

Models of Computation

—DRAFT—

Nachum Dershowitz

January 20, 2008

(c) 2006–8

(NOT FOR DISTRIBUTION)

Contents

1	Introduction	6
1.1	The Berry Paradox	6
1.2	The Busy Beaver	7
1.3	Bigger and Bigger	9
1.4	A Baby Programming Language	9
2	Transition Systems	10
2.1	State Transition Systems	10
2.2	Labelled Transition Systems	12
3	Formal Languages	13
4	Automaton Languages	15
4.1	Finite State Automata	15
4.2	Automaton Languages	16
4.3	Acceptance	16
4.4	Transducers	17
5	Nondeterminism	18
5.1	Angels and Demons	18
5.2	Nondeterministic Finite State Automata	19
5.3	Transducers	21
6	Closure Operations	22
6.1	Set Operations	22
6.2	Word Operations	24
6.3	Kleene's Theorem	25

7	Regular Languages	26
7.1	Regular Expressions	26
7.2	Regular Languages	27
8	Question Answering	30
8.1	Emptiness	30
8.2	Fullness	31
8.3	Equivalence	31
8.4	Minimization	32
9	Non-Regular Languages	34
9.1	Pumping Lemma	34
9.2	Beyond Finite Automata	35
9.3	Summary	36
10	Grammars	38
10.1	Generational Grammars	38
10.2	Context-Free Languages	39
10.3	Closure Properties	40
10.4	Linear Grammars	40
10.5	Pumping Lemma	40
10.6	Problems	41
11	Computability	42
11.1	Computable Languages	42
11.2	Computable Functions	43
11.3	Comparing Power	44
11.4	Fast Growing Functions	45
12	Descriptive Complexity	46
12.1	Kolmogorov Complexity	46
12.2	Incomputability	47
13	Interpretation	48
13.1	Interpreters	48
13.2	Compilers	48
13.3	Partial Evaluation	48
13.4	Checkers	49
13.5	Recursion Theorem	49

14 Decision Problems	51
14.1 Questions and Answers	51
14.2 Decision Problems	52
14.3 Semi-Decidability	53
14.4 Standard Models	55
15 Reductions	56
15.1 General Reductions	56
15.2 Rice’s Theorem	57
16 Enumerability	58
16.1 Recursive Enumerability	58
16.2 Mapping Reductions	58
16.3 Oracles	59
17 Turing Machines	60
17.1 Example	61
17.2 Universal Turing Machines	61
17.3 Finite State Automata Revisited	62
17.4 Context-Free Languages Revisited	62
17.5 Variations	63
18 Church-Turing Thesis	67
18.1 Turing Programs	67
18.2 Counter Machines	70
18.3 RAM Machines	72
18.4 Arbitrary Grammars	74
18.5 Equipotence	74
18.6 The Church-Turing Thesis	75
A The Baby Language	76

Preface

I thank Udi Boker, Matan Kalman, Ido Kasher, and Ricky Rosen for their comments.

Chapter 1

Introduction

Put the right kind of software into a computer,
and it will do whatever you want it to.

There may be limits on
what you can do with the machines themselves,
but there are no limits on
what you can do with software.

—*Time Magazine* (April 1984)

The opening quote is correct from the point of view that resources place limits on what can be computed. But we will see that there are very many problems that no software program can solve, even when provided with unlimited resources.

1.1 The Berry Paradox

God made the integers; all else is the work of man.

—Leopold Kronecker

Riddle. What is the smallest natural number that cannot be described in a dozen words?

By “natural” number, we mean a member of the set $\mathbf{N} = \{0, 1, 2, \dots, 11, \dots\}$.

1.2 The Busy Beaver

A more relevant question for computer science is, for each program size n , what is the largest integer that can be computed by a program with at most n characters (and no input). This is called *The Busy Beaver Problem*.

For example, the largest number we can get out of an 11-character Scheme program is 29512665430652752148753480226197736314359272517043832886063884637676943433478020332709411004889, produced by the size 11 program (expt 9 99), which makes use of the built-in exponentiation program `expt`.

Consider an arbitrary programming language \mathbf{L} . Let $|p|$ denote the size (number of characters) of a program $p \in \mathbf{L}$, and let $[p(x)]$ denote the output of running p on input x . Let \mathbf{L}_0 denote programs of \mathbf{L} that have numeric output (in \mathbf{N}), but no input. For example, there is a program in \mathbf{L}_0 , call it p_{99} , that computes $99!$. That is, $[p_{99}()] = 93326215443944152681699238856266700490715968264381621468592963895217599993229915608941463976156518286253697920827223758251185210916864000000000000000000000$.

So, once we have fixed the set of numerical programs \mathbf{L}_0 , we can define precisely the mathematical “Busy Beaver Function”:

$$bb(n) = \max\{[p()] : p \in \mathbf{L}_0, |p| \leq n\}$$

This function is perfectly well defined. For example, in Scheme $bb(1) = 9$, since 9 is valid code and returns the value 9. Similarly, if \mathbf{L} is Scheme, then $bb(11)$ is the 94-digit number 29512... Since we can program $99!$ in Scheme with a program of length 57, $bb(57)$ is at the very least $99!$, and actually is much, much bigger.

Theorem 1 *There is no Scheme program that computes the function bb for all values of n .*

Proof. The proof proceeds by contradiction. We show that any program for bb fail for some inputs.

1. Suppose that some program $b \in$ Scheme purports to compute bb faithfully.
2. It takes at least 10 symbols to define any function in Scheme, so let $N = |b| \geq 10$ be the size of this program.
3. Consider the Scheme program

```
(define (c)
  (define (b n) ...)
  (+ 1 (b (* N 5))))
```

4. The size of c (not counting unnecessary spaces) is

$$|c| = |b| + |N| + 27$$

where $|N|$ is the number of symbols needed to actually write the number N .

5. $\lceil \log_{10} N \rceil \leq N$ characters suffice to write out the number N . For example, it takes 4 (decimal) digits to write $N = 1000$.
6. Since $|N| \leq N$ and $N \geq 10$, we have

$$|c| \leq 2N + 27 \leq 5N$$

7. It follows by the definition of the Busy Beaver function that

$$bb(5N) \geq [c()]$$

8. The way the program is written, we have

$$[c()] = [b(5N)] + 1$$

9. It follows that

$$bb(5N) > [b(5N)]$$

10. Contradiction!

So, either b diverges (does not terminate) on $5N$, and returns no answer at all, and therefore $c()$ also diverges and has no impact on the value of $bb(N)$, or else $b(5N) < bb(5N)$, and b underestimates the true value of bb . In either case, b is a failure (for input $5N$ at least). \square

The same kind of proof works for any programming language.¹

The moral of the story is that there are perfectly good functions (like the Busy Beaver) that cannot be computed by any program, no matter how big and complicated it is.

¹The Busy Beaver problem is usually stated in terms of the number of states of a Turing Machine, a very simple programming device. See Chapter 17. For more on the subject, see: <http://grail.cba.csuohio.edu/~somos/bb.html>; <http://www-csli.stanford.edu/hp/Beaver.html>; <http://www.logique.jussieu.fr/~michel/ha.html>; <http://www.cs.rpi.edu/~kelleo/busybeaver>.

Problem. Show that the function giving the smallest natural number that cannot be computed with a program of size n is also not computable.

1.3 Bigger and Bigger

Suppose we ask only for a function $bb_k(n)$ such that $bb_k(n) = bb(n)$ for all $n \leq k$. We don't care what the program does for larger inputs. What can we say about the programmability of bb_k ?

Again, suppose that some program b computes bb_k , and let $N = |b| \geq 10$, as before. The size of

```
(define (c) (define (b n) ...) (+ 1 (b (* N 5))))
```

is bounded by $5N$, so $bb(5N) > b(5N)$. So, assuming b works as advertised, it must be that $5N > k$. In other words, to compute bb for size k programs, we need a program that is at least 20% of that size. It follows that no finite program can compute bb for *all* n .

So, either b diverges (does not terminate) on $5N$, and returns no answer at all, or else $b(5N) < bb(5N)$. In either case, b is a failure (for input $5N$ at least).

Comment. It should be obvious to anyone who studied even a smattering of set theory, that there are many such uncomputable numeric functions, since there are countably many (\aleph_0) programs and uncountably many more (\aleph) functions from \mathbf{N} to itself.

1.4 A Baby Programming Language

We are all accustomed to numeric functions $f : \mathbf{N} \rightarrow \mathbf{N}$. For example, the factorial function may be defined recursively, as follows:

$$n! := \begin{cases} 1 & n = 0 \\ n \cdot (n - 1)! & \text{otherwise} \end{cases}$$

This function can be programmed in all standard programming languages.

We will use this programming style, not just for numerical programs, but also for programs operating over other data types.

This *Baby* language is a subset of Scheme—see Appendix A, except that evaluation is always normal order.

Chapter 2

Transition Systems

Man/Wolf/Goat/Cabbage Problem The following puzzle is over a thousand years old:¹

- Consider a man with a wolf, goat and (big) cabbage on the left bank of a river.
- All must get to the right bank of the river.
- There is a boat large enough to carry only two items, including the man, who is the only one of the quartet who knows how to row.
- If the wolf and goat are left alone, unsupervised, the wolf will eat the goat.
- If the goat and the cabbage are left alone, the goat will eat the cabbage.
- Is it possible for all to get across the river safely?
- If so, how?

2.1 State Transition Systems

Programs can be modelled as “state transition systems”, where the *state* is a “snapshot” of all the data, including all that is needed to determine how the computation proceeds, and the *transition relation/function* determines how the computation proceeds, step-by-step.

¹A River-Crossing Problem in Cross-Cultural Perspective, Marcia Ascher, Mathematics Magazine, Vol. 63, No. 1 (Feb., 1990), pp. 26–29. For fun, see <http://perso.orange.fr/jeux.lulu/html/anglais/loupChe/loupChe1.htm>.

Definition 1 A (state) transition system (STS) consists of

- a (finite or infinite) set Q of states;
- a set of input (initial) states $I \subseteq Q$;
- a set of output (final) states $F \subseteq Q$; and
- a binary relation $\delta \subseteq Q \times Q$.

States can be anything: the contents of a pot of soup; the writing on a blackboard; the lines and arcs on a piece of paper; the state of affairs in the river-crossing problem. For the river problem, we can represent the state as a pair of sets $\langle L, R \rangle$, where $L, R \subseteq \{\text{man, wolf, goat, cabbage}\}$ denote what is currently on the left or right bank, respectively. (Man and boat are always on the same side.) For example, $\langle \{\text{man, wolf, goat}\}, \{\text{cabbage}\} \rangle$ has the cabbage alone on the right bank, and all else on the left.

A binary relation like δ can be viewed as a multivalued (partial) function, $\delta : Q \rightarrow \mathcal{P}(Q)$, where $\delta : q \mapsto \{r \in Q : q\delta r\}$. We can write $q \rightarrow_\delta r$, or just $q \rightarrow r$, when $r \in \delta(q)$.

Definition 2 A transition system is deterministic if δ is a function.

Definition 3 A computation is a finite or infinite sequence

$$s_0 \rightarrow s_1 \rightarrow \dots$$

such that

- $s_0 \in I$; and
- $s_i \delta s_{i+1}$ for all i .

Definition 4 A finite computation

$$s_0 \rightarrow s_1 \rightarrow \dots \rightarrow s_n$$

is terminating if $s_n \in F$.

In the river problem, we are looking for a terminating computation of the form

$$\langle \{\text{man, wolf, goat, cabbage}\}, \emptyset \rangle \rightarrow \dots \rightarrow \langle \emptyset, \{\text{man, wolf, goat, cabbage}\} \rangle$$

2.2 Labelled Transition Systems

Let's forget about output for a while, and imagine that a program's input is processed letter by letter, symbol by symbol. This way, the program need not store the whole input text unnecessarily.

A *labelled transition system (LTS)* is like a plain transition system, except that the relation δ incorporates labels.

Definition 5 A labelled transition system (LTS) consists of

- a finite alphabet Σ of symbols or “actions”;
- a (finite or infinite) set Q of states;
- a set of input (initial) states $I \subseteq Q$;
- a set of output (final) states $F \subseteq Q$; and
- a ternary relation $\delta \subseteq Q \times \Sigma \times Q$.

Again, we can view δ as a binary multivalued function: $\delta(q, a) \mapsto \{r \in Q : (q, a, r) \in \delta\}$. We can think of the $a \in \Sigma$ as an input symbol, and of $\delta(q, a)$ as the possible continuation states, given that the system is in state $q \in Q$ and the next input symbol is a .

An LTS is *finite* when Q is.

Real computers are finite; programs in many programming languages are not.

Chapter 3

Formal Languages

By an *alphabet*, we just mean a set of symbols. For example,

$$\Sigma = \{\alpha, \beta, \gamma, \delta, \epsilon, \zeta, \eta, \theta, \iota, \kappa, \lambda, \mu, \nu, \xi, \omicron, \pi, \rho, \sigma, \tau, \upsilon, \phi, \chi, \psi, \omega\}$$

is the alphabet consisting of the lower-case Greek letters. Symbols may be complex objects, like Egyptian and Mayan hieroglyphics, or Chinese ideograms, but usually alphabets contain only finitely many symbols.

A *word* or *string* is a finite sequence of zero, one, or more symbols of an alphabet Σ . Though alphabets are normally finite, they can be used to make infinitely many words. For example, $\alpha\gamma\rho\rho\alpha$ is a Greek word. Words, in this sense, need not be meaningful: both “syzygy”, “zygyty”, and “zigity” are all words over the Latin alphabet.

It is customary to use the symbol ε (“epsilon”) for the empty word (consisting of zero symbols). If u, v are words, then uv denotes the word obtained by *concatenating* them. Clearly $w\varepsilon = \varepsilon w = w$ for any word w .

We use Σ^* to denote the set of all words over Σ . It is always the case that $\varepsilon \in \Sigma^*$.

A (*formal*) *language* L over an alphabet Σ is a subset of its words: $L \subseteq \Sigma^*$.

