GRADUATE TEXTS IN COMPUTER SCIENCE

A Recursive Introduction to the Theory of Computation

Carl H. Smith

Springer-Verlag

Carl H. Smith

A RECURSIVE INTRODUCTION TO THE THEORY OF COMPUTATION

Springer-Verlag
New York Berlin Heidelberg London Paris
Tokyo Hong Kong Barcelona Budapest

Carl H. Smith
Department of Computer Science
University of Maryland
College Park, MD 20742 USA

Series Editors

David Gries
Fred B. Schneider

Department of Computer Science
Cornell University
Upson Hall
Ithaca, NY 14853-7501 USA

Library of Congress Cataloging-in-Publication Data
Smith, Carl. 1950–
A recursive introduction to the theory of computation / Carl Smith.
p. cm. — (Graduate texts in computer science)
Includes index.
ISBN 0-387-94332-3
1. Electronic digital computers—Programming. 2. Recursive functions—Data processing. I. Title. II. Series.
QA76.6.S61547 1994
511.3'5—dc20 94-21785

Printed on acid-free paper.

Production coordinated by Publishing Network and managed by Henry Krell; manufacturing supervised by Jacqui Ashri.
Photocomposed copy prepared from the author's LaTeX files.
Printed and bound by R.R. Donnelley and Sons, Harrisonburg, VA.
Printed in the United States of America.

9 8 7 6 5 4 3 2 1

ISBN 0-387-94332-3 Springer-Verlag New York Berlin Heidelberg
ISBN 3-540-94332-3 Springer-Verlag Berlin Heidelberg New York

Preface

Once upon a time, I was quite content to use the excellent text *An Introduction to the General Theory of Algorithms* by M. Machtey and P. Young for my graduate theory of computation course. After the publisher declined to reprint the book, I hastily threw together some notes. Consequently, much of the organization, notation and actual proofs were taken straight from Machtey and Young's book. The notes soon took an electronic form. What follows is the result of several semesters' worth of embellishments and classroom testing. Over the semesters, many students have found typos, suggested clarifications and come up with interesting questions that became exercises. This document has been significantly improved due to their efforts. I would also like to thank my colleague Bill Gasarch for proofreading, discussions, agreements, disagreements and for creating a large cardinal number of problems. Ken Regan and the anonymous referees made several valuable comments.

The material presented can be covered in a single semester. I have resisted the temptation to add material "to be covered, time permitting" since what material I would want to add changes from semester to semester. If time is tight, or some other pressing topic comes up that must be covered, then the material on other models, alternative characterizations of acceptable programming systems, as well as the section on restricted programming systems, may be omitted without loss of continuity.

The title of this book contains the word "recursion," which has several meanings, many of which are appropriate. The original meaning of the word *recurse* was "a looking back." Indeed, in this book we look back at the fundamental results that shaped the theory of computation as it is today. The object of study for most of the book is the partial recursive functions, functions that derive their name from the way they are defined, using an operator that "looks back" to prior function values. There is also a *recursion theorem*, which enables us to construct programs that can look back on their own code. In this treatment, recursion theorems are introduced relatively early and used often.

January 1994

Contents

Introduction

The goal of this book is to bring to light certain computational phenomena. The features we are interested in pertain to *computation* and not just some particular paradigm of computation such as programming in C++ on a workstation running UNIX. As a consequence, the conclusions we reach about computation will be valid for today's computing systems and the computing systems of the future. In gaining such generality, we lose immediate applicability of the results studied to contemporary computing practices. The main intellectual benefit to students in this course is a broadened intuition about computation. It is assumed that the reader has ample experience programming in more than one programming language. This book will make you think about computing in a new light. Some of your current intuitions about computation will be challenged. This challenge comes not from radical ideas, but rather from fundamental facts concerning computation. What is presented below is *not* speculation, every claim is proven using sufficiently rigorous mathematics. Consequently, the reader should have a firm background in discrete mathematics. A course in automata theory is also desirable as a prerequisite.

Most people think about computation in terms of their experiences with a real computing system or some formal model of computing. The traditional vehicle for studying computation in general is to introduce a model and then generalize. Indeed, we will do likewise. Before beginning our study, it may be helpful to attempt to define computation independently of any technical devices. The following definition is intended to be used as an intuition — nothing in the remainder of the course depends on it. *Computation is the movement and transformation of data. The transformations are accomplished by moving the data through entities that we will call processors.*

Some examples are in order. A personal computer can be regarded as a processor, transforming keystrokes, mouse movements, floppy disks and electricity from the wall socket into the image you see on the screen. At a finer level, each integrated circuit can be viewed as a processor, performing some computation by transforming its inputs and moving the result to some other integrated circuit. In this case the computation performed is a small piece of some larger computation. The above definition of computation is applicable at any level of detail. A logic gate is a processor that transforms

its inputs. With this general view, computation is ubiquitous. For example, your television set is a processor that moves inputs from the controls, the cable (or antenna) and the wall socket and transforms them into the vision you see on the screen.

The first section of the book deals with various models of computation. Each of these models implicitly moves and transforms data. A *model of computation* is essentially a programming language. However, one would not want to do any actual programming with them. They have few control structures, rigid variable names, only simple arithmetic operators and no built-in functions. The redeeming virtue of these models is that it is easier to prove things about them than with a full blown language like Pascal with a rich set of control structures. The proofs are complicated enough with a syntactically simple language. Using a real programming language would only make our arguments notationally extremely cumbersome, without adding any additional insight. Three models, each offering a different perspective on computing, are presented and shown to have equivalent computing power.

Armed with the intuition that it doesn't really matter which model of computation that we use for our study, we will pick a generic one. Programs will be referred to by name, functionality and arguments. Freed from dealing with the cumbersome details of syntax, we will explore the limitations of effective computation in the section on basic recursion theory. Self reference will be introduced as a fundamental and basic tool for constructing algorithms. Other tools for manipulating and constructing algorithms will be introduced and compared.

Given firm knowledge about what is and what is not computable, we move on to consider the complexity of computing the things that are computable. Again our study is quite general. The notion of a *complexity measure* is formalized. General properties of all complexity measures are shown to exist. In this way, we prove results that pertain to complexity when viewed as run time and also to complexity viewed as space used. Finally, we focus on time and space measures and distinguish the computable functions as being either feasible to compute or otherwise.

1

Models of Computation

We begin our study by developing several models of computation, each of which reflects all of the features inherent in computation. In an effort to simplify the arguments, all artifacts of computation will be absent from our models. We will start with existing computational paradigms and remove the "bells and whistles" of convenience, arriving at simplified versions of each paradigm that contain all the fundamentally important features of the original. The simplified versions will be shown to be equivalent in a strong sense. This will suggest a model-independent view of computation in terms of "programs" computing functions.

The first artifact of computation that we will dismiss is the notion that arguments may be of different types. All our inputs, outputs, temporary variables, etc. will be natural numbers (members of $\mathbb{N}$). We will argue informally that all the other commonly used argument types are included solely for convenience of programming and are not essential to computation. The argument is based on how computers encode everything into sequences of bits. Boolean arguments can be represented by using the first two members of $\mathbb{N}$, 0 and 1. Floating point numbers, as a consequence of their finite representation, are actually rational. Rational numbers are pairs of natural numbers. In the section on coding, we will show how to encode pairs of natural numbers as a single natural number. Consequently, members of $\mathbb{N}$ suffice to represent the rational numbers, and hence, the floating point numbers. Natural numbers represent character strings via an *index*, or position, in some standard list of all strings. For example, a standard list of all strings using the alphabet $\{a \ldots z\}$ would start by associating 0 with the empty string and then numbering the strings in lexicographical order: $a, b, \cdots, z, aa, ab, \cdots, az, ba, \cdots$.

Analog computation can also be viewed as computing with natural numbers. This follows from the observation that any voltage level can only be measured in increments determined by the measuring device. Even though the voltages are theoretically continuous, all our devices to measure voltages render discrete values. For common examples of turning essentially analog information into a numeric representation we need look no further than the digitally encoded music found in compact disk technology

or the digitally encoded images of high definition television and compact disk memories. We are now ready to present our first model of computation.

§1.1 Random Access Machines

All contemporary computers allow the "random" accessing of their memory. An address is sent to the memory which returns the data stored at the given address. The name "random access machine" stems from the fact that the earlier models were not random access; they were based on sequential tapes or they were functional in nature. We will consider these models later. As a starting point of our investigation, we will consider a model that strongly resembles assembly language programming on a conventional computer. The model that we introduce will perform very simple operations on registers. Every real-world computer has a fixed amount of memory. This memory can be arbitrarily extended, at great loss of efficiency, by adding a tape or disk drive. Then the ultimate capacity of the machine is limited only by one's ability to manufacture or purchase tapes or disks. Not wanting to consider matters of efficiency just yet, our *random access machine* (RAM) will have a potentially infinite set of registers, R1, R2, $\cdots$, each capable of holding any natural number. Notice that we have just eliminated main storage and peripheral storage (and their management) as artifacts of how we (necessarily) perform computations. We will be concerned with data, and computation on that data. The issues of input and output will not be addressed in any detail.

RAM programs will be abstractions of assembly language programs in the sense that they use a very limited set of instruction types and the only control structure allowed is the much maligned branch instruction. There is nothing special about our choice of instructions. Among the possible choices for instruction sets, the one chosen here is based on some nontechnical notion of simplicity. RAM programs are finite sequences of very basic instructions. Hence, each RAM program will reference only finitely many of the registers. Even though the memory capacity of a RAM is unlimited, any computation described by a RAM program will access only finitely much data, unless, of course, the computation described never terminates. In case of a nonterminating computation, the amount of data accessed at any given time is finite. Each instruction may have a label, where label names are chosen from the list: N0, N1, $\cdots$. Each instruction is of one of the following seven types:

1. INC Ri Increment (by 1) the contents of register Ri.
2. DEC Ri Decrement (by 1) the contents of register Ri. If Ri contains 0 before this instruction is executed, the contents of Ri remain unchanged.
3. CLR Ri Place 0 in register Ri.
4. Ri $\leftarrow$Rj Replace the contents of register Ri with the contents of register Rj. The contents of Rj remain unchanged.

5. JMP Nix If $x = a$ then the next instruction to execute is the closest preceding instruction with label Ni. If $x = b$ then the next instruction to execute is the closest following instruction with label Ni. The a stands for "above" and the b for "below." This unusual convention allows for the pasting together of programs without paying attention to instruction labels.
6. Rj JMP Nix Perform a JMP instruction as above if register Rj contains a 0.
7. CONTINUE Do nothing.

Definition 1.1: A RAM *program* is a finite sequence of instructions such that each JMP instruction (conditional or otherwise) has a valid destination, e.g., the label referred to in the instruction exists, and the final statement is a CONTINUE.

Definition 1.2: A RAM program *halts* if and when it reaches the final CONTINUE statement.

Definition 1.3: A RAM program P *computes* a partial function ϕ, of n arguments, iff when P is started with $x_1, \ldots, x_n$ in registers R1, ..., Rn respectively and all other registers used by P contain 0, P halts only if $\phi(x_1, \ldots, x_n)$ is defined and R1 contains the value $\phi(x_1, \ldots, x_n)$. A partial function is RAM *computable* if some RAM program computes it.

There is a subtle difference between asserting the existence of a RAM program computing some function and actually being able to produce the program. Consider, for example, the problem of trying to decide how many times the word "recursion" appears as a substring in some arbitrary but fixed infinite random string of symbols from the alphabet $\{a \ldots z\}$. No matter how much of the string we examine, we will never know if we have seen all of the occurrences of the word "recursion." However, there exists a RAM program that will tell us exactly how many occurrences there are. Owing to the possibility of there being infinitely many instances of the word "recursion" embedded in the infinite random string, we will use the convention that a RAM program can signal that there are exactly n repetitions of the word "recursion" in the mystery string by outputting $n + 1$. The output 0 will be reserved to indicate the situation where there are infinitely many occurrences of the substring we are trying to count. Now, for any natural number n, there is a RAM program that computes the constant n function. One of these programs will tell us exactly how many occurrences of the word "recursion" are in the string. Which program we cannot say, so it will be impossible to deliver the RAM program that solves our problem.

Most assembly languages have more powerful arithmetic instructions. Recall that our purpose here is not ease of writing RAM programs, but rather ease of proving things about RAM programs. As a first example, the following program computes the sum of two arguments.

```
N1  R2 JMP N2b
    INC R1
    DEC R2
    JMP N1a
N2  CONTINUE
```

Exercise 1.4: Show that exponentiation is RAM computable.

Exercise 1.5: Show that integer division is RAM computable.

It should be a straightforward, tedious exercise to show that all your favorite assembly language instructions have implementations in the RAM programming language described above. In fact, not all seven of the instruction types are necessary.

Proposition 1.6: For every RAM program P there is another RAM program P' computing the same function such that P' only uses statement types 1, 2, 6 and 7.

Proof: Suppose P is a RAM program. We show how to transform P into the desired P' in steps, eliminating one type of offending instruction at each step. First, we eliminate the unconditional jumps of statement type 5. Choose n least such that P makes no reference to register Rn. Form RAM program P'' from P by replacing each "Nk JMP Nix" instruction with the following code segment:

```
Nk  CLR Rn
    Rn JMP Nix
```

Next, we eliminate the register transfers of statement type 4. Choose m and n least such that P'' makes no reference to register Rm or register Rn. Let Nc and Nd be two labels not used in P''. Form RAM program P''' from P'' by replacing each "Nk Ri ←Rj" with the following code segment:

```
Nk  CLR Ri
    CLR Rn
    CLR Rm
Nc  Rj JMP Ndb
    DEC Rj
    INC Ri
    INC Rn
    Rm JMP Nca
Nd  Rn JMP Ncb
    DEC Rn
    INC Rj
    Rm JMP Nda
Nc  CONTINUE
```

Finally, we eliminate the register clear instructions. Let Nc be a label not used by P'''. Choose n large enough such that no register Rm is referenced by P''' for any $m \geq n$. This will guarantee that register Rn will initially contain a zero. Finally, form P' from P''' by replacing each "Nk CLR Ri" instruction with the following code segment:

```
Nk  Ri JMP Ncb
    DEC Ri
    Rn JMP Nka
Nc  CONTINUE
```

This completes the proof of the Proposition. The end of proofs will normally be indicated by the symbol ⊗.

§1.2 Partial Recursive Functions

The next model of computation that we will examine resembles programming in LISP. Actually, it is the other way around — LISP resembles the following computation paradigm. We start by defining the *base* functions.

The class of base functions contains the zero function Z where $Z(x) = 0$ for each x and the successor function S where $S(x) = x + 1$ for each x. The class of base functions also contains the *projection* functions. For each positive n and each positive $j \leq n$ there is a projection function U_j^n such that $U_j^n(x_1, \ldots, x_n) = x_j$. Essentially, the projection function selects one of its arguments. The situation is analogous to the UNIX convention of using \$1, \$2, $\cdots$ to represent individual arguments in the programs that we call shell scripts.

The base functions can be combined to obtain other functions. Large classes of functions can be obtained in this fashion. We will look at three operations for defining new functions. Since these operators map functions to functions, they can (and will) be viewed as closure operators. The first of these operators is an iteration operator that is analogous to the well-known "for loops" of FORTRAN and other subsequent programming languages.

Definition 1.7: A function f of $n + 1$ arguments is defined by *primitive recursion* from g, a function of n arguments, and h, a function of $n + 2$ arguments, iff for each $x_1, \ldots, x_n$:

$$f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n)$$
$$f(x_1, \ldots, x_n, y + 1) = h(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y)).$$

For the $n = 0$ case of the above definition we adopt the convention that a function of 0 arguments is a *constant*.

The recursion of the above definition is "primitive" because the value $f(x_1, \ldots, x_n, y)$ can be determined from the value of $f(x_1, \ldots, x_n, z)$ for some $z < y$. Forms of recursion that are not primitive will be discussed extensively later in the book.

Definition 1.8: A function f of n arguments is defined by *composition* from g a function of m arguments and functions h_1, h_2, ..., h_m, each of n arguments iff for each $x_1, \ldots, x_n$:

$$f(x_1, \ldots, x_n) = g(h_1(x_1, \ldots, x_n), \ldots, h_m(x_1, \ldots, x_n)).$$

The $m = 1$ case of the above definition yields the traditional composition scheme.

We can now define a very large class of functions. This class is large enough to include all the functions that we can actually compute today. In fact, all the functions we are likely to ever be able to compute are included in the following class.

Definition 1.9: The class of *primitive recursive functions* is the smallest class of functions containing all the base functions that is closed under the operations of primitive recursion and composition.

Some examples are now in order. First we will show that the addition function is primitive recursive. We do so in detail. By composition, a function h of three arguments is defined that increments its third argument.

$$h_1(x_1, x_2, x_3) = S(U_3^3(x_1, x_2, x_3))$$

Now the desired function A is defined by primitive recursion:

$$\begin{aligned} A(x, 0) &= U_1^1(x) \\ A(x, y+1) &= h_1(x, y, A(x, y)). \end{aligned}$$

A similar exercise in the formalization yields a multiplication function. By composition,

$$h_2(x_1, x_2, x_3) = A(U_1^3(x_1, x_2, x_3), U_3^3(x_1, x_2, x_3)).$$

By primitive recursion,

$$\begin{aligned} M(x, 0) &= Z(x) \\ M(x, y+1) &= h_2(x, y, M(x, y)). \end{aligned}$$

Division can be obtained by iterating subtraction, which can be obtained by iterating a function that simply subtracts 1 from its argument. This simple function, called the predecessor function, is defined below. The subtraction is "proper" in that no value less than zero can ever be produced. By primitive recursion:

$$\begin{aligned} PR(0) &= 0 \\ PR(x+1) &= U_1^2(x, PR(x)). \end{aligned}$$

In what follows we will use an informal style of showing that functions are primitive recursive. Rather than use a projection function, we will just give the desired argument omitting the others. The more conventional infix operator "+" will be used so that $A(x, y)$ becomes $x + y$. Similarly, $\times$ will be used for multiplication and $\dot{-}$ will be used for proper subtraction. To illustrate this informal style, we define some functions that will be useful for testing conditions.

$$sg(0) = 0$$
$$sg(x+1) = 1$$

$$\overline{sg}(0) = 1$$
$$\overline{sg}(x+1) = 0$$

$$|x - y| = (x \dot{-} y) + (y \dot{-} x)$$

$$\epsilon(x, y) = \overline{sg}(|x - y|)$$

$$\overline{\epsilon}(x, y) = sg(|x - y|)$$

The sg and $\overline{sg}$ functions can be used to turn any function into a function with range $\{0, 1\}$. By interpreting 0 as "false" and 1 as "true" a 0,1 valued function represents a predicate. Logical predicates that can be represented by 0,1 valued primitive recursive functions are called *primitive recursive predicates.*

Exercise 1.10: Supply the formal versions of the above definitions.

Exercise 1.11: Show that $rm(x, y)$ = 'the remainder left after dividing x by y' is primitive recursive.

Exercise 1.12: Show that integer division is primitive recursive. Make, and state, a convention about division by 0.

Exercise 1.13: Show that $f(x)$ = 'the largest integer less than or equal to $\sqrt{x}$' is primitive recursive.

Exercise 1.14: Suppose that f is primitive recursive. Show that

$$\Sigma_{y \leq z} f(x_1, \ldots, x_n, y) = f(x_1, \ldots, x_n, 0) + \ldots + f(x_1, \ldots, x_n, z)$$

is primitive recursive.

Exercise 1.15: Suppose that f is primitive recursive. Show that

$$\Pi_{y \leq z} f(x_1, \ldots, x_n, y) = f(x_1, \ldots, x_n, 0) \times \ldots \times f(x_1, \ldots, x_n, z)$$

is primitive recursive.

Exercise 1.16: Show that if f is primitive recursive then so is g defined by

$$g(x_1, \ldots, x_n, y) = \begin{cases} \max i \leq y[f(x_1, \ldots, x_n, i) = 0] & \text{if such an } i \text{ exists} \\ y + 1 & \text{otherwise.} \end{cases}$$

Exercise 1.17: Show that if f is primitive recursive then so is g defined by

$$g(x_1, \ldots, x_n, y) = \begin{cases} \min i \leq y[f(x_1, \ldots, x_n, i) = 0] & \text{if such an } i \\ & \text{exists} \\ y + 1 & \text{otherwise.} \end{cases}$$

Exercise 1.18: Suppose that g_1, g_2 and g_3 are primitive recursive functions of n arguments, and that h_1 and h_2 are primitive recursive predicates of n arguments. Show that f, a function of n arguments, defined as follows is primitive recursive:

$$f(x_1, \cdots, x_n) = \begin{cases} g_1(x_1, \cdots, x_n) & \text{if } h_1(x_1, \cdots, x_n), \\ g_2(x_1, \cdots, x_n) & \text{if } h_2(x_1, \cdots, x_n) \text{ and} \\ & \quad \text{not } h_1(x_1, \cdots, x_n), \\ g_3(x_1, \cdots, x_n) & \text{otherwise.} \end{cases}$$

Exercise 1.19: State and prove an n-ary version of the previous exercise. You start with n primitive recursive functions and $n-1$ primitive recursive predicates. The resulting scheme is called *course of values recursion.*

Exercise 1.20: Show that the primitive recursive predicates are closed under the operations of and, or, not, and implication.

The primitive recursive functions can be defined over *strings* directly instead of over the natural numbers. For example, consider the following definition of the primitive recursive functions over strings of 0's and 1's. The empty string is denoted by ϵ and plays the role of zero. x denotes an arbitrary string of 0's and 1's. Juxtaposition denotes concatenation of strings. The base functions include $E(x) = \epsilon$ instead of Z. The E stands for "erase." There are two successor functions: $S_0(x) = x0$ and $S_1(x) = x1$. The projection functions are defined in the same manner as for primitive recursive functions over natural numbers. The composition function is also defined in the same manner. Definition by primitive recursion is given by the following schema:

$$\begin{aligned} f(\epsilon, x_2, \ldots, x_n) &= g(x_2, \ldots, x_n) \\ f(y0, x_2, \ldots, x_n) &= h_0(y, f(y, x_2, \ldots, x_n), x_2, \ldots, x_n) \\ f(y1, x_2, \ldots, x_n) &= h_1(y, f(y, x_2, \ldots, x_n), x_2, \ldots, x_n) \end{aligned}$$

In the above case we say that f is defined from g, h_0 and h_1.

Exercise 1.21: Show, using the full formalism, that the functions *dell* and *con* below are primitive recursive:

$$dell(x) = \begin{cases} \epsilon & \text{if } x = \epsilon \\ x \text{ with the last character removed} & \text{if } x \neq \epsilon \end{cases}$$

$$con(x, y) = xy$$

Exercise 1.22: Define the primitive recursive functions over variables that are pairs of natural numbers.

Exercise 1.23: Prove that each primitive recursive function is defined on all its arguments. Hint: Use a structural induction over the number of applications of primitive recursion and composition used to define a function.

As a consequence of the previous exercise, each primitive recursive function is defined on all of its arguments. Such functions are called *total* because they are defined on the total possible domain ($\mathbb{N}$). We are seeking an alternative characterization of the RAM computable functions. Notice that not all RAM programs halt on all inputs. Below is a simple example of a RAM program that never halts.

```
N1  INC R1
    JMP N1a
    CONTINUE
```

Hence, there are RAM programs that compute functions that are not primitive recursive. Many people would argue that all the functions that are actually computed in practice are primitive recursive. The simple example above illustrates that there are computable functions that are not primitive recursive. This motivates the final operator for producing functions from functions. This operator is a "search" operator that is analogous to the well-known "while loops" of Pascal and other languages. There is a subtle point in the following definition.

Definition 1.24: A function f of n arguments is defined by *minimalization* from a function h of $n+1$ arguments iff $f(x_1, \ldots, x_n)$ is the least y such that $h(x_1, \ldots, x_n, y) = 0$ *and* (the subtle point) $h(x_1, \ldots, x_n, z)$ is defined for each $z \leq y$. Such a y may not exist, in which case $f(x_1, \ldots, x_n)$ is *undefined*. In any case, we will denote "f defined from h by minimalization" by:

$$f(x_1, \ldots, x_n) = (\mu y)[h(x_1, \ldots, x_n, y) = 0].$$

The only effective way of computing a function f defined by minimalization from h is to first compute $h(x_1, \ldots, x_n, 0)$, then $h(x_1, \ldots, x_n, 1), \cdots$, until the desired value is found. In other words, any calculation of such an f must search for the answer, testing each potential value in turn. Consequently, there are two ways for such a function f to be divergent. Firstly, there may be not be a value of y such that $h(x_1, \ldots, x_n, y) = 0$. Secondly, it may the case that $h(x_1, \ldots, x_n, y) = 0$, for some y, but there is also a value $z < y$ such that $h(x_1, \ldots, x_n, z)$ is undefined.

Definition 1.25: The class of *partial recursive functions* is the smallest class of functions containing the base functions that is closed under the operations of primitive recursion, composition and minimalization.

Definition 1.26: The class of *recursive functions* consists of all the partial recursive functions that are total functions.

Exercise 1.27: Prove or disprove: There exists a recursive function with range $\{0\}$ that is not primitive recursive.

Exercise 1.28: Prove or disprove: There exists a recursive function with range $\{0, 1\}$ that is not primitive recursive.

Exercise 1.29: Does the closure of the primitive recursive functions under the operation of minimalization yield precisely the partial recursive functions?

The partial recursive functions appear to be quite different from the RAM computable functions. The RAM programs suggest an imperative implementation while a functional programming approach is more suitable for the partial recursive functions. We will prove that the RAM computable functions are precisely the partial recursive functions. Toward that goal, we present our first theorem.

Theorem 1.30: Every partial recursive function is RAM computable.

Proof: This proof is a structural induction argument. For the base case, we show that the RAM computable functions contain the base functions. The inductive step(s) start with some partial recursive functions that are assumed to be RAM computable and show that if they are combined via the operations of primitive recursion, composition and/or minimalization, then the resultant function is RAM computable. The induction is over the implicit structure of the partial recursive functions. Each partial recursive function, ψ, is defined in terms of other partial recursive functions. Then ψ will follow the functions used to define it in the ordering of the partial recursive functions. Hence, an ordering of the partial recursive functions is induced by their definitions. The base functions form the beginning of the ordering. The implicit induction hypothesis assumes that all functions prior to ψ in this unspecified ordering are RAM computable. The induction step then uses the definition of ψ to show that ψ is RAM computable. There will be a case for each possible way that ψ could have been defined from partial recursive functions that appear before ψ in the ordering.

The proof makes no explicit reference to the time or space utilization of the RAM programs that are constructed. However, a careful reading of the proof will reveal that the number of steps needed by the RAM programs constructed in the proof is proportional to the number of function invokations required to unravel the partial recursive definition. The constant of proportionality is also small. The space utilized to store the variables will be the same in both models. Further discussions of resource utilization will be postponed to later chapters.

A program for Z is:

```
NO  CLR R1
    CONTINUE
```

A program for S is:

```
NO  INC R1
    CONTINUE
```

A program for U_j^n is:

```
NO  R1←Rj
    CONTINUE
```

Hence, all the primitive recursive base functions are RAM computable. This completes the base case of the structural induction. Next, we proceed to show that if we combine only functions that we know are RAM computable using the rules of constructing partial recursive functions, then the result will be a RAM computable function. Hence, there will be one inductive clause for each technique of constructing partial recursive functions. We start with the rule of primitive recursion, and then proceed to consider composition and minimalization.

Suppose that f, a function of $n+1$ arguments, is defined by primitive recursion from g and h. Suppose P_g is a RAM program computing g and P_h is a RAM program computing h. Suppose that neither P_g nor P_h references any register Rj for $j \geq k$. Let $N1$ be a label that is not used in P_h. Below is a program computing f.

The program initially saves the arguments in registers that are not used by either P_g or P_h and initializes an iteration counter. The first interesting step of the computation is to compute g on the given arguments. This is done by inserting a copy of P_g in the new program. Then the program below starts executing the main loop. The test for the termination condition is done first. Each pass through the loop starts with a restoration of arguments so that when P_h is run, its inputs will be found in the first few registers and all the other registers that it uses will contain 0. The iteration counters are updated, as part of the argument formation for the next iteration.

```
     Rk + 1 ←R1              save arguments
        ⋮
     Rk + n ←Rn
     CLR Rk + n + 1          initialization of
                               iteration number
     Rk ←Rn + 1              count the number of
                               iterations (y + 1)
     P_g                     compute g(x_1, ..., x_n)
     Rk + n + 2 ←R1          save function value
N1   Rk JMP N2b              test for termination
     R1←Rk + 1               restore arguments
        ⋮
     Rn ←Rk + n
     Rn + 1 ←Rk + n + 1
     Rn + 2 ←Rk + n + 2
     CLR Rn + 3              clear scratch registers
        ⋮
     CLR Rk − 1
     P_h                     compute h(x_1, ..., x_n,
                               Rk + n + 1,Rk + n + 2)
     Rk + n + 2 ←R1          save function value
     INC Rk + n + 1          prepare for next iteration
     DEC Rk                  update counter
     JMP N1a
N2   CONTINUE
```

Hence, the result of combining RAM computable functions by primitive recursion will always result in a RAM computable function. Now, continuing with the inductive step, we proceed to consider combining RAM programs by composition.

Suppose f, a function of n arguments, is defined by composition from g and $h_1, \ldots, h_m$. Let $P_g, P_1, \ldots, P_m$ be RAM programs computing g, $h_1, \ldots, h_m$ respectively. Suppose that no register Rj is referenced by any of those programs for any $j > k$. The following RAM program computes f.

As before, the program starts by saving the arguments. The $P1$ is computed, and the result is saved in a register that will not be used by any of the other programs $P_2, \ldots, P_m$. One by one, the arguments are restored and relevant registers are cleared before each of the programs $P_2, \ldots, P_m$ is run. After each program is executed, the result is stored in a register that will not be referenced by the other programs that are part of the composition. When all the subcomputations are completed, the results are gathered to be in position for the running of program P_g.

```
NO   Rk + 1 ←R1          save arguments
       ⋮
     Rk + n ←Rn
     P1                  compute h1(x1, ..., xn)
     Rk + n + 1 ←R1      save result
     R1←Rk + 1           restore arguments
       ⋮
     Rn ←Rk + n
     CLR Rn + 1          clear scratch registers
       ⋮
     CLR Rk
     P2                  compute h2(x1, ..., xn)
     Rk + n + 2 ←R1      save result
       ⋮
     R1←Rk + 1           restore arguments
       ⋮
     Rn ←Rk + n
     CLR Rn + 1          clear scratch registers
       ⋮
     CLR Rk
     Pm                  compute hm(x1, ..., xn)
     Rk + n + m ←R1      save result
     R1←Rk + n + 1       set up arguments to
                           compute g
       ⋮
     Rm ←Rk + n + m
     CLR Rm + 1          clear scratch registers
       ⋮
     CLR Rk
     Pg                  compute g
     CONTINUE
```

Hence, if RAM computable functions are combined by composition, the result is a RAM computable function. The proof is completed by considering the use of RAM computable functions in minimalizations. Suppose that f, a function of n arguments, is defined by minimalization from h. Let P_h be a RAM program computing h. Again, let k be such that no register Rj, for $j > k$, is referenced in P_h. Let $N1$ be a label that is not used in P_h. The following program computes f. As before, the initial arguments are saved in registers that won't be used by program P_h. The main loop is entered. The first part of the loop is to restore the arguments and clear all the other

registers that program P_h will access. The code for program P_h is executed and the result is saved. Finally, the loop ends with a test for termination.

```
      Rk + 1 ←R1              save arguments
          ⋮
      Rk + n ←Rn
      CLR Rk + n + 1          initialize search variable
N1    R1←Rk + 1               restore arguments
          ⋮
      Rn ←Rk + n
      Rn + 1 ←Rk + n + 1      set search variable
      CLR Rn + 2              clear scratch registers
          ⋮
      CLR Rk
      P_h                     compute h(x_1, ..., x_n, Rk + n + 1)
      R1 JMP N2b              check if done
      INC Rk + n + 1          increment search variable
      JMP N1a                 try again
N2    R1←Rk + n + 1
      CONTINUE
```

⊗

To show that the RAM computable functions are precisely the partial recursive functions, we must show that every RAM program computes a partial recursive function. To do so, we will encode RAM programs as natural numbers and devise a partial recursive function ϕ such that $\phi(x, y_1, \ldots, y_n)$ is the value obtained by running the x^{th} RAM program on arguments $y_1, \ldots, y_n$. As a bonus, the development of the coding scheme will yield some valuable insights about computation.

§1.3 Pairing and Coding

A *pairing function* is a one-to-one, invertible mapping from pairs of natural numbers onto $\mathbb{N}$. We will develop primitive recursive functions $\langle \cdot, \cdot \rangle$, π_1 and π_2 such that for any x and y, $\pi_1(\langle x, y \rangle) = x$ and $\pi_2(\langle x, y \rangle) = y$. Below is a picture of how we will map pairs of natural numbers onto the natural numbers.

	0	1	2	3	4	⋯
0	0	2	5	9	↗	
1	1	4	8	↗		
2	3	7	↗			
3	6	↗				
⋮	↗					

The above picture suggests the following enumeration of pairs of natural numbers:

$$\underbrace{\underbrace{(0,0)}_{0^{\text{th}}},\underbrace{(1,0),(0,1)}_{1^{\text{st}}},\underbrace{(2,0),(1,1),(0,2)}_{2^{\text{nd}}},\underbrace{(3,0),\ldots}_{3^{\text{rd}}}}_{\text{counterdiagonals}}$$

The counterdiagonals refer to tracing the counterdiagonals of the above diagram replacing each number with the pair that it represents. Notice that on the n^{th} counterdiagonal there are only pairs (x, y) such that $x + y = n$. Furthermore, all the pairs (x, y) such that $x + y = n$ are found on the n^{th} counterdiagonal. There are $n+1$ pairs on the n^{th} counterdiagonal. Suppose $x + y = n$. The pair (x, y) is the $y + 1^{\text{st}}$ pair on the n^{th} counterdiagonal. Hence,

$$\langle x, y\rangle = 1 + 2 + \ldots + (x + y) + y.$$

Clearly, this is a primitive recursive function.

To define the inverse functions π_1 and π_2 we employ the auxiliary function cd such that $cd(n)$ is the number of the counterdiagonal on which the n^{th} pair lies. The first entry on the counterdiagonal containing the n^{th} pair is $\langle cd(n), 0\rangle$. The key to this auxiliary function is the observation that n and $n+1$ lie on the same counterdiagonal iff $n+1 < \langle cd(n)+1, 0\rangle$. Hence,

$$\begin{aligned} cd(0) &= 0 \\ cd(n+1) &= cd(n) + ((n+2) \dot{-} \langle cd(n) + 1, 0\rangle)\,. \end{aligned}$$

Since $\pi_2(n)$ is the position of the n^{th} pair on the $cd(n)^{\text{th}}$ counterdiagonal we have:

$$\pi_2(n) = n \dot{-} \langle cd(n), 0\rangle$$

Recall that $\pi_1(n) + \pi_2(n) = cd(n)$. Hence:

$$\pi_1(n) = cd(n) \dot{-} \pi_2(n).$$

Clearly, $\langle \cdot, \cdot \rangle$, π_1 and π_2 are primitive recursive.

Exercise 1.31: Give full formal primitive recursive definitions of $\langle \cdot, \cdot \rangle$, π_1 and π_2.

Exercise 1.32: Give a closed form expression for $\langle \cdot, \cdot \rangle$.

Exercise 1.33: Show that $\langle x, y \rangle \geq x$ and $\langle x, y \rangle \geq y$, for all x and y.

Once we have a pairing function, it is an easy matter to devise one-to-one and invertible codings from n-tuples of natural numbers onto $\mathbb{N}$. For example $\langle x, y, z \rangle$ denotes $\langle x, \langle y, z \rangle \rangle$. Since 0 decodes to the pair (0,0), $\langle x_1, \ldots, x_n, 0 \rangle = \langle x_1, \ldots, x_n, 0, 0 \rangle = \langle x_1, \ldots, x_n, 0, 0, 0 \rangle$. Notice that if an n-tuple is encoded into some number x and more than n values are extracted from x then the decoding functions are still defined. This makes it possible to define a function Π of three arguments such that if $n \geq 2$, $1 \leq i \leq n$, and $\langle z_1, \ldots, z_n \rangle = x$ then $\Pi(i, n, x) = z_i$.

Exercise 1.34: Show that the function Π defined above is primitive recursive.

Exercise 1.35: Define a primitive recursive function F as follows:

$$F(x, y) = \Pi(y + 1, \pi_1(x) + 1, \pi_2(x)).$$

Now define a sequence of partial recursive functions as:

$$d_x(y) = \begin{cases} F(x, y) - 1 & \text{if } 0 < F(x, y) \text{ and } y < \pi_1(x) + 1, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Show that the sequence of functions d_0, d_1, $\cdots$ contains all the partial recursive functions with finite domain and that if $d_x = d_y$ then $x = y$, for any values x and y.

A very important observation can now be made. Since any number of arguments can be unambiguously coded into a single natural number, having multiple arguments to a program (or function) is a matter of convenience and not essential to the computation. Just as multiple argument types turned out to be an artifact of computation, so are the number of arguments. The important ramification for our study of computation is that we may assume without loss of generality that all the functions we are interested in computing are functions of a single argument. This will greatly simplify our proof that every RAM computable function is partial recursive and proofs of other, deeper, results yet to come.

We now describe how to code a RAM program into a single integer. To begin, the coding of individual instructions is described. There are only five types of instruction to consider. The instruction schemes and their codes are given in the table below:

Ni	INC Rj	coded as $\langle 1, i, j, 0 \rangle$
Ni	DEC Rj	coded as $\langle 2, i, j, 0 \rangle$
Ni	CONTINUE	coded as $\langle 3, i, 1, 0 \rangle$
Ni	Rj JMP Nka	coded as $\langle 4, i, j, k \rangle$
Ni	Rj JMP Nkb	coded as $\langle 5, i, j, k \rangle$

If the RAM program we are trying to encode does not have labels for each instruction, then supply each unlabeled instruction with the least label that is not used by the original program. It is now easy to define primitive recursive functions giving type, line label and the register referenced for any instruction. If the instruction is a JMP instruction, then there is a function giving the destination of the jump. Suppose x is the coding of some instruction.

$$Typ(x) = \Pi(1,4,x)$$
$$Nam(x) = \Pi(2,4,x)$$
$$Reg(x) = \Pi(3,4,x)$$
$$Jmp(x) = \Pi(4,4,x)$$

It is possible to tell if a given x encodes an instruction. There is a primitive recursive predicate $Inst(x)$ that is true iff x encodes an instruction. Informally, $Inst(x)$ is true iff $1 \leq Typ(x) \leq 5$ and $1 \leq Reg(x)$ and $Typ(x) \leq 3$ implies $Jmp(x) = 0$ and $Typ(x) = 3$ implies $Reg(x) = 1$.

Exercise 1.36: Give a primitive recursive definition of $Inst$.

Let P be a RAM program containing only instructions of the above five types. Say $P = I_1, \ldots, I_n$. Let $\hat{P}$ denote the code for program P, similarly for instructions. Then,

$$\hat{P} = \langle n, \hat{I}_1, \ldots, \hat{I}_n \rangle.$$

Useful primitive recursive functions giving the length, program part and individual instructions of some $\hat{P} = x$ are defined as:

$$Ln(x) = \Pi(1,2,x)$$
$$Pg(x) = \Pi(2,2,x)$$
$$Line(i,x) = \Pi(i, Ln(x), Pg(x))$$

As was the case with instructions, there is a primitive recursive predicate that is true of a given x iff there is some RAM program P such that $\hat{P} = x$. $Prog(x)$ is true iff

1. Every line of $Pg(x)$ is a valid instruction, and
2. The last instruction of $Pg(x)$ is a CONTINUE, and
3. Every JMP instruction has a destination label on some instruction of $Pg(x)$.

Exercise 1.37: Formally show that $Prog(x)$ is a primitive recursive predicate.

To simulate the execution of a RAM program with a partial recursive function we need to encode the contents of the registers. Suppose RAM program P never makes reference to any register Ri for any $i > n$. Let r_i denote the value held by register Ri. Initially, r_i will be 0, for $i > n$. No execution of P will modify any register Ri, for $i > n$. Hence,

$$\langle r_1, \ldots, r_n, 0, 0, \ldots, 0 \rangle = \langle r_1, \ldots, r_n, 0 \rangle.$$

To unambiguously code all the registers into a single natural number, we need only produce a bound on n, based only on the value of $\hat{P}$. This bound is supplied by the properties of our pairing function and the observation:

$$Reg(Line(i,x)) \leq Line(i,x) \leq Pg(x) \leq x.$$

As a consequence of our coding of RAM programs we can make an observation that will turn out to be fundamental. There is a list, P_0, P_1, ..., of all the RAM programs and nothing but RAM programs. If $Prog(i)$ is true, then P_i is the RAM program P such that $\hat{P} = i$. If $Prog(i)$ is false, then P_i is some innocuous program that never halts on any input.

