exercise 3.4-9 should be to Exercise 3.4-8.

Page 220 Exercises 4.1-1 and 4.1-2 should be marked with a "+" since they are used later.

Page 247 The reference to Figure 4.5 should be to Figure 4.19.

Page 267 In Exercise 4.7-15, replace "prefix equivalent" by "equivalent states."

Page 276 Exercise 4.7-18(c) should refer to the DFA in Figure 4.34, (not the DFR in Figure 4.7-18).

Page 288 In the first line of the proof of Corollary 4.46, replace "Let τ_1 and τ_2 be programs" by "Let τ_1 and τ_2 be transfer relations of programs"

Page 336 In the solution to Exercise 5.3-2, G's alphabet includes the symbols for empty set and empty string as well.

Page 338 In the solution to Exercise 5.3-4, the last occurrence of $L(G)$ should be $L(G')$.

Page 366 Exercise 5.8-3(h), which says to prove that the set of composites in decimal is not a CFL, is actually an open problem.

Pages 370 The first reference to Figure 5.30 should be Figure 5.13. The second reference to Figure 5.30 should be to Figure 5.14.

Page 371 In the grammar, replace $S \rightarrow T$ by the pair of productions

$$\begin{aligned} S &\rightarrow \texttt{statement}, \\ S &\rightarrow \texttt{if } C \texttt{ then } T \texttt{ else } S \end{aligned}$$

for consistency with Figure 5.15.

Page 372 The reference to Figure 5.30 should be to Figure 5.15.

Page 421 In Example 6.17, the reference to Figure 6.17 should be to Figure 6.7.

Pages 452, 453 When simulating a tape by two stacks, the instruction (MOVER, f) in P requires additionally in P' the instructions (PUSH $c, c \to \sqcup$, EMPTY, f), for all $c \in \Gamma$. An extra line should be added to Table 7.1.

Page 453 In Exercise 7.1-7, replace "construct" by "construct informally."

Page 455 In the second paragraph under the heading 3-CMs, in the last line replace "dyadic" by "k-adic."

Page 489 In the statement of Theorem 7.27, we should assume that every program in S computes a total recursive function.

Page 495 In the proof of Theorem 7.37, there is no need to replace blocking by infinite loops. Blocking does not count as halting.

Page 513 Relax the definition of underhang $(\cdot)$ to apply to any sequence of pairs, not necessarily beginning with the starting pair.

Pages 521, 523, 526 In Section 7.12, we must permit quantification over variables (but not over functions or predicates). Include inference rules for quantifiers.

In Table 7.5, the quantifier $\forall i$ should be added to each entry that involves the variable i, and the quantifier $\forall j$ should be added to each entry that involves the variable j. In Figure 7.5, the first line should start with $(\forall i)(\forall j)$.

Pages 529, 532, 533, and 534 Change "DSR language" to "DCFL."

Pages 540, 541 In Exercise 7.14-4 parts (a–b), g should have m parameters, not n.

Page 560 In both Step 1's, change "run φ_P on input P" to "run P on input P, without producing any output"

Page 570 Point (vi) should read "A theory of arithmetic is sound if all theorems are true."

Page 576 In Figure 8.1, replace A by B in the edge labels.

Page 589 In the 15th and 16th lines, i should be $i - 1$.

Page 602 The definition of NTIME() is not quite right. If a string is not accepted, there is no computation at all, hence no shortest computation. So the running time is undefined.

Instead, define running time as the length of the longest partial computation. This is infinite for programs that don't block or halt.

Acknowledgments. Thanks to Emile Roth, Jeffrey Oldham, Vaughan Pratt, and Daniel Leivant for pointing out our mistakes.

Instructor's Manual to Accompany *The Language of Machines: An Introduction to Computability and Formal Languages*

Richard Beigel

The single theoretical model of Floyd and Beigel's *The Language of Machines* encompasses all the traditional types of computing machines and even "real life" electronic computers, providing a framework on which students can build a rich and enduring body of knowledge. No other text offers this accessible, easy-to-learn-from approach. And to help instructors make the material even more accessible, the Instructor's Manual to Accompany *The Language of Machines* provides stimulating ideas for lectures and classwork. Organized into ten topics, each broken down into 50-minute lectures, the Manual includes a list of exercises and recommended reading for each topic, as well as additional exercises, solutions to exercises, and a take-home exam (with solutions).

CONTENTS

Computer Science Press
An imprint of W. H. Freeman and Company
41 Madison Avenue, New York, NY 10010
20 Beaumont Street, Oxford OX1 2NQ, England