Since (formal) languages are sets, we can use all the usual set operations on them. So, if K and L are languages over some alphabet Σ , then so are their union $K \cup L$ and intersection $K \cap L$. The empty language is denoted \emptyset .

Nota bene. The empty word ε , the empty language \emptyset , and the (nonempty) language $\{\varepsilon\}$ containing only the empty word are three very distinct things, and should never, ever be confused.

We say that a program P (in some programming language) *accepts* a language L if $P(w)$ returns *true*, for all words $w \in L$. For example, the following programs accept the empty language (\emptyset), the language containing only the empty word ($\{\varepsilon\}$), the language containing all but the empty word ($\Sigma^* \setminus \{\varepsilon\}$), and the language containing all words (Σ^*), respectively:

$$\begin{aligned} \text{Empty}(w) &:= \text{false} \\ \text{Almost-Empty}(w) &:= w = \varepsilon \\ \text{Almost-Full}(w) &:= w \neq \varepsilon \\ \text{Full}(w) &:= \text{true} \end{aligned}$$

Definition 6 (Concatenation) *If $K, L \subseteq \Sigma^*$ are languages, then the language*

$$K \cdot L = \{uv : u \in K, v \in L\}$$

is their concatenation.

We can drop the dot and just juxtapose for concatenation, as in KL .

We can abbreviate concatenations of the same language with itself by using exponents: $L^2 = LL$, $L^3 = LLL$, etc. Define $L^0 = \{\varepsilon\}$ and $L^1 = L$.

Definition 7 (Kleene Star) *If $L \subseteq \Sigma^*$ is a language, then*

$$\begin{aligned} L^* &= \{u_1u_2 \cdots u_n : u_i \in L\} \\ &= \bigcup_{i \geq 0} L^i \\ &= \{\varepsilon\} \cup L \cup L^2 \cup L^3 \cup \dots \end{aligned}$$

is its Kleene star language.

Chapter 4

Automaton Languages

The finite case of labelled transition systems is especially worth studying. A *finite automaton* is an LTS with finite alphabet ($|\Sigma| < \infty$), finitely many states ($|Q| < \infty$), functional transitions ($\delta : Q \times \Sigma \rightarrow Q$), and one initial state ($I = \{q_0\}$).

4.1 Finite State Automata

Definition 8 A (deterministic) finite (state) automaton (DFA) *consists of the following components:*

- *finite alphabet* Σ ;
- *finite state set* Q ;
- *transition function* $\delta : Q \times \Sigma \rightarrow Q$;
- *initial state* $q_0 \in Q$; and
- *accepting states* $F \subseteq Q$.

In programs, we would package these five components into a tuple, or record:

$$\langle \Sigma, Q, \delta, q_0, F \rangle$$

For example, we can have a two-state automaton, that switches from one to the other whenever any letter is received as input. Then, after any even number of letters, the automaton is back in the initial state; after an odd number it is in the other state. If the initial state is also the accepting

state, then when the input ends, the automaton is in an accepting state if and only if the input is of even length.

The transition function can be given as a finite table, where the rows are states $q \in Q$, the columns are symbols $a \in \Sigma$, and the entries contain states $\delta(q, a)$.

A sequence $a_1 a_2 \dots a_{n-1} a_n$ ($n \geq 0$) of input symbols, taken from an alphabet Σ , is *accepted* by an automaton A operating over Σ if (and only if) $\delta(\delta(\dots \delta(\delta(q_0, a_1), a_2), \dots, a_{n-1}), a_n) \in F$.

If we write $\delta_a(q)$ instead of $\delta(q, a)$, then this reads: $\delta_{a_n}(\delta_{a_{n-1}}(\dots \delta_{a_2}(\delta_{a_1}(q_0)) \dots)) \in F$.

Finite automata can be drawn as edge-labelled graphs. Then, a word is accepted if there is a path (sequence of edges) from the start state to a final state, the labels of which contain the letters of the word in sequence.

4.2 Automaton Languages

Definition 9 *The language $L(A)$ of an automaton A is the set of all words accepted by A .*

Definition 10 *A language is finite if only finitely many words belong to it ($|L| < \infty$).*

Definition 11 *A formal language is an automaton language if there is a finite state automaton that accepts it.*

We will see an alternative definition of the class of automaton languages in Chapter 7, which will justify the alternate term, “regular languages”, for this class.

Theorem 2 *All finite languages are automaton languages.*

4.3 Acceptance

The following program tests whether a finite automaton $A = \langle \Sigma, Q, \delta, q_0, F \rangle$ accepts a word $w \in \Sigma^*$:

$$\begin{aligned} \text{accept?}(w, \langle \Sigma, Q, \delta, q_0, F \rangle) &:= \widehat{\delta}(q_0, w) \in F \\ \text{where} &\begin{cases} \widehat{\delta}(q, \varepsilon) &:= q \\ \widehat{\delta}(q, aw) &:= \widehat{\delta}(\delta(q, a), w) \end{cases} \end{aligned}$$

Using λ -notation, the following program takes a finite state automaton A and returns a *program* that accepts its language $L(A)$:

$$\mathbf{acceptor}(\langle \Sigma, Q, \delta, q_0, F \rangle) := \lambda w. [\widehat{\delta}(q_0, w) \in F]$$

$$\text{where } \begin{cases} \widehat{\delta}(q, \varepsilon) & := q \\ \widehat{\delta}(q, aw) & := \widehat{\delta}(\delta(q, a), w) \end{cases}$$

4.4 Transducers

A (*finite-state*) *transducer* is a finite-state automaton that also produces output. Formally, in the deterministic case, we have an output alphabet Γ and the automaton is equipped with an output function $\gamma : Q \times \Sigma \rightarrow \Gamma^*$ that goes hand-in-hand with δ . With each state transition zero or any numbers of symbols are output.

An example is the famous Tic-Tac-Toe machine made of Tinker Toys, designed by Danny Hillis and Brian Silverman, when they were undergraduate students at MIT.

Comment. An important extension of the finite-state devices are tree automata and tree transducers.¹

Remark. Automata that operate over *infinite* input words can be used to model processes like operating systems, and have important practical applications in hardware validation and other fields. For this to make sense, one needs an infinite criterion for acceptance. For example, one can say that automaton A accepts an infinite word w if A goes through final states F *infinitely* often in the course of reading w .

¹For everything you want to know on the subject, see the Tree Automata Techniques and Applications website at <http://www.grappa.univ-lille3.fr/tata>.

Chapter 5

Nondeterminism

The Blind Bartender Problem A bartender holds a square tray with four glasses, one standing in each corner. Each glass can be either upright or upside down, but the bartender is blind and cannot tell the difference. Nevertheless, his goal is to turn the glasses so that they will all be up or all down. Between each round of this game, a customer rotates the tray through a multiple of 90° , after which the bartender may leave the glasses as they are, he may turn one of them over, or he may turn any two. If, after his action, all four glasses are up, or all four are down, then he wins. If not, the game continues with another round. Is there a strategy by which the bartender can guarantee an eventual win?¹

5.1 Angels and Demons

Nondeterminism (in programming) means that the progress and outcome of a computation might not be fully determined by the input, and may differ from run to run. Programs may contain explicit choice points, or may incorporate operations the effect of which is not fully specified, or there may be external (unknown or unpredictable) factors that can change the outcome. It is natural to view such programs as defining a “multivalued function”, with a (possibly empty) set of values as its output, or, equivalently, a (*binary*) *relation* between input and output values.

¹Martin Gardner, About rectangling rectangles, parodying Poe and many another pleasing problem, *Mathematical Games*, *Sci. Amer.*, vol 240, no. 2 (February 1979) pages 16-24. Martin Gardner, On altering the past, delaying the future and other ways of tampering with time, *Mathematical Games*, *Sci. Amer.*, vol 240, no. 3 (March 1979) pages 21-30.

This important notion in specification and programming comes in several flavors:

- *Angelic*, or “don’t know” nondeterminism. When the computation “flips a coin” to decide whether to “go left” or “go right”, the coin always gives a “good” answer—if there is one. In other words, the program does not “know” which choice leads to the desired outcome, and needs an “angel” to tell it which path to take. In the absence of an angel, one may need to try all paths, by keeping track of which routes have been explored and how far, or by backtracking upon failure, or by allocated the different choices to different processes.
- *Demonic*, or “don’t care” nondeterminism. In this version, all choices are assumed to be equally good. If any leads to success, then they all do. For example, we expect multi-threaded programs to work, regardless of scheduling choices. Put another way, the programmer needs to assume that if there is a path to failure, the adversarial demon will force the system to make the worst possible choice.
- *Probabilistic* computation. Here one “flips a coin” when faced with a choice, and the result of flipping obeys some probabilistic distribution, such as 50-50 odds for heads or tails, or some biased outcome, like 10-to-1 tails, or any other probabilistic assumptions.
- *Fair* computation. Here one only assumes that when faced over and over again with the same decision, the system does not *always* make the same choice.

We deal only with the angelic form.

5.2 Nondeterministic Finite State Automata

A *nondeterministic* finite state automaton (NFA) is like the deterministic case, except that the transition function δ is multi-valued, and gives a set of states from which to continue. That is, $\delta : Q \rightarrow \mathcal{P}(Q)$, where $\mathcal{P}(Q)$ is the powerset of states Q . If *any* choice of next state leads to success, then the word is accepted, even though other choices may lead to failure. In other words, the automaton—whenever possible—guesses correctly which choice to make. As before, we write $L(N)$ to denote the set of words accepted by an NFA N . If a word is not in $L(N)$, then all paths, no matter what choices are made, lead to rejection.

It will be more transparent to write $q \xrightarrow[\delta]{a} r$ for the relation $r \in \delta(q, a)$, or for $r = \delta(q, a)$ in the deterministic case.

The acceptance function, then, is:

$$\begin{aligned} \mathbf{accept-nfa?}(w, \langle \Sigma, Q, \delta, q_0, F \rangle) &:= acc?(q_0, w) \\ \text{where } acc?(q, \varepsilon) &:= q \in F \\ acc?(q, av) &:= \exists r \in Q. q \xrightarrow[\delta]{a} r \wedge acc?(r, v) \end{aligned}$$

So, if $w \in L(N)$, for nondeterministic automaton N , then there is at least one successful computation path given by δ . There may, in fact, be many paths that lead to rejection, in addition to one or more successful paths. Still the automaton accepts, since the nondeterminism is “angelic”.

Theorem 3 *All languages accepted by nondeterministic finite state automata are automaton language.*

Proof. The following function takes an NFA N and returns a DFA D with the same language:

$$\begin{aligned} \mathbf{n2dfa}(\langle \Sigma, Q, \delta, q_0, F \rangle) &:= \langle \Sigma, \mathcal{P}(Q), \delta', \{q_0\}, \mathcal{P}(Q) \setminus \mathcal{P}(Q \setminus F) \rangle \\ \text{where } \delta'(R, a) &:= \{q \in Q : \exists r \in R. r \xrightarrow[\delta]{a} q\} \end{aligned}$$

where $\mathcal{P}(Q) \setminus \mathcal{P}(Q \setminus F) = \{S \subseteq Q : S \cap F \neq \emptyset\}$ is simply all subsets S of A ’s states Q containing at least one final state from F .

One must show that N and D accept the same words exactly.

The idea is that the determinized machine D keeps track of *all* the states S in which N could have been at the point in the computation. So, the next state S' , after reading some input, would be the set of states in S that can be reached from *any* of the \square

The states of the DFA are each subsets of the states of the NFA. So, in general, the DFA can have exponentially more states than the original DFA.²

It has been found to be very convenient to also allow nondeterministic moves, called “epsilon moves”, that do not read from the input, and act like a “one way street”. Formally, $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ is now a relation between pairs of states in Q and letters in Σ or the empty string ε .

²Determinizing an automaton as efficiently as possible is a big business.

The acceptance criterion becomes the following:

$$\mathbf{accept-nfa}_\varepsilon?(w, \langle \Sigma, Q, \delta, q_0, F \rangle) := acc?(q_0, w)$$

where

$$acc?(q, w) := \left\{ \begin{array}{l} [w = \varepsilon \wedge q \in F] \vee \\ [\exists a \in \Sigma, v \in \Sigma^*, r \in Q. w = av \wedge q \xrightarrow[\delta]{a} r \wedge acc?(r, v)] \vee \\ [\exists r \in Q. q \xrightarrow[\delta]{\varepsilon} r. acc?(r, w)] \end{array} \right\}$$

Note that this introduces the possibility of an infinite loop, following a cycle of epsilon moves. In practice, one would need to keep track of paths to avoid loops, so that one could reject words for which there is no accepting path. Epsilon moves do not “add power” to automata and can be eliminated: For any given automaton, let $\xrightarrow[\delta]{\varepsilon} \dots \xrightarrow[\delta]{\varepsilon}$ denote “epsilon-reachability”, that is, the reflexive-transitive closure of the binary relation $\xrightarrow[\delta]{\varepsilon}$, meaning all paths of zero or any number of epsilon steps. Let

$$E(q) = \{r \in Q : q \xrightarrow[\delta]{\varepsilon} \dots \xrightarrow[\delta]{\varepsilon} r\}$$

be all the states epsilon-reachable from q without inputting any symbols. Another way of defining E is via the following axiom and inference rule:

$$\frac{}{q \in E(q)} \quad \frac{s \in E(q), s \xrightarrow{\varepsilon} r}{r \in E(q)}$$

meaning that all states are trivially reachable from themselves, and if s is epsilon-reachable from q and there is an epsilon step from s to r , then r is also epsilon-reachable.

With this notion of reachability, we can define a function to eliminate all epsilon moves:

$$\begin{aligned} \mathbf{remove-\varepsilon}?(\langle \Sigma, Q, \delta, q_0, F \rangle) &:= \langle \Sigma, Q \cup \{q_s\}, \delta', q_s, F' \rangle \\ \text{where } \delta'(q, a) &:= \bigcup_{r \in E(q)} \delta(r, a) \quad (q \in Q) \\ \delta'(q_s, a) &:= \bigcup_{r \in E(q_0)} \delta(r, a) \\ F' &:= \{q \in Q : E(q) \cap F \neq \emptyset\} \end{aligned}$$

5.3 Transducers

For non-deterministic transducers, just extend the transition relation $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ with $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \times Q$.