§1.4 Simulating an Execution of a RAM Program

Recall that we started encoding RAM programs so as to be able to show that every RAM computable function is partial recursive. In what follows, x will denote $\hat{P}$ for some RAM program P and y will denote the contents of P's registers at some point in the simulation.

Proposition 1.38: Suppose P is a RAM program, $x = \hat{P}$, and i is such that $1 \leq i \leq Ln(x)$. Then the following functions are primitive recursive:

1. $Nextline(i,x,y)$ is the number of the next instruction executed after the i^{th} instruction of P is executed with the contents of the registers coded by y;
2. $Nextcont(i,x,y)$ is the code of the register contents after executing the i^{th} instruction of P with register contents coded by y;
3. $Comp(x,y,m) = \langle i,z \rangle$ where after running program P for m steps starting with register contents coded by y, i is the number of the next instruction to be executed and z codes the contents of the registers.

Proof:

(1) In the event that the i^{th} instruction is not a JMP instruction we know that $Nextline(i,x,y) = i+1$. Suppose $Line(i,x)$ encodes a JMP instruction. Then it must be that $4 \leq Typ(Line(i,x)) \leq 5$ and the contents of the register that is tested is $\Pi(Reg(Line(i,x)),x,y)$. If this register contains a nonzero value, then again $Nextline(i,x,y) = i+1$. If the jump is "above" then $Nextline(i,x,y)$ is $\max j \leq i$ such that $[Nam(Line(j,x)) = Jmp(Line(i,x))]$. If the jump is "below" then $Nextline(i,x,y)$ is $\min i < j \leq Ln(x)$ such that $[Nam(Line(j,x)) = Jmp(Line(i,x))]$. Notice that the search for j is bounded above. The upper bound serves to contain the functions $Nextline$ within the primitive recursive functions. See exercise 1.16.

Exercise 1.39: Formalize the definition of $Nextline$.

Notice that as a nice consequence of the definition of $Nextline$ is that if the i^{th} instruction executed is the final `CONTINUE` then $Nextline(i,x,y) > Ln(x)$.

(2) Two auxiliary primitive recursive functions are needed: $Add(j,x,y)$, the (coded) result of incrementing `R`j in the group of registers coded by y, and $Sub(j,x,y)$, the analogous function for decrements. The parameter x serves as bound on the number of registers coded by y. Consequently, $Sub(j,x,y)$ is the least $z \leq y$ such that $\Pi(j,x,z) = \Pi(j,x,y) \dot{-} 1$ and for each $k \leq x$, if $k \neq j$ then $\Pi(k,x,z) = \Pi(k,x,y)$. Finding a bound on size of the code for the registers after an increment is a little more difficult. If y codes a group of (at most x) registers, then by the monotonicity of the pairing function, y is larger than the number stored in any of the registers. So $y+1$ will be a value that is at least as large as any value that is stored in the registers after one of them is incremented. Consequently, a bound on the size of the coded register contents after an increment is given by:

$$\underbrace{\langle y+1, \cdots, y+1 \rangle}_{x \text{ times}}.$$

Using the above bound suggests a search strategy to be used by Add that is similar to the one used by Sub.

Exercise 1.40: Give formal definitions of Add and Sub.

Now,

$$Nextcont(i,x,y) = \begin{cases} Add(Reg(Line(i,x)),x,y) & \text{if } Typ(Line(i,x)) = 1 \\ Sub(Reg(Line(i,x)),x,y) & \text{if } Typ(Line(i,x)) = 2 \\ y & \text{if } Typ(Line(i,x)) \geq 3. \end{cases}$$

(3) The final definition needed is simply:

$$\begin{aligned} Comp(x,y,0) &= \langle 1, y \rangle \\ Comp(x,y,m+1) &= \\ &\langle Nextline(\pi_1(Comp(x,y,m)),x,\pi_2(Comp(x,y,m))), \\ &\ Nextcont(\pi_1(Comp(x,y,m),x,\pi_2(Comp(x,y,m))))\rangle. \end{aligned}$$

⊗

Only two more pieces are needed to complete the simulation. These are the initial register contents and an indication of how many steps to simulate. Since we need only consider functions of a single argument (say y), the first piece is easy: $\langle y, 0 \rangle$. However, RAM program P may *never* halt on input y. Consequently, our calculation of how many steps to simulate will be given by a partial, not primitive, recursive function:

$$End(x,y) = \mu m[\pi_1(Comp(x,y,m)) = Ln(x)].$$

The result of running the x^{th} RAM program on input y is given by:

$$\psi(x,y) = \pi_1(\pi_2(Comp(x, \langle y, 0\rangle, End(x, \langle y, 0\rangle)))).$$

Some peculiar results may come out of trying to evaluate $\psi(x,y)$ if x does not code a RAM program. To rectify this we define a partial recursive function of a single argument thus:

$$\phi_{univ}(\langle x,y\rangle) = \begin{cases} \psi(x,y) & \text{if } Prog(x), \\ \uparrow \text{ (undefined)} & \text{otherwise.} \end{cases}$$

Theorem 1.41: Every RAM computable function is partial recursive.

Proof: Suppose f is a RAM computable function. Then there is a RAM program P computing f. Let $\hat{P} = x$. Then $f(y) = \phi_{univ}(x,y)$, for all y. Hence, f is partial recursive. ⊗

Recall that every partial recursive function is RAM computable. Then there is a RAM program computing ϕ_{univ}. This program is *universal* because on input $\langle x,y\rangle$, it produces the result of running RAM program P_x on input y. If one were to build a hardware device implementing the universal RAM program, one would have a machine capable of executing any RAM program. This machine would input an encoding of the RAM program that it was to execute, e.g., the universal RAM program is a blueprint for a *stored program computer*. Another pleasant ramification of our definition of ϕ_{univ} is the following:

Theorem 1.42: (Normal Form) There exist *primitive* recursive functions g and h such that for every partial recursive function f, there is an i such that:

$$f(x) = g(x, i, \mu y[h(x,i,y)]).$$

Exercise 1.43: Provide a proof for the above theorem.

Exercise 1.44: Find a RAM program that takes its input $\langle i,j\rangle$ and outputs the code for a program P ($\hat{P}$) that behaves as follows: P takes its input x and simulates program P_i on input x. If that computation converges, then P outputs the result of the computation $P_j(P_i(x))$. Is the function computed by your RAM program primitive recursive?

§1.5 Turing Machines

The first person to notice the existence of "universal programs" was Alan Turing, a man many consider the founder of computer science. He worked with a model that is very different from any contemporary computing paradigm. The simplicity of his model of computation has made it a popular base for the study of computational complexity where time is counted as the number of steps executed. Turing based his model on

an analysis of how people perform calculations. Using long division calculations as an example, Turing reasoned that any calculation proceeds in steps where in each step a single symbol is modified based on some "history" (or retained information) of the prior steps of the computation. The next step of the computation always involves a symbol adjacent to one just subjected to modification.

The various symbols in a human hand calculation appear on a (two dimensional) piece of paper. The multiple dimensionality of the storage medium was recognized by Turing to be an artifact and not crucial to the computation. With a little more effort, we can, in principle, arrange the storage to be a linear list of symbols. The advantage of displaying symbols in multiple dimensions is clarity with respect to human calculation. When performing a long division by hand, the divisor is typically displayed to the left of the dividend. The answer is accumulated, one digit at a time, above the dividend. The space below the dividend is used for the subtractions that implement the division. Over the centuries, this has proven to be an arrangement that facilitates the implementation of long division by humans. Having a computer take advantage of such a simple spatial orientation of data is still considered a difficult problem. Consequently, the memory of a Turing machine is a list of symbols that is maintained on a *tape* that is just a one-dimensional, potentially infinite, sequence of *cells*, each of which contains a single symbol. The symbols are from some *alphabet*, which is just a finite set of recognized characters. A marker is maintained indicating the symbol currently under examination. The history, or information "remembered" by the computation, is modeled by *states*. Each state is used to encode some finite amount of remembered information. We proceed with a formal definition of a *Turing machine* and then illustrate how it "computes."

Definition 1.45: A *Turing machine* is composed of an alphabet Σ that includes some symbol designated as a "blank," a finite set of states S, a distinguished initial state $s_0 \in S$, and a program consisting of a finite set of quintuples of the form $s_i\sigma_m\sigma_n D s_j$ where s_i and s_j are members of S, σ_m and σ_n are members of Σ, and $D \in \{L, R\}$.

Let M be a Turing machine. M starts in state s_0 with its input on a tape just large enough to accommodate it. The symbol designated by the pointer (the *scanned* symbol) is the leftmost symbol of the input. If at any point, the M attempts to move off the left or right end of the tape, another cell containing a blank is affixed to the appropriate end of the tape. Suppose M is in state s_i scanning symbol σ_m. M searches its program for a quintuple starting with "$s_i\sigma_m$." Suppose M finds $s_i\sigma_m\sigma_n D s_j$. M then replaces σ_m with σ_n and enters state s_j. If $D = L$, then M completes the instruction step by moving the pointer to the scanned symbol to the adjacent cell to the left. If $D = R$, M moves to the right instead. M halts when it enters some state s_i scanning σ_m for which there is no quintuple starting with "$s_i\sigma_m$." The result of the computation (output) is the leftmost consecutive string of nonblank characters.

Definition 1.46: Suppose M is a Turing machine with states S, initial state s_0, symbols Σ, and quintuples Q. A *configuration* of M is a string of symbols from $\Sigma \cup S$ containing exactly one symbol from S.

The idea of a configuration is to represent the status of a computation of a Turing machine. If the input to a Turing machine is the string "$x_1x_2\cdots x_n$" then the initial configuration would be "$s_0x_1\cdots x_n$" indicating that the machine is in state s_0 and scanning the symbol x_1. In general, the configuration "$x_1\cdots x_is_kx_j\cdots x_n$" represents the situation where the string on the tape is "$x_1\cdots x_n$" and the Turing machine is in state s_k scanning the symbol x_j. Configurations play a fundamental role in discussing Turing machine computations. Sometimes, configurations are also called *instanteous descriptions.*

If for each state s_i and symbol σ_m there is *at most* one quintuple starting with $s_i\sigma_m$, then M is called *deterministic.* Otherwise, M is called *nondeterministic.* A nondeterministic Turing machine is free to choose any matching quintuple to apply.

Previously, we concluded that it suffices to study the computation of functions with domain and range $\mathbb{N}$. Now we seem to be talking about computation over strings. For the purposes of discussion, fix some alphabet for all computations. An alphabet with 0,1, a blank and another delimiter will suffice, although the full ASCII alphabet may be more convenient to program. Suppose the alphabet that we just chose has k symbols. Then each finite string of symbols from the alphabet can be viewed as a base k integer. The symbols of the alphabet are used in place of the characters 0, 1, 2, $\ldots k-1$.

Exercise 1.47: Develop the details of coding of all strings over some alphabet to and from the natural numbers.

Definition 1.48: A function $\psi : \mathbb{N} \to \mathbb{N}$ is *Turing computable* iff there is a Turing machine that when started with a character string representing x on its tape halts iff $\psi(x)$ is defined. If the Turing machine halts, a string encoding the value of $\psi(x)$ is the only nonblank entry on the tape.

By way of example, we will show that the successor function is Turing computable. Suppose that all inputs are in binary notation. The alphabet we use contains 0, 1, and B (the blank). s_0 will be the initial state. The other states are the ones referenced by the following quintuples.

s_000Rs_0	find the end of the input
s_011Rs_0	
s_0BBLs_1	found the end
s_110Ls_1	increment, bit by bit
s_101Ls_2	
s_1B1Rs_2	found the left end of the tape

Exercise 1.49: Modify the above Turing machine so that, in addition to incrementing, it also halts scanning the leftmost symbol of the output.

The programming of Turing machines is rather tedious. The exercise becomes somewhat easier by using a *state transition* diagram to represent a Turing machine. Suppose M is a Turing machine. The state transition diagram of M is a collection of circles, called *nodes*, and arrows connecting the circles, called *arcs*. There is one node for every state of M. There is an arc from s_i to s_j labeled 'σ_m, σ_n, D' iff M has a quintuple $s_i\sigma_m\sigma_n D s_j$. The node corresponding to the initial state is designated by an unlabeled arc, without a source node, to the initial node.

Exercise 1.50: Show that addition is Turing computable. Devise your own alphabet and coding of integers.

Exercise 1.51: Devise a coding scheme to encode any Turing machine program into a string of symbols.

Exercise 1.52: Show that every Turing computable function is partial recursive.

Exercise 1.53: Show that every partial recursive function is Turing computable.

Exercise 1.54: Define a Turing machine with a two-dimensional "tape." Hint: define a *configuration* in such a way as to make the description of how the machine operates easy.

There are several possible embellishments that one can make to the basic Turing machine we have defined. One such popular enhancement is to allow the Turing machine to view more than one symbol at a time via multiple tape pointers. These pointers are typically called *heads*.

Exercise 1.55: Define formally what a k-head Turing machine is. Define what a configuration is, and how such a machine computes a function. Explain informally (but talk about tapes, heads, etc.) how to, given a k-head Turing machine, construct a 1-head Turing machine that computes the same partial function. If the original machine had n states, then (roughly) how many states does *your* machine have, and roughly how much time does it take to simulate (compared to the time on the old machine).

One of Turing's motivations was to define precisely the algorithmically (effectively) computable functions. He proposed what we now call the Turing computable functions. The definition of the partial recursive functions grew out of a similar goal. There were other proposals for formulating the effectively computable functions. These include functions computed by Post machines, functions computable by Markov algorithms (similar to Snobol programs), functions definable in the lambda calculus, and the Herbrand–Gödel computable functions.

All these various formalisms were shown to capture exactly the same class of functions. Furthermore, given any two formalisms, it was shown how to effectively transform programs in one formalism to equivalent programs in the other formalism. Recall that we have shown that the partial recursive

functions are precisely the RAM computable functions. Hidden in our proof was an algorithm to transform partial recursive function definitions into RAM programs, and vice versa. These led to the formulation of what is known as *Church's Thesis.* (Sometimes this is called the Church–Turing Thesis.)

> *The partial recursive functions are precisely the effectively computable functions.*

The thesis can never be proven. That would require a formal definition of "effectively computable" and that is precisely what the thesis says exists. It would be possible to refute Church's thesis. What is required is to exhibit a function that is effectively computable but is not partial recursive (or RAM computable, or Turing computable). No one has been able to do this, despite extensive study of the class of partial recursive functions.

Church's thesis will be used frequently in this course. Drawing on our programming experience, we will be able to say that a function is clearly computable, and hence, by Church's thesis, it is partial recursive.

A final note on the various models of computation mentioned above. The translation from one model to another is generally very efficient. Usually the translated program is of a size bounded by a small polynomial of the size of the initial program. Furthermore, the complexity of the translated program is within a (usually small) polynomial factor of the complexity of starting program. A discussion of the complexity of programs and functions will come later, after we codify a most succinct model of computation and prove some fundamental properties of that model.

§1.6 Some Other Models

We briefly consider some of the other models proposed to capture the class of effectively computable functions. The first, known as the Herbrand–Gödel model of computability, is a formalism that closely resembles the mathematical style of defining a function in terms of itself. The importance of this is that it is source of the word "recursion" in recursive function theory, a topic we will cover in the next chapter.

Suppose that ϕ denotes an unknown function, $\psi_1, \ldots, \psi_k$ are known functions and we are given $k+1$ equations relating the unknown function to the ones that are known. Consider various possible ways of substituting the ψ's and ϕ for variables in any of the equations under consideration. By equating certain pairs of the resulting expressions, a set of functional equations is created.

Thus we might have

$$\begin{aligned}
\phi(x,0) &= \psi_1(x),\\
\phi(0,y+1) &= \psi_2(x),\\
\phi(1,y+1) &= \psi_3(x),\\
\phi(x+2,y+1) &= \psi_4(\phi(x,y+2),\phi(x,\phi(x,y+2))).
\end{aligned}$$

If it turns out that the set of equations, such as the one above, has a unique solution for ϕ, then we say that ϕ is a recursive function. To make sure that the set of functional equations has a unique solution, we will impose some restrictions. Firstly, the left-hand side of each of the given functional equations defining ϕ shall be of the form

$$\phi(\psi_{i1}(x_1, \ldots, x_n), \psi_{i2}(x_1, \ldots, x_n), \ldots, \psi_{il}(x_1, \ldots, x_n)).$$

The second restriction (as stated below) is equivalent to the condition that all possible sets of arguments $(n_1, \ldots, n_l)$ of ϕ can be so arranged that the computation of the value of ϕ for any given set of arguments $(n_1, \ldots, n_l)$ by means of the given equations requires knowledge of the values of ϕ only for sets of arguments that precede $(n_1, \ldots, n_l)$.

From the given set of functional equations, we define inductively a set of derived equations:

(1) Any expression obtained by replacing all the variables of one of the given equations by a member of $\mathbb{N}$ is a derived equation.
(2) If $k_1, \ldots, k_n \in \mathbb{N}$ and $\psi_{ij}(k_1, \ldots, k_n) = m$ then $\psi_{ij}(k_1, \ldots, k_n) = m$ is a derived equation.
(3) If $\psi_{ij}(k_1, \ldots, k_n) = m$ is a derived equation, the equality obtained by substituting m for an occurrence of $\psi_{ij}(k_1, \ldots, k_n)$ in a derived equation is a derived equation.
(4) If $\phi(k_1, \ldots, k_l) = m$ is a derived equation where $k_1, \ldots, k_l, m \in \mathbb{N}$, the expression obtained by substituting m for an occurrence of $\phi(k_1, \ldots, k_l)$ on the right-hand side of a derived equation is a derived equation.

Our second restriction can now be stated. For each $k_1, \ldots, k_l \in \mathbb{N}$ there is a unique m such that $\phi(k_1, \ldots, k_l) = m$ is a derived equation.

If one takes $Z(x) = 0$, $S(x) = x + 1$ and projection functions $P_i^n(x_1, \ldots, x_n) = x_i$ as the only initially given functions, then the above constitutes what is known as Herbrand–Gödel model of computability, defining the H–G computable functions.

For example, below is a system of equations defining a function M that produces the result of multiplying its arguments:

$$g(x, 0) = x$$
$$g(x, S(y)) = S(g(x, y))$$

$$M(x, 0) = 0$$
$$M(x, S(y)) = g(M(x, y), x).$$

Exercise 1.56: Prove that the H–G computable functions include all the partial recursive functions.

As a final example of other models of computation, we present one that vaguely resembles a contemporary programming language. While modern programming languages have fancy constructs, the language below is exceptionally simple. Consider a programming language, that we will call QUAIL, where all the variables stand for natural numbers. The variable will be chosen from the list $v_1, v_2, \cdots$. The BNF for the language is very simple:

$$\begin{aligned}\langle \texttt{stmt} \rangle := \ & \texttt{clear } \langle \texttt{var} \rangle \\ | \ & \texttt{increment } \langle \texttt{var} \rangle \\ | \ & \texttt{decrement } \langle \texttt{var} \rangle \\ | \ & \texttt{while } \langle \texttt{var} \rangle \neq 0 \texttt{ do } \langle \texttt{stmt} \rangle \\ | \ & \langle \texttt{stmt} \rangle ; \langle \texttt{stmt} \rangle .\end{aligned}$$

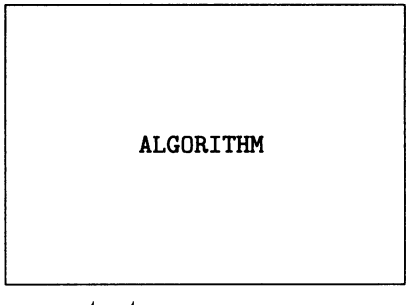
For an input/output convention, assume that the inputs are preloaded in $v_1, \cdots, v_n$ and that the output is to be left in v_1 when the computation finishes. The statements of the program are executed in sequential order, except as indicated by "while" constructs.

Exercise 1.57: Show that the language QUAIL can express exactly the Turing computable functions.

§1.7 A Representation for Programs

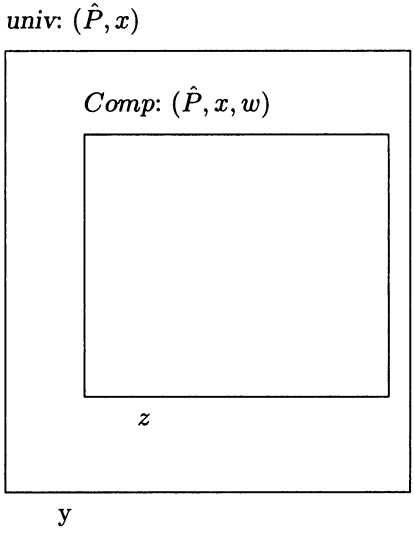
From the above comments, it seems that it does not matter which formalization of computable functions we choose for our study of the theory of computing. We will choose a generic one. In the next section, we will essentially give axioms for our generic model of computing. Here we give a pictorial notation for our generic programs. This representation gives the name of the program, its argument list, its output and the indication of the algorithm used.

NAME: (*argument list*)



output

For example, the universal RAM program developed above would be represented as:



Another handy notation for expressing functions is the lambda notation. $\lambda x, y[x + y]$ denotes a function of two arguments, x and y, which evaluates to the sum of the arguments, e.g., the addition function. In general, the string between the λ and the "[" denotes the argument list and the expression [...] denotes the algorithm for computing the function.

§1.8 Historical Notes

The RAM model was introduced in [S&S]. The particular formulation used here is from [M&Y]. The formulation of the partial recursive functions used here dates back to [Kle]. The coding techniques used to represent programs as natural numbers have their basis in the work of [Göd], although he used prime factorization instead of pairing functions. The exposition of pairing (and depairing) functions is from [Eng]. Turing machines were introduced in [Tur]. For the origins of Herbrand–Gödel computability, see the article "On undecidable propositions of formal mathematical systems" by Gödel in [Dav]. The λ notation for specifying functions is due to Church and Kleene [Chu]. The use of pictures to represent objects dates back to the caves of France.

[Chu] A. Church, An Unsolvable Problem of Elementary Number Theory, *American Journal of Mathematics*, Vol. 58, 1936, pp. 345–363.

[Dav] M. Davis, *The Undecidable*, Raven Press, New York, 1965.

[Eng] E. Engeler, *Introduction to the Theory of Computation*, Academic Press, Inc., New York, 1973.

[Göd] K. Gödel, Über formal unentscheidbare Sätze der Principia Mathematica und verwanter Systeme, I, *Monatsheft für Mathematik und Physik*, Vol. 38. pp. 173–198, 1931.

[Kle] S. Kleene, General Recursive Functions of Natural Numbers, *Mathematische Annalen*, Vol. 112, 1936, pp. 727–742.

[M&Y] M. Machtey and P. Young, An Introduction to the General Theory of Algorithms, Elsevier North-Holland, New York, 1978.

[S&S] J. Shepherdson and H. Sturgis, Computability of Recursive Functions, *Journal of the ACM*, Vol. 10, 1963, pp. 217–255.

[Tur] A. Turing, On Computable Numbers, with an Application to the Entscheidungsproblem, *Proceeding of the London Mathematical Society*, Vol. 42, 1936, pp. 230–265.

2

Basic Recursive Function Theory

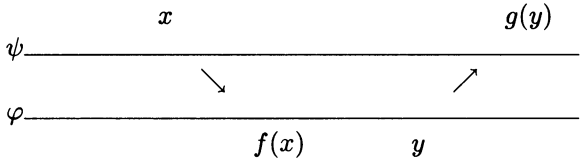
In the last section, we examined some reasonable models of computation. These models were found to be equivalent in the sense that they captured the same class of "computable" functions. Now we proceed to define our "generic" programming system syntactically. From our experience of coding each RAM program as a single natural number, we can use the natural numbers to serve as names for our programs. This will also make it easier to describe functions that take programs as arguments and produce programs as output. We will see that any way of effectively presenting the computable functions (via a list of all RAM programs, for example) will induce some structure in the way the programs appear. This structure will be evidenced by certain functional relationships among the various programs. Our generic programming system will be characterized by the specific relationships that hold for the system. The specific relationships are akin to control structures in that they specify how to modify and combine some programs to produce other programs. We will then discuss solvability (and partial solvability) of several fundamental problems. As a notational simplification, the use of pairing functions will be implicit, so we will write $f(x, y)$ instead of the more formal $f(\langle x, y\rangle)$. Since we will have several occasions to give proofs by contradiction, we will note the arrival at a contradiction by the symbol $(\Rightarrow\Leftarrow)$.

§2.1 Acceptable Programming Systems

Recall that P_0, P_1, $\cdots$ is a list of all and only the RAM programs. For each i, let ψ_i denote the function from $\mathbb{N}$ to $\mathbb{N}$ computed by P_i. Then, $\psi_0, \psi_1, \cdots$ is a list of all and only the RAM computable functions. To simplify our notation, we will say that *program* i computes the function ψ_i. Essentially what we have just done is to give every RAM program P_i the nickname i. Suppose $\varphi_0, \varphi_1, \cdots$ is a list of all and only the partial recursive (or Turing computable or H–G computable) functions. By Theorems 1.30 and 1.41 there are (primitive) recursive functions f and g such that:

$$(\forall x)\psi_x = \varphi_{f(x)}$$
$$(\forall x)\varphi_x = \psi_{g(x)}.$$

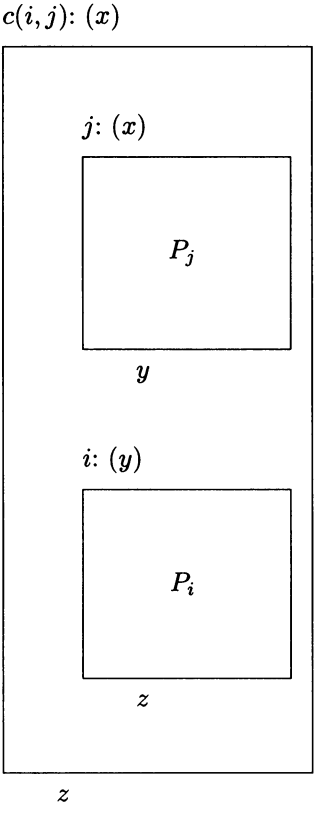
The functions f and g map from one system to the other, as indicated by the following picture:



For the proof of Theorem 1.41 we developed a universal function. Notice that:

$$\psi_{univ}(g(i), x) = \psi_{g(i)}(x) = \varphi_i(x).$$

So, the list $\varphi_0, \varphi_1, \cdots$ has a universal function (use f and some careful compositions). Recall that there is a RAM program that interprets its two inputs as program codes and outputs a program that executes the composition of the two input programs. Suppose this composition program is c (computing the function ψ_c). Then, $\psi_{c(i,j)}(x) = \lambda x[\psi_i(\psi_j(x))]$. Pictorially,



Hence, by Church's Thesis, c is a partial recursive function. Moreover, since c is defined on all possible arguments, it is a recursive function. Consequently,

$$\psi_{c(g(i),g(j))} = \psi_{g(i)} \circ \psi_{g(j)} = \varphi_i \circ \varphi_j.$$

So, $f(c(g(i), g(j)))$ is a composition function for $\varphi_0, \varphi_1, \cdots$. Notice that we have used the traditional notation for function composition $f \circ g$ to denote the function $\lambda x[f(g(x))]$. Church's Thesis and the above motivate the following:

Definition 2.1: A *programming system* is a list $\varphi_0, \varphi_1, \cdots$ of all and only the partial recursive functions. A programming system $\varphi_0, \varphi_1, \cdots$ is *universal* if it has a universal program: $\varphi_{univ}(i, x) = \varphi_i(x)$. A universal programming system is *acceptable* if it has a recursive composition function: $\varphi_{c(i,j)} = \varphi_i \circ \varphi_j$.

Some easy examples of acceptable programming systems come to mind. By the discussion above, both $\psi_0, \psi_1, \cdots$ and $\varphi_0, \varphi_1, \cdots$ are two such examples. Any sufficiently expressive programming language, like Pascal or C++, can be used as the basis for an acceptable programming system. All that is needed is to form a list of all the programs that can be written in the language. Below we give another example.

Theorem 2.2: The UNIX operating system is an acceptable programming system.

Proof: The Bourne shell (or csh or tcsh) is powerful enough to simulate any RAM program computing over strings instead of integers. Hence, UNIX is a programming system. The **sh** command is an implementation of a universal program for shell scripts. The pipe operator can be used to implement a compositor of shell scripts. Consequently, UNIX is an acceptable programming system. ⊗

The UNIX operating system has many useful features and programs built into it. We will show that, surprisingly, the two properties of Definition 2.1 suffice to *guarantee* that an acceptable programming system has many other features which are familiar from programming experience. As we have seen, acceptable programming systems are easy to create. What is difficult is to come up with a universal programming system that is not acceptable. To do so requires techniques that we have not yet presented.

Our first result concerning acceptable programming systems is that one can always store specific data in programs. In other words, there are special versions of programs that have some of their parameters fixed and these programs can be found uniformly and effectively from the original program. An often used computational technique is embodied in this result. The "stored" arguments can be viewed as default settings. Additionally, the aliasing feature of modern UNIX systems allows one to, for example, create a print command that has the name of the printer built in so that it is not necessary to specify your favorite printer every time you want a hard copy.

Theorem 2.3: (The $s-m-n$ Theorem) Suppose $\varphi_0, \varphi_1, \cdots$ is an acceptable programming system. Then there is a recursive function s such that $\forall m \geq 1$, $\forall n \geq 1$, $\forall x_1, \ldots, x_m$, $y_1, \ldots, y_n$, $\forall i$,

$$\varphi_{s(i,m,x_1,\ldots,x_m)}(y_1, \ldots, y_n) = \varphi_i(x_1, \ldots, x_m, y_1, \ldots, y_n).$$

Proof: The intuition of the proof is quite simple. We must take a program, find the first input statments, and replace them with assignment statements so that the appropriate constants are stored in the proper variables. This is quite a trivial process in most contemporary programming languages. Since we are proving the theorem for an arbitrary acceptable programming system, our job is slightly more complicated. We do the $m = 1$, $n = 1$ case only. For the general case replace x with $\langle x_1, \ldots, x_m \rangle$ and y with $\langle y_1, \ldots, y_n \rangle$ everywhere. Since $\varphi_0, \varphi_1, \cdots$ is an acceptable programming system, there is a recursive composition function c. Define $P(y) = \langle 0, y \rangle$, and suppose that program p computes P, e.g. $\varphi_p = P$. Let $Q(\langle x, y \rangle) = \langle x + 1, y \rangle$ with $\varphi_q = Q$. Finally define R by:

$$\begin{aligned} R(0) &= p \\ R(x+1) &= c(q, R(x)). \end{aligned}$$

A simple induction shows that $\varphi_{R(x)}(y) = \langle x, y \rangle$. For the base case, $\varphi_{R(0)}(y) = \varphi_p(y) = \langle 0, y \rangle$. Suppose inductively that $\varphi_{R(x)}(y) = \langle x, y \rangle$. Then

$$\begin{aligned} \varphi_{R(x+1)}(y) &= \varphi_{c(q,R(x))}(y) \\ &= \varphi_q \circ \varphi_{R(x)}(y) \\ &= \varphi_q(\langle x, y \rangle) \\ &= \langle x + 1, y \rangle. \end{aligned}$$

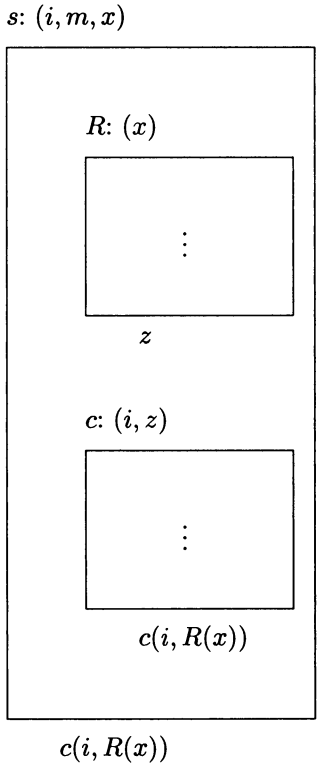
Let $s(i, m, x) = c(i, R(x))$. A pictorial rendition of s appears below.

$$\begin{aligned} \varphi_{s(i,m,x)}(y) &= \varphi_{c(i,R(x))}(y) \\ &= \varphi_i \circ \varphi_{R(x)}(y) \\ &= \varphi_i(x, y) \end{aligned}$$

$\otimes$

Notice that, in the above proof, if c is primitive recursive, then so is s. Furthermore, the universal program was not used. In what follows we will write $s(i, x_1, \ldots, x_m)$ for $s(i, m, x_1, \ldots, x_m)$. The function s will be called a *store* function.

Next we will show that any acceptable programming system can be effectively translated into any other acceptable programming system. Consequently, the choice of acceptable programming system does not effect what can and cannot be computed.



Theorem 2.4: Let $\varphi_0, \varphi_1, \cdots$ be a universal programming system. Let $\psi_0, \psi_1, \cdots$ be a programming system with an associated recursive function s such that:

$$(\forall i)(\forall x)(\forall y)\psi_{s(i,x)}(y) = \psi_i(x, y).$$

Then there is a recursive function t such that $(\forall i)\varphi_i = \psi_{t(i)}$.

Proof: Since $\psi_0, \psi_1, \cdots$ is a programming system there is a k such that $\psi_k = \varphi_{univ}$. Define $t(i) = s(k, i)$. Then, by unraveling the application of the store function:

$$\psi_{t(i)}(x) = \psi_{s(k,i)}(x) = \psi_k(i, x) = \varphi_{univ}(i, x) = \varphi_i(x).$$

$\otimes$

Exercise 2.5: Construct a universal programming system, $\psi_1, \psi_2, \psi_3, \ldots$, such that the following set is NOT recursively enumerable:

$$A = \{x | x \in \text{Domain}(\psi_x)\}.$$

Exercise 2.6: Construct a universal programming system, $\psi_1, \psi_2, \psi_3, \ldots$, such that the set

$$A = \{x | \text{ Domain}(\psi_e) \text{ is recursive } \}$$

is recursively enumerable but not recursive.

§2.2 Recursion Theorems

The original meaning of the word recursion was "a going backward." The word is no longer used in the general English language but was adopted by mathematicians in the 1800s to denote a style of function definition where a function is defined in terms of itself. The famous Fibonacci numbers are a (technically primitive) example. The study of functions that are defined in terms of themselves became known as "Recursive Function Theory." Another view of recursive definitions is that they are self-referential. In this section we will prove various theorems that embody more and more powerful forms of computational self-reference. Computational self-reference is a most powerful form of the programming technique known as recursion.

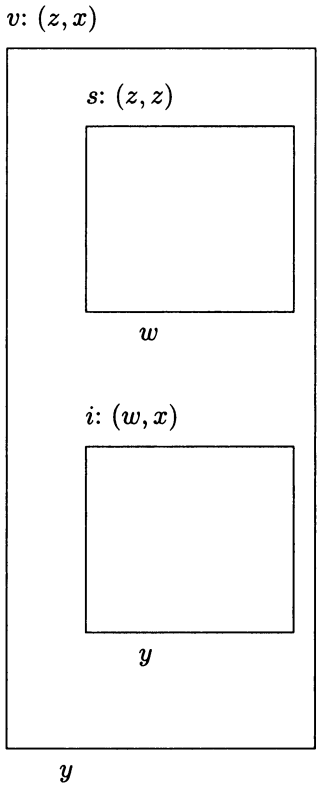
Of course, it is easy to get in trouble defining functions in terms of themselves. Circular definitions can easily result if one is not careful. The everywhere undefined function is a potential result of such mistakes. Some people find self-referential functions mysterious. However, computationally, the technique is well-founded. Consider writing a program on a contemporary computing system. When the process of entering code is begun, a text editor of some sort is invoked. Typically, the text editor will ask for the name of the file being created. Sometimes this is done before entering any text into the file. Some editors will only demand that the name of the file be selected before writing the file in long-term memory. In any event, the choice of the name of the file is up to the author. Manipulating the name of the program is the mechanism of self-reference. For the sake of discussion, call our file, containing the instructions for a program, *PROG*. In the process of creating the file named *PROG*, we can enter commands that access the file structure of computing system. Since we only need to know the name of the file to access it, instructions that access the file named *PROG* can be added to the program being written. The program can be instructed to access any file, interpret it as a program and simulate its execution on some arguments. Of course, the particular file we have in mind here is the one being created. Any algorithmic transformation on the contents of the file *PROG* may be specified. Hence, it is possible for a program to simulate modified versions of itself in order to decide what to output. In the many applications of self-reference that follow, we will see a wide variety of programs that simulate altered versions of themselves. Formally, the manipulation of program names to produce self-reference effects is given in the following.

Theorem 2.7: Suppose $\varphi_0, \varphi_1, \cdots$ is an acceptable programming system. Then for any program i, there is another program e such that $(\forall x)\ \varphi_e(x) = \varphi_i(e, x)$.

Proof: Given i, by implicit use of the *s-m-n* theorem construct a program v such that for all x and z:

$$\varphi_v(z, x) = \varphi_i(s(z, z), x).$$

A picture of v would look like:



The *s-m-n* theorem is often invoked in this way. Notice that the algorithm for program v intuitively says: "Run program i on arguments $s(z, z)$ and x." So the i is in fact a parameter to the algorithm that does not appear in the list of arguments given to v. That means that there was a parameter i at some point which was incorporated into the program v at some other point. To illustrate, we derive program v explicitly. By Church's Thesis there is a program j such that for all k, z and x:

$$\varphi_j(k, z, x) = \varphi_{univ}(k, \langle s(z, z), x \rangle).$$

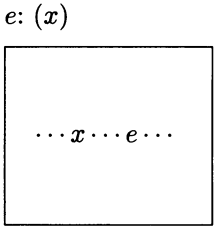
Now let $v = s(j, i)$.

Continuing with the proof, let $e = s(v, v)$, then:

$$\begin{aligned}\varphi_e(x) &= \varphi_{s(v,v)}(x)\\ &= \varphi_v(v,x)\\ &= \varphi_i(s(v,v),x)\\ &= \varphi_i(e,x).\end{aligned}$$

⊗

The recursion theorem is often invoked implicitly. That is, we will specify programs that use as an extra parameter the name of the program itself. Pictorially, the situation appears as follows:



As a first example application of the recursion theorem, we will show that, in an acceptable programming system, given a program, we can always find a syntactically different but semantically equivalent program. Generally, it is easier to think about writing a program that knows its own name and hence, can access its own code, than it is to figure out how to apply the recursion theorem. Consequently, applications of the recursion theorem are typically implicit. We illustrate this phenomenon below by first giving the explicit construction and then giving the implicit version.

Theorem 2.8: Suppose $\varphi_0, \varphi_1, \cdots$ is an acceptable programming system and k is a program. Then there is a j such that $k \neq j$ and $\varphi_k = \varphi_j$. Furthermore, j can be found effectively from k.

Proof: Let k be given. By Church's Thesis, there is a program i such that:

$$\varphi_i(e,x) = \begin{cases} \varphi_k(x) & \text{if } e \neq k;\\ \varphi_{e+1}(x) & \text{otherwise.}\end{cases}$$

Now, apply the recursion theorem to obtain a program e such that for all x:

$$\varphi_e(x) = \varphi_i(e,x).$$

If $e \neq k$ then set $j = e$ and the theorem follows. Suppose that $e = k$. Then, clearly $\varphi_e = \varphi_k$. By the construction of e, $\varphi_e = \varphi_{e+1}$. Let $j = e + 1$ and the theorem follows. ⊗

The proof above was given in full detail. As mentioned earlier, it is generally easier to think about applying recursion directly, rather than to figure out how to set up an application of the recursion theorem. Below we give the proof using the more common implicit application of the recursion theorem.

Proof: Let k be given. By an implicit use of the recursion theorem, there is a program e such that:

$$\varphi_e = \begin{cases} \varphi_k & \text{if } e \neq k; \\ \varphi_{e+1} & \text{otherwise.} \end{cases}$$

Program e essentially executes the following algorithm: First check to see if $e = k$. If not, then simulate program k; if so, then simulate program $e + 1$.

If $e \neq k$ then set $j = e$ and the theorem follows. Suppose that $e = k$. Then, clearly $\varphi_e = \varphi_k$. By the construction of e, $\varphi_e = \varphi_{e+1}$. Let $j = e + 1$ and the theorem follows. ⊗

Suppose we have some program r that had been constructed by the recursion theorem. Then program r can use the constant r as an extra, implicit, parameter in its effective calculations. We call r a self-referential program since it may refer to its own code. By the above theorem, there is a program j such that $j \neq r$ and $\varphi_j = \varphi_r$. Even though programs r and j compute the same function, j is *not* self-referential. Programs j and r refer to the constant r, not j.

Exercise 2.9: Prove that there is a program e such that φ_e computes the constant e function.

Exercise 2.10: Prove that there is a program e that halts on input e where it produces the value e^2.