Chapter 6

Closure Operations

We will see (Theorem 6) that automaton languages are “closed” under the set-union operation. This means that if L and L' are two automaton languages, then their union, $L \cup L'$ is also an automaton language; in other words, there is an automaton that accepts their union, too. It turns out that the automaton languages are closed under all the following operations: complementation, reversal, union, intersection, concatenation, and Kleene star.

For a family \mathcal{L} of languages to be *closed* under an operation f means that applying f to languages in \mathcal{L} results in a language in \mathcal{L} . So for an n -ary operation $o : \mathcal{L}^n \rightarrow \mathcal{L}$, we need that $o(L_1, \dots, L_n) \in \mathcal{L}$ for all $L_1, \dots, L_n \in \mathcal{L}$.

6.1 Set Operations

Theorem 4 *Automaton languages are closed under complementation.*

In other words, if there is a finite automaton, over an alphabet Σ , with language L , then there is a finite automaton with language $\Sigma^* \setminus L$.

Proof. All one needs to do to get the complement of a automaton language is to change final states of a deterministic automaton into non-accepting ones and vice-versa:

$$\mathbf{complement-dfa}(\langle \Sigma, Q, \delta, q_0, F \rangle) := \langle \Sigma, Q, \delta, q_0, Q \setminus F \rangle$$

Given a DFA A over an alphabet Σ , the result of applying $\mathbf{complement-dfa}(A)$ is an automaton A' such that $L(A') = \Sigma^* \setminus L(A)$.

□

Problem. This simple construction does not work for nondeterministic automata. Why? So what can one do?

Theorem 5 *Automaton languages are closed under intersection.*

Proof. The idea here is to keep one finger on the state of one of the given automata, while reading the input, and another finger on the current state of the second. We do this by creating an automaton, each state of which is a pair of states:

$$\begin{aligned} \mathbf{intersect-dfa}(\langle \Sigma, Q, \delta, q_0, F \rangle, \langle \Sigma, Q', \delta', q'_0, F' \rangle) &:= \\ &\langle \Sigma, Q \times Q', \delta'', [q_0, q'_0], F \times F' \rangle \\ \text{where } \delta''([q, q'], a) &:= [\delta(q, a), \delta'(q', a)] \end{aligned}$$

and \times is Cartesian product of sets.

Given DFAs A, A' over the same alphabet Σ , the result of applying this function is an automaton C such that $L(C) = L(A) \cap L(A')$. \square

In the same way, one can show that

Theorem 6 *Automaton languages are closed under union.*

Proof. Given DFAs A, A' over the same alphabet Σ , the result of applying the following function is an automaton C such that $L(C) = L(A) \cup L(A')$:

$$\begin{aligned} \mathbf{union-dfa}(\langle \Sigma, Q, \delta, q_0, F \rangle, \langle \Sigma, Q', \delta', q'_0, F' \rangle) &:= \\ &\langle \Sigma, Q \times Q', \delta'', [q_0, q'_0], (F \times Q') \cup (Q \times F') \rangle \\ \text{where } \delta''([q, q'], a) &:= [\delta(q, a), \delta'(q', a)] \end{aligned}$$

\square

If we use nondeterministic automata, the construction is almost the same:

Proof. Given NFAs A, A' over the same alphabet Σ , the result of applying the following function is an automaton C such that $L(C) = L(A) \cup L(A')$:

$$\begin{aligned} \mathbf{union-nfa}(\langle \Sigma, Q, \delta, q_0, F \rangle, \langle \Sigma, Q', \delta', q'_0, F' \rangle) &:= \\ &\langle \Sigma, Q \times Q', \delta'', [q_0, q'_0], (F \times Q') \cup (Q \times F') \rangle \\ \text{where } \delta''([q, q'], a) &:= \{[r, r'] : r \in \delta(q, a), r' \in \delta'(q', a)\} \end{aligned}$$

\square

Problem. What is the relation between the automaton C obtained by this method, and the one that would be obtained from complementation and intersection, using DeMorgan’s Laws?

6.2 Word Operations

Theorem 7 *Automaton languages are closed under reversal.*

By “reversal” we mean reading each word backwards, $a_1a_2 \cdots a_n$ becomes $a_n \cdots a_2a_1$:

$$\begin{aligned}\varepsilon^R &:= \varepsilon \\ (aw)^R &:= w^Ra\end{aligned}$$

The idea is to turn the arrows of the automaton around. But we must add a new start state, since only one is allowed.

Proof.

$$\begin{aligned}\text{reverse-nfa}(\langle \Sigma, Q, \delta, q_0, F \rangle) &:= \langle \Sigma, Q \cup \{q_s\}, \delta^{-1}, q_s, \{q_0\} \rangle \\ \text{where } \delta^{-1}(q_s, \varepsilon) &= F \\ \delta^{-1}(q, a) &= \{r : q \in \delta(r, a)\} \quad (q \in Q)\end{aligned}$$

□

Theorem 8 *Automaton languages are closed under concatenation.*

Given automata A, A' , we would like to run A first, and—when successful—try A' . But there may be many ways of splitting a word w into uv such that A accepts u and A' accepts v . It is therefore natural to use nondeterministic automata and guess where u ends.

Proof. Given NFAs A, A' over the same alphabet Σ , the result of applying the following function is an automaton C such that $L(C) = L(A) \cdot L(A')$:

$$\begin{aligned}\text{concatenate-nfa}(\langle \Sigma, Q, \delta, q_0, F \rangle, \langle \Sigma, Q', \delta', q'_0, F' \rangle) &:= \langle \Sigma, Q \uplus Q', \delta'', q_0, F \cup F' \rangle \\ \text{where } \delta''(q, x) &:= \begin{cases} \delta(q, x) & \text{if } q \in Q, x \in \Sigma \\ \delta(q, \varepsilon) \cup \{q'_0\} & \text{if } q \in F, x = \varepsilon \\ \delta'(q, x) & \text{if } q \in Q' \end{cases}\end{aligned}$$

where \uplus is a disjoint union, meaning that Q and Q' are presumed to have no elements in common. (One can always rename states, if necessary, to fulfil this condition.) □

Theorem 9 *Automaton languages are closed under Kleene star.*

Given an automaton A , we need to build a loop around it, without introducing any spurious accepting paths.

Proof. Given an NFA A , the result of applying the following function is an automaton C such that $L(C) = L(A)^*$:

$$\text{Kleene-nfa}(\langle \Sigma, Q, \delta, q_0, F \rangle) := \langle \Sigma, Q \uplus \{q_s\}, \delta', q_s, \{q_s\} \rangle$$

$$\text{where } \delta'(q, x) := \begin{cases} \delta(q, x) & \text{if } q \in Q \setminus F \\ \delta(q, x) \cup \{q_s\} & \text{if } q \in F \\ q_0 & \text{if } q = q_s, x = \varepsilon \end{cases}$$

□

6.3 Kleene's Theorem

Theorem 10 (Kleene's Theorem) *The automaton languages are the smallest family of languages that includes all the finite languages and are closed under union, concatenation, and star.*

Proof. We have already seen that finite languages are accepted by automata and that automaton languages are closed under these operations. The remaining issue is whether there is an automaton language that cannot be obtained by a finite sequence of closure operations. We defer this question till later. □

Chapter 7

Regular Languages

7.1 Regular Expressions

There is another, textual way of describing the languages accepted by finite-state automata, which can be quite convenient, namely *regular expressions*. Regular expressions are commonly used in textual search.¹

The syntax of regular expressions over some alphabet $\Sigma = \{a_1, \dots, a_z\}$ is conveniently expressed by the following “grammar”:

$$R \rightarrow \emptyset \mid \varepsilon \mid a_1 \mid \dots \mid a_z \mid R + R \mid RR \mid R^*$$

meaning that a regular expression R can be the empty-set symbol \emptyset , or the empty-string symbol ε , or any of the letters of the alphabet Σ . Moreover, a regular expression can be the sum $r + s$ or product rs of two regular expressions r, s . It can also be r^* , where r is any regular expression and the superscript is the Kleene star. In other words, the set R of regular expressions contains \emptyset , ε , and Σ , and is closed under the formation of sums, products, and star expressions. The intention is that regular expressions include these and nothing more. See the next chapter for details.

The following table summarizes the correspondence between regular expressions and sets of words:

¹In the grep family of Unix commands (<http://en.wikipedia.org/wiki/Grep>) and in the Perl programming language (<http://www.perl.org>).

<i>Expression</i>	<i>Language</i>
\emptyset	\emptyset
ε	$\{\varepsilon\}$
a	$\{a\}$ (for any $a \in \Sigma$)
$r + s$	$R \cup S$
rs	$R \cdot S$
r^*	R^*

where R and S are the languages corresponding to r and s , respectively. For example, if r, s are regular expressions denoting sets R and S of words, then $r + s$ denotes their union $R \cup S$.

7.2 Regular Languages

Let Reg be the set $\{L(r) : r \text{ is a regular expression}\}$.

Theorem 11 *There is a finite automaton that accepts the language of every regular expression.*

Proof. With the closure operations we have already programmed, it is a trivial matter to convert a regular expression into a nondeterministic automaton. Let Σ be the alphabet over which the regular expression is formed, and let the conversion function be defined by the following equations:

$$\begin{aligned}
\mathbf{r2a}(\emptyset) &:= \langle \Sigma, \{q_0\}, \lambda q, a.\emptyset, q_0, \emptyset \rangle \\
\mathbf{r2a}(\varepsilon) &:= \langle \Sigma, \{q_0\}, \lambda q, a.\emptyset, q_0, \{q_0\} \rangle \\
\mathbf{r2a}(a) &:= \langle \Sigma, \{q_0, q_f\}, q_0 \xrightarrow[\delta]{a} q_f, q_0, \{q_f\} \rangle \\
\mathbf{r2a}(r + s) &:= \mathbf{union-nfa}(\mathbf{r2a}(r), \mathbf{r2a}(s)) \\
\mathbf{r2a}(rs) &:= \mathbf{concat-nfa}(\mathbf{r2a}(r), \mathbf{r2a}(s)) \\
\mathbf{r2a}(r^*) &:= \mathbf{star-nfa}(\mathbf{r2a}(r))
\end{aligned}$$

By $q_0 \xrightarrow[\delta]{a} q_f$ we mean the transition relation δ such that $\delta(q, x) = \emptyset$ for all $q \in \{q_0, q_f\}$ and $x \in \Sigma$, except that $\delta(q_0, a) = \{q_f\}$. \square

The converse is more difficult:

Theorem 12 *Every automaton language can be expressed as a regular expression.*

Proof. Given a DFA, we use an auxiliary function $rg(q, r, S)$ that computes a regular expression for all paths from q to r that pass via any intermediate state in S .²

$$\begin{aligned} \mathbf{a2r}(\langle \Sigma, Q, \delta, q_0, F \rangle) &:= \sum_{f \in F} rg(q_0, f, Q) \\ \text{where } rg(q, r, \emptyset) &:= \begin{cases} \epsilon & \text{if } q = r \\ \emptyset & \text{otherwise} \end{cases} + \sum_{a \in \Sigma} \begin{cases} a & \text{if } \delta(q, a) = q \\ \emptyset & \text{otherwise} \end{cases} \\ rg(q, r, \{s\} \cup S) &:= [rg(q, r, S)] + [rg(q, s, S)][rg(s, s, S)]^*[rg(s, r, S)] \end{aligned}$$

where the summation operator $\sum_{x \in S} r(x)$ is being used here to mean forming a regular expression by summing the expression $r(x)$ for each $x \in S$:

$$\begin{aligned} \sum_{x \in \emptyset} r(x) &:= \emptyset \\ \sum_{x \in \{s\} \cup S} r(x) &:= r(s) + \sum_{x \in S} r(x) \end{aligned}$$

□

The second half of Kleene's Theorem (Theorem 10) follows, namely that automaton languages are obtainable by applying unions, concatenations, and Kleene star operations to finite languages (namely, \emptyset , $\{\epsilon\}$, and singleton sets of letters).

Example

Consider the following automaton for words over $\{a, b\}$ with an even number of a's:

$Q \setminus \Sigma$	a	b
0	1	0
1	0	1

It can be converted into an equivalent regular expression in the following manner:

$$\begin{aligned} rg(0, 0, \emptyset) &= rg(1, 1, \emptyset) = \epsilon + b \\ rg(0, 1, \emptyset) &= rg(1, 0, \emptyset) = a \\ rg(0, 0, \{1\}) &= rg(0, 0, \emptyset) + rg(0, 1, \emptyset)[rg(1, 1, \emptyset)]^*rg(1, 0, \emptyset) \\ &= (\epsilon + b) + a(\epsilon + b)^*a = \epsilon + b + ab^*a \\ rg(0, 0, \{0, 1\}) &= rg(0, 0, \{1\}) + rg(0, 0, \{1\})[rg(0, 0, \{1\})]^*rg(0, 0, \{1\}) \\ &= (\epsilon + b + ab^*a) + (\epsilon + b + ab^*a)(\epsilon + b + ab^*a)^*(\epsilon + b + ab^*a) \\ &= (b + ab^*a)^*(\epsilon + b + ab^*a) \\ &= (b + ab^*a)^* + (b + ab^*a)^*(b + ab^*a) \\ &= (b + ab^*a)^* \\ \mathbf{a2r}(A) &= rg(0, 0, \{0, 1\}) = (b + ab^*a)^* \end{aligned}$$

²The algorithm can be made more efficient by using dynamic programming to avoid repeatedly computing the same values of $rg(q, r, S)$.

The stages are given here “bottom-up”, though the algorithm **a2r** works top-down.

Chapter 8

Question Answering

8.1 Emptiness

The *emptiness problem*, for a formal language L , is the question if L is devoid of words, that is, whether or not $L = \emptyset$. For a finite-state automaton A , there are several ways to test whether $L(A) = \emptyset$, or not.

We say that a problem like emptiness is *decidable* if there is an algorithm that always gives the right answer.