Exercise 2.11: Prove that for every recursive function f there is a program e such that:

$$\varphi_e(x) = \begin{cases} e & \text{if } x = 0, \\ f(x-1) & \text{otherwise.} \end{cases}$$

Not only does the self-referential program e of Theorem 2.7 exist, it can be uniformly and effectively found from i. By "effectively" we mean that there is an algorithm for the transformation of i into e. The "uniformly" means that i is a parameter to the algorithm that produces the self-referential program. Now we re-prove the recursion theorem in its full effective and uniform incarnation. A solution to the above exercise, together with the proof of the following theorem, yields an algorithm for producing a program that outputs its own code. This algorithm will work independently of the choice of programming language.

Theorem 2.12: Suppose $\varphi_0, \varphi_1, \cdots$ is an acceptable programming system. Then there is a recursive function r such that $(\forall i)\ (\forall x)\ \varphi_{r(i)}(x) = \varphi_i(r(i), x)$.

Proof: Define program h by: $\varphi_h(x,y) = \langle s(x,x), y\rangle$. Define the recursive function g by: $g(i) = c(i,h)$. Finally, define r by: $r(i) = s(g(i),g(i))$. Then:

$$\begin{aligned}\varphi_{r(i)}(x) &= \varphi_{s(g(i),g(i))}(x)\\ &= \varphi_{g(i)}(g(i),x)\\ &= \varphi_i(\varphi_h(g(i),x))\\ &= \varphi_i(s(g(i),g(i)),x)\\ &= \varphi_i(r(i),x).\end{aligned}$$

⊗

Exercise 2.13: Pick a real programming language (C++, Pascal, etc.) and describe how to produce a program in that language that outputs its own code.

Exercise 2.14: Pick a real programming language (C++, Pascal, etc.) and write a program that outputs its own code.

The recursion theorem we have been discussing enabled us to write single programs that were self-referential. Our discussion now turns to more powerful forms of recursion. These more powerful forms will enable us to construct sets of self-referential programs. Cooperating self referential programming turns out to be a very powerful technique. For example, the next theorem essentially says that one can, uniformly and effectively, construct infinite *sequences* of self referential programs, each one of which knows its position in the sequence. A picture representing this situation is below.

Theorem 2.15: (Parametric recursion theorem) Suppose $\varphi_0, \varphi_1, \cdots$ is an acceptable programming system. Then for all i there is a recursive function pr such that

$$(\forall x)(\forall y)\varphi_{pr(x)}(y) = \varphi_i(x, pr(x), y).$$

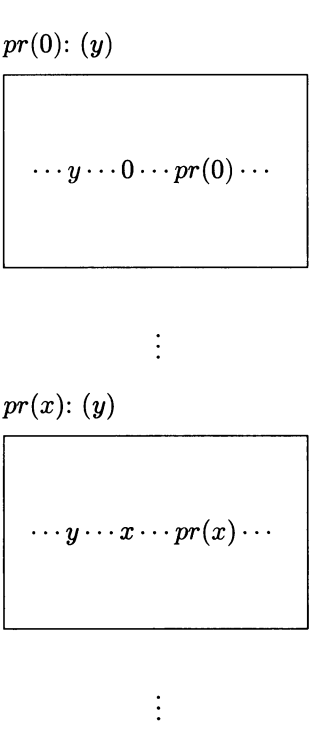
Proof: Let program i be given. By implicit use of the *s-m-n* theorem, there is a recursive function g such that for all j, x, y and z,

$$\varphi_{g(j)}(x,y,z) = \varphi_j(x, s(y,x,y), z).$$

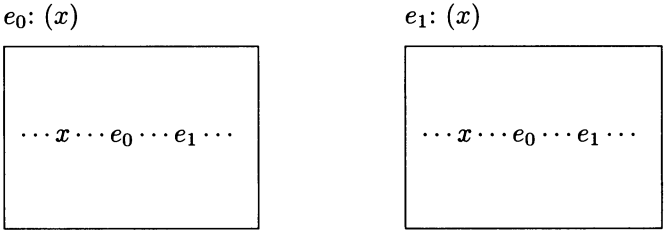
Define pr by $pr(x) = s(g(i),x,g(i))$. Then:

$$\begin{aligned}\varphi_{pr(x)}(y) &= \varphi_{s(g(i),x,g(i))}(y)\\ &= \varphi_{g(i)}(x,g(i),y)\\ &= \varphi_i(x, s(g(i),x,g(i)),y)\\ &= \varphi_i(x,pr(x),y).\end{aligned}$$

⊗



Now for a double recursion theorem. Essentially, this theorem allows us to write pairs of programs, each of which is defined in terms of itself and the other program of the pair. Again, we precede the proof by a picture.



Theorem 2.16: (Double recursion theorem) Suppose φ_0, φ_1, $\cdots$ is an acceptable programming system. For all i_0, i_1 and x there exist programs e_0 and e_1 such that:

$$\varphi_{e_0}(x) = \varphi_{i_0}(e_0, e_1, x)$$
$$\varphi_{e_1}(x) = \varphi_{i_1}(e_0, e_1, x).$$

Proof: Let i_0 and i_1 be given. By implicit use of the *s-m-n* theorem there is a recursive function g such that for all i, y_0, y_1 and x:

$$\varphi_{g(i)}(y_0, y_1, x) = \varphi_i(s(y_0, y_0, y_1), s(y_1, y_0, y_1), x).$$

Now define:

$$e_0 = s(g(i_0), g(i_0), g(i_1))$$
$$e_1 = s(g(i_1), g(i_0), g(i_1)).$$

Then

$$\begin{aligned}\varphi_{e_0}(x) &= \varphi_{s(g(i_0),g(i_0),g(i_1))}(x)\\ &= \varphi_{g(i_0)}(g(i_0), g(i_1), x)\\ &= \varphi_{i_0}(s(g(i_0), g(i_0), g(i_1)), s(g(i_1), g(i_0), g(i_1)), x)\\ &= \varphi_{i_0}(e_0, e_1, x).\end{aligned}$$

Similarly,

$$\varphi_{e_1}(x) = \varphi_{i_1}(e_0, e_1, x).$$

⊗

Exercise 2.17: State and prove a fully effective and uniform version of the double recursion theorem.

Of course, the above theorem can be generalized to the n-ary recursion theorem where program $e_0, e_1, \ldots, e_{n-1}$ give n self-referential programs, each of which knows the complete syntactic description of the other $n-1$ programs. As with all of our recursion theorems, the n-ary recursion theorem can be made effective and uniform. We will use the term *mutual recursion* to refer to an n-ary recursion theorem without being specific about the value of n.

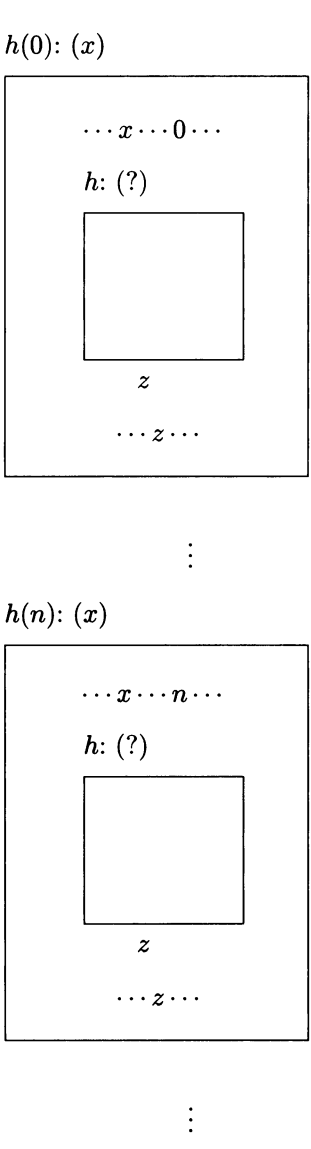
Exercise 2.18: Formally state and prove the n-ary recursion theorem.

The final recursion theorem that we will use is an infinitary one. This next theorem will allow us to construct sequences of self-referential programs where each program in the sequence knows its position in the sequence and an effective generator for the entire sequence. As is usual, there is a picture.

Definition 2.19: An *operator* is a mapping from functions to functions. Θ is an *effective operator* iff there is a recursive function f such that $(\forall i)\Theta(\varphi_i) = \varphi_{f(i)}$. The function f is called a *witness* to the effectiveness of Θ.

If Θ is an operator and $g = h$ then $\Theta(g) = \Theta(h)$. If Θ is an effective operator, with witness f, then if $\varphi_i = \varphi_j$ then $\varphi_{f(i)} = \varphi_{f(j)}$. Note that it is *not* necessary for $f(i)$ to be the syntactically same program as $f(j)$.

Theorem 2.20: (Operator recursion theorem) for any effective operator Θ there is a recursive function h such that for all n and x, $\varphi_{h(n)}(x) = \Theta(h)(n, x)$.



Before proving this theorem, a few remarks about its intended use are in order. In essence, it says that we can define what program $h(n)$ does on argument x in terms of x, n *and* h. Giving h to a program to use is different from giving it a program to compute h. What is meant here is that program $h(n)$ has use of the function h in a manner that is independent of any particular program that computes h. This is where the operator comes in. Operators are devices to describe effective computations involving functions where the computation reacts to the input/output behavior of the input function, and not to a particular program for it.

Proof: Let f be the witness for the effective operator Θ. By implicit use of the *s-m-n* theorem, there is a recursive function g such that for all m and y:

$$\varphi_{g(y)}(m) = s(y, y, m).$$

Hence,

$$\Theta(\varphi_{g(y)})(n, x) = \varphi_{f \circ g(y)}(n, x).$$

Choose d such that

$$\varphi_d(y, n, x) = \varphi_{f \circ g(y)}(n, x).$$

Consequently,

$$\varphi_{s(d,d,n)}(x) = \varphi_d(d, n, x) = \Theta(\varphi_{g(d)})(n, x).$$

Let $h(n) = s(d, d, n)$. Note that, by the selection of g above, $h = \varphi_{g(d)}$. So,

$$\begin{aligned} \varphi_{h(n)}(x) &= \varphi_{s(d,d,n)}(x) \\ &= \varphi_d(d, n, x) \\ &= \varphi_{f \circ g(d)}(n, x) \\ &= \Theta(\varphi_{g(d)})(n, x) \\ &= \Theta(h)(n, x). \end{aligned}$$

⊗

The h of the above theorem can also be made monotone increasing. This can be accomplished via a technique called *padding*. As an example of the above infinitary recursion theorem, we will prove that every acceptable programming system has a padding function. A circularity is avoided by claiming that less powerful techniques can be used to verify padding properties. The proof below is intended as a simple example of an operator recursion theorem argument.

Definition 2.21: A *padding function* is a recursive function p such that for all e, x and y, if $\varphi_{p(e,x)} = \varphi_e$ and $p(e, x) = p(e, y)$, then $x = y$.

At first glance, padding seems to be a useless side effect of the way we currently view computation. However, there is some evidence that padding techniques play a fundamental role in all models of computation. Part of this evidence is technical. In the next section we will present a characterization of acceptable programming systems in terms of padding and recursion instead of composition.

A form of padding is employed in all contemporary multiuser computer systems. One view is that when you run a program on such a system, you are in fact executing a padded version of your code. The "padding" is the operating system that protects other programs that are currently being executed from side effects caused by your programs. The time sharing aspect of the multiuser system is also handled in the padding.

Every time comments are added to a program, the size of text increases, yet the execution behavior of the program does not change. Hence, the act of placing documentation into a program also accomplishes a padding of the program. Furthermore, with suitable padding functions, the padding parameter x, representing the comments, can be extracted from the program.

Another example of padding comes from biology. A common view of DNA is that it is a "program" for the construction of an organism. Human DNA is known to contain much redundant and ostensibly superfluous information. Estimates of the amount of padding in the DNA range as high as 90%. One theory about the utility of the DNA padding is to make sure the relevant parts of the code are close to the outside when the DNA is folded. DNA also serves as a nice example of self-referential computation of the type enabled by the recursion theorem. A single strand of DNA contains enough information to produce an entire organism that will have as part of every one of its cells a copy of the original strand of DNA.

Theorem 2.22: Suppose $\varphi_0, \varphi_1, \cdots$ is an acceptable programming system. Then there is a padding function for $\varphi_0, \varphi_1, \cdots$.

Proof: By the operator recursion theorem we construct a sequence of programs below. The use of the theorem will, like all our applications of recursion theorems, be implicit. There will be no mention of the "operator" in the formal part of the proof. We will simply construct a sequence of programs where each program in the sequence has access to a subroutine computing the generator for the entire sequence. For convenience, we name the programs in the sequence: p, $h(0,0)$, $h(0,1)$, $\ldots h(x,y)$, $\cdots$. For convenience, let P denote φ_p.

$$P(e,x) = \begin{cases} h(e,x) & \text{if } h(e,x) \notin \{P(e,y) | y < x\}; \\ 1 + \max\{P(e,y) | y < x\} & \text{otherwise.} \end{cases}$$

$$\varphi_{h(e,x)} = \begin{cases} \varphi_e & \text{if } h(e,x) \notin \{P(e,y) | y < x\}; \\ \varphi_{1+\max\{P(e,y) | y<x\}} & \text{otherwise.} \end{cases}$$

To prove the theorem we must show that for all e and x, $\varphi_{P(e,x)} = \varphi_e$ and $P(e,x) \notin \{P(e,y) | y < x\}$. This is done by fixing e and performing an induction on x. For the base case, note that $P(e,0) = h(e,0)$ and $\varphi_{h(e,0)} = \varphi_e$.

Suppose inductively that, for all $x < z$, $\varphi_{P(e,x)} = \varphi_e$ and $P(e,x) \notin \{P(e,y) | y < x\}$. There are two cases to consider.
Case 1: $h(e,z) \notin \{P(e,y) | y < z\}$.
Then $P(e,z) = h(e,z)$ and $\varphi_{h(e,z)} = \varphi_e$.
Case 2: Otherwise.
Then $P(e,z) = 1 + \max\{P(e,y) | y < z\}$. Therefore, $P(e,z) \notin \{P(e,y) | y < z\}$. Since $h(e,z) \in \{P(e,y) | y < z\}$, by the induction hypothesis, $\varphi_{h(e,z)} = \varphi_e$. By construction, $\varphi_{h(e,z)} = \varphi_{P(e,z)}$. ⊗

For the construction of the above theorem, we will indicate what the implicit operator does. The operator maps the input function to a function of two arguments, n and $\langle e, x \rangle$, that behaves as follows: If $n = 0$, then execute that algorithm specified above as $P(e, x)$, using the function input to the operator to figure out what the value of the various $h(e, x)$'s are. If $n > 0$, then execute the algorithm specified by program $h(e, x)$ above, again using the function input to the operator to figure out the value of the various $P(e, y)$'s.

We now return to the issue of how to construct a padding function without using such a powerful recursion theorem. Along the way we will show that not only is it possible to map from one acceptable programming system to another effectively, it is possible to do so with a one-to-one function. The first step is to solve the following exercise.

Exercise 2.23: Suppose $\varphi_0, \varphi_1, \cdots$ is an acceptable programming system. Let g be a recursive function and s the store function. Prove that there is a program i such that for all x, $\varphi_{g(x)} = \varphi_{s(i,x)}$ and $\lambda x[s(i, x)]$ is a one-to-one function. HINT: Use the recursion theorem to write a program e that searches through the range of $\lambda x[s(e, x)]$ for the same value to appear twice.

Now Suppose $\varphi_0, \varphi_1, \cdots$ and $\psi_0, \psi_1, \cdots$ are acceptable programming systems and g is a recursive function such that for all x, $\psi_x = \varphi_{g(x)}$. Let e be the program produced by the solution to the above exercise using g and the acceptable programming system $\varphi_0, \varphi_1, \cdots$. Let $f = \lambda x[s(e, x)]$, a recursive one-to-one function. Then,

$$\varphi_{f(x)} = \varphi_{s(e,x)} = \varphi_{g(x)} = \psi_x.$$

Exercise 2.24: Use the techniques of the above observations to construct a padding function in any acceptable programming system.

A theorem that is often confused for a recursion theorem follows as the last result of this section. It is called the *fixed point* theorem since it asserts that every effective transformation of programs leaves the input/output behavior of at least one program unchanged. Uses of the fixed point theorem and the recursion theorem in an acceptable programming system are almost interchangeable. This theorem has been found to be very useful in the study of semantics. When replacing a use of the recursion theorem by an application of the fixed point theorem, an extra application of the $s - m - n$ theorem is usually invoked. As the exercises below indicate, the two theorems are interderivable. However, in the next section we will see some subtle differences between the two. For example, notice in the proof below of the fixed point theorem that the universal machine was used in its full generality. This application of the universal machine seems to be necessary. Note that none of the proofs of any of the above theorems used the universal function to construct a self-referential program. The above forms of recursion are all syntactic. The following result could be called a "semantic" recursion theorem.

Theorem 2.25: (The fixed point theorem) Suppose $\varphi_0, \varphi_1, \cdots$ is an acceptable programming system. for any recursive function f there is an n such that $\varphi_n = \varphi_{f(n)}$.

Proof: Suppose that f is a recursive function. By implicit use of the *s-m-n* theorem, define a recursive function g such that fo rall i and x:

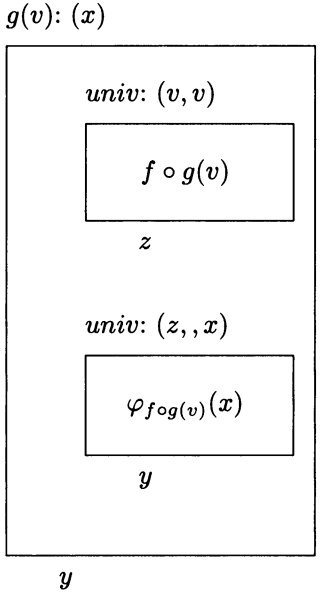
$$\varphi_{g(i)}(x) = \varphi_{univ}(\varphi_{univ}(i,i),x).$$

Since g is recursive so is $f \circ g$. Suppose that program v is such that $\varphi_v = f \circ g$. Then,

$$\begin{aligned}\varphi_{g(v)}(x) &= \varphi_{univ}(\varphi_{univ}(v,v),x)\\ &= \varphi_{univ}(\varphi_v(v),x)\\ &= \varphi_{univ}(f \circ g(v),x)\\ &= \varphi_{f(g(v))}(x).\end{aligned}$$

Choose $n = g(v)$. ⊗

A picture of the computation of $\varphi_{g(v)}(x)$ follows:



Notice that the choice of n was effective in a program for f and a program for g. The program for g can be effectively found from the *s-m-n* theorem. Consequently, there is a program fix such that if program i computes a recursive function then

$$\varphi_{fix(i)} = \varphi_{\varphi_i(fix(i))}.$$

Exercise 2.26: Show that in every acceptable programming system there are two consecutive programs computing the same function, e.g., there is an e such that $\varphi_e = \varphi_{e+1}$.

Exercise 2.27: Suppose f is a recursive function. Prove that f has arbitrarily large fixed points. In other words, prove that for all c there is an n such that $n > c$ and $\varphi_{f(n)} = \varphi_n$.

Exercise 2.28: Use the fixed point theorem to prove the recursion theorem.

Exercise 2.29: Use the recursion theorem to prove the fixed point theorem.

Exercise 2.30: A partial recursive function ψ is *universal* iff there is a recursive function f of two variables such that, for all x, $\varphi_x = \lambda y[\psi(f(x,y))]$. Prove that if ψ is universal and g is a recursive permutation (one to one and onto) then $g^{-1}\psi g$ is also universal.

§2.3 Alternative Characterizations

Each characterization of an acceptable programming system gives a minimal set of programming techniques that are necessary to manipulate programs and to build new ones from existing programs. It is very hard to construct an unacceptable programming system. The various characterizations of an acceptable programming system that we will discuss give insight into the relative power of several well-known programming techniques.

Our first characterization shows that the *s-m-n* theorem (Theorem 2.3) can be used instead of program composition. This is quite surprising since it suggests that the ability to set defaults for programs is as powerful a technique for the manipulation of programs as the ability to effectively compose programs.

Theorem 2.31: Suppose $\varphi_0, \varphi_1, \cdots$ is a universal programming system with a store function s. Then $\varphi_0, \varphi_1, \cdots$ is an acceptable programming system.

Proof: Suppose the hypothesis. We will use a single fixed composition to generate a uniform composition function. Let w be a program such that for all x, y and z:

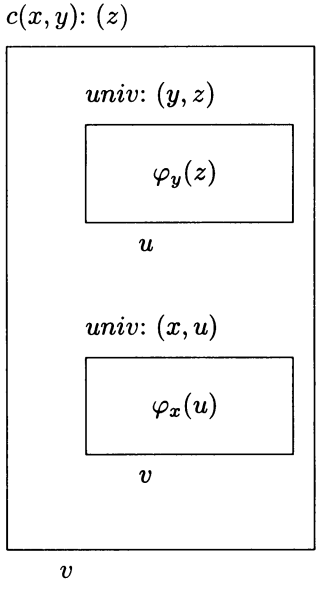
$$\varphi_w(x,y,z) = \varphi_{univ}(x, \varphi_{univ}(y,z)).$$

Define $c(x,y) = s(w,x,y)$. Then

$$\begin{aligned}\varphi_{c(x,y)}(z) &= \varphi_{s(w,x,y)}(z)\\ &= \varphi_w(x,y,z)\\ &= \varphi_{univ}(x, \varphi_{univ}(y,z))\\ &= \varphi_x(\varphi_y(z)).\end{aligned}$$

⊗

Again, we give a picture of the above construction. Notice that, in contrast to the proof of the fixed point theorem (Theorem 2.25), the first argument to the universal program (the name or description of the program to be simulated) is always a given input and never calculated. Program w of the above proof does not have a general composition function as a subroutine. This is evident from the following picture. Consequently, the above proof is not circular.



The next two theorems characterize acceptable programming systems in terms of the presence of various recursion theorems in a universal programming system. These results point out another facet of recursion: its use as a tool to construct programs based on the manipulation of the syntax of other programs. The interesting question then is to figure out just how powerful a recursion theorem is needed to guarantee acceptability. The theorems below show that either a parametric form or a mutual recursion is needed. In the first case, an infinite list of self-referential program is needed, along with a pointer into the sequence. The full power of the operator recursion theorem (Theorem 2.20), where every program not only has self-knowledge but knowledge of other programs in the sequence, is not needed. However, if we consider programs that are self-referential and also refer to other programs as well, then we only need two programs. The exercises point out that a suitable version of the ordinary recursion theorem and an effective padding program can be combined to yield acceptability in a universal programming system.

Theorem 2.32: Suppose $\varphi_0, \varphi_1, \cdots$ is a universal programming system with a parametric recursion theorem. Then $\varphi_0, \varphi_1, \cdots$ is an acceptable programming system.

Proof: Suppose the hypothesis. Define a partial recursive function ψ such that for all e, x, y and z:

$$\psi(e, x, y, z) = \varphi_{univ}(e, \langle x, z \rangle).$$

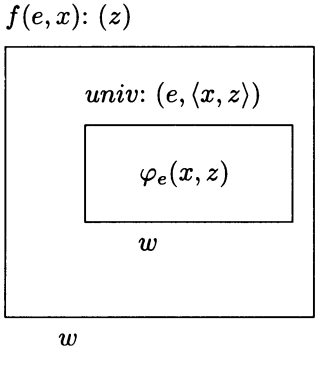
By the parametric recursion theorem there is a recursive function f such that for all e, x and z:

$$\varphi_{f(e,x)} = \lambda z[\psi(e, x, f(e, x), z)].$$

So,

$$\begin{aligned} \varphi_{f(e,x)}(z) &= \psi(e, x, f(e, x), z) \\ &= \varphi_{univ}(e, \langle x, z \rangle) \\ &= \varphi_e(x, z). \end{aligned}$$

As usual, we give a picture:



Hence, f is a store function for $\varphi_0, \varphi_1, \cdots$. Consequently, φ_0, φ_1, $\cdots$ is an acceptable programming system. ⊗

Theorem 2.33: Suppose $\varphi_0, \varphi_1, \cdots$ is a universal programming system with a double recursion theorem. Then $\varphi_0, \varphi_1, \cdots$ is an acceptable programming system.

Proof: Let r_0 and r_1 be the recursive functions witnessing the uniform double recursion in the programming system $\varphi_0, \varphi_1, \cdots$. Consequently, for any pair of programs i and j, program $r_0(i, j)$ computes the function $\lambda x[\varphi_i(r_0(i, j), r_1(i, j), x)]$. Similarly, program $r_1(i, j)$ computes the function $\lambda x[\varphi_j(r_0(i, j), r_1(i, j), x)]$. Define a function f by:

$$\begin{aligned} f(x, 0) &= 0 \\ f(x, y + 1) &= \mu y' > f(x, y)[(\forall z < y')r_1(x, z) \neq r_1(x, y')]. \end{aligned}$$

Suppose i and j are programs for two different constant functions, say, ($\varphi_i = \lambda x[a]$ and $\varphi_j = \lambda x[b]$ for $a \neq b$). Then, independently of z, program $r_1(z, i)$ computes $\lambda x[a]$. Similarly, program $r_1(z, j)$ computes $\lambda x[b]$.

Consequently, $(\forall z)[r_1(z,i) \neq r_1(z,j)]$. Hence, the search in the above definition of f *always* succeeds and f is recursive. By the definition of f, for any x, $\lambda z[r_1(x,f(x,z))]$ is one-to-one. To complete the proof is suffices to show that $\varphi_0, \varphi_1, \cdots$ has a uniform store function. Toward this end, we will apply mutual recursion to two programs, i_0 and i_1, computing partial recursive functions defined below. Before proceeding we will construct a program j:

$$\varphi_j(\langle e,y\rangle,z) = \varphi_{univ}(e,\langle y,z\rangle).$$

Now,

$$\varphi_{i_0}(u,v,x,y,z) = \begin{cases} \varphi_j(w,z) & \text{for } w \text{ such that } y = r_1(u,f(u,w)); \\ \text{undefined} & \text{otherwise.} \end{cases}$$
$$\varphi_{i_1} = \lambda u,v,x,y,z[\text{undefined}].$$

Let $p = r_0(i_0,i_1)$ and $q = r_1(i_0,i_1)$. Then

$$\begin{aligned}
&\varphi_{r_0(p,f(p,y))}(z) \\
&\quad = \varphi_p(r_0(p,f(p,y)), r_1(p,f(p,y)), z) \\
&\quad = \varphi_{r_0(i_0,i_1)}(r_0(p,f(p,y)), r_1(p,f(p,y)), z) \\
&\quad = \varphi_{i_0}(r_0(i_0,i_1), r_1(i_0,i_1), r_0(p,f(p,y)), r_1(p,f(p,y)), z) \\
&\quad = \begin{cases} \varphi_j(w,z) & \text{if } r_1(p,f(p,y)) = r_1(r_0(i_0,i_1), f(r_0(i_0,i_1),w)) \\ \text{undefined} & \text{otherwise.} \end{cases} \\
&\quad = \begin{cases} \varphi_j(w,z) & \text{if } r_1(p,f(p,y)) = r_1(p,f(p,w)) \\ \text{undefined} & \text{otherwise.} \end{cases}
\end{aligned}$$

Hence, $\varphi_{r_0(p,f(p,y))}(z) = \varphi_j(y,z)$. Define a recursive function h such that $h = \lambda y[r_0(p,f(p,y))]$ and let $s(e,y) = h(\langle e,y\rangle)$. Then, for all e, y and z:

$$\begin{aligned}
\varphi_{s(e,y)}(z) &= \varphi_{h(\langle e,y\rangle)}(z) \\
&= \varphi_{r_0(p,f(p,\langle e,y\rangle))}(z) \\
&= \varphi_j(\langle e,y\rangle,z) \\
&= \varphi_{univ}(e,\langle y,z\rangle) \\
&= \varphi_e(y,z).
\end{aligned}$$

⊗

Two more charactersizations of acceptable programming systems are given in the exercises below. A recursion theorem in an acceptable programming system is *one-to-one* if the witness function r is one-to-one.

Exercise 2.34: Prove that a universal programming system with padding and one-to-one recursion theorem is acceptable.

Exercise 2.35: Prove that a universal programming system with padding and a fixed point theorem is not necessarily acceptable.

Exercise 2.36: For any x let $s_x(i) = \lambda i[s(i,x)]$. The function s_x behaves like the store functions, excepts that it can only code in the fixed constant x. Prove that a universal programming system with store functions s_x and s_y for some $x \neq y$ is acceptable.

Exercise 2.37: Prove that there is a programming system with a composition function c but no fixed point function fix.

§2.4 The Isomorphism Theorem

In this section we will show that any two acceptable programming systems are not only mappable onto each other but are in fact isomorphic. This means that there is a one-to-one and onto mapping between any two acceptable programming systems. This means that, for many results, the choice of acceptable programming system does not matter. We start with a lemma.

Lemma 2.38: Suppose $\varphi_0, \varphi_1, \cdots$ and $\psi_0, \psi_1, \cdots$ are acceptable programming systems. Then there is a recursive, monotone increasing function g such that $g(0) > 0$ and $(\forall x)\varphi_x = \psi_{g(x)}$.

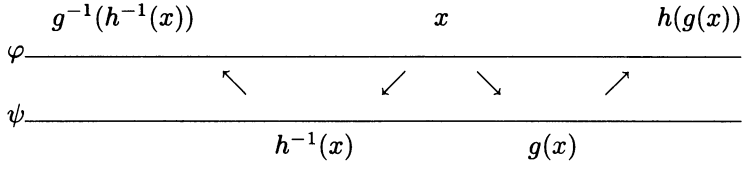
Proof: Let h be a translation function from the φ's to the ψ's obtained by Theorem 2.4 . Let p a padding function for the ψ's. The appropriate g is defined by:

$$g(0) = p(h(0), \min y[p(h(0), y) > 0])$$
$$g(x+1) = p(h(x+1), \min y[p(h(x+1), y) > g(x)]).$$

⊗

Theorem 2.39: (The Isomorphism Theorem) Suppose $\varphi_0, \varphi_1, \cdots$ and ψ_0, ψ_1, $\cdots$ are acceptable programming systems. Then there is a recursive, one-to-one and onto function f such that for all x, $\varphi_x = \psi_{f(x)}$.

Proof: By the above lemma, let g and h be positive valued, monotone increasing, recursive functions such that for all x, $\varphi_x = \psi_{g(x)}$ and $\psi_x = \varphi_{h(x)}$. Note that for any x, $h(x) > x$ and $g(x) > x$. For some arbitrary x we can start backtracking as in the following diagram.



If x is in the range of h, then $h^{-1}(x)$ exists. Furthermore, since h is monotone, one can effectively test whether or not x is in the range of

h. If $h^{-1}(x)$ exists and is in the range of g, then $g^{-1}(h^{-1}(x))$ exists. Each time an inverse is applied the resultant value decreases, so sooner or later, a value is reached which is not in the range of the other translation function. This is called a *dead end*.

For any recursive function f, let f^n denote the n-fold composition of f with itself, e.g., $f^0 = \lambda x[x]$, $f^1 = f$, and $f^{n+1} = f \circ f^n$. For any x *either* for some i, $(g^{-1} \circ h^{-1})^i(x)$ exists and is not in the range of h (a dead end in the φ's) *or* for some i, $h^{-1}((g^{-1} \circ h^{-1})^i(x))$ exists and is not in the range of g (a dead end in the ψ's). By the monotonicity of g and h, for any x and y, either the following two sequences are identical or they are disjoint.

$$\{\ldots, (g^{-1} \circ h^{-1})^2(x), (g^{-1} \circ h^{-1})^1(x), x, (h \circ g)^1(x), (h \circ g)^2(x), \ldots\}$$
$$\{\ldots, (g^{-1} \circ h^{-1})^2(y), (g^{-1} \circ h^{-1})^1(y), y, (h \circ g)^1(y), (h \circ g)^2(y), \ldots\}$$

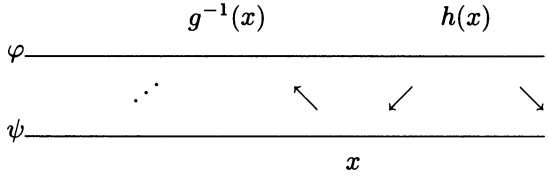
Now we can define the desired f:

$$f(x) = \begin{cases} g(x) & \text{if } x \text{ dead ends in the } \varphi\text{'s;} \\ h^{-1}(x) & \text{if } x \text{ dead ends in the } \psi\text{'s.} \end{cases}$$

Clearly, f is recursive and for all x, $\varphi_x = \psi_{f(x)}$. For the theorem, we must show that f is one-to-one and onto.

Suppose by way of contradiction that f is not one-to-one. Choose x and y least such that $x \neq y$ and $f(x) = f(y)$. Since g and h are one-to-one and monotone it must be that *either* $f(x) = g(x)$ and $f(y) = h^{-1}(y)$ *or* $f(x) = h^{-1}(x)$ and $f(y) = g(y)$. Suppose the former, the other case is similar. Since $g(x) = h^{-1}(y)$, $h(g(x)) = y$. Hence, both x and y dead end in the same system. $(\Rightarrow\Leftarrow)$

Suppose by way of contradiction that f is not onto. Choose x to be the least number not in the range of f. Actually, any x not in the range of f will do, but we must choose a specific one. Consider the value $f(h(x))$. $f(h(x)) \neq h^{-1}(h(x))$, as that would place x in the range of f. Consequently, $h(x)$ dead ends in the φ's. Consider the following picture:



Since $h(x)$ dead ends in the φ's, $g^{-1}(x)$ must exist. Furthermore, $g^{-1}(x)$ also dead ends in the φ's. So $f(g^{-1}(x)) = g(g^{-1}(x)) = x$. $(\Rightarrow\Leftarrow)\otimes$

§2.5 Algorithmically Unsolvable Problems

Computers have become ubiquitous in our society. Their awesome power gives the impression that computers can be programmed to do almost anything, given sufficiently many cycles and memory structures. Many people believe that, eventually, computers will be programmed to simulate a human mind. Others even believe that ultimately computers will surpass the ability of humans to perform tasks, such as language learning, where only humans have so far been successful. In order to better understand what it is that computers can do, we will explore the boundary between what they can do and what they can't. We will proceed by exhibiting some problems that cannot be solved by a computer, now or ever. Techniques to show problems unsolvable will be developed along the way.

The first problem that we will consider is called the *halting problem* because it is the problem of deciding whether or not a given program halts on a given input. Rather than write "the computation of program i on input x halts" we will write the notationally concise $\varphi_i(x)\downarrow$. Similarly, $\varphi_i(x)\uparrow$ denotes a computation that does not halt. Halting computations are called *convergent* and nonhalting computations are called *divergent*.

Theorem 2.40: (Unsolvability of the Halting Problem) Suppose $\varphi_0, \varphi_1, \cdots$ is an acceptable programming system. There is *no* recursive function f such that for all x and y:

$$f(x,y) = \begin{cases} 1 & \text{if } \varphi_x(y)\downarrow, \\ 0 & \text{if } \varphi_x(y)\uparrow. \end{cases}$$

Proof: Suppose by way of contradiction such an f exists. By the recursion theorem there is a program e such that:

$$\varphi_e(x) = \begin{cases} 1 & \text{if } f(e,x) = 0, \\ \uparrow & \text{otherwise.} \end{cases}$$

$\varphi_e(x)\downarrow \Rightarrow f(e,x) = 1 \Rightarrow \varphi_e(x)\uparrow \Rightarrow f(e,x) = 0 \Rightarrow \varphi_e(x)\downarrow$ $(\Rightarrow\Leftarrow)$ ⊗

It is common to associate a problem with a set. In this way, a problem is solvable iff there is a recursive function that decides membership in the associated set. The sets associated with solvable problems are called *recursive*. Given some set A, the *characteristic function* of A is the function:

$$C_A(x) = \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{if } x \notin A. \end{cases}$$

Now we can say that a set is recursive iff its characteristic function is a recursive function. The set associated with the halting problem is called K and formally is $\{x | \varphi_x(x)\downarrow\}$. The set used in the proof above is known as $K_0 = \{x, y | \varphi_x(y)\downarrow\}$. Usually, a direct recursion theorem argument can be used to show that some problem is undecidable.

In order to contrast proof techniques, we will now give a proof of the unsolvabilitiy of the halting problem that does not use the recursion theorem. As before, we suppose by way of contradiction that such an f exists. Then there is a program e such that:

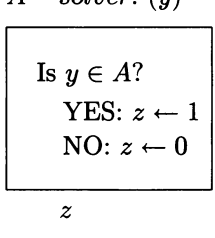
$$\varphi_e(x) = \begin{cases} 1 & \text{if } f(x,x) = 0, \\ \uparrow & \text{otherwise.} \end{cases}$$

Now we use a self-application technique:

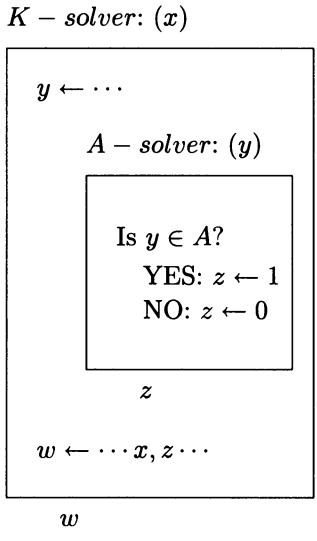
$$\varphi_e(e)\downarrow \Rightarrow f(e,e) = 1 \Rightarrow \varphi_e(e)\uparrow \Rightarrow f(e,e) = 0 \Rightarrow \varphi_e(e)\downarrow \ (\Rightarrow\Leftarrow).$$

This ends the alternative proof of the unsolvability of the halting problem. Proofs by various recursion theorems can always be replaced by proofs that don't make use of self-referential techniques. Although it is not evident with the above example, proofs with the recursion theorem is generally more succinct. We will see other, more typical, examples of this phenomenon later on. The more succinct a proof is, the easier it is for the reader to locate the key features.

We illustrate another technique for showing that certain problems are unsolvable, reduction from the halting problem, below. The formal notion of reduction is discussed in greater detail in Chapter 4. Reduction techniques take a problem of interest and transform its instances into instances of some other problem that is known to be unsolvable. Hence, a solution to the problem of interest would imply the existence of a solution to the unsolvable problem. For example, suppose we are interested in showing that some problem, represented by the set A is unsolvable. First we suppose by way of contradiction that there is a program $A - solver$:



Next, we use this program to develop a solution to the halting problem. This alleged solution will be a program called $K - solver$.



If K – *solver* turns out to be a program that decides membership in K, i.e., solves the halting problem, then we know that the program A – *solver* can not exist, hence A also represents an unsolvable problem. Some specific examples follow.

Theorem 2.41: For any parameters y and z, none of the following sets are recursive.

$$A = \{x | \varphi_x \text{ is a constant function } \}$$
$$B(y) = \{x | y \in \text{ range } \varphi_x\}$$
$$C(y,z) = \{x | \varphi_x(y) = z\}.$$

Proof: The halting problem is *partially solvable* since there is a program i such that:

$$\varphi_i(x,y,z) = \begin{cases} y & \text{if } x \in K; \\ \uparrow & \text{otherwise.} \end{cases}$$

Let $f(x,y) = s(i,x,y)$. If $x \in K$ then program $f(x,y)$ computes the constant y function. Now, it is easy to see:

$$x \in K \Leftrightarrow f(x,0) \in A$$
$$x \in K \Leftrightarrow f(x,y) \in B(y)$$
$$x \in K \Leftrightarrow f(x,z) \in C(y,z).$$

⊗

Exercise 2.42: Use the recursion theorem to show that the set A is not recursive.

Exercise 2.43: Use the recursion theorem to show that, for any y, the set $B(y)$ is not recursive.

Exercise 2.44: Use the recursion theorem to show that, for any y and z, the set $C(y, z)$ is not recursive.

Exercise 2.45: Use the mutual recursion theorem to show that the program equivalence problem is unsolvable, i.e., the set $\{(x, y) | \varphi_x = \varphi_y\}$ is not recursive.

In fact, no nontrivial property of the input/output behavior of programs can be decided uniformly by looking at the program. In other words, all interesting properties of programs are undecidable. To state the theorem verifying the above intuition, we must define what is commonly called an index set.

Suppose $\mathcal{C}$ is a class of partial recursive functions. Then $P_{\mathcal{C}} = \{x | \varphi_x \in \mathcal{C}\}$ is the *index set* of programs computing functions in $\mathcal{C}$. Notice that if $i \in P_{\mathcal{C}}$ and $\varphi_i = \varphi_j$ then $j \in P_{\mathcal{C}}$.

The word "index" is synonymous in its use in the above definition with our use of the word "program." In an acceptable programming system, the programs, because they are named by natural numbers, also serve to index the partial recursive functions. We will use the traditional terminology rather than risk confusion between *program set*, a defined technical term, and *set of programs*, a collection of programs that may or may not be an index set.

Exercise 2.46: Is the set $\{e | \varphi_e = \varphi_{e+1}\}$ an index set? Justify your answer.

Exercise 2.47: Is the set $\{e | \varphi_e$ is primitive recursive$\}$ an index set? Justify your answer.