Theorem 13 *Emptiness of finite state automata is decidable.*

Proof. The emptiness question can be solved directly by checking if there is any path from the start state to a final state that does not visit any state more than once. One way to do that, for any automaton (deterministic or nondeterministic, with or without epsilon moves) is as follows:

$$\begin{aligned} \text{empty?}(\langle \Sigma, Q, \delta, q_0, F \rangle) &:= s(q_0) \cap F = \emptyset \\ \text{where } s(q) &:= t(q, |Q|) \\ t(q, n) &:= \begin{cases} \{q\} & \text{if } n = 0 \\ t(q, n-1) \cup \{\delta(r, a) : r \in t(q, n-1), a \in \Sigma\} & \text{otherwise} \end{cases} \end{aligned}$$

Here $t(q, n)$ collects all states reachable from q within at most n steps. \square

Problem. Why does this work even in the presence of epsilon steps?

Another way of seeing essentially the same thing is to recall that, by the pumping lemma, for any A there is a constant n such that any word $z \in L(A)$ longer than n can be made shorter (by “pumping down” and

deleting v) and still be accepted. This means that we need only test words (over the alphabet of A) of length up to n for membership in $L(A)$. If any one of them is accepted, then the language is nonempty; if none are, it is empty.

Comment. The above algorithm can be inefficient, as cycles may be travelled over and over again.¹ By using standard marking methods of depth-first search (DFS), no edge need be traversed twice in determining whether there is any path from the start state to a final state.

8.2 Fullness

The *fullness problem*, for a formal language $L \subseteq \Sigma^*$, is the question if L contains all possible words, that is, whether or not $L = \Sigma^*$.

Problem. Show that the fullness problem for finite-state automata is decidable.

8.3 Equivalence

The *equivalence problem* for two formal languages L and L' is the question whether $L = L'$. So the equivalence problem for finite automata A and B is the question whether their languages, $L(A)$ and $L(B)$, are identical.

We say that automata A and B are *semantically equivalent* if they accept precisely the same language, that is, if $L(A) = L(B)$. In that case, we can write $A \equiv B$.

To determine if $L(A) = L(B)$, for DFAs or NFAs A and B , we can test whether both $L(A) \subseteq L(B)$ as well as $L(B) \subseteq L(A)$. Testing inclusion of languages, $L(A) \subseteq L(B)$, can be accomplished easily enough by using the algorithms of the previous chapter to construct an automaton for $L(A) \cap \overline{L(B)}$, where $\overline{L(B)}$ denotes the complement $\Sigma^* \setminus L(B)$, and then using the method of the previous section to test whether its language is empty, since $L(A) \setminus L(B) = L(A) \cap \overline{L(B)} = \emptyset$ iff $L(A) \subseteq L(B)$.

For DFAs, we have

$$\begin{aligned} \text{equiv?}(A, B) &:= \text{contain?}(A, B) \wedge \text{contain?}(B, A) \\ \text{where } \text{contain?}(A, B) &:= \\ &\quad \text{empty?}(\text{intersect-dfa}(A, \text{complement-dfa}(B))) \end{aligned}$$

¹Indeed, the algorithm takes $|\Sigma|^{|Q|}$ steps.

Thus:

Theorem 14 *Semantic equivalence of finite-state automata is decidable.*

8.4 Minimization

How can we find the best automaton for a given language? For one thing, we can remove any states that are unreachable from the initial states or from which no final state is reachable, along with all arrows of δ in and out of them:

$$\begin{aligned} \mathbf{clean}(\langle \Sigma, Q, \delta, q_0, F \rangle) &:= \langle \Sigma, Q', \delta \upharpoonright Q', q_0, F \cap Q' \rangle \\ \text{where } Q' &= \{q \in Q : q \in s(q_0), s(q) \cap F \neq \emptyset\} \end{aligned}$$

where s is as in Section 8.1, and $\delta \upharpoonright Q'$ is the same as δ , but is restricted to elements of $Q' \subseteq Q$.

There is more one can do to reduce the size of an automaton. Suppose we have a DFA A , and want to find a semantically equivalent automaton A' such that there is no other equivalent automaton with fewer states than A' . The naïve, grossly inefficient, “British Museum” method would be to construct every possible automaton A' over the same alphabet as A with fewer states than A , from smallest to largest, and test for semantic equivalence.

Comment. For any automaton A , not only is the language of all words obtainable beginning with start state $q_0 \in Q$ regular, but so are the languages obtained by starting at any other state in Q , in other words by making q the start state instead of q_0 . By the same token, changing the set of final states to any subset of Q yields a regular language.

For any finite automaton $A = \langle \Sigma, Q, \delta, q_0, F \rangle$, define

$$q \equiv r := \mathbf{equiv?}(\langle \Sigma, Q, \delta, q, F \rangle, \langle \Sigma, Q, \delta, r, F \rangle)$$

Denote by $[q] = \{r \in Q : r \equiv q\}$ the equivalence class of q . The minimal automaton is computed as follows:

$$\begin{aligned} \mathbf{minimize}(\langle \Sigma, Q, \delta, q_0, F \rangle) &= \mathbf{clean}(\langle \Sigma, \mathcal{P}(Q), \delta', [q_0], \{[f] : f \in F\} \rangle) \\ \text{where } \delta'([q], a) &:= [\delta(q, a)] \end{aligned}$$

A much better method works by first figuring out incrementally which states q, r are “inequivalent”, denoted $q \not\equiv r$. Inequivalence is symmetric.

Consider the following inference rules:

$$\frac{f \in F, q \notin F}{f \not\equiv q} \quad \frac{r \not\equiv r', q \xrightarrow{a} r, q' \xrightarrow{a} r'}{q \not\equiv q'}$$

These rules are applied to states of an automaton until nothing more can be inferred. Any states not found inequivalent in this manner are deemed equivalent.

The following is an alternative, very “neat” algorithm:

minimize(A) := **clean**(**n2dfa**(**reverse-nfa**(**n2dfa**(**reverse-nfa**(A))))))

Proof. Reversing twice clearly gives a semantically equivalent automaton. But why is it guaranteed to be minimal?! The trick is that determinization unites states that are always reached in tandem. The details are too complicated to include here. \square

Comment. Another way to test for emptiness would be to minimize A and see if what is obtained has only one, non-final state.

Theorem 15 *The minimal automaton is unique up to isomorphism.*

Proof. Suppose A and B are minimal automata for the same language L . Run the same procedure for determining equivalent states on the (disjoint) union of (the graphs of) A and B . If any (useful) state of A has no counterpart in B , then some word is accepted by A and not B . (Why?) On the other hand, if any state q in A corresponds to two states r, r' in B , then $r \equiv r'$ and B is not minimal. \square

Chapter 9

Non-Regular Languages

9.1 Pumping Lemma

Let $|w|$ be the length in symbols of a word w , that is, $|a_1 \cdots a_\ell| = \ell \geq 0$. We have the following interesting property of automaton languages:

Lemma 1 (Pumping Lemma for Automaton Languages) *For every automaton language $L \subseteq \Sigma^*$, there is a constant $n \in \mathbb{N}$ such that*

$$\begin{aligned} \forall z \in L. (|z| > n) \quad \rightarrow \quad & \exists u, v, w \in \Sigma^*. \\ & (z = uvw) \wedge (|v| > 0) \wedge (|uv| \leq n) \\ & \wedge \{uv^i w : i \in \mathbb{N}\} \subseteq L \end{aligned}$$

The point is that finite automata have only finitely many states, so if a long word is accepted, an automaton must pass through some loop at least once, reading some subword in the process. But if it may pass through once on the path to acceptance, then it could just as well pass through many times, or even take a shortcut and skip the loop altogether.

Proof. Let $n = |Q|$ be the number of states in some DFA $A = \langle \Sigma, Q, \delta, q_0, F \rangle$ that accepts L . Consider the sequence q_0, q_1, \dots, q_ℓ of states through which the automaton passes while accepting a word $z = a_1 \dots a_\ell$ of length ℓ , where $q_i = \widehat{\delta}(q_0, a_1 \dots a_i)$ and $q_\ell \in F$. By the Pigeon-Hole Principle, if $\ell \geq n$ then at least one of the first $n + 1$ states in the sequence repeats itself. That is, $q_j = q_k$ for some $0 \leq j < k \leq n \leq \ell$. Let $u = a_1 \cdots a_j, v = a_{j+1} \cdots a_k, w = a_{k+1} \cdots a_\ell$. We have $\widehat{\delta}(q_0, u) = q_j = q_k = \widehat{\delta}(q_j, v)$. So, for any i , we have

$$\widehat{\delta}(q_0, uv^i w) = \widehat{\delta}(q_j, v^i w) = \widehat{\delta}(q_k, w) = \widehat{\delta}(q_0, uvw)$$

□

9.2 Beyond Finite Automata

The main use of this lemma is in the reverse, to show that a language is *not* an automaton language.

To show that L is not accepted by any automaton, we need to prove that for every length $n \in \mathbf{N}$ there is a word $z \in L$, of length greater than n , that, regardless of how it is partitioned into uvw , has at least one pumped version $uv^i w \notin L$.

By logic, the contrapositive of the lemma is as follows:

L is not an automaton language if

$$\forall n \in \mathbf{N}. \exists z \in L. \quad |z| > n \wedge \\ \forall u, v, w \in \Sigma^*. (z = uvw) \wedge (|v| > 0) \wedge (|uv| \leq n) \rightarrow \\ [\exists x \in \Sigma^*. x \in \{uv^i w : i \in \mathbf{N}\} \wedge x \notin L]$$

The *palindromic language* $Pal(\Sigma)$ consists of all words over Σ that read the same left-to-right and right-to-left. For the Latin alphabet, it includes the English words: a, I, aha, bib, bob, toot, . . . , tattarrattat. Moving on to sentences in English, but ignoring spaces, punctuation and letter case, we have classics, like “Able was I ere I saw Elba” and “Madam, in Eden I’m Adam”.¹

The Pumping Lemma can be used to prove that $Pal(\{a, b\})$, or any palindromic language over an alphabet with more than one symbol, is not an automaton language.

For each $n > 0$, let $z = a^n b a^n$. For any partition of z into uvw , the limitation on the length of uv means that $v = a^k$ for some $k > 0$. But then $x = a^{n-k} b a^n \notin Pal(\{a, b\})$.

An Interlude

Alice: My Baa-Baa language consists of all strings with more a ’s than b ’s.

Bob: And I just invented a finite state automaton for your Baa-Baa language.

¹For a long palindromic “poem”, see http://graywyvern.blogspot.com/2004_09_19_graywyvern_archive.html#109576527901759797. For more information, see <http://www.palindromelist.com>, <http://en.wikipedia.org/wiki/Palindrome>, and <http://realchange.org/pal>.

Alice: How big is it?

Bob: It was really complicated; it has a whopping 9,999 states.

Alice: Impressive! So, let's test it on something big, but simple, like $b^{4999}a^{5000}$. What does it say?

Bob: No go. It's rejected.

Alice: Well, then, it's faulty, since my test case has more a 's.

Bob: Oops! Sorry, there was a tiny bug. Now it works fine.

Alice: Okay, so tell me: When does your machine first return to a prior state on my example, as you know it must?

Bob: Oh. I'll run it and watch this time. After reading the 150th letter, it's in the same state it was after the 50th.

Alice: I see. So try your machine out on this bigger one: $b^{5099}a^{5000}$. I'll bet you a fortune that it will accept it, even though it's *not* a Baa-Baa word. Not only that, it will also fail on $b^{5199}a^{5000}$, $b^{5299}a^{5000}$, ad infinitum.

Bob: You're right! How did you know, within even looking inside? Give me a few minutes.... It's okay now. I didn't even have to add any more states.

Alice: I see. So, now, when does the program repeat itself?

Bob: Oh. After reading the second a of $b^{4999}a^{5000}$, it is in the same state it was right after the run of b 's.

Alice: Aha! I can assure you, then, that it will give the wrong answer for $b^{4999}a^{4998}$.

Bob: You're fantastic! I better work on this some more.

Alice: I think you will find that you need to add a stack to your program.

Bob: Good idea. Thanks a million!

Moral. A language is not an automaton language if it *requires* more and more memory to process longer and longer inputs.

9.3 Summary

A language L is regular if any of the following hold true:

1. L is finite.
2. L is accepted by some DFA or NFA A .
3. L has a regular expression r .

4. L is obtainable from regular languages by a sequence of closure operations (union, intersection, complementation, concatenation, Kleene star, reversal).²

A language L is not regular if any of the following hold:

1. For every word-length n , L contains a word of length n that cannot be pumped.
2. There is a regular language R and a closure operation op such that $op(L, R)$ is a non-regular language.

²There are additional closure operations, such as homomorphism and inverse homomorphism, for regular languages.

Chapter 10

Grammars

Wouldn't the sentence I want to put a hyphen between the words fish
and and and and and
chips in my fish and chips sign have looked cleaner
if quotation marks had been placed before fish and between fish
and and and and and and and and and and and and and and and
and and and and and
chips as well as after chips?

—Martin Gardner

A convenient means of defining formal languages, and of computing with strings, is that of derivational grammars.

10.1 Generational Grammars

Definition 12 *A grammar consists of the following:*

- *an alphabet Σ of (terminal) symbols;*
- *a set of variables V , also called nonterminals;*
- *a set of rewrite rules P , called productions, each of the form $\alpha \rightarrow \beta$, where both sides of the rule are strings of symbols and/or variables ($\alpha, \beta \in (\Sigma \cup V)^*$);*
- *a start symbol $S \in V$.*

Let's stick to the convention of using upper-case Latin letters A, B, C, S for variables in V ; lower-case Latin letters u, v, w, x, y, z for words over Σ ;

and lower-case Greek $\alpha, \beta, \gamma, \delta$ for strings of symbols and/or variables over $\Sigma \cup V$.

A production rule $\alpha \rightarrow \beta$ is applied to a string $\gamma\alpha\delta$ containing the substring α by replacing the occurrence of α with β , in which case we write $\gamma\alpha\delta \Rightarrow \gamma\beta\delta$, and say that $\gamma\alpha\delta$ *derives* $\gamma\beta\delta$ in one step.