Armed with the notion of an index set, we are ready to prove that all interesting properties concerning the input/output behavior of programs are undecidable. By an "interesting property" we mean one that holds for some program and not for others. Interesting properties can be used to distinguish some of the programs from rest of the crowd. Index sets are natural, since if two programs have the same input/output behavior, then they are either both in some index set, or they are both outside of it. The next theorem shows that the only decidable properties of program input/output behavior, represented by the recursiveness of index sets, are ones that hold for all programs, or for none.

Theorem 2.48: Suppose $\mathcal{C}$ is a class of partial recursive functions. $P_{\mathcal{C}}$ is recursive iff $P_{\mathcal{C}} = \emptyset$ or $P_{\mathcal{C}} = \mathbb{N}$.

Proof: $\mathbb{N}$ and $\emptyset$ are clearly recursive sets. The converse is proved via the contrapositive. Suppose $x_1 \in P_{\mathcal{C}}$ and $x_2 \notin P_{\mathcal{C}}$. Suppose by way of contradiction that $P_{\mathcal{C}}$ is recursive. By the recursion theorem there is a program e such that :

$$\varphi_e = \begin{cases} \varphi_{x_1} & \text{if } e \notin P_{\mathcal{C}}; \\ \varphi_{x_2} & \text{if } e \in P_{\mathcal{C}}. \end{cases}$$

$e \in P_{\mathcal{C}} \Rightarrow \varphi_e = \varphi_{x_2} \notin \mathcal{C} \Rightarrow e \notin P_{\mathcal{C}} \Rightarrow \varphi_e = \varphi_{x_1} \in \mathcal{C} \Rightarrow e \in P_{\mathcal{C}}$ $(\Rightarrow\Leftarrow)$ $\otimes$

As an example, we will show that the set $Z = \{x \mid \varphi_x(0) = 0\}$ is not recursive. First we must show that Z is an index set. If $x \in Z$ then $\varphi_x(0) = 0$. If $\varphi_x = \varphi_y$, then $\varphi_y(0) = 0$. Therefore, $y \in Z$. A similar argument shows that if $x \notin Z$ and $\varphi_x = \varphi_y$, then $y \notin Z$. Hence, Z is an index set. Z is not empty, as witnessed by any program for $\lambda x[0]$. A program for the successor function serves as a witness that $Z \neq \mathbb{N}$. Therefore, by Theorem 2.48, Z is not recursive.

Again, to contrast proof techniques, we will present an alternative proof of Theorem 2.48. We will use a reduction to the halting problem. As in our original proof, $\mathbb{N}$ and $\emptyset$ are clearly recursive sets. Suppose by way of contradiction that $P_{\mathcal{C}} \neq \emptyset$, $P_{\mathcal{C}} \neq \mathbb{N}$ and $P_{\mathcal{C}}$ is recursive. Then there is an $x_0 \notin P_{\mathcal{C}}$ and an $x_1 \notin P_{\mathcal{C}}$. Suppose that u is a program for the everywhere undefined function, i.e. $\varphi_u = \lambda x[\uparrow]$. There are two cases to consider.

Suppose $u \in P_{\mathcal{C}}$. By an implicit use of the *s-m-n* theorem there is a recursive function f such that:

$$\varphi_{f(x)} = \begin{cases} \varphi_{x_0} & \text{if } \varphi_x(x) \downarrow, \\ \varphi_u & \text{otherwise.} \end{cases}$$

$$x \in K \Leftrightarrow \varphi_{f(x)} = \varphi_{x_0} \Leftrightarrow f(x) \notin P_{\mathcal{C}}$$
$$x \notin K \Leftrightarrow \varphi_{f(x)} = \varphi_u \Leftrightarrow f(x) \in P_{\mathcal{C}}.$$

Hence, $x \in K$ iff $f(x) \notin P_{\mathcal{C}}$. Consequently, K is recursive. $(\Rightarrow\Leftarrow)$

Suppose $u \notin P_{\mathcal{C}}$. By an implicit use of the *s-m-n* theorem

$$\varphi_{f(x)} = \begin{cases} \varphi_{x_1} & \text{if } \varphi_x(x) \downarrow, \\ \varphi_u & \text{otherwise.} \end{cases}$$

$$x \in K \Leftrightarrow \varphi_{f(x)} = \varphi_{x_1} \Leftrightarrow f(x) \in P_{\mathcal{C}}$$
$$x \notin K \Leftrightarrow \varphi_{f(x)} = \varphi_u \Leftrightarrow f(x) \notin P_{\mathcal{C}}$$

Hence, $x \in K$ iff $f(x) \in P_{\mathcal{C}}$. Consequently, K is recursive, $(\Rightarrow\Leftarrow)$.

In both cases, a contradiction was achieved, thus ending the alternative proof of Theorem 2.48.

§2.6 Recursively Enumerable Sets

We have seen that few interesting properties of programs are effectively decidable. All is not lost since there is a notion of partial decidability. Suppose we can devise an algorithm that outputs all and only members of some set A. We will call this algorithm a *generator* for A. If some x is a member of A, then, eventually, we will discover this by running the generator until x appears as output. For an arbitrary x, we will never know if it has not yet appeared or if it will never appear. Consequently, in this case, A is only partially solvable. Technically, we will say that A is *recursively enumerable*.

Definition 2.49: A set A is *recursively enumerable* (r.e.) iff $A = \emptyset$ or A is the range of a recursive function.

If a set is decidable, then so is its complement. However, for partial decidability this is not the case. To make our further discussion of this issue more precise, we introduce the usual notation that $\overline{A}$ denotes the complement of the set A, i.e. $\{x | x \in \mathbb{N} \text{ and } x \not\in A\}$.

Theorem 2.50: A set is recursive iff both it and its complement are r.e.

Proof: Suppose A is recursive. If $A = \emptyset$ then A and $\overline{A}$ are easily seen to be r.e. Suppose $y \in A$. Let C_A denote the characteristic function of A. Since C_A is recursive, so is the following function f:

$$f(x) = \begin{cases} x & \text{if } C_A(x) = 1; \\ y & \text{otherwise.} \end{cases}$$

Since $A = \{f(x) | x \in \mathbb{N}\}$, A is r.e. A similar argument shows that $\overline{A}$ is also r.e.

Suppose A and $\overline{A}$ are r.e. Suppose f and g are recursive functions generating A and $\overline{A}$ respectively. The characteristic function of A can be defined as:

$$C_A(x) = \begin{cases} 1 & \text{if } f(\mu y[f(y) = x \text{ or } g(y) = x]) = x; \\ 0 & \text{otherwise.} \end{cases}$$

⊗

Theorem 2.51: A is r.e. iff (A is the range of a partial recursive function) iff (A is the domain of a partial recursive function).

Proof: The empty set is both the domain and range of $\lambda x[\uparrow]$. Suppose $A \neq \emptyset$. We need to formalize the notion of the number of steps of a computation. While it may not be clear what a step is in some bizarre acceptable programming system, it is clear for the RAM programs. Hence, we can build a step counting function in the RAM programming system, and translate it over to any acceptable programming system. Our general *step* function below may really be measuring the number of steps taken by an equivalent RAM program, but that will be good enough for our purposes here.

$$step(x, y, z) = \begin{cases} 0 & \text{if } \varphi_x(y) \text{ not convergent after } z \text{ steps;} \\ \varphi_x(y) + 1 & \text{otherwise.} \end{cases}$$

The theorem will follow from the next two lemmas, which are interesting in their own right. The first lemma uses a technique called *dovetailing*. The essential idea is that you want to run more and more programs on more and more inputs or more and more steps. At the i^{th} stage of a dovetailing process you run programs 0 through i, on arguments 0 through i, for i steps each, looking for some property such as convergence to a particular value. A pictorial rendition of this process reminds some people of a dove's tail. Our description of this process is facilitated by the *step* function defined above.

Lemma 2.52: There is a recursive function g such that for all x the range of φ_x is the same as the domain of $\varphi_{g(x)}$.

Proof: (by dovetailing) Define program i such that :

$$\varphi_i(x,y) = \mu z[step(x, \pi_1(z), \pi_2(z)) = y + 1].$$

Let $g = \lambda x[s(i,x)]$. Then,

$$\begin{aligned}\varphi_{g(x)}(y) &= \varphi_{s(i,x)}(y) \\ &= \varphi_i(x,y) \\ &= \begin{cases} \downarrow & \text{if } y \in \text{range } \varphi_x; \\ \uparrow & \text{otherwise.} \end{cases}\end{aligned}$$

⊗ (Lemma)

Lemma 2.53: There is a recursive function h such that for all x the range of $\varphi_{h(x)}$ is the same as the domain of φ_x. Furthermore, if the domain of φ_x is not empty then $\varphi_{h(x)}$ is a recursive function.

Proof: Define program i such that:

$$\varphi_i(x,0) = \pi_1(\mu z[step(x, \pi_1(z), \pi_2(z)) \neq 0])$$

$$\varphi_i(x,y+1) = \begin{cases} \varphi_i(x,y) & \text{if } step(x, \pi_1(y+1), \pi_2(y+1)) = 0; \\ \pi_1(y+1) & \text{otherwise.} \end{cases}$$

Notice that $\varphi_i(x,0)$ converges (if at all) to the first argument y such that φ_x is defined on y. Let $h = \lambda x[s(i,x)]$. ⊗ (Lemma)

⊗ (Theorem)

Let $\psi(x) = \varphi_{univ}(x,x)$. ψ is partial recursive with domain K, hence K is r.e. Since K is r.e., its complement cannot be r.e., as otherwise, K would be recursive and the halting problem would be solvable, contradicting Theorem 2.40.

Although the range characterization of r.e. sets is a more natural fit with the intuition of being able to list effectively elements of an r.e. set, the domain characterization is often more convenient. Consider some r.e. set A. In general, one cannot effectively determine if some arbitrary x is a member of A. However, one can start a computation that will (eventually) converge iff $x \in A$. Consider the following computation using the range characterization of r.e. sets. Suppose that the range of f, a recursive function, is A. Look for the least y such that $f(y) = x$. This may involve several subcomputations ($f(0)$, $f(1)$, $\cdots$). Now consider using the following computation that uses the domain characterization of r.e. sets. Suppose that the domain of φ_i is A. Then we need only run a single computation: $\varphi_i(x)$. This computation will converge iff $x \in A$. Consequently, a special notation has been developed for r.e. sets. The domain of φ_i is denoted by W_i and is often called the i^{th} r.e. set.

Exercise 2.54: Is there a natural number m such that

a) $W_m = \{m^2\}$?
b) $W_m = \mathbb{N} - \{m\}$?
c) $W_m = \{x | \varphi_m(x) \uparrow\}$?
d) $W_m = K$ and $m \in K$?
e) $W_m = K$ and $m \notin K$?
f) $W_m = \{x | (\exists a, b, c, d) x = a^2 + b^2 + c^2 + d^2\}$
g) $W_m = \{x | x \notin K \text{ and } x \leq 1000\}$

Justify your answers.

Exercise 2.55: Use the function *Comp* to show that the function *step* is primitive recursive.

Exercise 2.56: Prove that the range of any monotone increasing recursive function is a recursive set.

Exercise 2.57: Is there an r.e. set of programs such that each program in the set computes a primitive recursive function and every primitive recursive function is represented by at least one program in the set? Justify your answer.

Exercise 2.58: Is there an r.e. set of programs such that each program in the set computes a recursive function and every recursive function is represented by at least one program in the set? Justify your answer.

Exercise 2.59: Is there an r.e. set of programs such that each program in the set computes a partial recursive function and every partial recursive function is represented by at least one program in the set? Justify your answer.

Exercise 2.60: Suppose A is an r.e. set such that for any $x \in A$, φ_x is total. Prove that there is a recursive function g such that for any y, if $\varphi_y = g$ then $y \notin A$.

Exercise 2.61: A set S is *productive* iff there is a recursive function g such that whenever $W_x \subset S$, $g(x) \in (S - W_x)$. (The symbol "$\subset$" denotes proper subset.) A set S is *creative* iff S is r.e. and $\overline{S}$ is productive. Prove that K is creative.

Exercise 2.62: Prove or disprove: The r.e. sets are closed under union.

Exercise 2.63: Prove or disprove: The r.e. sets are closed under complementation.

Exercise 2.64: Prove or disprove: The r.e. sets are closed under intersection.

Exercise 2.65: The *symmetric difference* of two sets consists of all the elements of the first set that are not in the second and all the elements of the second that are not in the first. Prove or disprove: The symmetric difference of two r.e. sets is r.e.

Exercise 2.66: A total function f is *monotone increasing* if $f(x+1) > f(x)$ for all x. Prove that a set is recursive iff it is the range of a monotone increasing recursive function.

Exercise 2.67: Prove or disprove: Every infinite r.e. set has an infinite recursive subset.

Exercise 2.68: Suppose A and B are two r.e. sets such that $A \cup B = \mathbb{N}$ and $A \cap B$ is a recursive set. Show that A and B are both recursive.

Exercise 2.69: Suppose A and B are two r.e. sets such that $A \cup B = \mathbb{N}$ and $A \cap B$ is a finite set. Show that A and B are both recursive.

Exercise 2.70: Show that if A is an r.e. set then there exists a recursive set of ordered pairs B such that $A = \{x | (\exists y)(x, y) \in B\}$.

Exercise 2.71: Does there exist a set $A \subset K$ such that $\overline{A}$ is r.e. and A is not r.e.?

For analytical functions it is often useful for comprehension to draw a graph of the function. This graph is actually the collection of points in two-dimensional space that collectively define the function. This suggests the following definition.

Definition 2.72: For ψ a partial recursive function, the *graph* of ψ is defined as:

$$graph_\psi = \{(x, \psi(x)) | \psi(x) \text{ is defined}\}.$$

Exercise 2.73: Prove or disprove: $graph_\psi$ is always a recursive set.

Exercise 2.74: Prove or disprove: If ψ is a recursive function then $graph_\psi$ is a recursive set.

Exercise 2.75: Prove or disprove: If $graph_\psi$ is a recursive set then ψ is a recursive function.

Exercise 2.76: Prove or disprove: If $graph_\psi$ is r.e. and for any x there is y such that $(x, y) \in graph_\psi$, then ψ is recursive.

Exercise 2.77: A total function f is called *decreasing* if $x < y$ implies $f(x) \geq f(y)$. For each of the following statements say whether they are true or false. Justify your answer with either a counterexample or a short proof.

a) If f is recursive and decreasing then range(f) is recursive.
b) If f is recursive and decreasing then range(f) is r.e.
c) If f is decreasing then range(f) is recursive.
d) If f is decreasing then range(f) is r.e.

Exercise 2.78: Show that if A is an infinite recursive set then there exists sets B and C such that $A = B \cup C$, $B \cap C = \emptyset$, and neither B nor C is r.e.

Exercise 2.79: Prove or disprove: Every infinite index set has an infinite r.e. subset.

Exercise 2.80: Prove or disprove: There is an infinite recursive subset of $\{x | \varphi_x \text{ is recursive } \}$.

Exercise 2.81: Prove or disprove: If A and B are r.e. then $\overline{A} \cap B$ is r.e.

When discussing Theorem 2.48, we noted that all interesting input/output behaviors of programs are undecidable. Perhaps some of them are r.e. If some property is undecidable, it could be of great use to be able to list effectively all the programs with the property. To investigate this issue, we will prove an analog of Theorem 2.48 for the r.e. sets.

Consider the process of effectively enumerating all programs with a certain property. Intuitively, when a particular program is added to the forming enumeration, the decision to include it was made based on some finite amount of information. This intuition is formalized in the proof below in what is sometimes called a *key array*. The test for inclusion in the enumeration is for some program to match one of the "keys" in the array. For the procedure of key matching to be effective, the keys must be some finite set. Not any listing of a finite set will do. We must be able to tell when the description of the finite set is complete, as otherwise, we may find a match with some potential program, include it on our enumeration and then later find out that the key was incomplete and the listed program should *not* have been enumerated.

Enumerations of finite sets where the end of the enumeration is also known are called *canonical*. We proceed to define a canonical enumeration of all and only the finite sets. For a given x, consider the binary representation of x and rewrite x as the sum of some powers of 2: $x = 2^{y_n} + \ldots + 2^{y_0}$ for some n and $y_0 < \ldots < y_n$. Then, $D_x = \{y_0, \ldots, y_n\}$. Clearly, $D_0, D_1, \ldots$ is an enumeration of all and only the finite sets. Furthermore, given x, not only can we enumerate all of D_x, but we can also know when we are done with the enumeration. Hence, the list above is canonical.

The test for recursive enumerability of index sets below depends on the existence of a key array of canonical finite sets. If a suitable key array exists, then the index set can be enumerated effectively by listing all programs that match one of the keys. This effective test is formalized in the next theorem.

Theorem 2.82: Let $\mathcal{C}$ be any class of r.e. sets. $P_{\mathcal{C}} = \{x | W_x \in \mathcal{C}\}$ is r.e. iff (there is an r.e. set A such that $W_x \in \mathcal{C}$ iff $(\exists y \in A) D_y \subseteq W_x$).

Proof: Suppose A satisfies the hypothesis. Then $P_{\mathcal{C}} = \{x | (\exists y \in A) D_y \subseteq W_x\}$. If $A = \emptyset$ then $P_{\mathcal{C}} = \emptyset$, an r.e. set. Suppose $A \neq \emptyset$ and f is a recursive function with range A. Define a partial recursive function:

$$\psi(z) = \begin{cases} \pi_1(z) & \text{if } D_{f(\pi_2(z))} \subseteq W_{\pi_1(z)} \\ \uparrow & \text{otherwise.} \end{cases}$$

So $P_{\mathcal{C}} = \text{range } \psi$. The other half of the theorem is proven with the aid of two lemmas that have independent interest.

Lemma 2.83: If $P_{\mathcal{C}}$ is r.e. and $W \in \mathcal{C}$ then there is a y such that $D_y \subseteq W$ and $D_y \in \mathcal{C}$.

Proof: Choose i such that domain $\varphi_i = P_{\mathcal{C}}$ and $W \in \mathcal{C}$. Let φ be a partial recursive function with domain W. By the recursion theorem there is a program e such that:

$$\varphi_e(x) = \begin{cases} \varphi(x) & \text{if } \varphi_i(e) \text{ does not converge in } x \text{ steps;} \\ \uparrow & \text{otherwise.} \end{cases}$$

If $e \notin P_{\mathcal{C}}$ then for every x it is discovered that $\varphi_i(e)$ does *not* converege within x steps, so $\varphi_e = \varphi$ $(\Rightarrow\Leftarrow)$. So $e \in P_{\mathcal{C}}$ and $\varphi_e = \{(x, \varphi(x)) | x < z\}$ where z is the least number such that $\varphi_i(e) \downarrow \leq z$ steps. Hence, φ_e is a finite function. Choose y such that $D_y =$ domain φ_e. $\bigotimes$ (Lemma)

Exercise 2.84: Find a proof of Lemma 2.83 that does not use the recursion theorem.

Lemma 2.85: If $P_{\mathcal{C}}$ is r.e., $W_j \in \mathcal{C}$ and W_k is an r.e. set such that $W_j \subseteq W_k$ then $W_k \in \mathcal{C}$.

Proof: Choose i such that domain $\varphi_i = P_{\mathcal{C}}$. Suppose $W_j \in \mathcal{C}$ and $W_j \subseteq W_k$. If $W_j = W_k$ the lemma follows. Suppose $W_j \subset W_k$. Suppose by way of contradiction $W_k \notin \mathcal{C}$. By the recursion theorem there is a program e such that

$$W_e = \begin{cases} W_k & \text{if } \varphi_i(e) \downarrow; \\ W_j & \text{otherwise.} \end{cases}$$

At first glance, the above may not seem like a description of an effective algorithm. Consider the following verbal description of how program e proceeds on input x. Initially, program e starts up two subprocesses that proceed either in parallel or in a time sharing fashion. The two subprocesses compute $\varphi_i(e)$, to check if $e \in P_{\mathcal{C}}$, and $\varphi_j(x)$, to check if $x \in W_j$. If neither subprocess terminates, then $\varphi_e(x) \uparrow$, in which case $x \notin W_e$. If the second subcomputation converges before the first subcomputation, then x was found to be in W_j. If the first subcomputation converges later on, then there is no conflict as $W_j \subset W_k$, so $x \in W_k$ also. In the case where the second subcomputation converges, the computation of $\varphi_e(x)$ halts, placing $x \in W_e$. The final case to consider is when the first subcomputation converges, but the second one does not. When the first subcomputation halts, program e terminates the other subcomputation and proceeds to compute $\varphi_k(x)$. In this case, program e will halt iff program k halts on input x. In any event, either $W_e = W_k$ or $W_e = W_j$.

$W_e = W_k \Rightarrow W_e \in \mathcal{C} \Rightarrow W_k \in \mathcal{C}$ $(\Rightarrow\Leftarrow)$

$W_e = W_j \Rightarrow W_e \notin \mathcal{C} \Rightarrow W_j \notin \mathcal{C}$ $(\Rightarrow\Leftarrow)$.

$\bigotimes$ (Lemma)

Exercise 2.86: Find a proof of Lemma 2.85 that does not use the recursion theorem.

Continuing with the theorem, choose i such that domain $\varphi_i = P_{\mathcal{C}}$. By the *s-m-n* theorem there is a recursive function k such that for all y and x:

$$\varphi_{k(y)}(x) = \begin{cases} 1 & \text{if } x \in D_y; \\ \uparrow & \text{otherwise.} \end{cases}$$

Note that $W_{k(y)} = D_y$. Since k is recursive, $\varphi_i \circ k$ is partial recursive. Let A be the domain of $\varphi_i \circ k$, an r.e. set. Now,

$$y \in A \Leftrightarrow \varphi_i(k(y)) \downarrow \Leftrightarrow W_{k(y)} \in \mathcal{C}.$$

Suppose $W_x \in \mathcal{C}$. We must show that there is a $y \in A$ such that $D_y \subseteq W_x$. By the first lemma, there is y such that $D_y \subseteq W_x$ and $D_y \in \mathcal{C}$. By the definition of k, $W_{k(y)} = D_y$. Hence, $W_{k(y)} \in \mathcal{C}$. Therefore, $y \in A$.

Suppose that there is a $y \in A$ such that $D_y \subseteq W_x$. We must show that $W_x \in \mathcal{C}$. Since $y \in A$, $W_{k(y)} \in \mathcal{C}$. By the second lemma, and the fact that $W_{k(y)} = D_y$, it follows that $W_x \in \mathcal{C}$. ⊗ (Theorem)

Notice that $\{x | \varphi_x \text{ total}\}$ is not r.e. by Lemma 2.83 and that $\{x | \varphi_x \text{ not total}\}$ is not r.e. by Lemma 2.85. Using the above theorem is the easiest way to show that some index sets are not r.e. Sometimes, however, we may be faced with sets that are not r.e. but either they are not index sets, or it is very awkward to use the above theorem. Some examples of these sets are in the exercise below. In these cases, either reduction techniques, or direct recursion theorem arguments give fairly simple solutions.

Using the canonical list of finite functions, $d_0, d_1, \ldots$, from Exercise 1.35 it is possible to give an alternative formulation of Theorem 2.82.

Exercise 2.87: Prove: Let $\mathcal{C}$ be any class of partial recursive functions. $P_{\mathcal{C}} = \{x | \varphi_x \in \mathcal{C}\}$ is r.e. iff (there is an r.e. set A such that $\varphi_x \in \mathcal{C}$ iff $(\exists y \in A) d_y \subseteq \varphi_x$).

Exercise 2.88: Which of the following sets are recursive? Which are r.e.? Justify your answers.

a) $\{\langle x, y\rangle | \varphi_x = \varphi_y\}$
b) $\{x | (\exists y, z) \varphi_x(y) \downarrow \text{ and } \varphi_y(z) \downarrow\}$
c) $\{x | (\exists y, z) \varphi_x(y) \downarrow \text{ and } \varphi_y(z) \uparrow\}$
d) $\{x | (\exists y) \varphi_x(y) \downarrow \text{ and } \varphi_y \text{ is total}\}$
e) $\{x | c \in W_x\}$, for some constant c
f) $\{x | W_x \neq \emptyset\}$
g) $\{x | W_x \text{ is infinite}\}$
h) $\{x | W_x \text{ is recursive}\}$
i) $\{x | W_x \cap K \neq \emptyset\}$
j) $\{x | W_x \text{ is finite }\}$
k) $\{x | W_x = \{x\}\}$
l) $\{x | \varphi_x(0) = 0\}$
m) $\{x | W_x \subseteq \{y | y \text{ is even }\}\}$
n) $\{x | \varphi_x \text{ is primitive recursive }\}$

o) $\{x | cardinality(W_x) = 1\}$
p) $\{x | cardinality(W_x) > 10\}$
q) $\{x | (\exists y)\varphi_x(y) \uparrow \text{ and } y \in K\}$
r) $\{x | W_x \subseteq K\}$
s) $\{x | \overline{W_x} \cap K \neq \emptyset\}$
t) $\{x | x \in K \text{ and } x < 1000\}$
u) $\{x | x \text{ is prime}\}$
v) $\{x | x \in K \text{ and } x \text{ is prime}\}$
w) $\{x | (\exists y)A(x, y)\}$ for some recursive set A
x) $\{x | \varphi_x \text{ is total and its range is included in the even numbers}\}$
y) $\{x | W_x \text{ is finite and } x > 1000\}$
z) $\{x | \varphi_x \text{ halts on exactly five elements }\}$
aa) $\{\langle x, y\rangle | W_x = W_y\}$
bb) $\{x | \text{ the set of primes} \subseteq \overline{W_x}\}$
cc) $\{x | W_x \text{ is an infinte union of finite sets }\}$
dd) $\{x | W_x \text{ is not an infinte union of finite sets }\}$
ee) $\{x | W_x \text{ is recursive and } x \leq 666\}$
ff) $\{x | W_x \text{ is recursive and } x \geq 666\}$
gg) $\{x | W_x \text{ is r.e. and } x \leq 666\}$
hh) $\{x | W_x \text{ is r.e. and } x \geq 666\}$
ii) $\{x | W_x \text{ is infinite and } \overline{W_x} \text{ is infinite}\}$
jj) $\{x | \varphi_x \text{ is recursive and } x < 1000\}$
kk) $\{x | (\forall y)\varphi_x(y) = y^2\}$
ll) $\{x | (\exists y)\varphi_x(y) = y^2\}$

Exercise 2.89: Suppose that A is an index set such that 1) $A \neq \emptyset$, 2) $A \neq \mathbb{N}$, and 3) A contains an index for the everywhere undefined function. Prove that A is not r.e.

§2.7 Historical Notes

The notion of an acceptable programming system stems from [Rog1], where the isomorphism theorem appears. The alternative characterizations can be found in [Ric] and [Roy]. The recursion theorem first appears in [Kle], the mutual recursion theorem in [Smu] and the operator recursion theorem in [Ca1]. For intuitive discussion and further examples of these recursion theorems see [Ca2]. The fixed point theorem is from [Rog2]. The parametric recursion theorem is also credited to Kleene. The unsolvability of the halting problem first appeared in [Tur]. Most of the results on recursive and r.e. sets come from [Pos], along with the (relatively) informal style of our presentation. The material on index sets is from [Rce]. For a much more comprehensive study of recursion theory, see [Rog2] and [Soa]. The proof of Theorem 2.8 is due to R. Byerly.

[Ca1] J. Case, Periodicity in Generations of Automata, *Mathematical Systems Theory*, Vol. 8, 1974, pp. 15–32.

[Ca2] J. Case, Infinitary Self-Reference in Learning Theory, *Journal of Experimental and Theoretical Artificial Intelligence*, Vol. 8, 1994, pp. 3–16.

[Kle] S. Kleene, On Notation for Ordinal Numbers, *Journal of Symbolic Logic*, Vol. 3, 1938, pp. 150–155.

[Pos] E. Post, Recursively Enumerable Sets of Positive Integers and their Decision Problems, Bulletin of the American Mathematical Society, Vol. 50, 1944, pp. 284–316.

[Rce] H. Rice, On Completely Recursively Enumerable Classes and Their Key Arrays, *Journal of Symbolic Logic*, Vol. 22, 1956, pp. 304–308.

[Ric] G. Riccardi, The Independence of Control Structures in Abstract Programming Systems, *Journal of Computer and Systems Sciences*, Vol. 22, 1981, pp. 107–143.

[Rog1] H. Rogers, Jr., Gödel Numberings of Partial Recursive Functions, *Journal of Symbolic Logic*, Vol. 23, 1958, pp. 331–341.

[Rog2] H. Rogers, Jr., Theory of Recursive Functions and Effective Computability, McGraw Hill, New York, 1967.

[Roy] J. Royer, A Connotational Theory of Program Structure, Lecture Notes in Computer Science 273, Springer Verlag, New York, 1987.

[Smu] R. Smullyan, Theory of Formal Systems, Annals of Mathematical Studies, No. 47, Princeton University Press, Princeton, 1961.

[Soa] R. Soare, Recursively Enumerable Sets and Degrees, Springer Verlag, New York, 1987.

[Tur] A. Turing, On Computable Numbers, with an Application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society*, Vol. 42, 1936, pp. 230–265.

3

Abstract Complexity Theory

In previous chapters we developed an implementation-independent model of computation. The fundamental features of this model were examined in the last chapter. As always, we assume that φ_0, φ_1, $\cdots$ is an acceptable programming system. Now we will develop a complexity theory that is independent of the machine model and of the resource being measured. Consequently, this theory will be insensitive to particular language features and input/output conventions, etc. Given a program i and some measure of complexity (like time or space), the complexity of φ_i is determined by a *function* that determines, for any input x, how much of the resource counted by the complexity measure is used in the computation of $\varphi_i(x)$. In general, the complexity of a function is another function. This is because programs consume different amounts of some resources depending on what inputs they are given. In the case where we are measuring time, any divergent computation will consume infinitely many time steps without converging. However, it is possible for a computation to diverge and only use a small, finite amount of space. If a computation is undefined, then we would like its complexity to be undetermined as well. Consequently, we adopt the convention that if it turns out that the computation of $\varphi_i(x)$ diverges, then we will say that program i on input x uses *infinitely* much of the resource counted by the complexity measure. To make the case of an undefined computation fit neatly into our results we will adopt the convention that in comparisons any natural number will be considered to be less than an undefined quantity.

We will prove results about the general nature of the complexity of functions. Often it is desirable to talk about the complexity of a function, not a program. For example, it is reasonable to ask about the time complexity of sorting. Unfortunately, one cannot define the complexity of a function to be the complexity of the program that computes the given function with the smallest amount of resource used. The reason is, as we will show, that there are functions with no best program. However, it is possible to discuss the collection of all functions that can be computed within some resource bound. First, we will go over some of the fundamental ideas in a less abstract setting.

§3.1 RAM Pseudospace Measure

Suppose $P_0, P_1, \ldots$ is a list of all and only RAM programs. In this section we will assume that not only that $\varphi_0, \varphi_1, \cdots$ is an acceptable programming system, but, furthermore, that RAM program P_i computes the function φ_i, for all i. The *pseudospace* function S_i for P_i is defined by:

$$S_i(x) = \begin{cases} \text{max integer stored in any register} & \text{if } P_i(x)\downarrow; \\ \uparrow & \text{otherwise.} \end{cases}$$

Clearly, for any i, S_i is partial recursive. Some infinite loops don't use an infinite amount of space. Complexity should be a measure of the use of some computational resource and not the intricacy of a program's structure. Notice that S_i doesn't measure all the space used by P_i. In this sense the pseudospace measure is only a crude approximation to the true amount of space used. Hence the name pseudospace. However, the difference is at most a multiplicative constant (the number of registers used by P_i). One feature of this theory is that we will not be distracted by such details. The general results we prove will apply to any "reasonable" complexity measure.

Exercise 3.1: Show that there is no recursive function g such that for each i and x, if $\varphi_i(x)$ converges then $S_i(x) \leq g(x, \varphi_i(x))$.

First, we will show that given any program i, an input x, and a bound y, we can tell if the pseudospace complexity of program P_i on input x is bounded by y.

Proposition 3.2: $S_i(x) \leq y$ is a primitive recursive predicate of i, x and y.

Proof: Let i be given and suppose that P_i has m instructions and references k registers. Then $(y+1)^k$ different register configurations are possible without storing a number larger than y. Hence, if P_i runs for more than $m \cdot (y+1)^k$ steps without using a number larger than y, then P_i is in an infinite loop. By our coding of RAM programs (see page 69), for all $k \leq i$, for all $m \leq i$:

$$S_i(x) \leq y \Leftrightarrow (\forall z \leq i)(\forall w \leq i \cdot (y+1)^i)$$
$$\left[\Pi(z, i, \pi_2(\overbrace{Comp(i, \langle x, 0\rangle, w)}^{w \text{ steps of } P_i})) \leq y \right]$$
$$\text{and } (\exists w \leq i \cdot (y+1)^i)\,[\pi_1(Comp(i, \langle x, 0\rangle, w)) = Ln(i)]$$

⊗

Note that for all i and x, $\varphi_i(x) \leq S_i(x)$. We say that program P_i is *optimal* if $\varphi_i = S_i$. Next we will show that optimal programs exist. In fact, there is a way to compute the pseudospace measure for any program in an optimal fashion.

Proposition 3.3: For any program i there is a program j such that $\varphi_j = S_j = S_i$.

Proof: Given program P_i form P_j by replacing each INC instruction by a subroutine (macro) that performs the INC instruction and compares the result with a monitor register. The monitor register is left holding the larger of the two values. If P_i halts, P_j moves the monitor register to $R1$. Clearly, $S_j = S_i$ and $\varphi_j = S_j$. ⊗

Since there are recursive functions with large values in their ranges and for any program i, $\varphi_i \leq S_i$, it follows that there are intrinsically difficult functions which require a large amount of space. It is easy to write programs that use a vast amount of space. Do all programs for some functions consume large amounts of space? The question is answered affirmatively in the next result. Notice that the function asserted to exist by the next proposition has a range restricted to $\{0,1\}$. The reason for this is to make sure that the function constructed consumes a large amount of space in its calculations, not just to print out the answer.

Proposition 3.4: For all recursive functions t there is a $\{0,1\}$ valued recursive function f such that if $\varphi_i = f$ then $S_i(x) > t(x)$ for infinitely many x.

Proof: Let t be given. Define f by:

$$f(x) = \begin{cases} 1 & \text{if } S_{\pi_1(x)}(x) \leq t(x) \text{ and } \varphi_{\pi_1(x)}(x) \neq 1; \\ 0 & \text{otherwise.} \end{cases}$$

First we show that f is a recursive function. Clearly, f is defined on all arguments. Consequently, we must show that the test "$S_{\pi_1(x)}(x) \leq t(x)$ and $\varphi_{\pi_1(x)}(x) \neq 1$" is effectively computable. Since t is recursive, by Proposition 3.2, one can test if $S_{\pi_1(x)}(x) \leq t(x)$. If this test returns an affirmative answer then the computation of program $\pi_1(x)$ on input x converges. In which case, it is possible to test if $\varphi_{\pi_1(x)}(x) \neq 1$. Hence, f is recursive.

By the definition of the pseudospace measure, for any x, the domain of $\varphi_{\pi_1(x)}$ is the same as the domain of $S_{\pi_1(x)}$. Suppose $\varphi_i = f$. As a consequence of our pairing function, there are infinitely many x such that $\pi_1(x) = i$. For those x's, $S_i(x) > t(x)$, as otherwise $f(x) \neq \varphi_i$. ⊗

The range of the function f above is $\{0,1\}$. The point of this restriction is to make sure that computing f intrinsically requires a lot of space. Restricting the range guarantees that computing f does not consume space because it takes a lot of space to form the output. Rather, the space is consumed in figuring out what the result of the computation is.

§3.2 Abstract Complexity Measures

Now we begin our study of complexity in full generality. Some fundamental properties of complexity measures are presented in this section. To

do so, we first state precisely what we will call a measure of complexity. The presentation is axiomatic and is based on the assumption that the complexity of a program is given by a function, that determines for each input, how much computational resources are consumed by the program on the given input. The first axiom asserts that the function giving the complexity of a program should be defined on all arguments for which the program converges, and no others. Intuitively, if a program does not halt on some input, then we would like the complexity of that computation to be infinite, or, technically, undefined. The second axiom asserts that you can always attach a meter to any computation and determine if the computation finishes within a certain bound.

Definition 3.5: Suppose $\varphi_0, \varphi_1, \cdots$ is an acceptable programming system. Then $\Phi_0, \Phi_1, \cdots$ is a *complexity measure* iff

1. for all i, domain φ_i = domain Φ_i, and
2. $\Phi_i(x) \leq y$ is a recursive predicate of i, x, and y.

The RAM pseudospace measure is a complexity measure, let $\Phi_i = S_i$. Via isomorphism, we can transfer any complexity measure to any acceptable programming system, so every acceptable programming system has a complexity measure. Most notions of time and space can form the basis for a complexity measure.

First we will verify that the two complexity axioms are independent. Suppose that for any i, Φ_i is defined to be equal to φ_i. Then the first axiom is satisfied but not the second. To see this, suppose by way of contradiction that R is a recursive predicate such that $R(i, x, y) = 1$ iff $\Phi_i(x) \leq y$. Then by the recursion theorem there is a program e such that for all x:

$$\varphi_e(x) = \begin{cases} 1 & \text{if } R(e, x, 0) = 1; \\ 0 & \text{otherwise.} \end{cases}$$

$$\varphi_e(x) = 1 \Rightarrow R(e, x, 0) = 1 \Rightarrow \Phi_e(x) = 0 \Rightarrow \varphi_e(x) = 0 \ (\Rightarrow\Leftarrow)$$
$$\varphi_e(x) = 0 \Rightarrow R(e, x, 0) = 0 \Rightarrow \Phi_e(x) > 0 \Rightarrow \varphi_e(x) > 0 \ (\Rightarrow\Leftarrow)$$

Now let $\Phi_i = \lambda x[0]$, for all i. Then axiom two is satisfied, but not axiom one, as witnessed by any partial, not total, recursive function.

Exercise 3.6: Show that there is a recursive function f such that, for any program i, $\varphi_{f(i)} = \Phi_i$.

One of the ramifications of VLSI is that it is now possible to implement a process on a chip and build a machine that executes that process very inexpensively. The analog in the theory is that there are complexity measures where some programs don't cost anything. Suppose $\Phi_0, \Phi_1, \cdots$ is a complexity measure for $\varphi_0, \varphi_1, \cdots$ an acceptable programming system. Suppose f is a recursive function and $\varphi_j = f$. Now define:

$$\Psi_i(x) = \begin{cases} \Phi_i(x) & \text{if } i \neq j; \\ 0 & \text{otherwise.} \end{cases}$$

Then $\Psi_0, \Psi_1, \cdots$ is also a complexity measure for $\varphi_0, \varphi_1, \cdots$.

Exercise 3.7: Which of the following are abstract complexity measures? Justify your answers.

a) The number of compositions used to define a partial recursive function.
b) The number of increment statements executed by a RAM program.
c) The number of times a Turing machine moves to the left.
d) The number of tape cells visited by a Turing machine.

Exercise 3.8: Suppose $\Phi_0, \Phi_1, \cdots$ is a complexity measure for $\varphi_0, \varphi_1, \cdots$ an acceptable programming system. For each of the following sets, determine if it is recursive, r.e. but not recursive, or not even r.e. Justify your answers.

a) $\{\langle x, y\rangle | (\exists t)\Phi_x(y) \leq t\}$
b) $\{\langle x, t\rangle | (\exists y)\Phi_x(y) \leq t\}$
c) $\{\langle y, t\rangle | (\exists x)\Phi_x(y) \leq t\}$
d) $\{\langle x, y, t\rangle | \Phi_x(y) \leq t\}$
e) $\{\langle x, y\rangle | (\neg\exists t)\Phi_x(y) \leq t\}$
f) $\{\langle x, t\rangle | (\neg\exists y)\Phi_x(y) \leq t\}$
g) $\{\langle y, t\rangle | (\neg\exists x)\Phi_x(y) \leq t\}$
h) $\{\langle x, y, t\rangle | \Phi_x(y) > t\}$

Complexity measures can be very pathological. However, any reasonable measure of complexity that anyone has thought of forms the basis for an abstract measure of complexity. Hence, results that are true for an arbitrary complexity measure will certainly be true for all the measures that we really care about.

Definition 3.9: $\psi_0, \psi_1, \cdots$ is an *r.e. sequence* of partial recursive functions if there is a recursive function f such that for any i, $\psi_i = \varphi_{f(i)}$.

As another example of "measure manipulation" techniques, we present the following.

Proposition 3.10: Let $f_0, f_1, \ldots$ and $g_0, g_1, \ldots$ be r.e. sequences of recursive functions. For any acceptable programming system $\varphi_0, \varphi_1, \cdots$ there is a complexity measure $\Phi_0, \Phi_1, \cdots$ such that for any j there is an i such that $\varphi_i = f_j$ and $\Phi_i = g_j$.