A word $w \in \Sigma^*$ belongs to the language $L(G)$ of a grammar G over Σ , if there is a derivation $S \Rightarrow \cdots \Rightarrow w$ of any length. This derivation relation \Rightarrow can be formalized by the following inference rules:

$$\frac{}{\alpha \Rightarrow^0 \alpha} \quad \frac{\alpha \rightarrow \beta}{\gamma\alpha\delta \Rightarrow^1 \gamma\beta\delta} \quad \frac{\gamma \Rightarrow^i \delta\alpha\delta', \alpha \Rightarrow^j \beta}{\alpha \Rightarrow^{i+j} \beta\gamma'\delta}$$

where $\alpha \rightarrow \beta$ means that this rule is one of the productions P of G , and the superscript \Rightarrow^i specifies the number of steps used in the derivation.

These rules can be specialized to capture the notions of “top-down” and “bottom-up” *parsing*. Top-down begins with the start symbol S and works its way towards words w :

$$\frac{}{S \Rightarrow^0 S} \quad \frac{S \Rightarrow^i \gamma\alpha\delta, \alpha \rightarrow \beta}{S \Rightarrow^{i+1} \gamma\beta\delta}$$

Bottom-up parsing starts with w and looks for ways to get to it.

$$\frac{}{w \Rightarrow^0 w} \quad \frac{\gamma\beta\delta \Rightarrow^i w, \alpha \rightarrow \beta}{\gamma\alpha\delta \Rightarrow^{i+j} w}$$

In either case, if $S \Rightarrow^n w$, for any n , then $w \in L(G)$.

10.2 Context-Free Languages

A grammar is *context free* if the left side α of every production rule is a lone variable.

Then top-down parsing looks like this:

$$\frac{}{S \Rightarrow^0 S} \quad \frac{S \Rightarrow^i \gamma A \delta, A \rightarrow \beta}{S \Rightarrow^{i+1} \gamma\beta\delta}$$

Definition 13 A context-free grammar is in Chomsky normal-form if the right side is either a symbol from Σ or else consists of just two variables from V . In addition, there may be a rule $S \rightarrow \varepsilon$, with just the empty word to the right of the start symbol S .

Theorem 16 For every context-free grammar G there is an equivalent grammar G' in Chomsky normal-form, that is, for which $L(G') = L(G)$.

10.3 Closure Properties

It is easy to see that context-free grammars are closed under union, concatenation, reversal, and Kleene star. For example, the union of two context-free languages is generated by the grammar obtained as follows:

$$\mathbf{union-cfg}(\langle \Sigma, V', P', S' \rangle, \langle \Sigma, V'', P'', S'' \rangle) := \langle \Sigma, V' \uplus V'', P' \cup P'' \cup \{S \rightarrow S' | S''\}, S \rangle$$

where S is a new variable, not in V' or V'' .

10.4 Linear Grammars

A grammar is (*right*) *linear* when the left-side α is always a variable from V , and the right side β has at most one variable, which appears at the end, if at all.

Theorem 17 *The linear languages are exactly the regular languages.*

10.5 Pumping Lemma

Lemma 2 (Pumping Lemma for Context-Free Languages) *For every context-free language $L \subseteq \Sigma^*$, there is a constant $\ell \in \mathbf{N}$ such that*

$$\begin{aligned} \forall z \in L. (|z| \geq \ell) \quad \rightarrow \quad & \exists u, v, w, x, y \in \Sigma^*. \\ & (z = uvwxy) \wedge (|v| > 0 \vee |x| > 0) \wedge (|vwx| \leq \ell) \wedge \\ & \{uv^iwx^iy : i \in \mathbf{N}\} \subseteq L \end{aligned}$$

It turns out that $\ell = 2^{n-1} + 1$, where $n = |V|$ is the size of the set of variables of a Chomsky-normal-form grammar for L .

This lemma can be used to show that a language (e.g. $\{a^i b^i a^i : i \in \mathbf{N}\}$) is not context-free, in the same way as was done for the regular languages.

Theorem 18 *The context-free languages are not closed under intersection.*

Proof.

$$\{a^i b^i a^i : i \in \mathbf{N}\} = \{a^i b^i a^* : i \in \mathbf{N}\} \cap \{a^* b^i a^i : i \in \mathbf{N}\}$$

□

10.6 Problems

Theorem 19 *The following problems for context-free languages are decidable:*

1. *Membership.*
2. *Emptiness.*
3. *Finiteness.*

Theorem 20 *The following problems for context-free languages are undecidable:*

1. *Fullness.*
2. *Equivalence.*

The proof is deferred to a later chapter.

Chapter 11

Computability

[The Analytical Engine is for] developing and tabulating any function whatever. . . . The engine [is] the material expression of any indefinite function of any degree of generality and complexity.

The Analytical Engine has no pretensions to originate anything. It can do whatever we know how to order it to perform.

The engine can arrange and combine its numerical quantities exactly as if they were letters or any other general symbols; and in fact might bring out its results in algebraic notation, were provision made.

The bounds of arithmetic were however outstepped the moment the idea of applying the cards had occurred; and the Analytical Engine does not occupy common ground with mere “calculating machines.”

In enabling mechanism to combine together general symbols in successions of unlimited variety and extent, a uniting link is established between the operations of matter and the abstract mental processes of the most abstract branch of mathematical science.

—Ada Lovelace

11.1 Computable Languages

We have seen that the context-free palindrome language is not recognizable by any finite-state automata, and that the language $L_3 = \{a^i b^i a^i : i \in \mathbf{N}\}$ cannot be generated by a context-free grammar, even though every programmer knows how to write a program $L_3(w)$ that will answer “yes” if $w \in \{a, b\}^*$ is of the form $a^i b^i a^i$, and answer “no”, otherwise.

We are interested in exploring the outer boundaries of which languages can be recognized or generated with the most powerful computational mechanisms.

Definition 14 (Computable Language) *We say that a formal language L over an alphabet Σ is computable, or recursive, or decidable, or in \mathbf{R} , if there is a computer program that returns T (for “true”) if the input $w \in \Sigma^*$ is in L , and returns F (“false”) if $w \notin L$. In that case, we say that the program decides the language L .*

Definition 15 (Computably Enumerable Language) *We say that a formal language L over an alphabet Σ is recursively enumerable, or r.e., or semi-decidable, or in $\mathbf{r.e.}$, if there is a computer program that returns T (for “true”) if and only if the input $w \in \Sigma^*$ is in L . In that case, we say that the program accepts the language L .*

We use \mathbf{R} to denote the set of all recursive languages and $\mathbf{r.e.}$, the set of recursively enumerable ones.

11.2 Computable Functions

Since programs do not always terminate, they should be viewed as computing “partial” functions from inputs to outputs.

Definition 16 (Computable Function) *We say that a function f over some domain D is computable, or recursive, or in \mathbf{Comp} , if there is a computer program that computes $f(x)$ for all $x \in D$.*

A *partial function* is like a function, but can be undefined for some arguments. For example, natural logarithms, $\ln : \mathbf{R} \rightarrow \mathbf{R}$, are undefined for $0 \in \mathbf{R}$. It is often convenient to use a special symbol, like \perp (“bottom”), to denote “undefined”, so that we can write, for example, $\ln 0 = \perp$. Then, if we want, we can think of partial functions as total functions with an extended range, like $\mathbf{R} \cup \{\perp\}$ (or \mathbf{R}^\perp , for short).

Definition 17 (Partially Computable Function) *We say that a partial function f over some domain D is partially computable, or partially recursive, or in \mathbf{Part} , if there is a computer program that computes $f(x)$ for all $x \in D$, and which diverges (does not terminate) whenever $f(x) = \perp$.*

Let **Comp** denote the computable functions and **Part**, the partially computable ones.

It turns out that there is no difference in the computational power of different programming languages, as long as we allocate unbounded resources to computations.

11.3 Comparing Power

By a *family* of formal languages we mean a set of languages, such as the set of languages that can be recognized by some mechanism. Specifically, **Reg** is the set of regular languages (over any alphabet), **CFL** is the set of context-free languages, **R** are the computable languages, and **r.e.** are the recursively-enumerable languages.

We have already seen that

$$\mathbf{Reg} \subsetneq \mathbf{CFL} \subsetneq \mathbf{R}$$

Theorem 21 $\mathbf{R} \subsetneq \mathbf{r.e} \subsetneq \mathcal{P}(\Sigma^*)$

Proof. There are countably many (\aleph_0) programs that accept the languages in **R**, and uncountably many (\aleph) languages in $\mathcal{P}(\Sigma^*)$. \square

In the same way, one can compare the power of sets of partial functions (over the same domain) computed by different mechanisms, by comparing the sets of functions they can compute. Let $[D \rightarrow R]$ denote the set of all functions (R^D) from D to R , and $[D \dashrightarrow R]$, the set of all partial functions from D to R . We have already seen (Busy Beaver) that

$$\mathbf{Comp} \subsetneq [\mathbf{N} \rightarrow \mathbf{N}]$$

Theorem 22 $\mathbf{Part} \subsetneq [\mathbf{N} \dashrightarrow \mathbf{N}]$

The proof is as for the previous theorem.

Nota bene. When models of computation operate over different domains (e.g. DFAs over strings vs. Scheme over lists or Algol over numbers), then more sophisticated means of comparing power will be needed.

11.4 Fast Growing Functions

Let's consider the class \mathcal{F} of numeric functions $f : \mathbf{N} \rightarrow \mathbf{N}$. We can define an ordering on functions in \mathcal{F} , as follows:

$$f > g \Leftrightarrow \exists n_0 \in \mathbf{N}. \forall n \geq n_0. f(n) > g(n)$$

In other words, f eventually dominates g .

There exists a total function s that dominates all computable numeric functions and is not itself computable:

$$s(n) = \max\{f_i(j) : i, j \leq n\} + 1$$

Chapter 12

Descriptive Complexity

entia non sunt multiplicanda praeter necessitatem.
[entities should not be multiplied beyond necessity.]

*Occam's razor, a.k.a. lex parsimoniae, a.k.a. law of
succinctness*

We are interesting in measuring the complexity of a string (or other data item) by how hard it is to define it. Think “Busy Beaver”.

12.1 Kolmogorov Complexity

Let $|f|$ denote the length of program f (in some programming language), and $|x|$ the length of an input or output x .

Definition 18 *The Kolmogorov complexity of (output string) x is defined as follows:*

$$K(x) = \min\{|f| + |y| : I(f, y) = x\}$$

for some interpreter I .

Theorem 23 *Kolmogorov complexity is well-defined (K is a total function), since there is some constant c such that*

$$K(x) \leq |x| + c$$

holds for all x .

Proof. Let $c = |\lambda z.z|$, the size of the identity function, since $I(\lambda z.z, x) = x$. \square

Theorem 24 For any f

$$K(x) \leq K_f(x) + |f|$$

where

$$K_f(x) = \min\{|y| : f(y) = x\}$$

Proof. If $f(y) = x$, then $I(f, y) = x$. \square

12.2 Incomputability

Not surprisingly:

Theorem 25 Kolmogorov complexity (the function K) is uncomputable.

Proof. Suppose, by way of contradiction, that $K : \Sigma^* \rightarrow \mathbf{N}$ were computable. Let e be an enumerator of Σ^* , and consider the program:

$$M(n) := C(n, 0)$$

where

$$C(n, i) := \begin{cases} e(i) & \text{if } K(e(i)) > n \\ C(n, i + 1) & \text{otherwise} \end{cases}$$

Note that M is a total function, since only finitely many $e(i)$ can have complexity n . We have the following:

- $K(M(n)) > n$, since that's the condition under which C terminates.
- $K(y) \leq K_M(y) + |M|$, for all y (Theorem 24).
- $K_M(M(n)) \leq |n| = \lceil \log n \rceil$, by definition of K_M .

Stringing these all together:

$$n < K(M(n)) \leq K_M(M(n)) + |M| \leq \lceil \log n \rceil + |M|$$

which cannot hold for all n , regardless of the size of M . \square

Chapter 13

Interpretation

For every programming language L , one can construct a program that enumerates it.

13.1 Interpreters

An *interpreter* is a program $I : L \times X^n \rightarrow X$ that takes a program p operating over domain X and written in some language L (or an encoding $\langle p \rangle$ of p) along with inputs $\bar{x} \in X^n$ intended for p (or representations thereof), and computes what p would have for those inputs. That is, $I(p, \bar{x}) = p(\bar{x})$. The interpreter I may be written in the language L or in any other language.

13.2 Compilers

A *compiler* $C : L \rightarrow M$ translates programs/machines in language L into programs/machines of M . This shows that M is at least as powerful as L . Another compiler is needed to show that L and M are of equivalent power. Often, the two models work over different data structures (e.g. as numbers, strings, terms), so we actually show that the models compute the “same” functions, but using different representations. We have already seen, for example, how to convert the program of a multi-tape Turing machine into one that works on a single-tape device.

13.3 Partial Evaluation

A *partial evaluator* $S(p, \bar{y})$ is a program $S : L \times X^i \rightarrow M$ that takes a program $p \in L$ (or an encoding of p) along with *some* inputs $\bar{y} \in X^i$ to p (or

representations thereof), and returns a program p' in language M , such that $p'(\bar{z})$ computes $p(\bar{y}, \bar{z})$. The partial evaluator S may be written in language L , or M or any other.

If one has a partial evaluator S and interpreter I , then one always has a compiler: $C(p) = S(I, p)$. In Scheme, one trivially has interpreters and partial evaluators for Scheme itself:

$$\begin{aligned} I(p, \bar{x}) &= p(\bar{x}) \\ S(p, \bar{y}) &= \lambda \bar{z}. p(\bar{y}, \bar{z}) \end{aligned}$$

In other languages, like C , it is a nontrivial matter.¹

Interpreters and partial evaluators can also be written in one language for another.

13.4 Checkers

A *computation* is a sequence of snapshots, called *configurations* of the state of execution of a program, starting with an initial configuration and ending in a terminal one. In Scheme, these are the terms obtained in the substitution model.

A *checker* is a program that takes a program p and computation c and checks whether *or not* c is a valid computation of p .

Usually, if one has a checker (C) then one also has an interpreter (I):

$I(f, x) := \text{last}(\text{seq}(\mu(\lambda n. C(f, x, \text{seq}(n)), 0)))$

where

seq enumerates sequences of terms

last gives last element of sequence

$\mu(p, 0)$ gives first n such that $p(n)$

13.5 Recursion Theorem

Let \equiv denote *semantic equivalence*. That is, $f \equiv g$ if $\forall \bar{x}. f(\bar{x}) = g(\bar{x})$.

Recall that Baby is a sublanguage of Scheme. Baby uses normal-order evaluation.

Theorem 26 *For every $f : \text{Baby} \rightarrow \text{Baby}$ there exists a Baby constant c (a single-argument function) such that $c \equiv f(c)$.*²

¹A few examples of interpreters may be found in [~nachum/models/programs](#).

²See Exercise 4.21 and the footnote 27 of the Scheme book at <http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-26.html>.

Such a c is called a *fixpoint* of f .

For example, consider the *non-recursive* definition:

$$f(g) = \lambda y. \text{if } y = 0 \text{ then } 1 \text{ else } y \cdot g(y - 1)$$

Then c is the factorial function and $c(3)$, for instance, evaluates to 6.

Proof. Let $c = k(k)$ where $k = \lambda w. f(w(w))$. Then

$$c \equiv k(k) \equiv (\lambda w. f(w(w)))(k) \equiv f(k(k)) \equiv f(c)$$

□

The Recursion Theorem implies the existence of “quines”. A *quine* (named after the American logician/philosopher Willard Van Orman Quine) is a program that prints itself.³

It also implies the existence of self-replicating *virus* programs!

³See <http://www.nyx.net/~gthomps/quine.htm>.

Chapter 14

Decision Problems

To be, or not to be: that is the question.

—William Shakespeare, *Hamlet*

14.1 Questions and Answers

For many yes/no questions, it is easier to decide one of the possible answers than the other. Consider, for example, Fermat’s Last Theorem,¹ claiming that there are no integer solutions to $x^n + y^n = z^n$, for $n > 2$. Clearly, were there a counterexample to Fermat’s claim, it could eventually be found by an exhaustive search, trying all positive integer values of x, y, z, n ($n \neq 1$) until a solution is found such that $x^n + y^n = z^n$. Such a “keep on trying” approach has been called the “British Museum Method”. Showing that no such solution exists, however, required radically different methods.

A *decision problem* is a yes/no question that takes some parameterized input. There are many questions one can ask regarding a formal language L . Before one can even ask a question about L , one must agree on the form in which L is presented. For example, regular languages $L \in \mathbf{Reg}$ may be given as a regular expression or as a state-transition graph; context-free languages $L \in \mathbf{CFG}$ may be given as grammars. Three of the most popular questions for families \mathcal{F} of languages over some alphabet Σ are:

¹ It was in his copy of *Arithmetica* by Diophantus of Alexandria that Pierre de Fermat wrote this frustratingly famous marginal note:

Cubem autem in duos cubos, aut quadratoquadratum in duos quadratoquadratos, et generaliter nullam in infinitum ultra quadratum potestatem in duos ejusdem nominis fas est dividere: cujus rei demonstrationem mirabilem sane detexi. Hanc marginis exiguitas non caparet.

- Membership: For $x \in \Sigma^*$ and $L \in \mathcal{F}$, is $x \in L$?
- Emptiness: For $L \in \mathcal{F}$, is $L = \emptyset$?
- Fullness: $L \in \mathcal{F}$, is $L = \Sigma^*$?

In each case, we may have methods for finding out if the answer is yes or if the answer is no, or for both.

When we have methods for always determining whether the answer is yes or no, we say that the problem is *decidable*; when we know how to determine if the answer is yes, but may never be able to figure out when the answer is no, we say that it is *semi-decidable* or *partially decidable*; when we know how to answer no, rather than yes, we say that it is *co-semi-decidable* or *co-r.e.*.

So far, we have seen the following:

Question	Reg	CFL	r.e.
Membership	decid.	decid.	at least semi-decid.
Emptiness	decid.	decid.	at least co-semi-decid.
Fullness	decid.	at least co-semi-decid.	??

By time we are done, all the maybes will turn out to be negative:

Question	Reg	CFL	r.e.
Membership	decid.	decid.	only semi-decid.
Emptiness	decid.	decid.	only co-semi-decid.
Fullness	decid.	only co-semi-decid.	not semi- or co-semi- decid.

That is, there is no way to check fullness for the languages of context-free grammars, nor even non-membership for the computably-enumerable languages.

14.2 Decision Problems

It has become customary² to present decision problems in the following (somewhat rigid) format:

Instance: Description of the parameters of the problem.

Question: The yes/no question that the problem asks.

For example, the *Halting Problem (HALT)* (for Scheme programs) is the following:

²Thanks to Garey and Johnson, in *Computers and Intractability*.

Instance: Program f in Scheme and some input value(s) x_1, \dots, x_n ($n \geq 1$).
Question: Does the evaluation of $(f\ x_1\ \dots\ x_n)$ in Scheme terminate?

Theorem 27 *The Halting Problem (for Scheme) is undecidable.*

Proof. Proof by contradiction.³ Suppose there were a (Scheme) program h that decided the problem and consider the following program:

```
(define (c y)
  (define (h f x) ...)
  (if (h y y) (c y) #T))
```

It cannot be that h always gives the correct answer; specifically, it cannot give the right answer for $(h\ c\ c)$. This is because c is designed so that $(c\ c)$ goes into an infinite loop if and only if $(h\ c\ c) = \#T$. So if $(h\ c\ c)$ answers yes, that is the wrong answer, and if it answers no, that is also wrong. The only other possibility is that $(h\ c\ c)$ gives no answer, in which case it is also faulty. \square

14.3 Semi-Decidability

We say that a set L is *semi-decidable* if the (partial) function

$$\lambda x. \left\{ \begin{array}{ll} T & \text{if } x \in L \\ \perp & \text{otherwise} \end{array} \right\}$$

is partially computable.

Theorem 28 *A set is semi-decidable if and only if it is recursively enumerable.*

Proof. Suppose $L \in \Sigma^*$ is recursively enumerable. Then to test whether $x \in L$, one can loop through all the natural numbers, looking for an n , such that $e_L(n) = x$

For the other direction, suppose we have a semi-decision procedure P for $x \in L$, an enumerator e_Σ of all words w_i over Σ , and a “stepper” P^n that

³See Doron Zeilberger’s

<http://www.math.rutgers.edu/~zeilberg/mamarim/mamarimhtml/halt.html>.

can run P a given number n of steps. Then we can construct the zig-zag sequence

$$P^0(w_0), P^1(w_0), P^0(w_1), P^2(w_0), P^1(w_1), P^0(w_2), \dots$$

and stop at the n th T , returning the argument w_j . \square

Just running $(f \ x_1 \ \dots \ x_n)$ until the computation terminates—if it terminates—shows that:

Theorem 29 *The Halting Problem is semidecidable.*

The complementary *Non-Halting Problem* (\overline{HALT}) (for Scheme programs) is the following:

Instance: Program f in Scheme and some inputs x_1, \dots, x_n ($n \geq 1$).

Question: Does the evaluation of $(f \ x_1 \ \dots \ x_n)$ in Scheme diverge?

Theorem 30 *The Non-Halting Problem (for Scheme) is not semidecidable.*

The proof is the same.

Theorem 31 *A decision problem P is decidable if and only if both it and its complement \bar{P} are semidecidable.*

The *Self-Halting Problem* ($SELF$) is the following:

Instance: Single-input program f in Scheme.

Question: Does the evaluation of $(f \ f)$ in Scheme terminate?

Theorem 32 *The Self-Halting Problem (for Scheme) is not semi-decidable.*

The proof is essentially same.

Theorem 33 *It is undecidable if two halting single-input programs ever compute the same value.*

We use a checker C ;

Proof.

$$\text{halt}_0(f) = \text{sameh}(\lambda c.C(f, c), \lambda c.T)$$

□

14.4 Standard Models

A programming language P in which one can write an interpreter (and other basic things) must include non-terminating programs. Just consider a program like $c(n) := I(e_P(n), n) + 1$, where I is the interpreter and $e_P(n)$ is the n th program in P .

Conversely, any programming language in which one can only write halting programs cannot compute all computable functions. In particular, if one has basic operations (like function composition), then one cannot write an interpreter within that language.

A language with non-terminating programs (and with standard operations) must have an undecidable halting problem, as we saw is true for Scheme.

Chapter 15

Reductions

I thought the following four [rules] would be enough, provided that I made a firm and constant resolution not to fail even once in the observance of them. The first was never to accept anything as true if I had not evident knowledge of its being so; that is, carefully to avoid precipitancy and prejudice, and to embrace in my judgment only what presented itself to my mind so clearly and distinctly that I had no occasion to doubt it. The second, to divide each problem I examined into as many parts as was feasible, and as was requisite for its better solution. The third, to direct my thoughts in an orderly way; beginning with the simplest objects, those most apt to be known, and ascending little by little, in steps as it were, to the knowledge of the most complex; and establishing an order in thought even when the objects had no natural priority one to another. And the last, to make throughout such complete enumerations and such general surveys that I might be sure of leaving nothing out.

—*René Descartes*

15.1 General Reductions

We say that problem A reduces to problem B , and write $A \times B$ (or $A \leq B$) if given a decision program for B , we can construct a program to decide A .

So if A reduces to B , but A is undecidable, then B is also undecidable.

Let \equiv denote *semantic equivalence*. That is, $f \equiv g$ if $\forall \bar{x}. f(\bar{x}) = g(\bar{x})$. (Also, if one diverges on input \bar{x} , so must the other.)

Theorem 34 *It is undecidable if two halting single-input programs always compute the same value.*

A *stepper* is like an interpreter, but has a bound n on the number of steps.

Proof. Let E be this problem and S be a stepper. We have the following reduction from the halting problem:

$$\text{halt}(f, x) := \neg E(\lambda n. S(f, x, n), \lambda n. \text{fail})$$

□

15.2 Rice's Theorem

A decision problem is *trivial* if the answer is always the same (yes or no), regardless of the input.

A *semantic* decision problem is one that takes programs as input and whose answer depends only the (partial) function computed by it. That is, P is a semantic problem if $P(f) \equiv P(g)$ whenever $f \equiv g$.

Theorem 35 (Rice's Theorem) *All non-trivial semantic problems are undecidable.*

Proof. We can classify semantic problems into two categories. Type A are those for which the problem answers no when asked about “loopy”, the program that loops endlessly; type B answers yes for *loopy*.

Let P be of type A; that is, $P(\text{loopy}) = F$. By non-triviality P answers yes for some program yea ; that is, $P(\text{yea}) = T$. Then the following reduction from HALT proves undecidability:

$$\text{halt}(f, \bar{x}) := P \left(\lambda \bar{y}. \left\{ \begin{array}{ll} \text{yea}(\bar{y}) & \text{if } f(\bar{x}) = f(\bar{x}) \\ \text{--} & \text{otherwise} \end{array} \right\} \right)$$

For type B use

$$\text{halt}(f, \bar{x}) := \neg P \left(\lambda \bar{y}. \left\{ \begin{array}{ll} \text{nay}(\bar{y}) & \text{if } f(\bar{x}) = f(\bar{x}) \\ \text{--} & \text{otherwise} \end{array} \right\} \right)$$

instead. □

For example, $\text{Idem} = \{f \in \mathbf{Baby} : \forall x. f(x) = f(f(x))\}$ is undecidable. This is a type B problem, since $\text{loopy} \in \text{Idem}$, and it is nontrivial because $\text{reverse} \notin \text{Idem}$. Since it is semantic (talks only about f), the theorem applies.

Chapter 16

Enumerability

16.1 Recursive Enumerability

Definition 19 A formal language L is recursively enumerable (r.e.) if there is a computer (Baby) program that accepts exactly the words in L .

Theorem 36 A formal language $L \subset \Sigma^*$ is r.e. if any of the following hold:

1. There is a program f_L such that $f_L(x) = T \Leftrightarrow x \in L$.
2. There is a program $e_L : \mathbf{N} \rightarrow \Sigma^*$ such that $L = \{e_L(n) : n \in \mathbf{N}\}$.
3. There is a 1-1 (injective) program $e_L : \mathbf{N} \rightarrow \Sigma^*$ such that $L = \{e_L(n) : n \in \mathbf{N}\}$.
4. There is a program p_L (without input) that prints all of L (but nothing not in L).
5. There is a program p_L that prints all of L without repetitions.

In particular, a formal language L is (recursively) enumerable (r.e.) if there is a Turing machine that accepts exactly the words in L . It is *co-r.e.* if its complement \bar{L} is r.e. Let \mathcal{RE} be the set of r.e. languages (over some alphabet).

16.2 Mapping Reductions

A *mapping* (or *many-one*) reduction from a set $A \subseteq L$ to a set $B \subseteq M$ is a computable function $f : L \rightarrow M$ such that $x \in A$ if and only if $f(x) \in B$.

If there exists a mapping reduction from A to B , we write $A \propto_m B$ (or $A \leq_m B$).

Theorem 37 *If $A \propto_m B$ and B is r.e., then so is A .*

By the same token:

Corollary 1 *If A is not r.e., then neither is B .*

The reduction we have seen for $\text{Halt} \propto \text{Halt}_0$ is a mapping reduction.

Question. Why don't ordinary reductions give the same result?

Definition 20 *A language $L_0 \in \mathcal{RE}$ is (r.e.) complete if for every $L \in \mathcal{RE}$ we have $L \propto_m L_0$.*

Theorem 38 *HALT is complete.*

Proof. Let ℓ be a program that accepts an r.e. language L and diverges for all inputs not in L . The mapping is $f : x \mapsto \langle \ell, x \rangle$. \square

16.3 Oracles

It follows that if one could miraculously know how to always answer the halting question, all r.e. languages would become recursive.

An *oracle* (Urim and Thummim) is a “magical” program statement that always answers some given (perhaps undecidable) problem.