Proof: Suppose the hypothesis. Let $\hat{f}$ be a recursive function such that $(\forall i)\varphi_{\hat{f}(i)} = f_i$. Suppose $\Psi_0, \Psi_1, \cdots$ is a complexity measure for $\varphi_0, \varphi_1, \cdots$. Let p be a padding function. Define F as a monotone increasing recursive function such that for all i, $\varphi_{F(i)} = f_i$:

$$F(0) = \hat{f}(0)$$
$$F(x+1) = p(\hat{f}(x+1), \mu y[p(\hat{f}(x+1), y) > F(x)]).$$

Now define:

$$\Phi_i = \begin{cases} g_j & \text{if } j \text{ is such that } i = F(j), \text{ for some } j; \\ \Psi_i & \text{otherwise.} \end{cases}$$

Exercise 3.11: Show that the list $\Phi_0, \Phi_1, \cdots$ defined above is a complexity measure.

All complexity measures, even the pathological ones, are related by no more than a computable factor. The statement of the next theorem also uses, for the first of many times in this treatment, the standard complexity theoretic phrase "for almost all x" to mean "all but finitely many x." The more concise "$\overset{\infty}{\forall} x$" will be used as an abbreviation for this phrase.

Theorem 3.12: (Recursive relatedness of complexity measures) Suppose $\Phi_0, \Phi_1, \cdots$ is a complexity measure for $\varphi_0, \varphi_1, \cdots$ an acceptable programming system and $\Psi_0, \Psi_1, \cdots$ is a complexity measure for $\psi_0, \psi_1, \cdots$ another acceptable programming system. Let t be a recursive function such that for all i, $\varphi_i = \psi_{t(i)}$. Then there is a recursive function r such that for all i, for almost all x:

$$\Phi_i(x) \leq r(x, \Psi_{t(i)}(x)) \qquad \text{and} \qquad \Psi_{t(i)}(x) \leq r(x, \Phi_i(x)).$$

Moreover, r can be made monotone nondecreasing in its second argument.

Proof: A technique called *maxing* is used. Define a recursive function h, using the second axiom for complexity measures, such that for all i, x and y:

$$h(i,x,y) = \begin{cases} \max\{\Phi_i(x), \Psi_{t(i)}(x)\} & \text{if } \Phi_i(x) = y \text{ or } \Psi_{t(i)}(x) = y; \\ 0 & \text{otherwise.} \end{cases}$$

Define the desired r by:

$$r(x,y) = \max_{i \leq x} \max_{z \leq y} h(i,x,z).$$

Now, $\forall i$, $\forall x \geq i$,

$$r(x, \Psi_{t(i)}(x)) \geq h(i,x,\Psi_{t(i)}(x)) \geq \Phi_i(x)$$
$$r(x, \Phi_i(x)) \geq h(i,x,\Phi_i(x)) \geq \Psi_{t(i)}(x).$$

⊗

As an application of the above theorem we will show that the value computed by any program is bounded above by the complexity of computing that value.

Proposition 3.13: Suppose $\Phi_0, \Phi_1, \cdots$ is a complexity measure on φ_0, φ_1, $\cdots$ an acceptable programming system. Then there is a recursive function b such that for all i and almost all x, $\varphi_i(x) \leq b(x, \Phi_i(x))$.

Proof: Let $\psi_0, \psi_1, \cdots$ be the acceptable programming system formed from $P_0, P_1, \ldots$, the list of all RAM programs. Then the pseudospace functions $S_0, S_1, \ldots$ form a complexity measure for $\psi_0, \psi_1, \cdots$. By the isomorphism theorem (Theorem 2.39) there is a recursive, one-to-one, and onto function t such that for all i, $\psi_i = \varphi_{t(i)}$. By the recursive relatedness theorem, there is a recursive function r such that for all i and almost all x,

$$r(x, \Phi_{t(i)}(x)) \geq S_i(x) \geq \psi_i(x) = \varphi_{t(i)}(x).$$

Since t is onto $\mathbb{N}$, let $b = r$. ⊗

Exercise 3.14: Show that the recursive relatedness theorem cannot be improved to make the bounding function r a function of only the complexity in the related system.

Exercise 3.15: Show that the recursive relatedness theorem cannot be improved to make the inequalities hold for all values of x.

Exercise 3.16: Prove or disprove: The function r from the recursive relatedness theorem can be made primitive recursive.

Exercise 3.17: Show that there is no recursive function b such that for all i, for all but finitely many x, $\Phi_i(x) \leq b(x, \varphi_i(x))$.

§3.3 Fundamental Results

The first result that we will examine should come as no surprise to anyone who has graded programs for an introductory programming course. There are arbitrarily bad ways of computing any function, with respect to any complexity measure.

Proposition 3.18: Suppose $\Phi_0, \Phi_1, \cdots$ is a complexity measure for φ_0, φ_1, $\cdots$ an acceptable programming system. For any program i and any recursive function h, there is a program e such that $\varphi_e = \varphi_i$ and for all x in the domain of φ_i, $\Phi_e(x) > h(x)$.

Proof: Suppose the hypothesis. By the recursion theorem there is a program e such that for all x,

$$\varphi_e(x) = \begin{cases} \varphi_i(x) & \text{if } \Phi_e(x) > h(x); \\ \uparrow & \text{otherwise.} \end{cases}$$

Program e, on input x, begins by computing $h(x)$. The second axiom of complexity measures guarantees the comparison between $\Phi_e(x)$ and $h(x)$ can be done effectively. For any x, if $\Phi_e(x) \leq h(x)$ then $\varphi_e(x)$ is undefined, $(\Rightarrow\Leftarrow)$. Hence, $\varphi_e = \varphi_i$ and Φ_e bounds h. ⊗

Exercise 3.19: Suppose f and h are recursive functions. Show that there is an r.e. list of programs $g(0), g(1), \cdots$ such that:

a. $\forall i,\ \varphi_{g(i)} = f$,

b. $\forall x,\ \Phi_{g(0)}(x) > h(x)$, and

c. $\forall i \forall x,\ \Phi_{g(i+1)}(x) > h(\Phi_{g(i)}(x))$.

Next, we will show that there are arbitrarily difficult to compute functions. In other words, no matter how much of some computational resource you have available for use, there will be some functions that you will not be able to compute. Such functions will be called *arbitrarily difficult* or *arbitrarily complex*. The function that we will construct will have range $\{0,1\}$. This means that the function is difficult to compute, not because the answer is difficult to output, but because it is intrinsically difficult to figure out what the answer is.

The proof below uses three fundamental proof techniques: *finite extension*, *diagonalization* and *cancellation*. The idea of a finite extension argument is to construct a function a (finite) piece at a time. Suppose we want to construct some function f. Starting with the nowhere defined finite function, a procedure (called a *stage*) is executed that determines the value of f on some initial segment of arguments. Suppose $f(0), f(1), \cdots, f(n)$ are determined in this manner. Invoking the procedure again will extend the initial segment of arguments on which f has been defined. The second procedure call may define, say, $f(n+1), \cdots, f(m)$, for some $m > n$. In this way, the value of $f(x)$, for any x, can be determined by executing enough stages, in order, for x to be included in the domain of f. In the argument below, each stage defines a function f on a single point, i.e., $f(x)$ is defined at stage x.

The reason that a particular function f is being constructed in the first place is that we want this f to have certain properties. In the argument below, we want f to have the property that any program computing it is particularly complex. One way to make sure that the constructed f has the desired properties is to make f different from all the functions that don't have the property. For the argument below, we want to make f different from any function that is computed by a fast program. Suppose we discover, in the course of the construction, that φ_i does not have the property that we want our f to have. Then we will define $f(x) \neq \varphi_i(x)$, for some x. In doing so, we will have *diagonalized* against program i on argument x. The term "diagonalization" is an historical artifact. The first diagonalization argument constructed a function f that was different from each of $\psi_0, \psi_1, \cdots$ by making $f(x) \neq \psi_x(x)$. Consider an infinite table with one column for each function and one row for each argument:

	ψ_0	ψ_1	ψ_2	ψ_3	ψ_4	$\cdots$
0	$\psi_0(0)$	$\psi_1(0)$	$\psi_2(0)$	$\psi_3(0)$	$\psi_4(0)$	
1	$\psi_0(1)$	$\psi_1(1)$	$\psi_2(1)$	$\psi_3(1)$	$\psi_4(1)$	
2	$\psi_0(2)$	$\psi_1(2)$	$\psi_2(2)$	$\psi_3(2)$	$\psi_4(2)$	
3	$\psi_0(3)$	$\psi_1(3)$	$\psi_2(3)$	$\psi_3(3)$	$\psi_4(3)$	
$\vdots$						

The "diagonalization" then is to traverse the diagonal of the above table, selecting in turn a value $\psi_x(x)$, and making $f(x)$ different from that value.

In other diagonalization arguments, like the one below, the diagonalization against a program i may not occur at stage i. So we must have some way of keeping track of which programs we have already diagonalized against, and which ones we have yet to consider. This is where cancellation comes in. Once program i is diagonalized against, we will cancel it. Canceled programs are never considered later for future diagonalizations.

Theorem 3.20: Suppose $\Phi_0, \Phi_1, \cdots$ is a complexity measure for $\varphi_0, \varphi_1, \cdots$ an acceptable programming system. For all recursive functions h there is a $\{0,1\}$ valued recursive function f such that for all programs i, if $\varphi_i = f$ then $\Phi_i(x) \geq h(x)$, for almost all x.

Proof: The desired f is constructed via diagonalization and cancellation in effective stages of finite extension below. Let h be as in the hypothesis. Execute the following stages for $x = 0, 1, \cdots$.

> *Stage x.* Let i be the least *uncancelled* program such that $i \leq x$ and $\Phi_i(x) < h(x)$. If there is no such i, set $f(x) = 0$. Otherwise, set $f(x) = 1 \dot{-} \varphi_i(x)$ and cancel program i. Go to stage $x + 1$.
> *End stage x.*

The comparison between $\Phi_i(x)$ and $h(x)$ can be done effectively, due to the second axiom of complexity measures. Each stage performs this test for finitely many values of i. Consequently, each stage finishes and f is recursive. Furthermore, by the construction, $f(x) \leq 1$ for each x. Hence, f is a recursive $\{0,1\}$ valued function.

Suppose by way of contradiction that $\varphi_i = f$ and $\Phi_i(x) < h(x)$ for infinitely many x's. Choose the least stage x such that $x \geq i$, $\Phi_i(x) < h(x)$ and all the programs $j < i$ that are ever canceled are canceled before stage x. Choosing x such that $\Phi_i(x) < h(x)$ is no problem as we have assumed that there are infinitely many such x's. Almost all of them will be larger

than i. Some of the programs will be canceled in the construction of f, and some will not. It would take a solution to the halting problem to sort the $j < i$ into two sets: the eventually canceled and the forever uncancelled. However, since there are only finitely many $j < i$ that ever get canceled, these cancellations will take place before some stage. The choice of a stage x such that all the programs $j < i$ that are ever canceled are canceled before stage x is *noneffective*. However, we are assured that such a stage does *exist*. For that stage x, program i will be the target of our diagonalization, and i will be canceled. Hence, $f(x)$ will be made not equal to $\varphi_i(x)$ at stage x, a contradiction. ⊗

Exercise 3.21: Prove that there are infinitely many different functions f satisfying the above theorem.

Exercise 3.22: For recursive functions f and g, a function f is g *sparse* if, for any x, if $f(x) \neq 0$ then $f(y) = 0$ for $x < y \leq x + g(x)$. Prove that, for any recursive function g, there are arbitrarily complex, g sparse, $\{0,1\}$ valued recursive functions.

Definition 3.23: For recursive functions f and g, f is a *finite variant* of g if $f(x) = g(x)$ for all but finitely many x.

Exercise 3.24: Prove that there are recursive functions so complex that even their finite variants are arbitrarily complex.

Exercise 3.25: Prove that there are $\{0,1\}$ valued recursive functions so complex that even their finite variants are arbitrarily complex.

§3.4 Complexity Gaps

Let t be a recursive function. Consider the class of functions computable using some resource bounded by t, i.e., $\{f \mid f$ is recursive and there exists an i such that $\varphi_i = f$ and for almost all x, $\Phi_i(x) \leq t(x)\}$. Now suppose that g is some very large recursive function. We will show that the class of functions computable with resource bound $g \circ t$ is not necessarily larger than the class of functions computable with resource bound t. If the two classes are the same, then there is a "gap" in the complexity between t and $g \circ t$, as no new functions become computable if the resource available is expanded from t to $g \circ t$. Arbitrarily large gaps, represented by the function g, do not start at arbitrary points in the complexity spectrum. For the familiar measures of complexity there is some evidence indicating that the gaps only start occuring above resource bounds that are considered high. The theorem we will prove below starts with an arbitrary gap size, represented by the recursive function g, and produces a recursive function t, that is at the beginning of a size g gap.

A concrete example may illustrate this point better. Suppose that we are interested in measuring the memory usage of programs. Then the

bounding function t would be a constant function, where the constant was the memory size of the machine that we were using. Now suppose that we have ordered another block of memory, expanding the machine's storage capacity to $g \circ t$. Here, $g = \lambda x[x + y]$, where y is the amount of memory just added to the machine. The theorem below tells us that the newly enhanced machine may not be capable of computing any functions that weren't computable with the old configuration. Concisely, the moral of the following result is that adding resources to a machine is no guarantee that you have enhanced its computational power.

Theorem 3.26: (The Gap Theorem) Suppose $\Phi_0, \Phi_1, \cdots$ is a complexity measure for $\varphi_0, \varphi_1, \cdots$ an acceptable programming system. Suppose g is a recursive function such that for all x and y, $g(x, y) > y$. Then, from g one can effectively find a recursive function t such that for all i and x if $t(x) < \Phi_i(x) < g(x, t(x))$ then $x \leq i$.

Proof: The intuition behind the proof is to make t so large that g becomes almost insignificant. Suppose the hypothesis. Clearly, the following function t suffices:

$$t = \lambda x \left[\mu y[(\forall i) y < \Phi_i(x) < g(x, y) \Rightarrow x \leq i]\right].$$

To complete the proof, we must show that t is recursive. Toward this end, rewrite t as:

$$t = \lambda x \left[\mu y[(\forall i < x)\Phi_i(x) \leq y \text{ or } g(x, y) \leq \Phi_i(x)]\right].$$

By the second axiom of complexity measures all the conditions in the above definition of t are effectively testable. Now, it remains to show that for each x, a suitable y exists. Inductively define $y_0 = 0$ and $y_{j+1} = g(x, y_j)$. By the monotonicity of g, for all x, $y_0 < y_1 < y_2 < \ldots$. Furthermore, for all j,

$$y_j < g(x, y_j) = y_{j+1} < g(x, y_{j+1}).$$

Note that the cardinality of $X = \{\Phi_i(x) \mid i < x\} \leq x$. There are $x + 1$ values in the sequence: $y_0, y_1, \ldots, y_{x+1}$. Hence, at least one of the values is different from each of the at most x members of X. In other words, there exists a $j \leq x + 1$ such that for all $i < x$:

$$\Phi_i(x) \leq y_j \text{ or } g(x, y_j) \leq \Phi_i(x).$$

$\otimes$

Note that in the above proof, it turns out that for all x, $t(x) \leq y_{x+1}$. As an illustration of the gap theorem, we will show that there are programs that can run as fast on some slow machine as they can on a fast one. Suppose you have two computers, one very fast and one very slow. Base an acceptable programming system on each of them. For each acceptable programming system define an associated complexity measure to be the total running time of the program under consideration. Let r be the function from the recursive relatedness theorem that relates the two complexity measures. Let t be a total recursive function at the bottom of an r-gap for the slow machine. If a program runs in time bounded by t on the fast machine, then it runs in time $r \circ t$ on the slow one. But any program that runs in time $r \circ t$ on the slow machine actually runs in time t for all sufficiently large inputs on the slow machine.

Exercise 3.27: Show that the function t in the gap theorem can be made arbitrarily large. More specifically, let g be any recursive function with $y < g(x,y)$, for all x and y. Let b be any recursive function. Show that there exists a recursive function t such that $b(x) \leq t(x)$ for all x and if $t(x) < \Phi_i(x) < g(x,t(x))$ then $x \leq i$.

§3.5 Complexity Compression

In this section we will construct an r.e. sequence of programs for partial recursive functions such that each program in the sequence is optimal (with respect to a recursive factor) among all programs computing the same function. Each of the programs in the sequence has its complexity "compressed" between its known lower bound and some recursive factor of that bound.

Theorem 3.28: (The Compression Theorem) Suppose $\Phi_0, \Phi_1, \cdots$ is a complexity measure for $\varphi_0, \varphi_1, \cdots$ an acceptable programming system. Then there are recursive functions f and g such that the following all hold:

1. $\forall i\, \forall x, \Phi_i(x) \downarrow \Rightarrow \varphi_{f(i)}(x) \leq x$
2. $\forall i\, \forall j, \varphi_j = \varphi_{f(i)} \Rightarrow \overset{\infty}{\forall} x, \Phi_j(x) \geq \Phi_i(x)$
3. $\forall i\, \overset{\infty}{\forall} x, \Phi_{f(i)}(x) \leq g(x, \Phi_i(x))$.

Proof: The idea is to make $\varphi_{f(i)}$ different from the result of each program with complexity smaller than Φ_i infinitely often. The set of programs whose complexity is too low is given by:

$$C(i,x) = \{j < x | \Phi_j(x) < \Phi_i(x)\}.$$

Now define:

$$\varphi_{f(i)}(x) = \begin{cases} \mu y[\forall j \in C(i,x), y \neq \varphi_j(x)] & \text{if } \varphi_i(x) \downarrow; \\ \uparrow & \text{otherwise.} \end{cases}$$

The second complexity measure axiom guarantees that if $\varphi_i(x) \downarrow$ then $C(i,x)$ is recursive. The first axiom ensures that if $j \in C(i,x)$ then $\varphi_j(x) \downarrow$. Hence, if $\varphi_i(x) \downarrow$ then $\varphi_{f(i)}(x) \downarrow$. Since the cardinality of $C(i,x)$ is no more than x, $\varphi_{f(i)}(x) \leq x$. f is recursive by the *s-m-n* theorem. Therefore, 1. has been established. To verify 2. consider j such that $\varphi_j = \varphi_{f(i)}$ and $\Phi_j(x) < \Phi_i(x)$ for infinitely many x's. Then for some $x > j$, $j \in C(i,x)$. $\varphi_{f(i)}(x)$ is made to differ from $\varphi_j(x)$ for that x. To prove 3. we start by defining a recursive function h such that for all i, x and y:

$$h(i,x,y) = \begin{cases} \Phi_{f(i)}(x) & \text{if } \Phi_i(x) = y; \\ 0 & \text{otherwise.} \end{cases}$$

The desired g is given by:

$$g(x,y) = \max_{i \leq x} h(i,x,y).$$

If $\varphi_i(x) \downarrow$ and $x \geq i$ then

$$\Phi_{f(i)}(x) = h(i,x,\Phi_i(x)) \leq g(x,\Phi_i(x)).$$

$\otimes$

Suppose f and g are as promised by the above result. Then for all i and almost all x:

$$\Phi_i(x) \leq \Phi_{f(i)}(x) \leq g(x,\Phi_i(x)).$$

Furthermore, Φ_i is a lower bound on the complexity of any program computing $\varphi_{f(i)}$. So program $f(i)$ is optimal for $\varphi_{f(i)}$, within a factor of g. Also, the complexity of program $f(i)$ is "compressed" between Φ_i and $\lambda x[g(x,\Phi_i(x))]$.

Exercise 3.29: Prove a version of the above theorem where the range of $\varphi_{f(i)} \subseteq \{0,1\}$.

§3.6 Speed-up

In this section we will show that there are functions that have no best program. This will be accomplished by constructing a recursive function such that, given any program for that function, we will be able to find a faster one. The intuitive idea is to construct a sequence of programs via diagonalization. All the programs in the sequence do the same diagonalization, except that the i^{th} program in the sequence doesn't consider programs $0, 1, \ldots i-1$ for cancellation. So, the $i+1^{\text{st}}$ program is faster than the i^{th} one since it simulates one less program in its decision as to which program to cancel. The programs not considered for cancellation will affect only finitely many values. Hence, each program in the sequence will differ from the first program (which considers all programs for cancellation) on only finitely many values. A simple patching argument rectifies the situation. Since the patching will alter the program somewhat, we will use recursion to diagonalize against the patched versions of the programs.

Theorem 3.30: (The Speed-up Theorem) Suppose $\Phi_0, \Phi_1, \cdots$ is a complexity measure for $\varphi_0, \varphi_1, \cdots$ an acceptable programming system. Let h be a recursive function that is monotone nondecreasing in its second argument. Then there is a recursive function f such that for any program i, if $\varphi_i = f$ then there is a program j such that $\varphi_j = f$ and for almost all x, $h(x,\Phi_j(x)) \leq \Phi_i(x)$.

Proof: Suppose the hypothesis. Using the operator recursion theorem (Theorem 2.20) we will construct an r.e. sequence of programs named $t(0,0)$, $t(0,1)$, $t(1,0)$, $\cdots$, $t(i,j)$, $\cdots$. $\varphi_{t(0,0)}$ will be our f. The construction below attempts to make $\varphi_{t(0,0)}$ different from any function φ_n such that program $t(n+1,0)$ is not h faster than program n. Program $t(n+1,0)$ will compute basically the same function as program $t(0,0)$. However, $t(n+1,0)$ will be faster than program $t(0,0)$ since it will consider fewer programs for cancellation. The lack of attention program $t(n+1,0)$ pays to some diagonalizations will cause it to compute a function that is different from the one computed by program $t(0,0)$. As we will see, this difference will be small and can be rectified by a suitable patch. Program $t(n,y)$ will compute a patched version of $\varphi_{t(n,0)}$. In Exercise 1.35, a canonical listing d_0, d_1, $\cdots$, of all and only the finite functions was developed. This listing is used below to implement the patching over of one program with a finite set of argument/output pairs. Recall that d_0 is the empty function, e.g., its domain is empty. Program $t(n,0)$ is unpatched. Program $t(n,q)$ will be the program $t(n,0)$ patched with d_q. For each n and q, $\varphi_{t(n,q)}$ is constructed in effective stages of finite extension.

Stage x in the construction of $\varphi_{t(n,q)}$.

Firstly, compute $\varphi_{t(j,p)}(x)$ for $n+1 \leq j \leq x$ and $p \leq x$. These values will be used below. In fact, if all of these computations are convergent, then this stage will converge. If even one of these computations diverges, then this stage will diverge, not continuing past this point. If x is in the domain of d_q, then $\varphi_{t(n,q)}(x) = d_q(x)$. If not, then

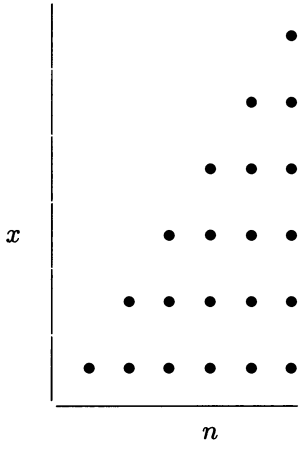
$$\varphi_{t(n,q)}(x) = 1 + \max\{\varphi_i(x) | n \leq i < x \text{ and } i \text{ not yet canceled and } \Phi_i(x) < \max\{h(x, \Phi_{t(i+1,p)}(x)) | p < x\}\}.$$

Notice that, since $\{j | n+1 \leq j \leq x\} = \{i+1 | n \leq i < x\}$, the values needed for the max above to be defined are from precisely the same computations that started this stage. Cancel all i's such that $\varphi_i(x)$ was included in the max above. Go to stage $x+1$.

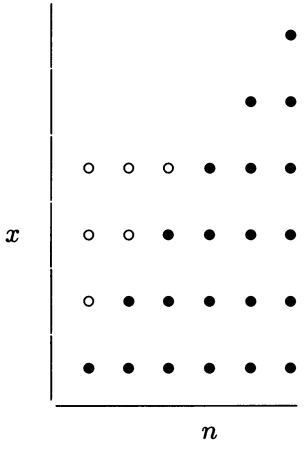
End Stage x.

The hard part of this proof is to show that all the $\varphi_{t(n,q)}$'s are total. Recall that d_0 is the empty finite function. Hence, if $\varphi_{t(i,0)}$ is total, then for any y, $\varphi_{t(i,y)}$ is total.

We will show, by induction on x that for all n, $\varphi_{t(n,0)}(x)$ is defined. Note that if $n \geq x$ then there are no j's such that $n+1 \leq j \leq x$. Hence, $\varphi_{t(n,0)}(x)$ is defined when $n \geq x$. To keep track of for which values of n and x we know that $\varphi_{t(n,0)}(x)$ is defined, we use pictures. In the picture below, and all other pictures in this proof, we use a "•" in row x and column n to indicate that $\varphi_{t(n,0)}(x)$ is defined.



Suppose inductively that for all n and for any $x < x'$, $\varphi_{t(n,0)}(x)$ is defined. By the induction hypothesis, all stages prior to stage x' have completed and the construction reaches stage x'. We use "$\circ$" in our pictures to indicate that in row x and column n, $\varphi_{t(n,0)}(x)$ is defined because of the induction hypothesis. Our picture now looks like:



It remains to show that $\varphi_{t(0,0)}(x)\downarrow$, $\varphi_{t(1,0)}(x)\downarrow$, $\cdots$, $\varphi_{t(x-1,0)}(x)\downarrow$. This is done by a subinduction. Formally, we will show that if

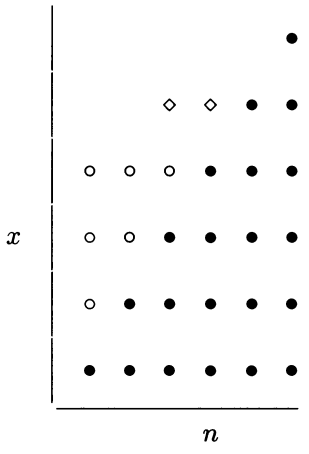
$$\varphi_{t(x,0)}(x)\downarrow, \varphi_{t(x-1,0)}(x)\downarrow, \cdots, \varphi_{t(x-i,0)}(x)\downarrow$$

Then $\varphi_{t(x-i-1,0)}(x)\downarrow$. The subinduction is on i. The base case is when $i = 0$. Note that, since $x \geq x$, $\varphi_{t(x,0)}(x)\downarrow$. Now, $\varphi_{t(x-1,0)}(x)\downarrow$ if $\varphi_{t(j,0)}(x)\downarrow$, for $x \leq j \leq x$. Therefore $\varphi_{t(x-1,0)}(x)\downarrow$.

Suppose subinductively that

$$\varphi_{t(x,0)}(x)\downarrow, \varphi_{t(x-1,0)}(x)\downarrow, \cdots, \varphi_{t(x-i,0)}(x)\downarrow .$$

Using "$\diamond$" in row x and column n to indicate that $\varphi_{t(n,0)}(x)$ is defined because of our subinduction hypothesis, our picture now looks like:



Now, $\varphi_{t(x-i-1),0}(x)\downarrow$ when $\varphi_{t(j,0)}(x)\downarrow$ for $x-i \leq j \leq x$. The convergence of these computations is stipulated by the subinduction hypothesis. Therefore $\varphi_{t(x-i-1,0)}(x)\downarrow$.

This completes the subinduction and the induction. Hence, all the $\varphi_{t(n,0)}$'s are total. By the remarks above about the effect of patching on the $t(n,0)$'s, $\varphi_{t(n,q)}$ is total for each n and q. Now to complete the proof. Let $f = \varphi_{t(0,0)}$. Suppose $\varphi_i = f$. Consider the difference between f and $\varphi_{t(i+1,0)}$. Program $t(i+1,0)$ ignores all the $j \leq i$ when considering which programs to cancel. Only finitely many of the j's less than i ever get canceled during the construction of f. After they are canceled, these programs have no effect on the calculation of f. Let $S = \{j \mid j \leq i$ and j is canceled in the construction of $\varphi_{t(0,0)}\}$. Choose stage s so large that all the programs in S are canceled before stage s. Then $\varphi_{t(i+1,0)}(x) = f(x)$ for all $x \geq s$. Choose p such that the domain of $d_p = \{z < s\}$ and $d_p(z) = \varphi_{t(0,0)}(z)$ for all $z < s$. Then $\varphi_{t(i+1,p)} = f$. Suppose $\Phi_i(x) < h(x, \Phi_{t(i+1,y)}(x))$ for some $x > \max\{i,y\}$. Then $h(x, \Phi_{t(i+1,y)}(x)) \leq \max\{h(x, \Phi_{t(i+1,y)}(x)) \mid y < x\}$. Since $\{h(x, \Phi_{t(i+1,y)}(x)) \mid y < x\} \subseteq \{h(x, \Phi_{t(j,y)}(x)) \mid y < x\}$ for $j < x$ and j not canceled prior to stage s, $\varphi_{t(0,0)}$ would be defined differently from φ_i at stage s ($\Rightarrow\Leftarrow$). $\otimes$

Exercise 3.31: The speed-up theorem shows that there is a function f for which there is no fastest program. Suppose $\varphi_{i_0} = f$. By the speed-up theorem, there is a program i_1 such that $\varphi_{i_1} = f$ and $h(x, \Phi_{i_1}(x)) \leq \Phi_{i_0}(x)$, for all but finitely many x. Similarly, there is a still faster program i_2 for f, etc. Show that the speed-up theorem is noneffective by showing that there does not exist a recursive function g such that $g(i_j) = i_{j+1}$, for all j.

§3.7 Measures of Program Size

So far, we have been concerned with calculating the amount of some resource consumed by various computations. Now we will consider measures of program size. These measures are static in that the size of a program does not vary with its inputs. However, as we will see, there are some relationships between the size and the complexity of programs. Throughout this section, $\varphi_0, \varphi_1, \cdots$ will denote an acceptable programming system and $\Phi_0, \Phi_1, \cdots$ an associated complexity measure.

Definition 3.32: A *size measure* is any recursive function sz such that for any n, $\{\varphi_i | sz(i) = n\}$ is finite.

The following are all measures of program size: the number of characters in a RAM program, the number of quintuples in a Turing machine program, and the index of the program in its associated acceptable programming system. Note that the number of statements in a Pascal program is *not* a measure of program size. By substituting a different natural number for n in the following program schema will produce infinitely many Pascal programs all with four instructions and all computing a different constant function.

```
program constant()
begin
    writeln(n)
end.
```

Size measures as we have defined them have some pathologies. For example, suppose p is a padding function. Define a recursive function p' such that for all i and x:

$$p'(i, x) = p(i, \mu y[p(i, y) > \max\{i, x\}]).$$

Now define s, a size measure, as follows:

$$sz(i) = \begin{cases} p'(j, 0) & \text{if } \exists j, y \text{ such that } \max\{j, y\} \leq i \text{ and } p'(j, y) = i; \\ i & \text{otherwise.} \end{cases}$$

Then, for any i, $p'(i, 0)$, $p'(i, 1)$, $\cdots$ all have the same size. By our definition, this is not a problem because they all compute the same function. We can fix this pathology by demanding that any size measure be such that there are only finitely many *programs* (as opposed to functions) of a given size. In symbols, we say that for all n, $\{i | sz(i) = n\}$ is finite. The important property that makes the proofs below go through is that for any n, there are only finitely many functions computed by programs with size n. Next, we present a very general padding theorem.

Theorem 3.33: (Length Padding) For any measure of program size, there is a recursive function p such that $\forall i$, $\varphi_i = \varphi_{p(i)}$ and $sz(i) < sz(p(i))$.

Proof: Let some measure of program size be given. Let s be the store function of the *s-m-n* theorem. By the parametric recursion theorem there is a recursive function f such that

$$\varphi_{f(x)}(i,y) = \begin{cases} \varphi_i(y) & \text{if } sz(f(x)) > sz(i); \\ x & \text{otherwise.} \end{cases}$$

It cannot be the case that $sz(f(x)) \leq sz(i)$ for all but finitely many x, as then there would be infinitely many constant functions of size less than $sz(i)$. A similar argument shows that $sz(s(f(x),i))$ is greater than $sz(i)$ for some x. Choose x' such that for any $x \geq x'$, $sz(f(x)) > sz(i)$ and $sz(s(f(x),i)) > sz(i)$ The desired p is given by:

$$p(i) = s\left(f\left(\mu z\left[\min\left\{sz(f(z)), sz(s(f(z),i))\right\} > sz(i)\right]\right), i\right).$$

Then, clearly, $sz(p(i)) > sz(i)$ and for all z:

$$\begin{aligned} \varphi_{p(i)}(y) &= \varphi_{s(f(\mu z[\min\{sz(f(z)),sz(s(f(z),i))\}>sz(i)]),i)}(y) \\ &= \varphi_{f(x)}(i,y) \qquad \text{for some } x \geq x' \\ &= \varphi_i(y) \end{aligned}$$

⊗

Exercise 3.34: Prove that for any measure of program size and any recursive function h, there is a recursive function p such that for all i, $\varphi_i = \varphi_{p(i)}$ and $h(sz(i)) < sz(p(i))$.

In contrast to padding techniques, which are useful theoretically, in practice, a *minimal* size program is often sought. A program i is *minimal* iff $\varphi_i \neq \varphi_j$ for all j such that $sz(j) < sz(i)$. The set of minimal size programs is not r.e. Moreover, the following result tells us that the set of minimal size programs is not even close to being effectively enumerable.

Theorem 3.35: Every r.e. subset of the set of minimal size programs contains programs for only finitely many functions.

Proof: Choose an arbitrary size measure. We will prove that any r.e. subset of the set of minimal size programs contains programs of only finitely many different sizes. The theorem will follow. Suppose by way of contradiction that f is a recursive function with an infinite range such that, for all i, $f(i)$ is minimal and the set $\{sz(f(i)) | i \in \mathbb{N}\}$ is infinite. By the recursion theorem there is a program e that is described as follows.

e: (x)

Choose i least such that $sz(f(i)) > sz(e)$.
$y := \varphi_{f(i)}(x)$.

y

The above construction will work, provided such an i always exists. Since f is assumed to enumerate programs of infinitely many different sizes, a suitable i will always be found. The contradiction is that program e is smaller that program $f(i)$. Both programs compute the same function, but $f(i)$ is supposed to be minimal. ⊗

We will continue our study of size measures by considering preprocessors. Preprocessors change the syntax, but not the semantics, of programs. Optimizing compilers are examples of preprocessors. Unfortunately, every preprocessor is very far from being an optimizer.

Definition 3.36: A recursive function f is *a preprocessor* if, $\forall i$, $\varphi_i = \varphi_{f(i)}$.

Theorem 3.37: Suppose $\Phi_0, \Phi_1, \cdots$ is a complexity measure for $\varphi_0, \varphi_1, \cdots$ an acceptable programming system. Let sz be a measure of program size. Suppose f is a preprocessor. Let g and h be recursive functions. For any program i, one can effectively find a program j such that

1. $\varphi_j = \varphi_{f(j)} = \varphi_i$
2. $sz(f(j)) \geq g(sz(i))$ and
3. for all x in the domain of $\varphi_i, \Phi_{f(j)}(x) \geq h(x, \Phi_i(x))$.

Proof: Suppose the hypothesis and let i be given. By the parametric recursion theorem there is a recursive function m such that for all x and y:

$$\varphi_{m(x)}(y) = \begin{cases} \varphi_i(y) & \text{if } sz(f(m(x))) \geq g(sz(i)) \text{ and} \\ & \quad \Phi_{f(m(x))}(y) \geq h(y, \Phi_i(y)) \\ x & \text{if } sz(f(m(x))) < g(sz(i)) \\ \uparrow & \text{if } sz(f(m(x))) \geq g(sz(i)) \text{ and} \\ & \quad \Phi_{f(m(x))}(y) < h(y, \Phi_i(y)). \end{cases}$$

Since, for any x, $\varphi_{f(m(x))} = \varphi_{m(x)}$, there can only be finitely many x's such that $sz(f(m(x))) < g(sz(i))$, as otherwise there would be infinitely many constant functions with size bounded by $g(sz(i))$. This would lead to a contradiction. Hence, the second case of the definition of $\varphi_{m(x)}$ can apply for only finitely many x's. Choose an x such that $sz(f(m(x))) \geq g(sz(i))$. So, the first or third case of the definition of $\varphi_{m(x)}$ must apply. If, for some y, $\varphi_{m(x)}(y) = \uparrow$, then it must be that the third case is being applied and, hence, $\Phi_{f(m(x))}(y) < h(y, \Phi_i(y))$. Consequently, $\varphi_{f(m(x))}(y) \downarrow$ However, if $\varphi_{f(m(x))}(y) \downarrow$ then $\varphi_{m(x)}(y) \downarrow$, $(\Rightarrow\Leftarrow)$. Therefore, for all y, $\Phi_{f(m(x))}(y) \geq h(y, \Phi_i(y))$ and $\varphi_{m(x)}(y) = \varphi_i(y)$. In summary:

$$\begin{aligned} &\varphi_{f(m(x))} = \varphi_{m(x)} = \varphi_i \\ &sz(f(m(x))) \geq g(sz(i)) \text{ and} \\ &\forall y, \Phi_{f(m(x))}(y) \geq h(y, \Phi_i(y)). \end{aligned}$$

The desired j is then $m(x)$. ⊗

Suppose f is the identity function and g and h are very fast growing recursive functions. Then f is a preprocessor and the above theorem tells us that there are arbitrarily poor programs for any partial recursive function. By arbitrarily poor, we mean larger than necessary by a factor of g and more complex (using more resources) than necessary by a factor of h.

We will strengthen this result to show that if we are given any r.e. list of programs that compute recursive functions, then that list will necessarily contain some programs that are excessively slow and overly large. To do so, we will need to argue that applying a recursion theorem does not add much to the complexity of the algorithm being modified. The following theorem will tell us that when a self-referential version of some program is formed, the complexity only increases by a recursive factor. In reasonable, and imaginable, acceptable programming systems, this recursive factor is very small, sometimes as small as a multiplicative constant, or even smaller. Tracing through the proof of the recursion theorem reveals that the extra complexity comes from computing the store function. Such an analysis is not needed for the following:

Theorem 3.38: (Complexity Theoretic Recursion Theorem) Suppose Φ_0, Φ_1, $\cdots$ is a complexity measure for $\varphi_0, \varphi_1, \cdots$ an acceptable programming system. There are recursive functions r and h such that h is monotone nondecreasing in its second argument and for any i:

1. $\forall x, \varphi_{r(i)}(x) = \varphi_i(r(i), x)$ and
2. $\overset{\infty}{\forall} x \in \text{ domain } \varphi_{r(i)}, \Phi_{r(i)}(x) \leq h(x, \Phi_i(r(i), x))$.

Proof: The r is supplied by the fully effective proof of the recursion theorem. Define:

$$k(i, x, y) = \begin{cases} \Phi_{r(i)}(x) & \text{if } \Phi_i(r(i), x) = y; \\ 0 & \text{otherwise.} \end{cases}$$

Notice, that if $\Phi_i(r(i), x) = y$ then $\varphi_i(r(i), x)$ is defined, and consequently, $\varphi_{r(i)}(x)$ is defined. Hence, k is recursive. Let

$$h(x, y) = \max_{i \leq x} \max_{z \leq y} k(i, x, z)$$

The function h is monotone, since on larger arguments the same values as before, and more, are included in the max. Suppose x is in the domain of $\varphi_{r(i)}$ and $x \geq i$. Then,

$$\Phi_{r(i)}(x) = k(i, x, \Phi_i(r(i), x)) \leq h(x, \Phi_i(r(i), x)).$$

$\otimes$

Exercise 3.39: Show that the above theorem can be strengthened to also get:

$$\Phi_i(r(i), x) \leq h(x, \Phi_{r(i)}(x)).$$

Exercise 3.40: Consider the fully effective form of the the fixed point theorem (2.25). Modify its statement to come up with the complexity theoretic fixed point theorem. Prove this new theorem.

The next theorem shows that any effective list of programs that contains some arbitrarily long ones must contain some excessively long ones that could be replaced by a shorter, almost as fast, semantically equivalent program.

Theorem 3.41: For all recursive functions f and g such that $\{sz(f(x))|x \in \mathbb{N}\}$ is infinite, one can effectively find i and j such that for h from the complexity theoretic fixed point theorem:

1. $\varphi_i = \varphi_{f(j)}$
2. $g(sz(i)) \leq sz(f(j))$ and
3. $\overset{\infty}{\forall} x, \Phi_i(x) \leq h(x, \Phi_{f(j)}(x))$.

Proof: Suppose the hypothesis. Define a recursive function k such that:

$$k(y) = f(\mu z[g(sz(y)) \leq sz(f(z))]).$$

k is recursive since f was assumed to enumerate arbitrarily large programs. Note that if $k(y) = f(z)$ then $f(z)$ is the first program in the list which is larger than (or the same size as) $g(sz(y))$. Now, take a fixed point of k, so that $\varphi_i = \varphi_{k(i)}$. Choose $j = \mu z[g(sz(i)) \leq sz(f(z))]$. By the definition of k, $k(i) = f(j)$. So,

$$\varphi_i = \varphi_{k(i)} = \varphi_{f(j)} \text{ and } g(sz(i)) \leq sz(f(j)).$$

By the complexity theoretic fixed point theorem, for almost all x,

$$\Phi_i(x) \leq h(x, \Phi_{k(i)}(x)) = h(x, \Phi_{f(j)}(x)).$$

⊗

§3.8 Restricted Programming Systems

One way to eliminate some of the pathologies that we have found is to restrict the systems we consider. Below we will restrict both our programming system and our size measures.

Definition 3.42: A recursive set *RPS* is a *restricted programming system* if $\{\varphi_i | i \in RPS\}$ is an infinite set of recursive functions.

Note that no RPS can include all the recursive functions. An example RPS is the set of programs computing the primitive recursive functions.

Definition 3.43: A size measure is *canonical* if there is a recursive function b such that if $sz(i) \leq j$ then $i \leq b(j)$.

If a size measure is canonical then $b(j)$ bounds the finite set of programs with size $\leq j$. In canonical size measures, there are only finitely many programs of a given size.

$$\{\text{programs of size } j\} = \{i | i \leq b(j) \text{ and } sz(i) = j\}$$

Most reasonable ways of measuring program size yield canonical measures.

Proposition 3.44: Suppose we are given an RPS with a canonical size measure. Then the minimal size programs in the RPS are r.e.

Proof: We must show that there is a recursive function f such that $\{\varphi_{f(i)} | i \in \mathbb{N}\} = \{\varphi_i | i \in \text{ RPS}\}$ and if for some $k \in$ RPS, $\varphi_k = \varphi_{f(i)}$, then $sz(k) \geq sz(f(i))$. Suppose the hypothesis. Now define a partial recursive function ψ that on input x diverges if x is not in the RPS and otherwise searches for the least z such that for all y in the RPS with $sz(y) < sz(x)$ there is a $w \leq z$ such that $\varphi_x(w) \neq \varphi_y(w)$. If x is in the domain of ψ then x is in the RPS and, furthermore, φ_x is different from all programs in the RPS with size smaller than the size of x. The canonical nature of the size measure guarantees that all the suitable y's can be found, making ψ partial recursive. Find a recursive function f with range the same as the domain of ψ. ⊗

So, we get some improvement (less pathology) by looking at an RPS. However, every RPS has some excessively long programs.

Theorem 3.45: Suppose we are given an RPS with a canonical size measure. Let h be from the complexity theoretic fixed point theorem. For all recursive functions g we can effectively find i and j such that:

1. $i \notin \text{RPS}, j \in \text{RPS}, \varphi_i = \varphi_j$
2. $sz(j) \geq g(sz(i))$
3. $\forall k, k \in \text{RPS}$ and $\varphi_k = \varphi_i \Rightarrow sz(k) \geq sz(j)$ and
4. $\overset{\infty}{\forall} x, \Phi_i(x) \leq h(x, \Phi_j(x))$.

Proof: Use f from the proof of 3.44 and the g given in the statement of the theorem in 3.41 to find the desired i and j. ⊗

Exercise 3.46: Show that for any acceptable programming system φ_0, φ_1, $\cdots$ there is a total recursive function p such that for all j, if φ_j is a total $\{0,1\}$ valued function such that for some i, $\varphi_j(i) = 1$ and $(\varphi_j(x) = 1$ implies $\varphi_x = \varphi_i)$, then $\varphi_{p(j)} = \varphi_i$ and $\varphi_j(p(j)) = 0$.

Exercise 3.47: Suppose ψ_0, ψ_1, $\cdots$ is also an acceptable programming system and that t is the recursive function witnessing the isomorphism between them. Suppose S_φ and S_ψ are canonical size measures for the programming systems. Prove that the size of programs in the two systems are recursively related, e.g. prove that there is a recursive function g such that for all i

a) $S_\varphi(i) \leq g(S_\psi(t(i)))$, and
b) $S_\psi(i) \leq g(S_\varphi(t^{-1}(i)))$.

§3.9 Historical Notes

The pseudospace measure was introduced in [M&Y]. Abstract complexity measures were introduced in [Bl1], a paper that also contains results on arbitrarily difficult functions, speed-up and compression. A machine dependent version of the result on arbitrarily difficult functions appeared in [Rab]. The gap theorem first appeared in [Bor]. The proof of the gap theorem that appears in this work is adapted from [You]. The proof of the speed-up theorem is based on a suggestion in [Cas]. The material on program size measure is largely from [Bl2].

[Bor] A. Borodin, Computational complexity and the existence of complexity gaps, *Journal of the ACM*, Vol. 19, 1972, pp. 158–174.

[Bl1] M. Blum, A machine-independent theory of the complexity of recursive functions, *Journal of the ACM*, Vol. 14, 1967, pp. 322–336.

[Bl2] M. Blum, On the size of machines, *Information and Control*, Vol. 11, 1967, pp. 257–265.

[Cas] J. Case, Operator speed-up for universal machines, *IEEE Transactions on Computers*, to appear, and personal notes.

[M&Y] M. Machtey and P. Young, An Introduction to the General Theory of Algorithms, North Holland, New York, 1978.

[Rab] M. Rabin, Degree of difficulty of computing a function, Hebrew University Technical Report 2, 1960.

[You] P. Young, Easy constructions in complexity theory: gap and speed-up theorems, *Proceedings of the AMS*, Vol. 37, 1973, pp. 555–563.

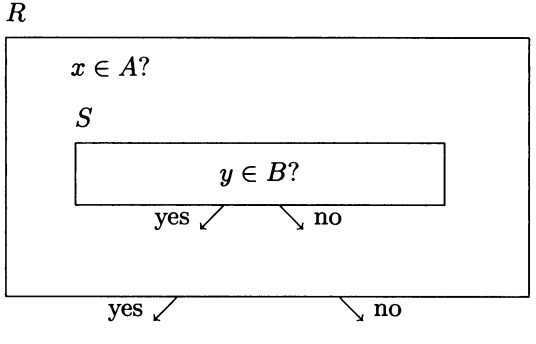
4

Complete Problems

In our study of unsolvable problems, we associated a set with each problem. In essence, we discussed only problems of the form "does x have a certain property." By coding programs onto $\mathbb{N}$, many natural problems in computer science can be represented as sets. From our study of abstract complexity we know that there are sets (problems) for which deciding membership is arbitrarily difficult. A problem is "complete" for some class if it is the "hardest" problem in the class. If you have some class of sets and a complete problem for the set, then that complete problem embodies all that is difficult about *any* problem in the class. For example, we will show that K is complete for r.e. sets.

§4.1 Reducibilities

Earlier, in Theorem 2.41, we showed that $\{x \mid \varphi_x$ is a constant function$\}$ was not recursive by a reduction from the halting problem (K). Now, we will formalize the notion of reduction. As a preliminary step, we give an informal definition. For two sets A and B, we say that A *is algorithmically reducible* from B if there is an algorithm R that, given a subroutine to decide membership in B, can decide membership in A.



Even though the above picture depicts an algorithm that makes only *one* call to the subroutine for deciding membership in B, finitely many such calls are admissible. The reduction is a "polynomial time reduction" if R runs in polynomial time (complexity bounded by some polynomial of the length of the input) exclusive of the time spent in S. Note that if only polynomially many calls to S are made and S runs in a polynomial amount of time then so does R. Hence, if we have a lower bound for deciding membership in A, then we may also have one for deciding membership in B (within a polynomial factor).

First we will take a simplistic view and say that recursive sets are easy and all the other sets are not easy. This is simplistic because we know that there are arbitrarily difficult to decide recursive sets (Theorem 3.20). Later, we will redraw the line between easy sets and hard sets by tightening the notion of reducibility. We proceed by considering reductions where only a single question may be asked about membership in the auxiliary set.

Definition 4.1: A is *many-one reducible* from B (written: $A \leq_m B$) if there is a recursive function f such that $x \in A$ iff $f(x) \in B$.

In Theorem 2.41 we really showed that for any y and z:

$$K \leq_m \{x | \varphi_x \text{ is a constant function}\}$$
$$K \leq_m \{x | \exists z, \varphi_x(z) = y\}$$
$$K \leq_m \{x | \varphi_x(y) = z\}.$$

It is easy to see that $\leq_m$ is transitive and reflexive. Also, if $A \leq_m B$ and A is undecidable then B is undecidable. Some examples follow.

Proposition 4.2: Let $E = \{\langle x, y\rangle | \varphi_x = \varphi_y\}$. Then $K \leq_m E$.

Proof: The idea is to use E to solve K. We start by defining a program k:

$$\varphi_k(x, y) = \begin{cases} 1 & \text{if } x \in K \\ \uparrow & \text{otherwise.} \end{cases}$$

Notice, that if $x \in K$, then (and only then) $\varphi_k = \lambda x, y[1]$. Let program l be such that $\varphi_l = \lambda x[1]$. So programs l and $s(k, x)$ will compute the same function iff $x \in K$. Now, applying the *s-m-n* theorem, let $f(x) = \langle s(k, x), l\rangle$.

$$x \in K \Rightarrow \varphi_{s(k,x)} = \lambda x[1] = \varphi_l \Rightarrow f(x) \in E$$
$$x \notin K \Rightarrow \varphi_{s(k,x)} = \lambda x[\uparrow] \neq \varphi_l \Rightarrow f(x) \notin E.$$

⊗

The following proposition shows that $\leq_m$ relation is not symmetric.

Proposition 4.3: $E \not\leq_m K$

Proof: Suppose by way of contradiction that f is a recursive function such that for all x, y, $\langle x, y\rangle \in E$ iff $f(x, y) \in K$. Let $\varphi_{i_0} = \lambda x[0]$ and $\varphi_{i_1} = \lambda x[\uparrow]$. Define a recursive function g, by implicit use of the *s-m-n* theorem, such that for all i,

$$\varphi_{g(i)}(z) = \begin{cases} 0 & \text{if } i \in K; \\ \uparrow & \text{otherwise.} \end{cases}$$

Since $\varphi_{g(x)} = \varphi_{i_0}$ or $\varphi_{g(x)} = \varphi_{i_1}$, one of $f(i_0, g(x))$ or $f(i_1, g(x))$ must appear (eventually) in the enumeration of K. If $f(i_0, g(x))$ appears in K then $x \in K$. If $f(i_1, g(x))$ appears in K then $x \notin K$. Hence, K is recursive, $(\Rightarrow\Leftarrow)$. ⊗

Definition 4.4: Let $\mathcal{C}$ be a collection of sets. A set A is *m-hard for* $\mathcal{C}$ if $\forall B \in \mathcal{C}$, $B \leq_m A$. A is *m-complete* for $\mathcal{C}$ if $A \in \mathcal{C}$ and A is m-hard for $\mathcal{C}$.

Proposition 4.5: K is m-complete for the r.e. sets.

Proof: K is r.e. so it remains to show that K is m-hard for the r.e. sets. Suppose A is an r.e. set. Choose i such that domain $\varphi_i = A$. By implicit use of the *s-m-n* theorem there is a recursive function f such that :

$$\varphi_{f(x)}(y) = \begin{cases} 1 & \text{if } \varphi_i(x)\downarrow; \\ \uparrow & \text{otherwise.} \end{cases}$$

$x \in A \Rightarrow \varphi_{f(x)}$ is total $\Rightarrow \varphi_{f(x)}(f(x))\downarrow \Rightarrow f(x) \in K$
$x \notin A \Rightarrow$ domain $\varphi_{f(x)} = \emptyset \Rightarrow f(x) \notin K$.

Hence, $x \in A$ iff $f(x) \in K$. ⊗

Exercise 4.6: Show that $K \leq_m \{x | W_x \text{ is finite }\}$. (Recall that W_x is the domain of φ_x.)

Exercise 4.7: Show that $\{x | W_x \text{ is infinite }\} \leq_m \{x | \varphi_x \text{ is total }\}$.

Exercise 4.8: Show that $\{x | W_x \text{ is finite }\} \leq_m \overline{\{x | \varphi_x \text{ is total }\}}$.

Exercise 4.9: Show that $\{x | \varphi_x \text{ is total }\} \leq_m \{x | W_x \text{ is infinite }\}$.

Exercise 4.10: Show that $E \leq_m \{x | W_x \text{ is infinite }\}$.

Exercise 4.11: Show that $\{x | W_x \text{ is infinite }\} \leq_m E$.

Exercise 4.12: Show that $E \leq_m \{x | W_x \text{ is infinite }\}$.

Exercise 4.13: Show that $\{x | W_x \text{ is infinite }\} \leq_m E$.

Exercise 4.14: Show that $\{x | W_x \neq \emptyset\}$ is m-complete for the r.e. sets.

There are other kinds of reducibilities that give rise to notions of completeness. Consider, for example, modifying the definition of $\leq_m$ so as to require that the function f be one-to-one. The result of this modified definition would be what is known as $\leq_1$ reducibility. The notion of 1-completeness is defined analogously.

Exercise 4.15: Formally state the definition of $\leq_1$ reducibility and 1-completeness.

Exercise 4.16: Show that $A \leq_1 \{x | W_x \cap A \neq \emptyset\}$.

Exercise 4.17: Show that K is 1-complete.

Exercise 4.18: Show that $K_0 \leq_1 K$.

Exercise 4.19: Which of the above reductions can be strengthened from an $\leq_m$ reduction to an $\leq_1$ reduction?

§4.2 Polynomial Computability

There has been considerable interest recently in the class of functions that can be computed in a polynomial amount of time (or sometimes space). There are a number of reasons. Firstly, each of the translations from one programming system to another that we studied in the beginning of the course were not only effective, but can be accomplished within a polynomial amount of time where the polynomial is a function of the length of the input. For the sake of concreteness, consider the translation from RAM programs to Turing machines. That translation, given a RAM program, will produce, in a polynomial amount of time (and space), a Turing machine computing the same function as the original RAM program. Furthermore, if the RAM program had its time complexity bounded by a polynomial, then so will the Turing machine that results from the translation. Note, however, that the polynomial bounding the time complexity of the Turing machine will most likely be larger than the bounding function for the RAM program. Similar remarks can also be made for space complexity.

The point of the discussion above is that if some function is computable in polynomial time in one model of computation, then it is computable in a polynomial amount of time in any model that we have studied and, most likely, in any model anybody will ever care about. Hence, the class P of functions computable in a polynomial amount of time can be defined without reference to a particular model of computation. In fact, we just did so. Similarly, *P-space* is the class of functions computable with space bounded by a polynomial. In all cases, the polynomial is a function of the size of the input. We will denote the *size* of an input x by $|x|$. Since integers are commonly represented in binary on computers, it is common to consider $|x|$ to be $\log x$.

Another major reason why P has been so extensively studied is that the class of polynomials is closed under composition. This adds a certain amount of mathematical tractability to the study of P. Finally, it is believed that P is large enough to include all we will ever be able to reasonably compute anyway. Certainly polynomials can express large enough run times to satisfy even the most patient. Who is going to wait around for even a super computer to execute $|x|^{1,000,000,000,000,000,000,000}$ ($|x|$ to the 1 sextillion) instructions when it is given input x? On the other hand, there are exponential time algorithms in use today for computations with relatively small inputs. For example, consider **grep** and its derivatives that are in the collection of programs that come with UNIX.

Definition 4.20: A is *polynomial time reducible* from B (written: $A \leq_m^p B$) if there is a recursive function f such that $x \in A$ iff $f(x) \in B$ and f can be computed in time bounded by a polynomial in $|x|$.

Note that, by the properties of polynomials, $\leq_m^p$ is reflexive and transitive. Also, if $A \leq_m^p B$ and "$x \in B$?" can be decided in polynomial time, then "$x \in A$?" can be decided in polynomial time. We say that a set A is p-hard if $B \leq_m^p A$, for any $B \in P$. Furthermore, A is p-complete if A is p-hard and $A \in P$. Often, P is called the *polytime* computable functions. Sets and functions are purposely confused by saying that a set A is in P if its characteristic function is.

Proposition 4.21: For all sets A, if $A \in P$, $A \neq \emptyset$ and $A \neq \mathbb{N}$ then A is p-complete for P.

Proof: Suppose the hypothesis. Let $a_0 \in A$ and $a_1 \notin A$. Suppose $B \in P$. Let g be the witness, i.e., $g \in P$ and g is the characteristic function for B. The poly-time function reducing B to A, on input x, first runs g (in polytime). Then, if $x \in B$ the procedure outputs a_0, otherwise a_1 is output. ⊗

Consequently, to study structure within P, one needs a finer reducibility notion, such as that of linear time or logarithmic space reductions.

Exercise 4.22: Let $\mathcal{C}$ be any time bounded class of sets containing P that is closed under the operation of composition with a polynomial. Show that if $B \in \mathcal{C}$, $A \leq_m^p B$, and A is p-hard for $\mathcal{C}$, then both A and B are p-complete for $\mathcal{C}$.

Exercise 4.23: Is there a recursive function f such that $P = \{\varphi_{f(i)} | i \in \mathbb{N}\}$? Justify your answer.

§4.3 The Deterministic Time Hierarchy

In this section we will show that there is a hierarchy of larger and larger sets of computable functions based on an upper bound on compute time. This will not contradict the gap theorem (Theorem 3.26) since the bounding functions will be chosen carefully so as to guarantee that no two functions that mark consecutive levels of the hierarchy will fall in the same gap. The impact of this result is most profound. It means that we can characterize functions based on how much time they take to compute in the best case. The practice of classifying functions in this way is widespread. Much of the research in theoretical computer science has been devoted to classifying various computational tasks according to their time complexity, e.g., their position in the time hierarchy. The most popular version of the hierarchy theorem involves Turing machines with some extra bells and whistles.

Definition 4.24: A *multitape* Turing machine is an algorithmic device that behaves just like a Turing machine except that it has more than one tape, each with its own read/write head.

In the operation of a Turing machine, there will be several data areas. For example, the universal Turing machine has one area for data and one for the program to be simulated. If both the data areas are on the same tape, then the Turing machine must move its read/write head back and forth between the areas, usually doing nothing useful in between. Multiple tapes allow for computations to proceed without having to execute any of these useless moves.

Definition 4.25: Suppose T is a recursive function. DTIME(T) is the class of all sets whose membership question can be decided by a multi-tape Turing machine that executes a sum total of no more than $T(n)$ quintuples on inputs of length n.

Theorem 4.26: (Constant Speed-up) Any Turing machine can be sped up by any constant factor.

Proof: (Sketch) Suppose M is a Turing machine that we want to speed up by a factor of n. M', the faster Turing machine, works just like M except that every one of M''s tape symbols is an encoding of some n-tuple of M's tape symbols. If Σ is M's tape alphabet, then M''s tape alphabet will have size $|\Sigma|^n$. So, M', essentially in one step, simulates n of M's steps.

Since each one of M''s symbols encodes a block of M's symbols, creation of M''s transition function is bit complicated. Fortunately, the complexity of M''s computation is not affected by how long it takes to come up with M''s program. To figure out what M' does with one of its symbols, we will have to simulate M on the group of symbols represented by it.

Suppose M entered some state q while moving one symbol to the right. The case of a left move is analogous. Suppose further that this move crossed a boundary from one block of symbols (coded as one symbol in M') to the next adjacent block. Then M' must also change state. The state that M' changes to must encode not only q, but the fact that at this point in the simulation, M is looking at the leftmost symbol of the block.

Part of the figuring out what M''s transitions are is to consider what happens when M enters a block from the left. One of four things will happen. Firstly, M could halt. In this case M' also halts, after rewriting the single symbol that encodes the current block. Secondly, M could leave the block moving right. To do so, M would have had to execute at least n steps (the length of the block). In this case, M', in one step, rewrites the entire block as a single symbol, based on what M would have done before leaving the block. The third possible outcome is for M to leave the block moving left. Again, M' can simulate the action of M on the block in a single step. Finally, M could go into an infinite loop, never leaving the block. Since there are only n symbols, chosen from a finite alphabet, in any block and there are only finitely many states of M, if M goes into an infinte loop without leaving some block, a configuration will be repeated. M' can detect the repeated configuration in its simulation of M. ⊗

As a consequence of Theorem 4.26 it appears that constant factors do not hold much influence when trying to characterize a collection of functions by a bound on the time it takes to compute them. To neatly handle situations where constant factors obstruct the view of deeper, foundational issues we will use the notion of the order of a function.

Exercise 4.27: Fill in missing details in the proof of Theorem 4.26.

Exercise 4.28: What is the complexity of transforming M into M' in the proof of Theorem 4.26?

Exercise 4.29: Show how to simulate a two-tape Turing machine that runs in time T by a one tape Turing machine in time T^2.

Definition 4.30: For total functions f and g we say that f is of *order* g if there are constants c and x' such that for any $x > x'$, $f(x) \leq c \cdot g(x)$. If f is of order g we write $f = O(g)$.

Intuitively, if T_1 bounds T_2 by a large enough margin then DTIME(T_2) is strictly included in DTIME(T_1). By definition, if $T_2(n) < T_1(n)$, for all n, then DTIME(T_2) is included in DTIME(T_1). In order to obtain a set in DTIME(T_1) that is not in DTIME(T_2) we will need to simulate a Turing machine for $T_2(n)$ steps and diagonalize. Consequently, T_2 must be "well behaved" if we are to complete the simulation *and* the diagonalization within time T_1. A notion of "well behaved" that works is given in the following.

Definition 4.31: A recursive function T is *time constructible* if there is a Turing machine that, on inputs of length n, runs for exactly $T(n)$ steps.

Exercise 4.32: Prove that there is a recursive function f such that f is *not* time constructible.

Exercise 4.33: Prove that for any recursive function r there is a time constructible function r' such that $r'(x) \geq r(x)$, for all x.

Exercise 4.34: Is the set of time constructible functions r.e.? Justify your answer.

A similar notion of constructibility holds for space bounds as well:

Definition 4.35: A recursive function S is *space constructible* if there is a Turing machine that, on inputs of length n, visits exactly $S(n)$ tape cells.

Exercise 4.36: Prove that there is a recursive function f such that f is *not* space constructible.

Proposition 4.37: For any time constructible function T, there is a list of Turing machines, $M_0, M_1, \cdots$ containing all and only the Turing machines that decide membership in sets that are members of DTIME(T).

Proof: Suppose T is a time constructible function. Let M be a Turing machine that executes exactly $T(n)$ steps on inputs of length n. Let $N_0, N_1, \ldots$ be a list of all and only the multitape Turing machines. Machine M_i, on input x, simultaneously simulates $N_i(x)$ and $M(x)$. If M halts before N_i then x is not of member of the set being determined. If N_i halts before M, then N_i determines whether or not x is in the set. M_i may take twice (or at most thrice) as long as allowed, but we can apply Theorem 4.26. ⊗

In order to simulate the machines described above, we will need a subroutine for a universal Turing machine that is efficient.

Theorem 4.38: Suppose T is a time constructible function and $M_0, M_1, \cdots$ is the list of Turing machines from Theorem 4.37. Then there is a two-tape Turing machine U such that for all i and x, $U(i, x) = M_i(x)$ and U runs in time $T(|x|)\log(T(|x|))$ times a constant dependent on i.

Proof: Machine U is given inputs i and x. The interpretation of i is that it is the encoding of some k tape Turing machine M. For notational simplicity, we will present the proof of the $k = 1$ case only. The decoding of i, e.g., a description of the quintuples of M, is placed on the scratch tape. Each tape of M will be simulated by multiple tracks on the first tape of U. For example, two tapes can be simulated using two tracks on a single tape by expanding the alphabet of tape symbols as follows. If the first tape to be simulated has an "a" in position one, and the second tape has a "b" in position one, then the single tape will have the symbol "$\frac{a}{b}$" in position one. Here, the "a" is on the first *track*, and "b" is on the second. The second tape of U will be used for scratch space to copy data to and from the first tape. U will simulate each step of M by a series of steps we will refer to as an A-move (for *augmented* move). After each A-move, the cells of M that would be under the tape heads of M will all be under the tape head of U. Thus instead of marking where the tape heads of M would be, we move the data in such a way that U's head is always over the current cell of each tape when an A-move is completed.

We now describe an A-move. The simulation of one tape by two tracks will be described, leaving the generalization to $2k$ tracks to the reader. The tracks have their cells divided into blocks as follows. Block B_0 is the *home* block. It consists of one cell from each of the two tracks aligned vertically. The upper track contains a special marker so that U can always find the home block. This marker symbol is never replaced or copied elsewhere during the simulation. The lower track contains the data that would be under the head of the tape being simulated. To the right of the home block are blocks numbered B_1, B_2, $\cdots$, B_i, $\cdots$. Each block B_i is of size 2^{i-1} tape cells, containing, perhaps, twice that number of symbols distributed on the two tracks. To the left of the home block, the blocks are numbered B_{-1}, B_{-2}, $\cdots$, B_{-i}, $\cdots$. Again, these blocks are of size 2^{i-1}. At the beginning of the simulation, all the data are on the lower track in the same order as on M's tape, and the lower track of the home block contains the data that

would be under the head of Turing machine M. At any time during the simulation, the contents of M's tape will be distributed, in some fashion, over the blocks of U's primary tape. To read the data off of U's tape, in the order that they would be appearing on M's tape, start with the leftmost block that contains any nonblank symbols. There are three cases for the identity of the leftmost empty block:

B_{-j}: Read the nonblank symbols from the lower track, left to right, then the symbols on the upper track, left to right. If $j > 1$, repeat this procedure for block $B_{-(j-1)}$. If $j = 1$, continue with the next case.

B_0: Read the symbol in the lower track and proceed with the next case, for $j = 1$.

B_j: Read the nonblank symbols from the upper track, left to right, then the symbols on the lower track, left to right. If block B_{j+1} contains some nonblank symbols, then repeat this procedure for block B_{j+1}, otherwise stop.

To make the mathematics work out smoothly, we will want to make sure that if any track of any block has a nonblank symbol in it, then it is completely filled. The initial configuration of all of M's data on the lower track of U's tape may not conform to this convention. To get around this minor difficulty, we will have U distinguish two types of "blank" symbols, M-blanks, and regular blanks. Before any data are placed on U's tape it contains only the home block marker and the rest of the tape is filled with regular blanks. If the input data do not quite fill the lower track of some block, then it is filled with M-blanks since, if more symbols were to be taken from M, they would be blank. Before and after each A-move the following *invariant* will be true:

> For all $i > 0$, either B_i is full and B_{-i} is empty *or* B_i is empty and B_{-i} is full *or* both B_i and B_{-i} have their lower tracks full and their upper tracks empty.

To simulate a move of M's head, U must move the data on the appropriate two tracks. A move of M's head to the left means U must move its data to the right. We now describe such an A-move (a move in the other direction is symmetric).

First we move the head of U to the right from B_0 until we find the first block that is not full, copying all the data to the second tape of U as we go. Suppose B_j is the first block encountered that is not full. If B_j is empty, its lower track will be used, otherwise the upper track will be employed. Copy all the data from the second tape to the lower tracks of $B_j, \cdots, B_1$. (If the lower track of B_j already had data, we used the upper track for that particular block.) Note that the data exactly fit, since the total amount of data from the upper and lower tracks of B_0 to B_{j-1} is

$2^j - 1$. The larger half gets deposited in the cells of B_j, while the smaller portion is distributed across the lower tracks of blocks B_1 through B_{j-1}.

As a consequence of the above A-move, block B_0 can now be used for other purposes. Move U's head to block B_{-j}, using the second tape as a counter. The previous A-move transferred $2^j - 1$ symbols from the second tape to the first, and the symbols are still on the tape, so we may assume that an end marker was included. Since B_j was not full when we began our A-move, it must be the case that B_{-j} is not empty. Furthermore, all of the blocks between B_{-j} and B_0 must be empty, since B_j was the first block to the right of the home block that was not full.

If both tracks of B_{-j} are full, move the upper track to the second tape. If only the lower track is full, move that. Then distribute the 2^{j-1} symbols now on the second tape across the lower tracks of the blocks between B_{-j} and B_0, including B_0. Note that the data that before were logically to the left of the data in B_0 are now themselves in B_0. Using the third case of the rule to determine the leftmost empty block to read the data in the tracks, the data are still in order and readable as the tape of M. Note that the data exactly fit on the lower tracks of blocks B_{-j} to B_0. The redistribution of the data guarantees that invariant has not been violated.

The proof is completed by comparing the complexity of U's A-move to a single step of M. Call an A-move that finds B_j to be the first block that is not full an *A_j-move*.

The first observation is that an A_j-move takes time linear in the size of block B_j. It requires a single pass in both directions of the home symbol. In the worst case, we could require a couple of passes for separate tapes of M with heads moving in different directions, but the A_j-move could still be completed in time proportional to the size of B_j. Hence, the time taken for an A_j-move is $O(2^{j-1})$.

An A_j-move is performed at most once per 2^{j-1} moves of M. Since half of the data in B_1 through B_{j-1} get put in B_j, another 2^{j-1} A-moves would be required before the worst case amount of data could build up in B_1 through B_{j-1}.

An A_j-move cannot be performed until M has made at least 2^{j-1} moves. Again, it would take that long for the data to build up in B_1 to B_{j-1}.

Using the above facts, we can place a bound on the moves of U. Note that M was assumed to operate in time $T(x)$, for T a time constructible function. Next we calculate the largest j such that an A_j move is made.

$$\begin{aligned} 2^{j-1} &\leq T(|x|) \\ \log 2^{j-1} &\leq \log T(|x|) \\ j - 1 &\leq \log T(|x|) \\ j &\leq \log(T(|x|) + 1 \end{aligned}$$

By the above, an A_j-move takes $O(2^j)$ steps. If M makes $T(|x|)$ moves on input x, then U makes at most, say $T'(|x|)$ moves where, for some constant m,

$$\begin{aligned} T'(|x|) &= \sum_{i=1}^{\log(T(|x|))+1} m \cdot \overbrace{2^i}^{\text{the } A_i \text{ move}} \cdot (T(|x|)/ \overbrace{2^{i-1}}^{\text{rate of occurence}}) \\ &= 2 \cdot m \cdot T(|x|) \cdot \log[T(|x|)] + 1 \\ &< 2 \cdot m \cdot T(|x|) \cdot (\log T(|x|) + \log T(|x|)) \\ &< 4 \cdot m \cdot T(|x|) \cdot \log T(|x|) \end{aligned}$$

And since we can speed up any Turing machine by a constant factor, we can speed up U by a factor of $4m$ to get a universal Turing machine that runs in at most $T(|x|)\log(T(|x|))$ steps. ⊗

Exercise 4.39: Prove a version of the Theorem 4.38 for a single-tape Turing machine U and time bound $T(|x|)^2$.

Theorem 4.40: (Time Hierarchy Theorem) If T_1 and T_2 are time constructible functions such that

$$\lim_{n\to\infty} \frac{T_2(n)\log T_2(n)}{T_1(n)} = 0$$

then DTIME(T_2)⊂DTIME(T_1).

Proof: The logarithmic factor comes into play because of the model we are using. Similar proofs for other models do not need the logarithmic factor. Suppose the hypothesis. First we will show that DTIME(T_2) ⊆ DTIME(T_1). By the limit condition there exists an $n_0 \in \mathbb{N}$ such that for any $n > n_0$:

$$\begin{aligned} \frac{T_2(n)\log T_2(n)}{T_1(n)} &< 1 \\ T_2(n)\log T_2(n) &< T_1(n) \\ T_2(n) &< T_1(n). \end{aligned}$$

Notice that we have also just shown that $T_2(n)\log T_2(n) = O(T_1(n))$. Let M_0, M_1, ... be the list of DTIME(T_2) Turing machines constructed in Proposition 4.37. If $S \in$DTIME(T_2) then there is a machine M_i that decides membership in S and runs in time bounded by $T_2(n)$ on inputs of length n. In forming a witness that $S \in$DTIME(T_1), care must be taken since it is not necessarily true that $T_1(n) \geq T_2(n)$ for all n. An A_i algorithm that arbitrates membership in S using at most T_1 time behaves as follows:

A_i: (x)

On input x, if $|x| \leq n_0$ then look the answer up in a finite table, placing the value in y.
Otherwise, run $M_i(x)$ and copy its answer to y.

y

By replacing M_i with A_i in the list above, we have that DTIME(T_2) $\subseteq$DTIME(T_1). To complete the proof, it suffices to construct a set $S \in$ DTIME(T_1) $-$ DTIME(T_2). Let U be the efficient universal Turing machine of Theorem 4.38. Note that, since T_1 is time constructible, there is a process that will stop in precisely $T_1(|x|)$ steps. Hence, we can effectively run $U(x,x)$ for precisely $T_1(|x|)$ steps. Let S be the set determined as follows:

DECIDE.S: (x)

Run $U(x,x)$ for $T_1(|x|)$ steps.
If $U(x,x)$ halts and $U(x,x) = 0$
then $y = 1$
else $y = 0$

y

By Theorem 4.38, and the time constructibility of T_1, it follows that $S \in$DTIME(T_1). Suppose by way of contradiction that S is a member of DTIME(T_2). Choose a witness i such that M_i is the characteristic function of S, M_i runs in time bounded by T_2 and $T_2(x) \leq T_1(x)$, for all $x > i$. The membership of $S \in$DTIME(T_2) and padding guarantee that such an i exists. If $i \in S$, then $M_i(i) = 1$, in which case $i \notin S$. If $i \notin S$ then $M_i(i) = 0$, in which case $i \in S$. ($\Rightarrow\Leftarrow$) $\otimes$

Exercise 4.41: Explain why Theorem 4.40 does not contradict Theorem 3.26.

Exercise 4.42: Consider the following pair of functions:

$$T_1(n) = \begin{cases} n^2 & \text{if } n \text{ is even} \\ n & \text{if } n \text{ is odd} \end{cases}$$

$$T_2(n) = \begin{cases} n^2 & \text{if } n \text{ is odd} \\ n & \text{if } n \text{ is even} \end{cases}$$

Prove that DTIME(T_1) and DTIME(T_2) are incomparable, e.g., DTIME(T_1) − DTIME(T_2)$\neq \emptyset$ and DTIME(T_2) − DTIME(T_1)$\neq \emptyset$. Can you think of other, more natural examples of pairs of complexity classes that have this property?

Exercise 4.43: Show that $P \subset$DTIME($n^{\log n}$).

Exercise 4.44: Let $A = \{0^n | n \text{ is prime } \}$. Let T be any time constructible function. Construct a recursive set B which is a subset of A and is NOT in DTIME(T). Justify your answer.

Exercise 4.45: Given any infinite recursive set A and a total recursive function T, show that there exists a recursive set B such that $B \subset A$ and $B \notin DTIME(T(n))$ (There are no conditions on A except that it be infinite and recursive.)

Exercise 4.46: Let T_1 and T_2 be the functions:

$$T_1(n) = \begin{cases} n^4 & \text{if } n \text{ is odd} \\ n^2 & \text{if } n \text{ is even} \end{cases}$$

$$T_2(n) = \begin{cases} n^2 & \text{if } n \text{ is odd} \\ n^4 & \text{if } n \text{ is even} \end{cases}$$

Show that there exist a set S with the property that $S \in$DTIME(T_1) but $S \notin$ DTIME(T_2).

§4.4 Nondeterminism

One of the consequences of Theorem 4.40 is that our patience will always limit what we can compute. As machines get faster and faster, we will be able to compute more and more functions, modulo the gaps from Theorem 3.26, in a day's worth of computer time. The nature of science and technology indicates that there will always be some program that someone wants to run that takes too much time with the current computers. Hence, the search for faster and faster computers will continue. There are some physical limits as to how fast data can be moved and manipulated. This limit puts constraints on how fast computers as we know them can go. Consequently, the search for faster computers has led to investigations of massively parallel computers and techniques to squeeze a little more power out of traditional designs.

One technique to circumvent the difficulty of computing functions not known to be in P that has received considerable attention is *nondeterminism*. Nondeterminism can be introduced easily into most models of computation. For Turing machines, all that one has to do is to allow as legal programs that have multiple quintuples with the same *state, symbol* prefix. For RAM programs the JMP instruction is augmented to have more than one next possible instruction. For the purposes of using a particular

machine model as an example, we will use Turing machines. When a Turing machine, in its execution of a nondeterministic program, runs across an instantaneous description such that more than one quintuple applies, one is chosen, nondeterministically, and the computation proceeds. The computation structure then becomes tree-like, with the branch points corresponding to the choices made and the leaves corresponding to possible outputs.

A first observation is that, for Turing machines, allowing nondeterminism does not enlarge the class of functions computed. Consider the following outline of a deterministic simulation of a nondeterministic Turing machine. The program and data are presented to the simulator just as the universal Turing machine receives its inputs. The computation proceeds just as in the universal Turing machine until an instantaneous description with more than one applicable quintuple arises. Then, the simulator copies the instantaneous description and applies the different relevant quintuples to the copies. Now the simulator has two instantaneous descriptions to contend with. In general, the number of instantaneous descriptions will continue to grow. If *any* instantaneous description yields a terminating configuration, then the simulation stops after erasing everything except the final result of the instantaneous description that halted.

There are a variety of ways to make sense of the nondeterministic computation of functions. To avoid the possibility of multiple answers, we can consider all nondeterministic Turing machines as computing characteristic functions. In this situation, any *positive* result is interpreted as meaning that the input is in the set. Hence, all possible results must be 0 in order to conclude that a number is not a member. Alternatively, we may suppose that a nondeterministic computation tree is such that there is a unique result, but some paths may be infinite, returning no value.

Even though nondeterminism does not enlarge the class of computable functions, it may enable us to compute more functions within a certain time bound. To investigate this issue, we need some definitions. NP is the class of sets that are decidable nondeterministically by some program in polynomial time. In other words, if $S \in NP$ then there is some nondeterministic program that, on input x, if all the right choices are made at the nondeterministic points, returns a value larger than 0 in time bounded by a polynomial in $|x|$. As was the case with P, NP is invariant across computation models. In addition to studying time bounded resource classes, we will also study complexity classes where membership is determined by space usage. *NP-Space* has an analogous definition. Nondeterminism does not expand the class of functions computable. Does it allow some functions to be computed faster in less space? To do so, we must compare P, NP, *P-Space* and *NP-Space*. Unfortunately, we do not know how to make all the comparisons in a mathematically rigorous manner.

In our notation, what is commonly called an "NP-complete problem" is just a problem that is complete for the class NP under the $\leq_m^p$ reduction. By viewing a deterministic program as a nondeterministic one without any

choices we have that $P \subseteq NP$. Whether or not the containment is proper has been a major open problem in theoretical computer science for over two decades now. Part of the interest in the question comes from the staggering collection of problems that have been shown to be NP-complete. This collection includes most of the computational problems in operations research and artificial intelligence. Since the results on nondeterministic space classes are clearer, we start our formal study of nondeterminism with some results concerning space.

Exercise 4.47: Let A and B be sets in P. Show that $A \cup B$ and $A \cap B$ are in P.

Exercise 4.48: Let A and B be sets in NP. Show that $A \cup B$ and $A \cap B$ are in NP.

Just as we previously defined the deterministic time bounded classes DTIME(T) for T a recursive function, we can, and do, define similar space bounded complexity classes.

Definition 4.49: Suppose S is a recursive function. DSPACE(S) is the class of all sets whose membership question can be decided by a Turing machine that visits at most $S(n)$ tape cells on inputs of length n. If the Turing machine used is *nondeterministic*, then we have the class NSPACE(S).

Exercise 4.50: State precisely the definition of a class that we would call NTIME(T).

Exercise 4.51: Give two different definitions of what one might mean by computing a function f in NTIME(T).

Exercise 4.52: Show that a set A is in NP iff there exists a polynomial p and a polynomial predicate R such that

$$A = \{x | (\exists y)[|y| \leq p(|x|) \text{ and } R(x, y)]\}.$$

Exercise 4.53: Let NNP be the set of all sets A such that there exist polynomials p and q, and a polymomial predicate R such that

$$A = \{x | (\exists y)[|y| \leq p(|x|) \text{ and } \forall z[|z| \leq q(|x|) \rightarrow R(x, y, z)]]\}.$$

Show that $P = NP$ iff $P = NNP$.

Exercise 4.54: Let T be a recursive function. Show that NTIME(T) $\subseteq$ DSPACE(T^2).

Our first result shows that deterministic simulations of nondeterministic computations can be accomplished with a relatively small amount of extra space. As a corollary of this result, we will be able to show that everything that is computable nondeterministically within a polynomial amount of space can also be computed deterministically within a polynomial amount of space.

Theorem 4.55: If $\lambda n[\log n] = O(S)$ then NSPACE$(S) \subseteq$DSPACE(S^2).

Proof: Suppose that f is computed by a nondeterministic Turing machine M in space bounded above by $S(n)$ for inputs of size n. Suppose also that $\lambda n[\log n] = O(S)$. This ensures that the given space bound is sufficient to store the input. Suppose without loss of generality that S is space constructible. We will informally describe a deterministic Turing machine M' computing f in space on the order of $S^2(n)$. Let c be the sum of the number of tape symbols of M and the number of states of M. Then there are at most $c^{S(n)+1}$ unique instantaneous descriptions containing at most $S(n)$ used tape cells. If any of M's terminating computations has more than $c^{S(n)+1}$ steps, then some instantaneous description is repeated and M has a computation with the same result in at most $c^{S(n)+1}$ steps. The deterministic simulator M' of M will be an implementation of the following recursive algorithm *TEST*, which on inputs D_1, D_2, and i returns *true* iff there is a computation of at most i steps that takes M from instantaneous description D_1 to instantaneous description D_2. The deterministic simulation will look for a computation from the initial instantaneous description to a halting one in at most $c^{S(n)+1}$ steps.