Suppose we had such an oracle. Would all uncomputable functions become computable? Would there be anything uncomputable?

Chapter 17

Turing Machines

The most popular model of computation used in theoretical computer science is that of Turing Machines. A *Turing machine* is a finite program operating on strings of symbols from some alphabet Γ) and consisting of a labelled sequence of the following operations:

- Move left ($[\ell]$ Left)
- Move right ($[\ell]$ Right)
- Branch on input ($[\ell]$ Case a : goto ℓ_a ; ...; z : goto ℓ_z)
- Write a ($[\ell]$ Out a) for any $a \in \Gamma$
- Accept ($[\ell]$ Yes)
- Reject ($[\ell]$ No)

We imagine the input given as a finite sequence of symbols, on a one-dimensional “tape”, with the current symbol at the start of the computation being the first input letter. The program can move back and forth over the sequence and can overwrite any symbol it wants. When it tries to move further to the left or right than the sequence of written symbols, the tape gets extended with a *blank* symbol, which can be overwritten any way the program wishes. In this way, the program’s memory is unbounded.

A machine M accepts or rejects an input word $w \in \Gamma^*$ if a computation beginning with input w reaches an Accept or Reject command, respectively.

Let **TM** be the set of languages accepted by Turing machines. We will see that TMs are no more or less powerful than ordinary programming languages. So:

Theorem 39 $\text{CFL} \subsetneq \text{TM} = \text{r.e.}$

It is also clear that

Theorem 40 $\text{TM} \subsetneq \mathcal{P}(\Sigma^*)$

Proof. There are only countably many Turing machine programs and uncountably many languages $L \subseteq \Sigma^*$ \square

17.1 Example

The following is a Turing machine's transition function for the language $\{a^n b^n c^n : n \in \mathbf{N}\}$:

input	a	b	c	X	Y	Z	\square
q_0	q_1, X, R	no	no	no	q_4, Y, R	no	yes
q_1	q_1, a, R	q_2, Y, R	no	no	q_1, Y, R	no	yes
q_2	no	q_2, b, R	q_3, Z, L	no	no	q_4, Z, R	no
q_3	q_3, a, L	q_3, b, L	no	q_0, X, R	q_4, Y, L	q_3, Z, L	no
q_4	no	no	no	q_4, X, R	q_4, Y, R	q_4, Z, R	yes

Its tape alphabet is $\Gamma = \{a, b, c, X, Y, Z, \square\}$, where \square is the blank symbol. Its states are $\{q_0, q_1, q_2, q_3, q_4, \text{yes}, \text{no}\}$, where q_0 is the initial state and yes and no are the accept and reject states, respectively.

17.2 Universal Turing Machines

A *universal machine* is a machine U that acts as an interpreter of its own language. For example, the following is said to be the smallest known universal Turing machine:

state	1	2	3	4	5	6	7
Y	\square L1	\square L1	YL3	YL4	YR5	YR6	\square R7
\square	\square L1	YR2	Halt	YR5	YL3	AL3	YR6
1	1L2	AR2	AL3	1L7	AR5	AR6	1R7
A	1L1	YR6	1L4	1L4	1R5	1R6	\square R2

17.3 Finite State Automata Revisited

Consider a programming language with the following operations:

- Next input symbol ($[\ell]$ Right)
- Branch on input ($[\ell]$ Case a : goto ℓ_a ; ...; z : goto ℓ_z)
- Accept ($[\ell]$ Yes)
- Reject ($[\ell]$ No)

This language accepts exactly the same family of languages as do finite state automata. (Why?) The point is that once makes a move right it can never go back, so there is no need for long-term memory.

Question. What happens if DFAs are enhanced with left moves ($[\ell]$ Left)?

Question. What happens if DFAs are enhanced with write operations ($[\ell]$ Out a)?

17.4 Context-Free Languages Revisited

For Turing machines, configurations $(\alpha q \beta)$ consist of the internal state ($q \in Q$), the position of the head on the tape, and the finite written portion of the tape ($\alpha \in \Sigma^*$ to the left of the head and $\beta \in \Sigma^*$ from the head to the right end).

We already know that the context-free languages are not closed under complement. In fact the fullness problem for CFLs is undecidable. This is because the *non-computations* of a Turing machine M with input x can be presented as a context-free language $L(G)$, which include

- computations that do not begin with an initial configuration;
- computations that do not end in a final state;
- configurations displaying more than one state;
- adjacent configurations that are impossible;
- etc.

17.5 Variations

A large variety of variations on the basic Turing machine model have been proposed.

One-way infinite tape. Instead of a tape that can be extended either to the left or right with as many blanks as one wants, we can fix the beginning of the tape. Any attempt to move left of the beginning is ineffectual. It is often convenient to place a special symbol (e.g. \blacktriangleright) to indicate the left end of the tape. (If it is not supplied with the input, then it can be inserted by the program with one quick pass over the input to shift the real input right.)

Two-dimensional sheet. If we imagine that the machine operates on a grid, rather than on a tape, then we need to incorporate up and down motions, as well as left and right.

Multiple heads. One can also imagine that the machine has more than one read/write head that operate in tandem.

Bundled instructions. The formal definition of Turing machines usually bundles the different instructions into one transition that depends on both an internal state and the symbol below the read/write head.

Definition 21 *A Turing machine is composed of the following components:*

- *Tape alphabet* (Γ)
- *Blank symbol* ($\square \in \Gamma$)
- *Input alphabet* ($\Sigma \subset \Gamma$)
- *Internal memory: finite set of states* (Q)
- *External memory: unbounded sequence of symbols (from alphabet Γ) on a tape*
- *Program: transition function* ($\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$)
- *Initial state* ($q_0 \in Q$)
- *Accept states* ($F \subseteq Q$)

In this version, with every read, a write operation is performed, as well as a move left or right. Turing machines may also have a reject state (halt, but don't accept).

It is convenient to represent the complete (internal and external) state—the *configuration*—of such a Turing machine as two words, the symbols α to the left of the read/write head and the rest β , separated by the current state q , as in $\alpha q \beta$. So, the initial configuration is $q_0 w$, where $w \in \Sigma^*$ is the input.

For every TM M over an alphabet Σ , there is a Scheme program B_M such that, for any input string $x \in \Sigma^*$, machine M accepts x iff program $B_M(x)$ evaluates to T .

$$\begin{aligned}
 \text{accept-tm?}(\langle \Sigma, Q, \Gamma, \square, \delta, q_0, F \rangle, w) &:= tm(q_0, \epsilon, w) \\
 \text{where } tm(q_f, u, v) &:= T \text{ if } q_f \in F \\
 tm(q, u, \epsilon) &:= tm(q, u, \square) \text{ if } q \notin F \\
 tm(q, \epsilon, av) &:= \\
 &\begin{cases} tm(q', \epsilon, bv) & \text{if } \delta(q, a) = (q', b, L), q \notin F \\ tm(q', b, v) & \text{if } \delta(q, a) = (q', b, R), q \notin F \end{cases} \\
 tm(q, cu, av) &:= \\
 &\begin{cases} tm(q', u, cbv) & \text{if } \delta(q, a) = (q', b, L), q \notin F \\ tm(q', bcu, v) & \text{if } \delta(q, a) = (q', b, R), q \notin F \end{cases}
 \end{aligned}$$

In Scheme, TM computations of functions (not just acceptance) is implemented as follows:

```

(define (run m q0 F i)
  (define (tm m l r q)
    (if (null? r)
        (tm m l (cons 'B '()) q)
        (let ((mv (move m q (car r))))
          (cond
           ((member (car mv) F)
            (append (reverse l) (cons (cadr mv) (cdr r))))
           ((equal? (caddr mv) 'L)
            (tm m (cdr l) (cons (car l) (cons (cadr mv) (cdr r)))
                 (car mv)))
           ((equal? (caddr mv) 'R)
            (tm m (cons (cadr mv) l) (cdr r) (car mv)))))))
  (define (move m q a)

```

```

(if (equal? (list q a) (caar m))
    (cadr (car m))
    (move (cdr m) q a))
(tm m nil i q0))

```

Stay-put instruction. The above model can be extended with the stay-put option, denoted by direction S.

Multiple tapes. One can imagine several tapes, each with its own read/write head. We will delve more deeply into this variant and show why it does not add computational power. By the same token, none of the other variants are more powerful than the original.

To simulate a two-tape machine with a one-tape machine, we use a 2-channel tape. The states are triples $Q \times \Gamma \times \{0, 1, 2, \dots, 7\}$, where the second component remembers a symbol from one of the tapes, and the last component keeps track of which stage of the simulation process is in progress. The tape symbols are pairs, $(\Gamma \cup \underline{\Gamma} \cup \{\blacktriangleright\})^2$, for the upper and lower channel, each representing one of the tapes, and with the position of the reading head noted by an underlined symbol from a copy $\underline{\Gamma} = \{\underline{a} : a \in \Gamma\}$ of the tape alphabet. Also, we mark the beginning of the tape with the pair $[\blacktriangleright, \blacktriangleright]$.

So to simulate a two-tape step $\delta : (q, a, a') \mapsto (r, b/d, b'/d')$ requires a sequence of moves, changing a to b and a' to b' and moving the underline according to directions d and d' . For example, if the configuration looks like

$$[q, \blacktriangleright, 0] \ [\blacktriangleright, \blacktriangleright][a_1, a'_1] \cdots [\underline{a}, a'_i][a_{i+1}, a'_{i+1}] \cdots [a_{j-1}, a'_{j-1}][a_j, \underline{a}'] \cdots [\square, \square]$$

and $\delta : (q, a, a') \mapsto (r, b/R, b'/L)$ then after the sequence of moves, it should be

$$[r, \blacktriangleright, 0] \ [\blacktriangleright, \blacktriangleright][a_1, a'_1] \cdots [b, a'_i][\underline{a_{i+1}}, a'_{i+1}] \cdots [a_{j-1}, \underline{a'_{j-1}}][a_j, a'] \cdots [\square, \square]$$

Let $\Gamma' = \Gamma \cup \{\blacktriangleright\}$ and $\Gamma'' = \Gamma' \cup \underline{\Gamma}$. The stages are as follows:

0. Scan top channel and memorize symbol

$$\begin{aligned} [q, \blacktriangleright, 0], [x, y] &\mapsto [q, \blacktriangleright, 0], [x, y], R & (q \in Q, x \in \Gamma', y \in \Gamma'') \\ [q, \blacktriangleright, 0], [\underline{a}, y] &\mapsto [q, a, 1], [a, y], L & (\underline{a} \in \underline{\Gamma}) \end{aligned}$$

1. Return to start

$$\begin{aligned} [q, a, 1], [x, y] &\mapsto [q, a, 1], [x, y], L & (q \in Q, x \in \Gamma, y \in \Gamma \cup \underline{\Gamma}) \\ [q, a, 1], [\blacktriangleright, \blacktriangleright] &\mapsto [q, a, 2], [\blacktriangleright, \blacktriangleright], R \end{aligned}$$

2. Scan bottom channel, memorize symbol, write

$$\begin{aligned} [q, a, 2], [x, y] &\mapsto [q, a, 2], [x, y], R & (q \in Q, x \in \Gamma'', y \in \Gamma') \\ [q, a, 2], [x, \underline{a}'] &\mapsto [q, a', 3], [x, b'], d' & (\delta : (q, a, a') \mapsto (r, b/d, b'/d')) \end{aligned}$$

3. Move head on bottom channel

$$[q, a', 3], [x, y] \mapsto [q, a', 4], [x, \underline{y}], L$$

4. Return to start

$$\begin{aligned} [q, a', 4], [x, y] &\mapsto [q, a', 4], [x, y], L \\ [q, a', 4], [\blacktriangleright, \blacktriangleright] &\mapsto [q, a', 5], [\blacktriangleright, \blacktriangleright], R \end{aligned}$$

5. Scan and write top channel and change state

$$\begin{aligned} [q, a', 5], [x, y] &\mapsto [q, a', 5], [x, y], R \\ [q, a', 5], [\underline{a}, y] &\mapsto [r, a', 6], [b, y], d & (\delta : (q, a, a') \mapsto (r, b/d, b'/d')) \end{aligned}$$

6. Move head on top channel and return to start

$$\begin{aligned} [r, a', 6], [x, y] &\mapsto [r, \blacktriangleright, 6], [\underline{x}, y], L \\ [r, \blacktriangleright, 6], [x, y] &\mapsto [r, \blacktriangleright, 6], [x, y], L \\ [r, \blacktriangleright, 6], [\blacktriangleright, \blacktriangleright] &\mapsto [r, \blacktriangleright, 0], [\blacktriangleright, \blacktriangleright], R \end{aligned}$$

Nondeterminism A nondeterministic Turing machine (NTM) is like an ordinary deterministic one (DTM), except that there may be more than one transition for any given current state and symbol.

Nondeterminism does not add power to Turing machines, in the sense that for any formal language that is accepted by an NTM, there is also a DTM for the same language.

Let **NTM** denote the set of languages recognizable by a nondeterministic Turing machine.

Theorem 41 **NTM = TM.**

Proof. The computations of an NTM can be simulated breadth-first. \square

Chapter 18

Church-Turing Thesis

*“is an expression which, when preceded by its translation, placed in
quotation marks,
into the language originating on the other side of the Channel, yields
a falsehood”
est une expression qui, quand elle est précédée de sa traduction, mise
entre guillemets,
dans la langue provenant de l’autre côté de la Manche, cre une
fausseté.*

—Douglas R. Hofstadter, *Gödel, Escher, Bach: An Eternal Golden Braid*

We will show that various models of computation are equipotent, that is, they have the same computational power and compute the same set of partial functions, though perhaps using different representations. If model L computes a set F of partial functions, then we will need to show that the other model, M , computes the same set F of functions, no more and no less.

18.1 Turing Programs

Definition 22 *A Turing program over a tape alphabet Γ consists of a finite sequence of optionally labelled instructions (labels taken from some set Λ) of the following types:*

- Right, meaning move the head right.
- Left, meaning move the head left, if possible.
- Write a , where $a \in \Gamma$, meaning write the letter a under the head (and don’t move).