TEST: (D_1, D_2, i)

If $i = 1$
then *RESULT* is true iff $D_1 = D_2$ or M reaches instantaneous description D_2 in one step from instantaneous description D_1
else *RESULT* is true iff there exists an instantaneous description D_3 of size $\leq S(n) + 1$ such that
$TEST(D_1, D_3, \lceil i/2 \rceil)$ and
$TEST(D_3, D_2, \lfloor i/2 \rfloor)$

RESULT

M' will use a standard stack implementation of TEST. Note that on each recursive call to TEST the value of i is essentially halved. So the number of stack frames that must be remembered at any time in the simulation is at most $1 + \lceil \log c^{S(n)+1} \rceil$, which is of the order $S(n)$. The size of each stack frame is $S(n) + 1 + k_1$ where k_1 is a constant denoting the space used to hold the value i. There is another constant k_2 that represents the space taken in addition to the stack frames to run the simulation. Hence the total amount of space used is:

$$k_2 + \left(1 + \left\lceil \log c^{S(n)+1} \right\rceil\right)(S(n) + 1 + k_1).$$

This latter term is of the order $S^2(n)$, so M' deterministically computes f in roughly the square of the space used by M. ⊗

We can now state an important result about nondeterministic space bounded complexity classes.

Theorem 4.56: *P-Space* = *NP-Space*.

Proof: Clearly, *P-Space* $\subseteq$ *NP-Space*. The converse follows immediately from Theorem 4.55. ⊗

Exercise 4.57: Show that NSPACE(S) $\subseteq$ DTIME(2^S), for any space constructible function S such that $\lambda n[\log n] = O(S)$.

Recall that the r.e. sets are not closed under complementation. We showed that if $\overline{K}$ were, r.e. then K would be recursive, leading to a contradiction. Given the nature of nondeterministic computation, we cannot expect that the complement of a set in NP is also in NP. The above discussion also holds when any of the time or space bounded complexity classes is substituted for NP.

Definition 4.58: Let C denote some time or space bounded class. Then co-C denotes the set of complements of the set in C.

Theorem 4.59: Suppose S is any space constructible recursive function such that $S(|n|) \geq \log|\, n|$, for all n. Then, NSPACE(S)=co-NSPACE(S).

Proof: Suppose S is any recursive function such that $S(|n|) \geq \log|\, n|$. Let M be a nondeterministic Turing machine given input n. Let C be the initial configuration. One way to include the state information in the configuration is to place a new symbol, representing the current state, just before the symbol to be read next on the tape of the Turing machine. Thus, a configuration conveys tape contents, machine state and read head position. Consequently, we assume that C appears as the symbol representing the initial state followed by the symbols of n. As long as S is at least logarithmic, the size of $C \leq S(|n|)$. If S is less than logarithmic, then $\log|\, n|$ bits are needed to encode the position of the read head in the configuration. Hence the lower bound on S in the statement of the theorem.

First we count the exact number of configurations that are reachable from C using at most order of $S(|n|)$ space. The calculation of this value will also be done in space bounded by $S(|n|)$. Let r_i denote the number of configurations reachable in at most i computation steps from C using at most $S(|n|)$ space. Clearly, $r_0 = 1$. We show how to calculate r_{i+1} from r_i. The total number of possible qualifying configurations is bounded above by $k^{S(|n|)}$, for some constant k, so the sequence $r_0, r_1, \cdots$ reaches a limit. The total number of configurations that we seek then is r_i where i is chosen least such that $r_i = r_{i+1}$.

Suppose inductively that r_i is given. Initialize a counter *cnt* to 0. A string of symbols is a *potential configuration* of M if it contains only one

instance of a symbol representing a state and all the other symbols of the string are from the tape alphabet of M. Every potential configuration of size bounded by $S(|n|)$ is called a *target*. These targets are considered one at a time in lexicographical order. We want to test if the current target is a configuration that is reachable in $i+1$ computation steps. The number of targets representing one of our desired configurations that we have found so far is kept in *cnt*. For each target, consider each potential configuration of size bounded by $S(|n|)$ one at a time in lexicographical order. For each target, another counter, *cntconfig* is initialized to 0. The new counter will keep track of how many configurations reachable in i computation steps we have considered so far for the current target. For each potential configuration, check to see of it is one of the r_i configuration that is reachable in i steps. If so, perform two actions:

(1) Increment *cntconfig*, and
(2) Check if this potential configuration is the same as the current target, or if the target is reachable from the current configuration in one computation step. If so, then the current target is another configuration that is reachable in $i+1$ steps.

If the current target has just been found to be reachable in $i+1$ steps, then increment *cnt*, and proceed to the next target. If *cntconfig* $= r_i$, then all the configurations reachable in i steps have been considered for this target, so proceed to the next target without incrementing *cnt*. The default is to consider the next potential configuration for the current target. Notice that each configuration reachable in i steps is generated again for each target. It would be more efficient with respect to time to calculate these configurations once and remember them. Unfortunately, this cannot be done in $O(S(|n|))$ space. When all targets have been checked in this manner, *cnt* contains the value r_{i+1}.

Let $R = r_i$ where i is chosen least such that $r_i = r_{i+1}$. Then R is the total number of configurations reachable by M from C using at most space $S(|n|)$. We complete the proof by showing how to test, in space $S(|n|)$, if M rejects its input. Again we call each potential configuration of size $S(|n|)$ a target and consider them one at a time in lexicographical order. For each target, nondeterministically guess a computation path of M from C to the current target. If unsuccessful in guessing the computation path, go on to the next target. Otherwise, increment *cnt* and test if the target is an accepting configuration of M. If it is, then the input cannot be in the complement of the inputs accepted by M. If the counter reaches the value R and we have not found any accepting computations of M then we know that M does not accept the input. ⊗

Exercise 4.60: Prove that it is not possible to store the R reachable configurations in $O(S(|n|))$ space.

Notice that in the above proof, the same space is used over and over again. Since time cannot be reused in the same manner, the proof above does not carry over to case of time bounded complexity classes. The same is true for trying to adapt the proof of Theorem 4.55 to the case of complexity classes determined by a function bounding execution time. For nondeterministic time bounded classes, we only have results concerning completeness.

Exercise 4.61: Show that there exists a recursive set A such that $A \notin \text{DSPACE}(S(n))$.

Exercise 4.62: Show that $\text{NSPACE}(\log n) \subseteq P$.

§4.5 An *NP*-Complete Problem

Once an *NP*-Complete problem is found, reduction techniques can be used to show that other problems are also *NP*-Complete. Currently, there are thousands of problems known to be *NP*-Complete. We will show that a problem called *CNFSAT* is *NP*-Complete by translating every computation of a RAM program into an instance of this problem. After defining the problem *CNFSAT* we will introduce a simple, yet computationally complete, nondeterministic RAM model of computation used in the proof that follows.

Definition 4.63: A *Boolean expression* is composed of variables that range over *true*, *false* connected by the logical operators $\wedge$ (*and*) $\vee$ (*or*) and $\neg$ (*not*). The last connective is a unary one, the others are binary.

For example, $p_1 \vee (p_2 \wedge \neg p_1)$ is a Boolean expression. Such expressions are not unique in form. $p_1 \vee p_2$ is a Boolean expression that is equivalent to the first example. Often, a Boolean expression will contain a subexpression within pairs of parenthesis. Such subexpressions are called *clauses*. A clause is *conjunctive* if it contains only the operators $\wedge$ and $\neg$. Similarly, a clause is *disjunctive* if it contains only the operators $\vee$ and $\neg$.

Definition 4.64: A Boolean expression is *satisfiable* iff there is an assignment of values to the variables of the expression that makes the expression evaluate to *true*.

The above example is satisfiable as witnessed by the assignment: ($p_1 = \textit{false}$, $p_2 = \textit{true}$).

Definition 4.65: A Boolean expression is in *conjunctive normal form* if it is a conjunction of disjunctive clauses.

In other words, an expression in conjunctive form looks like $C_1 \wedge C_2 \wedge \ldots \wedge C_n$ where $C_1, C_2, \ldots, C_n$ are disjunctive clauses.

Definition 4.66: *CNFSAT* is the problem of deciding whether or not a given Boolean expression in conjunctive normal form is satisfiable.

A Boolean expression is in *disjunctive normal form* if it is the disjunction of a number of conjunctive clauses. It is easy to determine if an arbitrary formula in disjunctive normal form is satisfiable. In fact, the task can be done in time proportional to the length of the formula. The best known algorithm for converting from conjunctive normal form to disjunctive normal form takes time exponential in the length of the formula.

Proposition 4.67: *CNFSAT*$\in$ *NP*.

Proof: We describe the computation tree of a nondeterministic polynomial time decision procedure. The computation tree has one level for each variable. At each level, a nondeterministic choice is made determining the truth assignment of the variable associated with the level. At the bottom level, a complete assignment has been determined, in essentially the same number of steps as the number of variables. One can decide if the chosen assignment satisfies the formula in time proportional to the length of the formula. If the expression is satisfiable then the associated branch of the computation converges with a positive answer, otherwise it diverges. If the expression is satisfiable, and the satisfying branch is chosen at every choice, then the above procedure will terminate in a polynomial amount of time. $\otimes$

To show that *CNFSAT* is *NP*-Complete, we will transform every nondeterministic RAM computation into a Boolean expression in conjunctive normal form that is satisfiable iff the computation halts within a polynomial time bound. To do so, we must develop a notion of a nondeterministic RAM program. This will be essentially the model of RAM's over natural numbers that we coded up earlier with two exceptions. First, we will assume a unique labeling so there will be no jump above's versus jump below's, only jumps. Furthermore, in addition to a jump on condition type of instruction there will be an unconditional jump to one of (perhaps several) possible destinations.

Definition 4.68: A *nondeterministic* RAM program is a RAM program in the usual sense that is composed of the following 5 types of instructions:

`INC R`j
`DEC R`j
`IF R`$j = 0$ `JMP L`
`JMP (L1, L2, ..., L`n`)`
`CONTINUE`

Computations on a nondeterministic RAM are defined in a manner analogous to the way computations on a regular (deterministic) RAM were defined in Definition 1.3.

Theorem 4.69: *CNFSAT* is *NP*-Complete.

Proof: Suppose S is a set in *NP*. Then there is a nondeterministic RAM program P, with n instructions, that decides membership in S in time bounded by some polynomial p. If some computation of P, on input x, takes more than $p(|x|)$ steps, then that computation will not indicate membership of x in S. If all possible computations of P on input x take more than $p(|x|)$ steps, then this indicates that $x \notin S$. To show that $S \leq_m^p CNFSAT$ it suffices to write down a Boolean expression, B_x, in conjunctive normal form, such that B_x is satisfiable if and only if P on input x has a valid halting computation of length at most $p(|x|)$ steps. Provided B_x can be produced in a polynomial in x number of steps, we will have $S \leq_m^p CNFSAT$. This part of the proof involves a lot of notation, but is not conceptually dificult.

On input x, if P halts in $p(|x|)$ steps, then the largest possible value stored in any register during the computation is $p(|x|) + x = m$. Furthermore, P will only use finitely many registers, say $r + 1$ of them. Each of the instructions of P is one of five types: `INC`, `DEC`, conditional `JMP`, `CONTINUE`, and nondeterministic `JMP`. These considerations lead us to define, for a fixed x, the following Boolean constants and variables. These are presented along with their intuitive meanings in the expression we construct modeling the computation of P on input x. First, the constants that will guarantee that the program described by the expression we construct is precisely P.

INST1(i) for $1 \leq i \leq n$ is *true* iff the i^{th} instruction of P is an `INC` instruction.

INST2(i) for $1 \leq i \leq n$ is *true* iff the i^{th} instruction of P is a `DEC`

INST3(i) for $1 \leq i \leq n$ is *true* iff the i^{th} instruction of P is a conditional `JMP` instruction.

INST4(i) for $1 \leq i \leq n$ is *true* iff the i^{th} instruction of P is a `CONTINUE` instruction.

INST5(i) for $1 \leq i \leq n$ is *true* iff the i^{th} instruction of P is a nondeterministic `JMP` instruction.

USE(i, j) for $1 \leq i \leq n$, $0 \leq j \leq r$ is *true* iff the i^{th} instruction of P uses register `R`j. Notice that only the `INC`, `DEC` and conditional `JMP` instructions use any registers.

Next are the variables that will be used to guarantee that the Boolean expression we construct describes a valid computation of P.

REG(j, t, v) for $0 \leq j \leq r$, $0 \leq t \leq p(|x|)$, $0 \leq v \leq m$ is *true* iff the value of `R`j after t steps of the computation is v.

INSTRUCTION(t, i) for $0 \leq t \leq p(|x|)$, $1 \leq i \leq n$ is *true* iff the instruction of P executed at time t is the i^{th} instruction of P.

Note that there are on the order of m^2 variables defined above. Before proceeding to write down the formula B_x, we will introduce some convenient

abbreviations. "$\wedge_{0\leq i\leq k} x_i$" abbreviates "$x_0 \wedge x_1 \wedge \ldots \wedge x_k$." Similarly for the $\vee$ operator. $JUSTONE(x_1, \ldots, x_k)$ is an abbreviation for:

$$(\vee_{1\leq i\leq k} x_i) \wedge (\wedge_{1\leq i<j\leq k} (\neg x_i \vee \neg x_j)) .$$

Note that $JUSTONE(x_1, \ldots, x_k)$ is a statement in conjunctive normal form that is true just in case exactly one of $x_1 \ldots, x_k$ is true. The logical statement $C \wedge D \Rightarrow E$ is equivalent to $\neg C \vee \neg D \vee E$ which is in conjunctive normal form. The expression B_x is presented below in pieces, with each section preceded by an informal interpretation of its intended meaning. The idea is to have each of the pieces of the formula model some aspects of the computation. A weakness of the proof is that there is no formal way to prove that all aspects of the computation have been accounted for. Trying to account for all neccessary components of the computation can be very tricky. For example, we do not need to have B_x make sure that every instruction is of only one type and uses at most one register since $USE(i, j)$, $INST1(i)$, $\cdots$, $INST5(i)$ are all constants based on P which we are assuming is a well-formed RAM program.

At any step of the computation there can be at most one instruction being executed:

$$\wedge_{0\leq t\leq p(|x|)} \; JUSTONE(INSTRUCTION(t, 1), \ldots, INSTRUCTION(t, n)) \qquad (i)$$

Every register holds only one value at any given time:

$$\wedge_{0\leq t\leq p(|x|)} \wedge_{0\leq j\leq r} JUSTONE(REG(j, t, 0), \ldots, REG(j, t, m)) \qquad (ii)$$

If register Ri isn't referenced in the instruction executed at time t, then Ri won't change value:

$$\wedge_{1\leq i\leq n} \wedge_{0\leq t<p(|x|)} \wedge_{0\leq j\leq r} \wedge_{0\leq v\leq m} \; INSTRUCTION(t, i) \wedge \neg USE(i, j) \wedge REG(j, t, v) \Rightarrow REG(j, t + 1, v) \qquad (iii)$$

Conditional jump statements don't alter the contents of registers:

$$\wedge_{1\leq i\leq n} \wedge_{0\leq t<p(|x|)} \wedge_{0\leq j\leq r} \wedge_{0\leq v\leq m} \; INSTRUCTION(t, i) \wedge INST3(i) \wedge REG(j, t, v) \Rightarrow REG(j, t + 1, v) \qquad (iv)$$

The computation starts with the first instruction with the input x in R0, and the rest of the registers containing 0:

$$INSTRUCTION(0,1) \wedge REG(0,0,x) \wedge_{1 \le j \le r} REG(j,0,0) \qquad (v)$$

The computation terminates:

$$\vee_{0 \le t \le p(|x|)} INSTRUCTION(t,n) \qquad (vi)$$

Every instance of an INC instruction is performed correctly:

$$\wedge_{0 \le t < p(|x|)} \wedge_{0 \le v < m} \wedge_{1 \le i \le n} \wedge_{0 \le j \le r} INSTRUCTION(t,i) \wedge USE(i,j) \wedge REG(j,t,v) \wedge INST1(i) \Rightarrow REG(j,t+1,v+1) \qquad (vii)$$

Every instance of an DEC instruction is performed correctly:

$$\wedge_{0 \le t < p(|x|)} \wedge_{0 \le v \le m} \wedge_{1 \le i \le n} \wedge_{0 \le j \le r} INSTRUCTION(t,i) \wedge USE(i,j) \wedge REG(j,t,v) \wedge INST2(i) \Rightarrow REG(j,t+1,v \dot{-} 1) \qquad (viii)$$

If an INC, DEC or CONTINUE instruction is executed at time t then the next sequential instruction of P is executed at the $t+1^{st}$ step of the computation (unless the final continue statement has been reached):

$$\wedge_{0 \le t < p(|x|)} \wedge_{1 \le i < n} INSTRUCTION(t,i) \wedge INST1(i) \Rightarrow INSTRUCTION(t+1,i+1) \qquad (ix)$$

$$\wedge_{0 \le t < p(|x|)} \wedge_{1 \le i < n} INSTRUCTION(t,i) \wedge INST2(i) \Rightarrow INSTRUCTION(t+1,i+1) \qquad (x)$$

$$\wedge_{0 \le t < p(|x|)} \wedge_{1 \le i < n} INSTRUCTION(t,i) \wedge INST4(i) \Rightarrow INSTRUCTION(t+1,i+1) \qquad (xi)$$

When the final continue statement is reached, the computation does not proceed:

$$\wedge_{0 \le t < p(|x|)} INSTRUCTION(t,n) \Rightarrow INSTRUCTION(t+1,n) \qquad (xii)$$

The Test and Branch instruction "IF Rj = 0 JMP Lk" behaves correctly when the test fails:

$$\bigwedge_{0\le t<p(|x|)} \bigwedge_{1\le i<n} \bigwedge_{0\le j\le r} INSTRUCTION(t,i) \wedge INST3(i) \wedge USE(i,j) \wedge \neg REG(j,t,0) \Rightarrow INSTRUCTION(t+1,i+1) \quad (xiii)$$

The Test and Branch instruction "IF Rj = 0 JMP Lk" behaves correctly when the test succeeds: Let i_k denote the instruction of P with label Lk. Notice that we are sure that the label exists since the program we started with was a well-formed RAM program.

$$\bigwedge_{0\le t<p(|x|)} \bigwedge_{1\le i<n} \bigwedge_{0\le j\le r} INSTRUCTION(t,i) \wedge INST3(i) \wedge USE(i,j) \wedge REG(j,t,0) \Rightarrow INSTRUCTION(t+1,i_k) \quad (xiv)$$

The nondeterministic branch instruction performs as specified: Suppose the i^{th} instruction of P is "JMP L1, ..., Lk." Suppose label Lj is attached to the i_j^{th} instruction of P, for $1 \le j \le k$.

$$\bigwedge_{0\le t<p(|x|)} \bigwedge_{1\le i<n} INSTRUCTION(t,i) \wedge INST5(i) \Rightarrow JUSTONE(INSTRUCTION(t+1,i_1),\ldots, INSTRUCTION(t+1,i_k)) \quad (xv)$$

Let B_x be $(i) \wedge \ldots \wedge (xv)$. Since each of the subexpressions are in conjunctive normal form, B_x is in conjunctive normal form. The number of occurrences of variables is proportional to $p(|x|)$ or $p(|x|)^2$. So the length of B_x is proportional to $p(|x|)^3$ at worst. Hence, given P and x, B_x can be written out in polynomial time. Furthermore, by construction, B_x is satisfiable just in case there is a computation of P on input x that halts in at most $p(|x|)$ steps.

To see this, imagine the state of a computation of a RAM program as a grid plus an instruction counter. There is one column for each register and one row for each possible time. The value of the register, at the time given by the corresponding row is placed at coordinates in the grid for that particular register at that particular time. For example, such a grid might look something like:

	$\vdots$	$\vdots$	$\vdots$	0	0	0
	$x-2$	1	1	0	0	0
$\vdots$	$x-1$	1	1	0	0	0
	$x-1$	1	0	0	0	0
1	$x-1$	0	0	0	0	0
0	x	0	0	0	0	0
	R1	R2	$\cdots$			Rn

Actually, we must use a three-dimensional grid. Since we are constructing a Boolean expression that mimics, in some way, the RAM computation, we can only use 0's and 1's. So our grid takes on a third dimension, with coordinate value ranging from 0 to maximum value that any register may take on during the simulation. Hence, row i, column t, depth v takes on a value 1 iff at time t of the computation register Ri contains the value v. If the value of register Ri is not v at time t, then there is a 0 at coordinates (t, i, v) in our grid.

Notice that as each instruction of the program executes, the grid representing the values of the registers will change in at most two points. This is reflected in B_x above. What changes actually happen are also reflected. The execution of the RAM program will cause the grid to change in a particular way. If there is a sequence of grid changes that satisfies B_x, then there is a corresponding valid execution on the RAM program which gave rise to B_x.

If values are assigned to the *USE*, *REG* and *INSTRUCTION* variable according to their intended meaning, as dictated by some accepting computation of P, then B_x will be satisfiable. The construction of B_x guarantees that any satisfying assignment corresponds to an accepting computation of P. Hence, $x \in S$ iff $B_x \in CNFSAT$. ⊗

Exercise 4.70: Show that a set $A \in NP$ iff there is a set $B \in P$ and a polynomial p such that A can be defined as:

$$A = \{x | (\exists y) | y | \leq p(|x| \text{ and } \langle x, y \rangle \in B\}.$$

Exercise 4.71: For each of the following sets, determine whether it is in P, in NP but not known to be in P, recursive but not known to be in NP, r.e. but not recursive, or not even r.e.

a) $\{\phi | \phi$ is a Boolean formula that is NOT satisfiable $\}$
b) $\{(\phi, i) | \phi \in$ SAT or $\varphi_i(i) \downarrow\} \cup \{(\phi, i) | \phi \notin$ SAT $\}$
c) $\{x | M_x(x)$ halts in $|x|$ steps$\}$
d) $\{1^n |$ there exists a and b such that $a^2 + b^2 = n\}$
e.) $\{(\phi, k) |$ there exists exactly k satifying assignments for $\phi\}$

Exercise 4.72: For each of the following functions say whether it is known to be computable in polynomial time, known to be not computable in polynomial time, or currently not known whether it is in polynomial time or not. Justify your answer.

a) For k expressed in binary,

$$f(k) = \begin{cases} 1 & \text{if there is a DTIME}(n^k) \text{ algorithm for SAT} \\ 0 & \text{otherwise.} \end{cases}$$

b) Let M_i denote the i^{th} Turing machine from the acceptable programming system based on Turing machines. Suppose x expressed in binary and i is expressed so that the code for M_i is easily obtained from i.

$$f((i, x, 0^k)) = \begin{cases} M_i(x) & \text{if } M_i(x) \text{ halts in } k \text{ steps} \\ 0 & \text{otherwise.} \end{cases}$$

c) For both input and output expressed in binary,

$$f(1^k) = \begin{cases} 1^{2^k} & \text{if } k \text{ is divisible by 17} \\ 1^k & \text{otherwise.} \end{cases}$$

§4.6 More *NP*-Complete Problems

In this section, we will verify that four other problems are *NP*-Complete. The purpose of doing this is two-fold. Firstly, some examples of polynomial reductions will be given. Secondly, the more problems that you know are *NP*-Complete, the greater the variety of ways you have to show some other problem is also *NP*-Complete. In fact, there are thousands of *NP*-Complete problems. Each problem is preceded by the technical definitions necessary to define the problem.

Definition 4.73: A *graph* is an ordered pair (V, E) such that V is a finite set of objects called *vertices* and $E \subseteq V \times V$. Elements of E are called *edges*. The graph is *complete* if $E = V \times V$, e.g., every pair of vertices is connected by an edge.

Definition 4.74: A graph $G = (V, E)$ is *undirected* iff whenever $\overline{uv} \in E$, so is $\overline{vu}$, for an $u, v \in V$. A *directed* graph (digraph) is any graph that is not undirected.

Definition 4.75: A *subgraph* of a graph $G = (V, E)$ is a graph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq (E \cap (V' \times V'))$.

Definition 4.76: A *k-clique* of a graph G is a complete subgraph of G with exactly k vertices.

The *CLIQUE* problem is to determine if an arbitrary graph has a k-clique, for some value of k.

Theorem 4.77: The *CLIQUE* problem is *NP*-Complete.

Proof: First we must show that *CLIQUE*$\in$ *NP*. A nondeterministic algorithm starts by guessing a size k subset of the vertices from the input graph, G. This can be done in time proportional to the number of vertices of G. Next, the algorithm, in time $O(k^2)$, verifies that for every pair of vertices in the chosen subset, there is an edge connecting them in G.

To complete the proof, we must show *CNFSAT*$\leq_m^p$*CLIQUE*. This step involves showing how to transform uniformly in polynomial time an instance of *CNFSAT* into an instance of *CLIQUE*. Then we must show that the starting instance of *CNFSAT* was satisfiable iff the transformed instance has a k clique, where k is chosen as part of the transformation. When all this is done, Theorem 4.69 will immediately imply the result we seek. Now we proceed to describe the necessary transformation. The initial step is to develop notation to describe an arbitrary expression in conjunctive normal form. Let us suppose that a formula α looks like:

$$\alpha = \alpha_1 \wedge \alpha_2 \wedge \cdots \wedge \alpha_k,$$

where each

$$\alpha_i = \alpha_{i,1} \vee \alpha_{i,2} \vee \cdots \vee \alpha_{i,j_i},$$

where each $\alpha_{i,j}$ is a Boolean variable, or the negation of one. The $\alpha_{i,j}$'s are called *atomic formulae*.

The graph we construct will have one vertex for every literal of α. Two vertices will be connected by an edge if there is a possibility of some assignment satisfying both of them. The number of edges will be at most the square of the number of vertices.

$$V = \{[i,j] | 1 \leq i \leq k \text{ and } 1 \leq j \leq j_i\}$$

$$E = \{\overline{[i,j]\ [l,m]} | i \neq l \text{ and } \neg\, \alpha_{i,j} \neq \alpha_{l,m} \text{ and } \alpha_{i,j} \neq \neg\, \alpha_{l,m}\}$$

Here is an example. Let

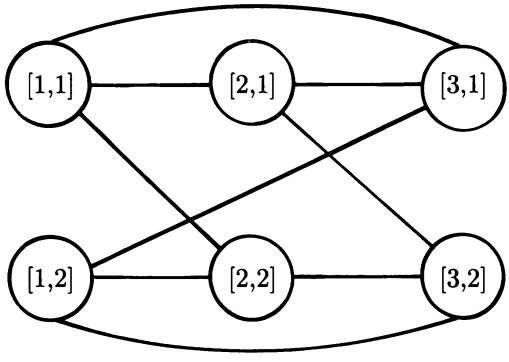
$$\alpha = (\beta_1 \vee \neg\, \beta_2) \wedge (\beta_2 \vee \neg\, \beta_3) \wedge (\beta_3 \vee \neg\, \beta_1).$$

For this example,

$$\alpha_{1,1} = \beta_1 \qquad \alpha_{2,1} = \beta_2 \qquad \alpha_{3,1} = \beta_3$$

$$\alpha_{1,2} = \neg\, \beta_2 \qquad \alpha_{2,2} = \neg\, \beta_3 \qquad \alpha_{3,2} = \neg\, \beta_1$$

Continuing with the example, G looks like:



Let $n = \sum_{i=1}^{k} j_i$ = the number of literals. Then there are n vertices in G and at most n^2 edges. So, the time to construct G is $O(n^2)$, which is polynomial in the number of literals. Hence, the transformation is polynomial. It remains to show that α is satisfiable iff G has a k clique.

Suppose that α is satisfiable. Then there is an assignment of TRUE or FALSE to each variable that makes α evaluate to TRUE. This means that in each α_i $(1 \leq i \leq k)$ there is an α_{i,j^i} (for some $1 \leq j^i \leq j_i$) that receives value TRUE. We claim that $\{[i, j^i] | 1 \leq i \leq k\}$ is a k clique. To show this, we must verify that $e = \overline{[i, j^i]\,[l, j^l]} \in E$ for all $1 \leq i \neq l \leq k$. If $i \neq l$ and $e \notin E$, then either $\alpha_{i,j^i} = \neg\, \alpha_{l,j^l}$ or $\neg\, \alpha_{i,j^i} = \alpha_{l,j^l}$. Both cases lead to a contradiction since we know that both α_{i,j^i} and α_{l,j^l} get value TRUE.

Suppose that G has a k clique. If $[i, j]$ and $[l, m]$ are in the clique, then, by the choice of E, it must be the case that $i \neq l$. Consider the following truth assignment to the literals of α. For each β, a Boolean variable referenced in α, assign the value TRUE to β if, for some $[i, j]$ in the clique, $\beta = \alpha_{i,j}$. If for some $[i, j]$ in the clique, $\neg\, \beta = \alpha_{i,j}$, then assign the value FALSE to β. Independently of how the other variables referenced in α are assigned, at least one atomic formula in each α_i will evaluate to TRUE, hence, α evaluates to TRUE.

The only problem may be that we have attempted to assign both TRUE and FALSE to some β. Suppose, by way of contradiction, that we have assigned true to β because $\beta = \alpha_{i,j}$ and we have assigned FALSE to β because $\neg\beta = \alpha_{l,m}$. Since $[i, j]$ and $[l, m]$ are in the clique, so $\overline{[i, j]\,[l, m]} \in E$. Consequently, invoking the definition of E,

$$\beta = \alpha_{i,j} \neq \neg\, \alpha_{l,m} = \neg\, \neg\, \beta = \beta.$$

The desired contradiction is evident. ⊗

Exercise 4.78: Show that for the reduction of the above result, the number of cliques in the graph constructed is the same as the number of satisfying assignments of the original Boolean expression.

Definition 4.79: A *vertex cover* of a graph is a subset of the set of vertices (called a *cover*) with the property that every edge is incident with one of the vertices in the cover.

Definition 4.80: The *VERTEX COVER* problem is to determine if an arbitrary graph has a vertex cover of size at most k, for an arbitrary value of k.

Theorem 4.81: *VERTEX COVER* is *NP*-Complete.

Proof: To see that *VERTEX COVER* is in *NP*, note that a nondeterministic algorithm need only guess a set of k vertices, and then check each edge to make sure one of its end points is in the guessed set. To show completeness, we must polynomially transform some *NP*-complete problem into *VERTEX COVER*. Now we have a choice now of two *NP*-Complete problems to choose from. Much of the prior reduction involved transforming Boolean expressions into graphs. Such a complicated transformation can be avoided by showing $CLIQUE \leq_m^p$ *VERTEX COVER*. Suppose $G = (V, E)$ is a graph. Let n be the number of vertices. Construct $\overline{G} = (V, \overline{E})$ where

$$\overline{E} = \{\overline{v\,w} | v, w \in V, v \neq w, \text{ and } \overline{v\,w} \notin E\}.$$

The transformation from G to $\overline{G}$ takes time proportional to the sum of the number of vertices and the number of edges, clearly polynomial. The proof is completed by showing that G has a k clique iff $\overline{G}$ has a size $n - k$ vertex cover.

Suppose S is a clique in G. Then no edge in $\overline{G}$ connects any two vertices in S. Hence, every edge in $\overline{E}$ is incident with some vertex in $V - S$. Since S is of size k and V is of size n, $\overline{G}$ has a vertex cover of size $n - k$.

Suppose that $V - S$ is a vertex cover of $\overline{G}$. Then every edge in $\overline{E}$ is incident with some vertex in $V - S$. So no edge in $\overline{E}$ connects two vertices in S. So S is a clique in G. ⊗

Definition 4.82: A *cycle* in a (di)graph $G = (V, E)$ is a sequence of vertices $v_0, v_1, \cdots, v_n$ such that $v_i \in V$, $0 \leq i \leq n$, $\overline{v_i v_{i+1}}$, $0 \leq i < n$, and $v_0 = v_n$. A *Hamiltonian* cycle is cycle with two additional properties: $v_i \neq v_j$ for $0 \leq i < j < n$ and $\{v_i | i \leq n\} = V$.

Definition 4.83: The *DIRECTED HAMILTONIAN CIRCUIT* problem is to determine if there is a Hamiltonian circuit in an arbitrary digraph.

Theorem 4.83: The *DIRECTED HAMILTONIAN CIRCUIT* problem is *NP*-Complete.

Proof: Again, it is easy to show that *DIRECTED HAMILTONIAN CIRCUIT* $\in NP$. The necessary nondeterministic algorithm guesses a permutation of the vertices and verifies that edges exists between the vertices that are adjacent in the permutation.

To show completeness, we will prove that *VERTEX COVER* $\leq_m^p$ *DIRECTED HAMILTONIAN CIRCUIT*. Let $G = (V, E)$ and k some number less than the number of vertices in V. We proceed to construct a graph $G' = (V', E')$ such that G' has a Hamiltonian cycle iff G has a vertex cover of size k.

Choose $a_1, a_2, \cdots, a_k$ new elements that are easily distinguished from all elements of V and E. These new elements are in V' along with four new vertices for each $\overline{u\,v} \in E$. These four vertices will be named:

$$(u, \overline{u\,v}, 0) \quad (v, \overline{u\,v}, 0)$$

$$(u, \overline{u\,v}, 1) \quad (v, \overline{u\,v}, 1)$$

E' contains three classes of edges. For each $\overline{u\,v} \in E$, there are four edges in the first class of E''s edges:

$$(u, \overline{u\,v}, 0) \quad \longleftrightarrow \quad (v, \overline{u\,v}, 0)$$

$$(u, \overline{u\,v}, 1) \quad \longleftrightarrow \quad (v, \overline{u\,v}, 1)$$

Let v be an arbitrary vertex in V. Suppose that $w_1, w_2, \cdots, w_m$ is a complete list of all the vertices adjacent to v. Then the following edges for $0 < i < m$ are in E':

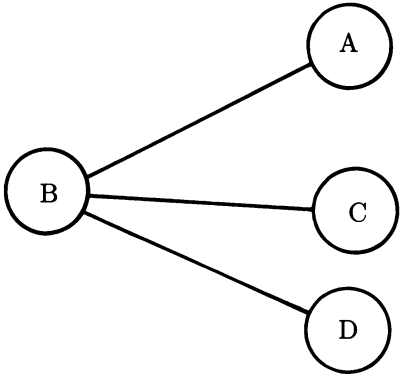
$$\begin{array}{c} (v, \overline{v\,w_i}, 0) \\ \downarrow \\ (v, \overline{v\,w_i}, 1) \\ \downarrow \\ (v, \overline{v\,w_{i+1}}, 0) \end{array}$$

The third and final class of edges in E' contains two edges for each $1 \leq i \leq k$:

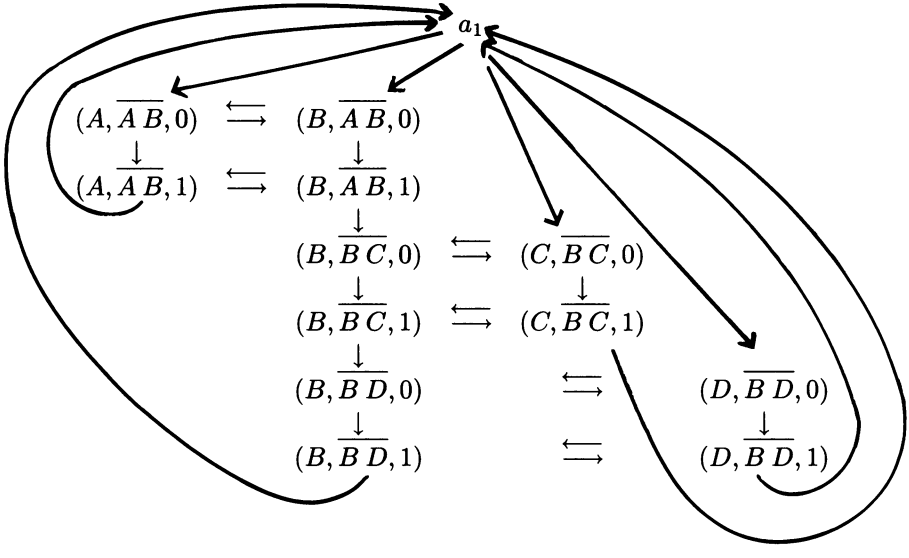
$$\begin{array}{cc} a_i & (v, \overline{v\,w_m}, 1) \\ \downarrow & \downarrow \\ (v, \overline{v\,w_1}, 0) & a_i \end{array}$$

If G has n vertices and m edges, then G' has $4m + k$ vertices and $6m + 2k$ edges. Clearly, the transformation is polynomial in m.

An example is in order. Consider the following graph G with a 1 vertex cover:



The conversion of this graph G into G' yields:



Now we return to the proof. As a first observation, notice that for each edge $\overline{v\,w} \in E$, there are the following edges in E':

$$\begin{array}{ccc} \downarrow & & \downarrow \\ (v, \overline{v\,w}, 0) & \leftrightarrows & (w, \overline{v\,w}, 0) \\ \downarrow & & \downarrow \\ (v, \overline{v\,w}, 1) & \leftrightarrows & (w, \overline{v\,w}, 1) \\ \downarrow & & \downarrow \end{array}$$

Consequently, any Hamiltonian circuit that enters $(v, \overline{v\,w}, 0)$ must exit the quartet of vertices from $(v, \overline{v\,w}, 1)$, as otherwise one of the four nodes cannot be visited in a Hamiltonian cycle.

Now we complete the proof by showing that the transformation has the desired properties. Suppose that G' has a hamiltonian cycle. We can view this cycle as consisting of k parts, each from a_i to a_j for some i and j such that no other vertex in $\{a_1, \cdots, a_k\}$ occurs in that segment. By the previous observation about the traversal of each quartet of vertices, each segment starting at a_i enters some $(v\overline{v\,w_1}, 0)$ and exits from $(v, \overline{v\,w_m}, 1)$. This segment could also contain a traversal through $(w_i, \overline{v\,w_i}, 0), \cdots, (w_i, \overline{v\,w_i}, 1)$, but all these w_i's are adjacent to v. Hence, each of the k segments $a_i, \cdots, a_j$ has a single vertex associated with it. These k vertices are a vertex cover of G, since for every $(w, e, b) \in V'$, e must be incident with one of the selected vertices.

Suppose $\{v_1, \cdots, v_k\}$ is a vertex cover of G. Consider the following cycle in G', using primarily the second class of edges:

$$
\begin{aligned}
&a_1, (v_1, \overline{v_1\,w_{1,1}}, 0), \cdots, (v_1, \overline{v_1\,w_{1,1}}, 1), (v_1, \overline{v_1\,w_{2,1}}, 0), \cdots,\\
&\qquad (v_1, \overline{v_1\,w_{m_1,1}}, 0), \cdots, (v_1, \overline{v_1\,w_{m_1,1}}, 1)\\
&a_2, (v_2, \overline{v_2\,w_{1,2}}, 0), \cdots, (v_2, \overline{v_2\,w_{1,2}}, 1), (v_2, \overline{v_2\,w_{2,2}}, 0), \cdots,\\
&\qquad (v_2, \overline{v_2\,w_{m_2,2}}, 0), \cdots, (v_2, \overline{v_2\,w_{m_2,2}}, 1)\\
&\qquad\qquad \vdots\\
&a_k, (v_k, \overline{v_k\,w_{1,k}}, 0), \cdots, (v_k, \overline{v_k\,w_{1,k}}, 1), (v_k, \overline{v_k\,w_{2,k}}, 0), \cdots,\\
&\qquad (v_k, \overline{v_k\,w_{m_k,k}}, 0), \cdots, (v_k, \overline{v_k\,w_{m_k,k}}, 1), a_1
\end{aligned}
$$

If all of the vertices from V' are in the above path, then we are done. Suppose, on the other hand, that some $(w, e, b) \in V'$ is not in the cycle above. Since the v_i's form a vertex cover for G, there is a $v \in \{v_1, \cdots, v_k\}$ such that $e = \overline{v\,w}$. The part of the cycle that goes

$$(v, \overline{v\,w}, 0), (v, \overline{v\,w}, 1)$$

can be replaced with:

$$(v, \overline{v\,w}, 0), (w, \overline{v\,w}, 0), (w, \overline{v\,w}, 1), (v, \overline{v\,w}, 1).$$

The resulting path is still a cycle and it includes (w, e, b). Since (w, e, b) was chosen arbitrarily, this process can be repeated for all omitted vertices from V', thus forming a Hamiltonian cycle in G'. ⊗

Exercise 4.84: Show *DIRECTED HAMILTONIAN CIRCUIT* $\leq_m^p$ *UNDIRECTED HAMILTONIAN CIRCUIT*.

Exercise 4.84: The subgraph isomorphism problem is: Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, does G_1 contain a subgraph that is isomorphic to G_2? In other words, is there a $V' \subseteq V_1$ and $E' \subseteq E_1$ such that V' and V_2 have the same number of vertices, E' and E_2 have the same number of edges, and there is a one-to-one function f mapping V_2 to V' such that $(u, v) \in E_2$ iff $(f(u), f(v)) \in E'$. Prove that the subgraph isomorphism problem is *NP*-Complete.