- If A go to ℓ , where $A \subseteq \Gamma$, $\ell \in \Lambda$, meaning to continue with the (first) instruction labelled ℓ if the letter under the head belongs to the list (or set) A , and to continue with the sequentially following instruction, otherwise.
- Halt, meaning that computation halts, and the output is what is written on the tape.

One can use “Go to ℓ ” as short for “If Γ go to ℓ ”.

For example, the following is the same example as in Section 17.1, in this programming language:

<p>0: if b,c,X,Z go to no if \square go to yes if Y go to 4 Write X</p> <p>1: Right if c,X,Z go to no if \square go to yes if a,Y go to 1 Write Y</p> <p>2: Right if a,X,Y,\square go to no if b go to 2 if Z go to 4 Write Z</p>	<p>3: Left if c,\square go to no if a,b,Z go to 3 if Y go to 4a Right Go to 0</p> <p>4a: Left</p> <p>4: if a,b,c go to no if \square go to yes Right Go to 4</p> <p>yes: Halt yes</p> <p>no: Halt no</p>
---	--

It is easy to see that Turing programs is no more or less powerful than standard Turing machines. The translation from the latter is as follows: For each transition table row for $\delta(q, a_i) = (r_1, b_i, L/R)$, as above, the program contains the following instructions:

q : If a_1 go to q_1
If a_2 go to q_2
:
If a_n go to q_n
 q_1 : Write b_1
Left (if $d_1 = L$) / Right (if $d_1 = R$)

Go to r_1
 q_1 : Write b_1
 Left/Right
 Go to r_1
 \vdots
 q_n : Write b_n
 Left/Right
 Go to r_n

In addition, we have

q_f : Halt

The program begins with the start-state's statement, q_0 .

For the other direction, let's number each statement sequentially. The translation is straightforward:

Program	Machine
k : Right	$\delta(k, a) = (k + 1, a, R)$
k : Left	$\delta(k, a) = (k + 1, a, L)$
k : Write b	$\delta(k, a) = (k + 1, b, S)$
k : If A go to ℓ	$\delta(k, a) = (\ell, a, S)$, for all $a \in A$ $\delta(k, b) = (k + 1, b, S)$, for all $b \notin A$

If we are only interesting in acceptance of languages, then we would want two separate Halt instructions, for acceptance and rejection, namely:

- Halt Yes
- Halt No

Definition 23 A k -tape Turing program over a tape alphabet Γ consists of a finite sequence of optionally labelled instructions (labels taken from some set Λ , tape names from Ω , $|\Omega| = k$) of the following types:

- Right i , where $i \in \Omega$, meaning: move the head on the tape named i
- Left i , where $i \in \Omega$
- Write a i , where $a \in \Gamma$, $i \in \Omega$
- If $i \in A$ go to ℓ , where $A \subseteq \Gamma$, $\ell \in \Lambda$, $i \in \Omega$
- Halt

One of the tapes in Ω is designated for input/output.

These can be simulated by ordinary Turing programs in the same fashion as done for Turing machines in Chapter 17.

Definition 24 A nondeterministic Turing program consists of the same types of instructions as the ordinary Turing program (Definition 22 above), plus

- Try ℓ , where $\ell \in \Lambda$ (the set of labels), meaning nondeterministically choose between continuing with the next instruction or else jumping to ℓ .

We already saw how to simulate nondeterminism with a breadth-first search using a 3-tape Turing machine.

18.2 Counter Machines

Counter machines have a fixed, finite number of registers, each of which can contain any natural number. One of the registers is designated for input/output.

Definition 25 A k -register Counter program consists of a finite sequence of optionally labelled instructions (labels taken from some set Λ , counter names from Ω , $|\Omega| = k$) of the following types:

- Inc i , where $i \in \Omega$, meaning add 1 to register i
- Dec i , where $i \in \Omega$, meaning add 1 to register i , and doing nothing if it's 0
- If $i = 0$ go to ℓ , where $\ell \in \Lambda$, $i \in \Omega$, meaning to go to instruction ℓ if $i = 0$, and continue with the next instruction, otherwise
- Halt

For example, to copy the contents of (register) i to (register) j , zapping i in the process:

```
 $\ell$ : if  $i=0$  go to cont  
Dec  $i$ ; Inc  $j$ ; Go to  $\ell$   
cont: ...
```

(We use semicolons to separate instructions written on the same line.) Let's use the following macro for this sequence of instructions:

j := i

The following program computes $\lfloor \log_2 z \rfloor$, the integer part of the binary logarithm of the contents of register z , placing the answer in register i , and using a third register y :

```

start: Dec z
      If z=0 go to halt
      ℓ: Dec z
      If z=0 go to copy
      Inc y
      Dec z
      Go to ℓ
copy: z:= y
      Inc i
      Go to start
halt: Halt

```

To convert a counter program in a Turing one, replace as follows:

Inc i	→	Right i
Dec i	→	Left i
If i = 0 go to ℓ	→	If i • go to ℓ

Three counters suffice for any counter-computable numeric function by representing the k registers as a counter $c = 2^{r_1} 3^{r_2} \cdots p_k^{r_k}$, where the bases of the exponentiation are prime numbers. We will also use register a , leaving it zeroed between operations.

For example, to increment r_2 , we need to multiply by $p_2 = 3$, so we do the following:

```

1: if c=0 go to copy
  Dec c; Inc a; Inc a; Inc a; Go to 1
copy: c := a

```

To check whether $r_2 = 0$, we need to see whether c is divisible without remainder:

```

1: if c=0 go to yes
  Dec c; Inc a; if c=0 go to no

```

Dec c; Inc a; if c=0 go to no
 Dec c; Inc a; Go to 1
 yes: c := a; Go to ℓ
 no: c := a

Abbreviating this as the macro

If $c \bmod 3 > 0$ go to ℓ

we can decrement as follows:

If $c \bmod 3 > 0$ go to cont
 1: If c=0 go to cont
 Dec c; Dec c; Dec c; Inc a; Go to 1
 cont: c := a

Comment. A 2-counter machine, however, cannot compute the function $n \mapsto 2^n$. On the other hand, if a 3-counter machine computes a function $f(n)$, then there is a 2-counter machine that computes $2^n \mapsto 2^{f(n)}$.

18.3 RAM Machines

Definition 26 A RAM program consists of a finite sequence of optionally labelled instructions (labels taken from some set Λ , register names from Ω) of the following types:

- $i := n$, where $i \in \Omega$, $n \in \mathbf{N}$
- $i := j + k$, where $i, j, k \in \Omega$
- $i := j - k$, where $i, j, k \in \Omega$, and the result is zero when $k > j$
- $i := m[j]$, where $i, j \in \Omega$, and $m[0..\infty]$ is an unbounded “external” memory, each entry of which is a natural number, all initialized to 0 at the start of computation
- $m[j] := i$, where $i, j \in \Omega$
- *Halt*

To simulate these by Counter programs, represent pairs (x, y) by $2^x(2y + 1)$. It is easy to write Counter macros for all the following operations:

- $i := n$
- $i := j$
- $i := j + k$
- $i := j - k$
- $i := j + n$
- $i := j - n$
- $i := 2j$
- $i := j \div 2$
- $i := j \bmod 2$
- $i := 2^j$
- if $i > 0$ go to ℓ

where i, j are registers and n is any natural number. We also need

$z := pr(x, y) [= 2^x(2y + 1)]$	$x := 1^{st}(z)$	$y := 2^{nd}(Z)$
$z := 2y$ $z := z + 1$ $t := x$ $a : \text{if } t = 0 \text{ go to } c$ $z := 2z$ $t := t - 1$ $\text{go to } a$ $c :$	$t := z$ $x := 0$ $\text{if } z = 0 \text{ go to } c$ $a : j := t \bmod 2$ $\text{if } j > 0 \text{ go to } c$ $t := t \div 2$ $x := x + 1$ $\text{go to } a$ $c :$	$t := z$ $\text{if } z = 0 \text{ go to } c$ $a : j := t \bmod 2$ $\text{if } j > 0 \text{ go to } c$ $t := t \div 2$ $\text{go to } a$ $c : t := t - 1$ $y := t \div 2$

The memory operations are implemented by a counter m that contains $pr(m[1], pr(m[2], \dots))$, and operates as follows:

$i := m[j]$	$m[j] := i$	
$t := j$ $y := m$ $a : \text{if } t = 0 \text{ go to } c$ $b : y := 2^{nd}(y)$ $t := t - 1$ $\text{go to } a$ $c : i := 1^{st}(y)$	$t := j$ $y := 0$ $a : \text{if } t = 0 \text{ go to } e$ $h := 1^{st}(m)$ $m := 2^{nd}(m)$ $y := pr(h, y)$ $t := t - 1$ $\text{go to } a$	$e : m := 2^{nd}(m)$ $m := pr(i, m)$ $t := j$ $d : \text{if } t = 0 \text{ go to } c$ $h := 1^{st}(y)$ $y := 2^{nd}(y)$ $m := pr(h, m)$ $t := t - 1$ $\text{go to } d$ $c :$

RAM machines, in turn, can simulate Scheme programs: witness the compiler in the Scheme book.¹

18.4 Arbitrary Grammars

An *arbitrary (type 0) grammar* allows productions rules with any left side that has at least one nonterminal. Turing machines can be simulated by such arbitrary grammars:

Symbols: Tape symbols plus X, Y .

Variables: Symbols Q_i for states q_i , plus A_j for each tape symbol a_j , plus S, B, C

The simulation proceeds in three stages:

1. $S \Rightarrow \dots \Rightarrow wXQ_0wX$ (initial state q_0)
2. $Q_0w \Rightarrow \dots \Rightarrow uQ_fv$ (final state q_f)
3. $XuQ_fvX \Rightarrow \dots \Rightarrow \varepsilon$

$$\begin{array}{l}
 S \rightarrow CB \\
 B \rightarrow YX \\
 B \rightarrow a_j A_j B \quad \text{for each } a_j \\
 A_j a_k \rightarrow a_k A_j \\
 A_j Y \rightarrow Y a_j \\
 C a_k \rightarrow a_k C \\
 CY \rightarrow XQ_0 \\
 \hline
 Q_i a_j \rightarrow a_k Q_l \quad \text{for each right move} \\
 Q_i X \rightarrow a_k Q_l X \quad a_j = \square \\
 a_m Q_i a_j \rightarrow Q_l a_m a_k \quad \text{for each left move} \\
 a_m Q_i X \rightarrow Q_l a_m a_k X \quad a_j = \square \\
 \hline
 Q_f a_j \rightarrow Q_f \\
 a_m Q_f \rightarrow Q_f \\
 XQ_f X \rightarrow \varepsilon
 \end{array}$$

18.5 Equipotence

To summarize, the following propositions hold:

Theorem 42

¹See [nachum/models/programs/C.scm](#).

- a. *Turing programs simulate k-tape Turing programs (and vice-versa).*
- b. *k-tape Turing programs simulate k-register Counter programs.*
- c. *k-register Counter programs simulate RAM programs.*
- d. *RAM programs simulate Scheme programs.*
- e. *Scheme programs simulate Turing programs.*
- f. *2-register Counter programs simulate 3-register Counter programs.*
- g. *3-register Counter programs simulate k-register Counter programs.*
- h. *Turing programs simulate non-deterministic Turing programs (and vice-versa).*
- i. *Unrestricted grammars simulate non-deterministic Turing machines.*

18.6 The Church-Turing Thesis

The *Church-Turing Thesis* states that these models comprise all the functions that can be humanly computed.

Appendix A

The Baby Language

Our programming language, called **Baby**, has the following commands and Scheme equivalents:

Baby	Scheme	Meaning
$\alpha := \beta$	(define $\alpha \beta$)	let (the function or constant) α be defined to have the value of β
$\alpha(\beta_1, \dots, \beta_n)$	($\alpha \beta_1 \dots \beta_n$)	apply function α to arguments β_1, \dots, β_n
γ where $\alpha_1 := \beta_1 \dots \alpha_n := \beta_n$	(let (($\alpha_1 \beta_1$) \dots ($\alpha_n \beta_n$)) γ)	let the α_i be defined to have the value β_i in the expression γ
$\lambda x_1 \dots x_n. \alpha$	(lambda ($x_1 \dots x_n$) α)	an anonymous function, with parameters x_1, \dots, x_n and body α
$\begin{cases} \beta_1 & \alpha_1 \\ \beta_2 & \alpha_2 \\ \vdots & \\ \alpha_n & \text{o/w} \end{cases}$	(cond ($\alpha_1 \beta_1$) ($\alpha_2 \beta_2$) \dots (else β_n))	conditional statement: if α_1 is true then β_1 , otherwise, if α_2 then β_2, \dots , otherwise β_n
\mathbf{N}	0, 1, ...	natural numbers
$\in \mathbf{N}$	number?	test for numbers
$+, \cdot$	+, *	addition, multiplication
$\alpha - \beta$	(if ($> \beta \alpha$) 0 ($- \alpha \beta$))	natural subtraction (never negative)
Σ	'a','b',..., 'z, '0',..., '9', 'lambda', '=', ...	finite set of atoms
$\in \Sigma$	symbol?	test for atomic symbol
$=, \neq$	=, \neq	equality, disequality of naturals and atoms
$>, <, \geq, \leq$	>, <, >=, <=	inequalities of naturals
true, false	#T, #F	true, false
\neg, \vee, \wedge	not, or, and	logical negation, disjunction, conjunction
ϵ	'()	empty list
$\langle \alpha_1, \dots, \alpha_n \rangle$	(list $\alpha_1 \dots \alpha_n$)	list containing $\alpha_1, \dots, \alpha_n$
$\alpha : \beta$	(cons $\alpha \beta$)	list with first element α and continuation β
$= \epsilon$	null?	test for empty list
hd(α)	(car α)	head element of (nonempty) list α
tl(α)	(cdr α)	all but first element of (nonempty) list α
" $a_1 a_2 \dots a_n$ "	('a ₁ 'a ₂ ... 'a _n)	string containing symbols a_1, a_2, \dots, a_n