Exercise 4.85: An independent set of a graph $G = (V, E)$ is a subset $V' \subset V$ such that for any $u, v \in V'$ if $u \neq v$ then $\overline{uv} \notin E$. The *Independent Set* problem is to determine for a given graph $G = (V, E)$ and an integer k whether or not G has an independent set of size at least k. Prove that *Independent Set* is *NP*-Complete.

Exercise 4.86: Let $F - SAT = \{\phi |$ there are at least four different satisfying truth assignments for $\phi\}$. Show that $F - SAT$ is *NP*-Complete.

§4.7 Historical Notes

The notions of reducibility and completeness stem from [Tur]. The idea of $<_m$ and $<_1$ reductions are from [Pos]. The linear speed-up theorem and the time hierarchy theorem are from [H&S1]. Theorem 4.38 was based on a result from [H&S2]. Theorem 4.55 is from [Sav]. The closure of nondeterministic space classes under complementation is from [Imm] but also appeared independently in [Sze]. The *NP*-Completeness of *CNFSAT* is from [Coo]. The series of polynomial reductions from *CNFSAT* to *CLIQUE* to *VERTEX* cover to *DIRECTED HAMILTONIAN CIRCUIT* is from [Kar]. An extensive collection of *NP* complete problems can be found in [G&J]

[Coo] S. Cook, The complexity of theorem-proving procedures, *Proceedings of the 3rd Annual ACM Symposium on the Theory of Computing*, ACM, New York, 1971, pp. 151-158.

[G&J] M. Garey and D. Johnson, Computers and Intractability, A Guide to *NP*-Completeness, W. H. Freeman and Co., San Francisco, 1979.

[H&S1] J. Hartmanis and R. Sterns, On the computational complexity of algorithms, *Transactions of the AMS*, Vol. 117, 1965, pp. 285–306.

[H&S2] F. Hennie and R. Sterns, Two-tape simulation of multiple tape Turing machines, *Journal of the ACM*, Vol. 13, 1966, pp. 533–546.

[Imm] N. Immerman, Deterministic space is closed under complementation, *SIAM Journal of Computing*, Vol. 7, 1988, pp. 935–938.

[Kar] R. Karp, Reducibility among combinatorial problems, in Complexity of Computer Computations, edited by R. Miller and J. Thatcher, Plenum Press, New York, 1972.

[Pos] E. Post, Recursively enumerable sets of positive integers and their decision problems, *Bulletin of the American Mathematical Society*, Vol. 50, 1944, pp. 284–316.

[Sav] W. Savitch, Relationships between nondeterministic and deterministic tape complexities, *Journal of Computer and Systems Sciences*, Vol. 4, 1970, pp. 177–192.

[Sze] R. Szelpcsenyi, The method of forced enumeration for nondeterministic automata, *Acta Informatica*, 1988, pp. 279–284.

[Tur] A. Turing, On computable numbers, with an application to the entscheidungsproblem, *Proceedings of the London Mathematical Society*, Vol. 42, 1936, pp. 230–265, with a correction in Vol. 43, 1937, pp. 544-546.

Solutions to Selected Exercises

Exercise: 1.5.

To divide R1 by R2, returning 0 if R2 is 0.

```
    R3←R2          save the divisor
    R2 JMP N3b     check if divisor is 0
N1  R1 JMP N3b     check if done
    DEC R1         this loop subtracts R2 from R1
    DEC R2
    R2 JMP N2b     check if finished with this iteration
    JMP N1a        continue subtracting R2 from R1
N2  INC R4         increment the answer
    R2←R3          restore divisor
    JMP N1a        subtract R2 from R1 again
N3  R1←R4          move result to output register
    CONTINUE
```

Exercise: 1.11.

$rm(x, y)$ = remainder of x divided by y. We use the following recursive definition:

$$rm(0, y) = 0$$

$$rm(x+1, y) = (rm(x, y) + 1) \times \begin{cases} 0 & \text{if } rm(x, y) + 1 = y; \\ 1 & \text{otherwise.} \end{cases}$$

By primitive recursion:

$$\begin{aligned} rm(0, y) &= Z(y) \\ rm(x+1, y) &= h(x, y, rm(x, y)) \\ &= M(S(rm(x, y)), \overline{\epsilon}(S(rm(x, y)), y)). \end{aligned}$$

Exercise: 1.15.

$\Pi(x_1, \ldots, x_n, z) = \Pi_{y \le z} f(x_1, \ldots, x_n, y) = f(x_1, \ldots, x_n, 0) \times \ldots \times f(x_1, \ldots, x_n, z)$.

We use the following recursive definition (where $\overline{x} = x_1, \ldots, x_n$):

$$\begin{aligned}\Pi(\overline{x},0) &= f(\overline{x},0)\\ \Pi(\overline{x},z+1) &= (\Pi_{y\le z} f(\overline{x},y)) * f(\overline{x},z+1)\end{aligned}$$

By primitive recursion:

$$\begin{aligned}\Pi(\overline{x},0) &= g(\overline{x})\\ &= f(U_1^n(\overline{x}),\ldots,U_n^n(\overline{x}),Z(U_1^n(\overline{x})))\\ \Pi(\overline{x},z+1) &= h(\overline{x},z,\Pi(\overline{x},z))\\ &= M(\Pi(\overline{x},z),f(U_1^n(\overline{x}),\ldots,U_n^n(\overline{x}),S(z))).\end{aligned}$$

Exercise: 1.16.

The idea is to define g via primitive recursion, keeping the value of the function to be one larger than the variable of recursion, as long as a zero does not appear in the range of f. In the recursive step, care must be taken not to take the previous function value if no zeros have been found in the range of f.

$$g(x_1,\ldots,x_n,0) = \begin{cases} 0 & \text{if } f(x_1,\ldots,x_n,0)=0,\\ 1 & \text{otherwise}\end{cases}$$

$$g(x_1,\ldots,x_n,y+1) = \begin{cases} y+1 & \text{if } f(x_1,\ldots,x_n,y+1)=0,\\ g(x_1,\ldots,x_n,y) & \text{if } f(x_1,\ldots,x_ny+1)\neq 0\\ & \quad\text{and } g(x_1,\ldots,x_n,y)\\ & \quad\neq y+1,\\ y+2 & \text{otherwise.}\end{cases}$$

The last clause of g can be written more formally as:

$$\begin{aligned}g(x_1,\ldots,x_n,y+1) &= (y+1)\times \overline{sg}(f(x_1,\ldots,x_n,y+1))\\ &+ g(x_1,\ldots,x_n,y)\\ &\quad\times sg(f(x_1,\ldots,x_n,y+1))\times \overline{\epsilon}(g(x_1,\ldots,x_n,y),y+1)\\ &+ (y+2)\times sg(f(x_1,\ldots,x_n,y+1))\times \epsilon(g(x_1,\ldots,x_n,y),y+1).\end{aligned}$$

Exercise: 1.17.

We will use the following strategy to define the function g by primitive recursion. First we will sum up $sg(f(x_1,\ldots,x_n,0)),\cdots,sg(f(x_1,\ldots,x_n,y))$. If the sum is $y+1$, then all the values $f(x_1,\ldots,x_n,0),\cdots,f(x_1,\ldots,x_n,y)$ evaluate to a nonzero value. In this case we want to add 1 to the previous value of g. If the sum is less than $y+1$, then there is a $z<y$ such that $f(x_1,\ldots,x_n,z)=0$, and we take the previous function value as is. More formally,

$$\begin{aligned}g(x_1,\ldots,x_n,0) &= 0\\ g(x_1,\ldots,x_n,y+1) &= g(x_1,\ldots,x_n,y)\\ &\quad+\epsilon((\Sigma_{i\le y} sg(f(x_1,\ldots,x_n,i))),y+1).\end{aligned}$$

Exercise: 1.31.

Since the identity function is primitive recursive and bounded summations of primitive recursive functions are primitive recursive, there is a primitive recursive function

$$SUM(x, y) = \Sigma_{i \leq x+y} i.$$

Then

$$\langle x, y \rangle = SUM(x, y) + U_2^2(x, y).$$

Next we show that cd is primitive recursive. First define:

$$h_1(x, y) = \langle S(x), Z(y) \rangle.$$

Now, reverse the arguments:

$$h_2(x, y) = h_1(U_2^2(x, y), U_1^2(x, y)).$$

Using two applications of composition, define:

$$h_3(x, y) = S(S(U_1^2(x, y))).$$

Now to subtract,

$$h_4(x, y) = h_3(x, y) \dot{-} h_2(x, y).$$

A final auxiliary function is needed:

$$h(x, y) = U_2^2(x, y) + h_4(x, y).$$

Now,

$$cd(0) = 0$$
$$cd(n + 1) = h(n, cd(n)).$$

For π_2, a single auxiliary function is needed: $h_5(x) = \langle cd(x), Z(x) \rangle$. Now,

$$\pi_2(n) = n \dot{-} h_5(n).$$

The final function, π_1, is obtained by a single composition as given in the text.

Exercise: 1.33.

Recall that $< x, y >= 1 + 2 + \cdots + (x + y) + y$. If $x = y = 0$ then $< x, y >= 0$. If $x > 0$ or $y > 0$ then $(x+y) \geq 1$ and by the definition above, $< x, y > \geq (x + y) + y$ which is $\geq$ both x and y.

Exercise: 1.35.

Suppose ψ is a partial recursive function with domain $\{x_0, \cdots, x_n\}$. Suppose without loss of generality that x_n is the largest element of the domain of ψ. Define z_i for $i \leq x_n$ as follows:

$$z_i = \begin{cases} \psi(x_j) + 1 & \text{if } i = x_j, \\ 0 & \text{otherwise.} \end{cases}$$

Let x be the coded $x_n + 2$ tuple: $\langle x_n, z_0, \cdots, z_{x_n} \rangle$. We claim that $d_x = \psi$. Suppose that $y = x_j$, i.e. y is in the domain of ψ. Then $y < \pi_i(x) + 1$ and $F(x, y) = \Pi(x_j + 1, x_n + 1, \pi_2(x)) = \psi(x_j) + 1$. Since $\psi(x_j) + 1 > 0$, $d_x(x_j)) = \psi(x_j)$. Suppose the y is not in the domain of ψ. If $y > \pi_1(x) = x_n$, then $d_x(y)$ is undefined as desired. Suppose then that $y \leq x_n$. Then $F(x, y) = \Pi(y + 1, x_n + 1, \pi_2(x)) = z_y = 0$. So, again, $d_x(y)$ is undefined as desired.

Exercise: 1.40.

In order to define *Add* and *Sub*, we will use the maximum in a bounded range function:

$$g(x_1, \ldots, x_n, y) = \max z \leq y[f(x_1, \ldots, x_n, z) = 0]$$

which is primitive recursive by 1.16.

For *Sub*, given $y =< r_1, \ldots, r_x >$ we want to find the maximum $z =< r_1, \ldots, r_x >$ such that all the register values remain the same, except for the jth register, which is decremented. We use the following predicate function f, which has the value zero only in this situation:

$f = 0$ if jth register is decremented in z relative to y and the number of registers in z that are different from the corresponding register in y equals 1.

$$f = \overline{\epsilon}(\pi(j, x, z), \pi(j, x, y) - 1) \times \overline{\epsilon}\left(\left(\sum_{k=1}^{x} \overline{\epsilon}(\pi(k, x, z), \pi(k, x, y))\right), 1\right).$$

More formally,

$$Sub(j, x, y) = \max z \leq y \left[M\left(\overline{\epsilon}(\pi(j,x,z), PR(\pi(j,x,y))), \overline{\epsilon}\left(\left(\sum_{k=1}^{x} \overline{\epsilon}(\pi(k,x,z), \pi(k,x,y))\right), 1\right)\right)\right].$$

For ADD, we can define a similar function, replacing the PR above with S, *but we need to find an upper bound*, since the z we are looking for will be larger than y. Note that by problem 1.25 above, $y \geq r_i, i = 1, \ldots, x$ and so $y + 1 \geq r_i + 1, i = i, \ldots, x$. Let our bound be

$$y' = \underbrace{y + 1, \ldots, y + 1}_{\text{x times}}.$$

Exercise: 1.49.

$s_0 00 R s_0$	find the end of the input
$s_0 11 R s_0$	
$s_0 BBL s_1$	found the end
$s_1 10 L s_1$	increment, bit by bit
$s_1 01 L s_2$	end of increment
$s_2 00 L s_2$	find left end
$s_2 11 L s_2$	
$s_2 BBR s_4$	
$s_1 B1 L s_3$	found the left end of the tape
$s_3 BBR s_4$	

Exercise: 1.50.

The coding that makes the problem easiest is the unary encoding. The number n is represented by $n+1$ 1's. The alphabet is $\{B, 1\}$.

$s_0 1BR s_1$
$s_1 1BR s_2$
$s_2 11 R s_2$
$s_2 B1 L s_3$

Exercise: 1.57.

To show that the BNF language expresses exactly the Turing computable functions requires proof in two directions. The first is to show that every partial recursive function can be computed by a program in the BNF language. The second is to show that every program in the BNF language is a partial recursive function. We do not show this second direction but it is a simple task to convert a BNF program into a RAM program (this is the job of a compiler). Alternatively, we could just assume Church's thesis.

Theorem: Every partial recursive function can be computed by a program in the BNF language.

Proof: Our convention will be that $X1, \ldots, Xn$ are variables containing the input values $x_1, \ldots, x_n$ and that the output value is contained in $X1$.

First we define a BNF subprogram $Xi \leftarrow Xj$ that *copies* the value of Xj into Xi. Note that it is necessary for the remainder of the proof that the value in Xj remain intact.

$Xi \leftarrow Xj$: $\{Xk$ is a variable not used by the rest of the program$\}$
case 1: $i \neq j$

```
clear Xi
clear Xk
while Xj ≠ 0 do
    decrement Xj
    increment Xi
    increment Xk
```

```
            while Xk ≠ 0 do
                decrement Xk
                increment Xj
```
case 2: $i = j$
```
            clear Xk
```
The base functions can be represented as follows:

$Z(x)$:	`clear` $X1$
$S(x)$:	`increment` $X1$
$U_j^n(x_1, \ldots, x_n)$:	$X1 \leftarrow Xj$

The remainder of the proof follows the format for representation of the partial recursive functions by RAM programs. We show only minimalization. Suppose that f, a function of n arguments, is defined by minimalization from h. Let P_h be a BNF program computing h. Let m be such that no variable Xk for $k \geq m$ is referenced in P_h. The following program computes f:

```
X(m+1) ← X1                                  save arguments
        ⋮
X(m+n) ← Xn
clear X(m+n+1)                    initialize search variable
X(n+1) ← X(m+n+1)
P_h                                    compute h(x_1, ... x_n, 0)
while X1 ≠ 0 do                                 check if done
    X1 ← X(m+1)                               restore arguments
    ⋮
    Xn ← X(m+n)
    clear X(n+1)                      clear scratch variables
    ⋮
    clear X(m-1)
    increment X(m+n+1)                      set search variable
    X(n+1) ← X(m+n+1)
    P_h                          compute h(x_1, ... x_n, X(m+n+1))
X1 ← X(m+n+1)
```

Exercise: 2.9.

Apply the recursion theorem (Theorem 2.7) to the projection function U_1^2. Then, for any x, $\varphi_e(x) = U_1^2(e, x) = e$.

Exercise: 2.13.

Use the following algorithm:

1. Write a program, called P, that takes two strings as as input, and outputs the first one.
2. Write a program, called S, that takes two strings, interprets the first argument as a program that we will call T. The second argument will

be called Y. Suppose that the first input statement of T inputs data into some variable, say Z, and maybe some other variables as well. The point is that Z is the first variable read. S then outputs a string representing a program T' that is just like T except that Z is initially set to Y, instead of being read in.

3. Write a program, called V, that takes two strings, A and B, as input. V runs program S on inputs A and A (again). The result of this computation is a string, say O. V then simulates program P on inputs O and B.
4. Program V, like all programs is represented as a character string. Run program S on argument strings V and V (again). The output of this computation will be a string that, when interpreted as a program, will produce its own code.

Exercise: 2.14.
From the electronic bulletin boards:

In LISP:

```
((lambda (x) (list x (list (quote quote) x)))
(quote (lambda (x) (list x (list (quote quote) x)))))
```

In Pascal:

```
program repro(output);
const d=39;
b=';begin writeln(c,chr(d),b,chr(d),chr(59));
writeln(chr(67),chr(61),chr(d),c,chr(d),b) end.';
c='program repro(output); const d=39; b=';
begin
writeln(c,chr(d),b,chr(d),chr(59));
writeln(chr(67),chr(61),chr(d),c,chr(d),b)
end.
```

In C:

```
main(a){a="main(a){a=%c%s%c;printf(a,34,a,34);}";
printf(a,34,a,34);}
```

Exercise: 2.18.

Theorem: (n-ary recursion theorem) Suppose $\varphi_0, \varphi_1, \cdots$ is an acceptable programming system. $(\forall i_1) \cdots (\forall i_n)(\forall x)$ there exists programs $e_1, \ldots, e_n$ such that:

$$\varphi_{e_1}(x) = \varphi_{i_1}(e_1, \ldots, e_n, x)$$
$$\vdots$$
$$\varphi_{e_n}(x) = \varphi_{i_n}(e_1, \ldots, e_n, x).$$

Proof: Let $i_1, \ldots, i_n$ be given. By implicit use of the s-m-n theorem, there is a recursive function g such that $\forall i, x_1, \ldots, x_n, z$:

$$\varphi_{g(i)}(x_1, \ldots, x_n, z) = \varphi_i(s(x_1, x_1, \ldots, x_n), \ldots, s(x_n, x_1, \ldots, x_n), z).$$

Now define:

$$\begin{aligned} e_1 &= s(g(i_1), g(i_1), \ldots, g(i_n)) \\ &\vdots \\ e_n &= s(g(i_n), g(i_1), \ldots, g(i_n)) \end{aligned}$$

Then

$$\begin{aligned} &\varphi_{e_1}(x) \\ &\quad = \varphi_{s(g(i_1), g(i_1), \ldots, g(i_n))}(x) \\ &\quad = \varphi_{g(i_1)}(g(i_1), \ldots, g(i_n), x) \\ &\quad = \varphi_{i_1}(s(g(i_1), g(i_1), \ldots, g(i_n)), \ldots, s(g(i_n), g(i_1), \ldots, g(i_n)), x) \\ &\quad = \varphi_{i_1}(e_1, \ldots, e_n, x) \end{aligned}$$

And similarly for $i_2, \ldots, i_n$.

Exercise: 2.23.

By the recursion theorem there is a program e such that

$$\varphi_e(j, y) = \begin{cases} 0 & \text{if } (\exists k < j) s(e, k) = s(e, j); \\ 1 & \text{if } (\forall k < j) s(e, k) \neq s(e, j) \text{ and} \\ & \quad (\exists k) j < k \leq y \text{ and } s(e, k) = s(e, j); \\ \varphi_{g(j)}(y) & \text{otherwise.} \end{cases}$$

Suppose by way of contradiction that $\lambda x[s(e, x)]$ is not a one-to-one function. Choose $j > k$ least such that $s(e, j) = s(e, k)$. Then $\forall y$,

$$\varphi_{s(e,j)}(y) = \varphi_e(j, y) = 0.$$

But, $\forall y > j$,

$$\varphi_{s(e,k)}(y) = \varphi_e(k, y) = 1.$$

Hence $\varphi_{s(e,k)} \neq \varphi_{s(e,j)} (\Rightarrow\Leftarrow)$
Therefore, $\lambda x[s(e, x)]$ is a one-to-one function and $\forall x, y$:

$$\varphi_{s(e,x)}(y) = \varphi_e(x, y) = \varphi_{g(x)}(y).$$

Exercise: 2.26.

Apply the fixed point theorem (Theorem 2.25) to the successor function S. Then there is an i such that $\varphi_{i+1} = \varphi_{S(i)} = \varphi_i$.

Exercise: 2.27.

Let f and c be given. Find a k such that $\varphi_i \neq \varphi_k$, for all $i \leq c$. Such a k must exist as there are only finitely many functions computed be the first $c+1$ programs in any acceptable programming system. Now define:

$$g(x) = \begin{cases} k & \text{if } x \leq c, \\ f(x) & \text{otherwise.} \end{cases}$$

Since f is recursive, so is g. Take a fixed point n of g. So $\varphi_n = \varphi_{g(n)}$. Suppose by way of contradiction that $n \leq c$. Then $g(n) = k$ where k was chosen such that $\varphi_k \neq \varphi_n$, a contradiction. Hence, $n > c$ and $g(n) = f(n)$, so $\varphi_n = \varphi_{g(n)} = \varphi_{f(n)}$. Consequently, n is also a fixed point for f.

Exercise: 2.28.

Suppose i is a program computing a function of two arguments. Define $f = \lambda x[s(i,x)]$. Since the store function is total, the function f just defined is recursive. Applying the fixed point theorem to f yields an e such that for any x:

$$\varphi_e(x) = \varphi_{f(e)}(x) = \varphi_{s(i,e)}(x) = \varphi_i(e,x).$$

Exercise: 2.29.

Let f be a recursive function. Define program i by:

$$\varphi_i(e,x) = \varphi_{univ}(f(e),x).$$

Applying the recursion theorem to program i yields an e such that for all x:

$$\varphi_e(x) = \varphi_i(e,x) = \varphi_{univ}(f(e),x) = \varphi_{f(e)}(x).$$

Exercise: 2.30.

By the *s-m-n* theorem and the universal machine theorem there is a recursive function k such that $\varphi_{k(x)} = g \circ \varphi_x$. Define $\hat{f}(x,y) = g^{-1}(f(k(x),y))$. Then, for any x and y:

$$\begin{aligned} \lambda y[g^{-1}\psi g(\hat{f}(x,y))] &= \lambda y[g^{-1}\psi g g^{-1} f(k(x),y)], \\ &= \lambda y[g^{-1}\psi f(k(x),y)], \\ &= g^{-1}\varphi_{k(x)}, \\ &= g^{-1} g \varphi_x, \\ &= \varphi_x. \end{aligned}$$

Exercise: 2.34.

See G. Riccardi, The independence of control structures in abstract programming systems, *Journal of Computer and Systems Sciences*, Vol. 22, 1981, pp. 107–143.

Exercise: 2.35.

See M. Machtey, K. Winklmann and P. Young, Simple Gödel numberings, isomorphisms and programming properties, *SIAM Journal of Computing*, Vol. 7, 1978, pp. 39–59.

Exercise: 2.36.

See J. Royer, A Connotational Theory of Program Structure, Lecture Notes in Computer Science Vol. 273, Springer Verlag, New York, 1987.

Exercise: 2.37.

See C. Smith, Applications of classical recursion theory to computer science, in "Recursion theory: its generalisations and applications," edited by S. Wainer and E. F. Drake, London Mathematical Society Lecture Notes Series Vol. 45, Cambridge University Press, Cambridge, 1980, pp. 236–247.

Exercise: 2.42.

Let $A = \{x | \varphi_x \text{ is a constant function }\}$. If A is recursive, then there is a recursive characteristic function f for A:

$$f(x) = \begin{cases} 1 & \text{if } \varphi_x \text{ is a constant function} \\ 0 & \text{otherwise.} \end{cases}$$

By the recursion theorem, there is a program e such that

$$\varphi_e(x) = \begin{cases} 1 & \text{if } f(e) = 0 \\ \uparrow & \text{otherwise.} \end{cases}$$

Now we show the contradiction:
$\varphi_e \in A \Rightarrow \varphi_e$ is a constant function $\Rightarrow f(e) = 1 \Rightarrow \forall x \varphi_e(x) \uparrow \Rightarrow \varphi_e \notin A$
$\varphi_e \notin A \Rightarrow \varphi_e$ is *not* a constant function $\Rightarrow f(e) = 0 \Rightarrow \forall x \varphi_e(x) = 1 \Rightarrow \varphi_e \in A$.

Exercise: 2.45.

Let $D = \{(x, y) | \varphi_x = \varphi_y\}$. If D is recursive, then there is a recursive characteristic function f for D:

$$f(x, y) = \begin{cases} 1 & \text{if } \varphi_x = \varphi_y \\ 0 & \text{otherwise.} \end{cases}$$

By the mutual recursion theorem, there are two programs e_0 and e_1 such that:

$$\varphi_{e_0}(x) = \begin{cases} 0 & \text{if } f(e_0, e_1) = 0 \\ 1 & \text{otherwise} \end{cases}$$

$$\varphi_{e_1}(x) = \begin{cases} 0 & \text{if } f(e_0, e_1) = 0 \\ 2 & \text{otherwise.} \end{cases}$$

Now we show the contradiction:
$(e_0, e_1) \in D \Rightarrow \varphi_{e_0} = \varphi_{e_1} \Rightarrow f(e_0, e_1) = 1 \Rightarrow \varphi_{e_0} = \lambda x[1]$ and $\varphi_{e_1} = \lambda x[2] \Rightarrow (e_0, e_1) \notin D$
$(e_0, e_1) \notin D \Rightarrow \varphi_{e_0} \neq \varphi_{e_1} \Rightarrow f(e_0, e_1) = 0 \Rightarrow \varphi_{e_0}(x) = \varphi_{e_1} = \lambda x[0] \Rightarrow (e_0, e_1) \in D$.

Exercise: 2.46.

No. Suppose by way of contradiction that $S = \{e | \varphi_e = \varphi_{e+1}\}$ is an index set. By the recursion theorem there is an e such that $\varphi_e = \varphi_{e+1}$. (Alternatively, apply the fixed point theorem applied to the successor function.) Consequently, there is an $e \in S$. We show by induction that for all $i \in \mathbb{N}$, $\varphi_{e+i} = \varphi_e$. Since $e \in S$, $\varphi_e = \varphi_{e+1}$. Since S is an index set, $\varphi_{e+1} \in S$. Suppose inductively that $\varphi_e = \varphi_{e+1} = \cdots = \varphi_{e+i}$. Then $\varphi_{e+i} \in S$. By the definition of S, $\varphi_{e+i} = \varphi_{e+i+1}$, completing the induction. Consequently, $\{\varphi_i | i \in \mathbb{N}\} = \{\varphi_i | i \leq e\}$. This implies that that are only finitely many partial recursive functions, a contradiction.

Exercise: 2.54.

b) Yes, by the recursion theorem there is a program e such that for any x:

$$\varphi_e(x) = \begin{cases} \downarrow & \text{if } x \neq e, \\ \uparrow & \text{otherwise.} \end{cases}$$

Clearly, $W_e = \mathbb{N} - \{e\}$.

d) Yes. Since K is r.e., by the recursion theorem there is a program e such that for all x:

$$\varphi_e(x) = \begin{cases} \downarrow & \text{if } x = e \text{ or } x \in K, \\ \uparrow & \text{otherwise.} \end{cases}$$

Clearly, $W_e = K \cup \{e\}$. Notice that $\varphi_e(e) \downarrow$. Hence, $e \in K$ and $K \cup \{e\} = K$.

Exercise: 2.60.

Suppose $A \neq \emptyset$ is r.e. and f is a recursive function with range A. Define g such that :

$$g(x) = \varphi_{f(x)}(x) + 1.$$

Since every program in the range of f computes a recursive function, g is recursive. Suppose by way of contradiction $\varphi_y = g$ and $y \in A$. Then there is a z such that $f(z) = y$.
$\varphi_y(z) = g(z) = \varphi_{f(z)}(z) + 1 = \varphi_y(z) + 1$ $(\Rightarrow\Leftarrow)$

Exercise: 2.67.

Suppose A is an infinite recursively enumerable set. Let f be a recursive function with range A. Define:

$$g(0) = f(0)$$
$$g(x+1) = f(\mu y[f(y) > g(x)])$$

Since A is infinite, g is recursive. Furthermore, the range of g is a subset of the range of f, so the range of g is a subset of A. Note that g is monotone increasing. To decide if x is in the range of g, find the least y such that $x \leq g(y)$. Such a y must exist since g is monotone increasing. Now, x is in the range of g iff $x = g(y)$. Hence, the range of g is an infinite, recursive subset of A.

Exercise: 2.88.

a) Suppose by way of contradiction that $A = \{\langle x, y\rangle | \varphi_x = \varphi_y\}$ is r.e. A is not an index set, so a reduction technique should be used. Choose z and u such that $\varphi_z = \lambda x[0]$ and $\varphi_u = \lambda x[\uparrow]$. By an implicit application of the *s-m-n* theorem, there is a recursive function g such that:

$$\varphi_{g(x)}(y) = \begin{cases} 0 & \text{if } x \in K, \\ \uparrow & \text{otherwise.} \end{cases}$$

For any x, either $\varphi_{g(x)} = \varphi_z$ or $\varphi_{g(x)} = \varphi_u$, depending on whether or nor $x \in K$. So, to decide if $x \in K$ wait to see which pair, $\langle g(x), z\rangle$ or $\langle g(x), u\rangle$ shows up first in the enumeration of A. One of the two pairs must show up, hence if A is r.e. then K is recursive, a contradiction.

c) Suppose by way of contradiction that the given set $C = \{x | (\exists y, z)\varphi_x(y) \downarrow$ and $\varphi_y(z) \uparrow\}$ is r.e. Let z be such that $\varphi_z = \lambda x[0]$. By the mutual recursion theorem, there are programs e_0 and e_1 such that:

$$\varphi_{e_0}(x) = \begin{cases} 0 & \text{if } x = e_1 \\ \uparrow & \text{otherwise.} \end{cases}$$

$$\varphi_{e_1}(x) = \begin{cases} 0 & \text{if } e_0 \in C, \\ \uparrow & \text{otherwise.} \end{cases}$$

Note that $W_{e_0} = \{e_1\}$.
$e_0 \in C \Rightarrow (\exists z)\varphi_{e_1}(z) \uparrow \Rightarrow \varphi_{e_1} = \lambda x[\uparrow] \Rightarrow e_0 \notin C$ $(\Rightarrow\Leftarrow)$
$e_0 \notin C \Rightarrow \varphi_{e_1} = \lambda x[\uparrow] \Rightarrow e_0 \in C$ $(\Rightarrow\Leftarrow)$

d) Suppose by way of contradiction $D = \{x | (\exists y)\varphi_x(y) \downarrow$ and φ_y is total$\}$ is r.e. By an implicit use of the *s-m-n* theorem there is a recursive function f such that $W_{f(i)} = \{i\}$. Define a partial function:

$$\psi(y) = \begin{cases} 1 & \text{if } f(y) \in D, \\ \uparrow & \text{otherwise.} \end{cases}$$

Since D is assumed to be r.e., ψ is a partial recursive. Hence, the domain of ψ is r.e. Notice that $\psi(y)$ converges iff $f(y) \in D$ iff φ_y is total. Hence, $\{y | \varphi_y$ is total$\}$ is r.e., a contradiction.

g) Suppose by way of contradiction that $G = \{x | W_x$ is infinite$\}$ is r.e. Suppose $W_n = \mathbb{N}$. Then $n \in G$. By Lemma 2.83, there is a finite set $W_f \subset W_n$ with $f \in G$, $(\Rightarrow\Leftarrow)$.

j) Suppose by way of contradiction that $J = \{x | W_x$ is finite $\}$ is r.e. Suppose $W_n = \mathbb{N}$ and W_f is finite. Then $f \in J$. By Lemma 2.85, since $W_f \subset W_x$, $x \in J$, $(\Rightarrow\Leftarrow)$.

k) Suppose by way of contradiction that $L = \{x | W_x = \{x\}\}$ is r.e. By the recursion theorem, there is a program e such that

$$\varphi_e(x) = \begin{cases} 1 & \text{if } x = e \text{ or } e \in L, \\ \uparrow & \text{otherwise.} \end{cases}$$

$e \in L \Rightarrow \varphi_e = \lambda x[1] \Rightarrow e \notin L$ $(\Rightarrow\Leftarrow)$
$e \notin L \Rightarrow W_e = \{e\} \Rightarrow e \in L$ $(\Rightarrow\Leftarrow)$

o) Suppose by way of contradiction that $O = \{x | cardinality(W_x) = 1\}$ is r.e. Let l be such that $W_l = \{0\}$. Clearly, $l \in O$. Let n be such that $W_n = \mathbb{N}$. Then $W_l \subset W_n$, so by Lemma 2.85, $n \in O$, $(\Rightarrow\Leftarrow)$.

p) Clearly, $P = \{x | card(W_x) > 10\}$ is an index set. Let A be the set of all x such that D_x has cardinality 10. If $W_y \supset D_x$ for some $x \in A$, then it has cardinality at least 10. Furthermore, of W_y has cardinality at least 11, then there is a $D_x \subset W_y$ for some $x \in A$. Therefore, P is r.e.

Choose n and t such that $W_n = \mathbb{N}$ and $W_t = \{x | x < 10\}$. Then $t \in P$ and $n \notin P$, therefore, by Theorem 2.48 , P is not recursive.

q) Suppose by way of contradiction that $Q = \{x | (\exists y)\varphi_x(y) \uparrow \text{ and } y \in K\}$ is r.e. Choose u and z such that $\varphi_u = \lambda x[\uparrow]$ and $\varphi_z = \lambda x[0]$. Notice that $u \in Q$ and $z \notin Q$. However, $\varphi_u \subset \varphi_z$, hence by exercise 2.87, $z \in Q$, $(\Rightarrow\Leftarrow)$.

r) Suppose by way of contradiction $R = \{x | W_x \subseteq K\}$ is r.e. Choose n such that $W_n = \mathbb{N}$. Since R is clearly not empty, by Lemma 2.85 , $n \in R$, $(\Rightarrow\Leftarrow)$.

s) This is same set as Q above in disguise.

v) The answer depends on the acceptable programming system in use.

Exercise: 3.1.

Let $P_0, P_1, \ldots$ be an infinite sequence of RAM programs such that on all inputs, P_i puts i in register R_1 then clears R_1 and halts. Clearly, $\forall i \forall x, S_i(x) \geq i$ and $\varphi_i(x) = 0$. Now suppose there exists a function g such that for all i, $S_i(x) \leq g(x, \varphi_i(x))$. Then for all the functions described above $S_i(x) \leq g(x, 0)$, but this is a contradiction. Consider $x = 1$. $g(1,0)$ is finite and $S_{g(1,0)+1}(1) > g(1,0)$.

Exercise: 3.11.

Notice that if $\Phi_i = g_j$ for some j, then $\varphi_i = f_i$, a recursive function. In this case, the domain of Φ_i is $\mathbb{N}$, the same as the domain of g_j. If $\Phi_i = \Psi_i$, then Φ_i, Ψ_i and φ_i all have the same domain. Hence, the first axiom of abstract complexity measures holds for Φ_0, Φ_1, $\cdots$. Let $R(i,x,y)$ be the recursive predicate that is guaranteed to exist by the second axiom of abstract complexity measures for the complexity measure Ψ_0, Ψ_1, $\cdots$. Define:

$$\hat{R}(i,x,y) = \begin{cases} 1 & \text{if } i = F(j) \text{ and } y \geq g_j, \\ 0 & \text{if } i = F(j) \text{ and } y < g_j, \\ R(i,x,y) & \text{otherwise.} \end{cases}$$

Since F is monotone increasing, its range is a recursive set. Hence, $\hat{R}$ is recursive and it satisfies the second axiom of abstract complexity measures for Φ_0, Φ_1, $\cdots$.

Exercise: 3.17.

Suppose such a recursive function b exists. Then by implicit use of the recursion theorem there is a program E such that:

$$\varphi_e(x) = \begin{cases} 0 & \text{if } \Phi_e(x) > b(x,0) \\ \uparrow & \text{otherwise.} \end{cases}$$

Notice that $\varphi_e(x)$ is the everywhere zero function. This is so because Φ_e is a complexity measure and so must have the same domain as φ_e. But by the definition of φ_e, if $\varphi_e(x)\uparrow$, then $\Phi_e(x) \leq b(x,0)$, which is a contradiction. Now, $\forall x, \Phi_i(x) > b(x,0) = b(x,\varphi_e(x))$, so the function b can not exist.

Exercise: 3.21.

The idea is to run the same construction as in Theorem 3.20, only starting with an identifying set of initial values. Let h be as in the hypothesis. For each $j \in \mathbb{N}$, we construct an arbitrarily complex f_j, in effective stages of finite extension, such that if $i \neq j$ then $f_i \neq f_j$. Let j be given. To reduce notational complexity, we will write simply f for f_j. Let f^s denote the finite amount of f determined prior to stage s. let x^s denote the least value not in the domain of f^s. By way of initialization, let $f^0 = \{(y,1) |\ y \leq j\} \cup \{(j+1,0)\}$. Execute the following stages for $s = 0, 1, \cdots$.

> *Stage s.* Let i be the least *uncancelled* program such that $i \leq x^s$ and $\Phi_i(x^s) < h(x^s)$. If there is no such i, set $f(x^s) = 0$. Otherwise, set $f(x^s) = 1 \dot{-} \varphi_i(x^s)$ and cancel program i. Go to stage $s+1$.
> *End stage s.*

The proof that f is h-complex is the same argument as used in the proof of Theorem 3.20. Since the range of f starts with a string of j ones followed by a zero, the f we get for each j is unique. Hence, there are infinitely many f's satisfying Theorem 3.20.

Exercise: 3.22.

The solution to this problem is superficially similar to the proof of Theorem 3.20, but there are several key points of difference.

- In order to make f sparse, there will be stages in which we have to define f on more than one value.
- We *must* test complexity functions Φ_j against all domain values. Specifically, we can not skip over the domain values that we use to make f sparse because there are an infinite number of these and we don't want to produce a function j with $\Phi_j < h$ on these domain values.
- We must be sure that φ_i converges in order to diagonalize against it.

The second point above is a key one. Let x_s be the least undefined domain element of f at the beginning of stage s. A naive approach to this

problem would be to find an i such $\Phi_i(x_s) < h(x_s)$ and to diagonalize. But, if $f(x_s) = 1$ by the diagonalization, then we also have to define f at $x_s + 1, x_s + 2, \ldots x_s + g(x_s)$. Then for stage $s + 1$, $x_{s+1} = x_s + g(x_s) + 1$. There is a problem, though, in skipping over the elements we use to make f sparse. Suppose there exists some $j < i$ with $\varphi_j = f$ and some y among the elements we used to make f sparse, that is, $x_s + 1 < y \leq x_s + g(x_s)$ such that $\Phi_j(y) < h(y)$. Further suppose that by making f sparse (in all stages), we miss an infinity of elements where $\Phi_j < h$. Then we have not diagonalized against function j and the construction fails.

Our construction is by finite extension. Let x_s be the least undefined element in the domain of f at the beginning of stage S, so $x_0 = 0$.

Stage s: If for all $i \leq s$, $\Phi_i(x_s) \geq h(x_s)$ then set $f(x_s) = 0$, set $x_{s+1} = x_s + 1$ and go to stage $s + 1$. Otherwise, let i be the least uncancelled program such that $i \leq s$ and $\Phi_i(x_s) < h(x_s)$.

Now we have to ensure that we do not skip over some function $j < i$ when making f sparse.

Let $y = x_s$.

While there is a $j < i$ and a z such that $y < z \leq y + g(y)$ and $\Phi_j(z) < h(z)$, set $y = z$ and $i = j$.

For x such that $x_s \leq x < y$, set $f(x) = 0$. Set $f(y) = 1 \dot{-} \varphi_i(y)$. For x such that $y < x \leq y + g(y)$, set $f(x) = 0$. Set $x_{s+1} = y + g(y) + 1$ and go to stage $s + 1$.

By our construction, f is sparse. Suppose that there exists a $\varphi_i = f$ and $\Phi_i(x) < h(x)$ for infinitely many x. Let D_s be the elements for which f is defined in stage S. Choose the least stage s such that $i \leq s$ and $\Phi_i(y) < h(y)$ for some $y \in D_s$ and all functions $j < i$ that are ever canceled are canceled before stage s. Then $f(y)$ will be made $\neq \varphi_i(y)$ at stage s, a contradiction.

Exercise: 3.25.

See C. Smith, A note on arbitarily complex recursive functions, *Notre Dame Journal of Formal Logic*, Vol. 29, 1988, pp. 198–207.

Exercise: 3.27.

Modify the proof of the gap theorem (Theorem 3.26) as follows. Instead of finding $y_0 < y_1 < \cdots < y_x$, find $y_0 < y_1 < \cdots y_{b(x)+x}$ in the same way. Since $y_{b(x)} > b(x)$ by the monotonicity of g, there is a string of at least $x + 1$ larger and larger values, each larger than $b(x)$ in the y's just constructed. The result follows.

Exercise: 3.40.

Theorem: (Complexity Theoretic Fixed Point Theorem) There are recursive functions r and h such that h is monotone nondecreasing in its second argument and $\forall i$ such that φ_i is total:

(1) $\forall x \quad \varphi_{r(i)}(x) = \varphi_{\varphi_i(r(i))}(x)$ and

(2) $\overset{\infty}{\forall} x \in \text{domain } \varphi_{r(i)}, \quad \Phi_{r(i)}(x) \leq h(x, \Phi_{\varphi_i(r(i))}(x))$.

Proof: The r is supplied by the fully effective proof of the fixed point theorem. Define:

$$k(i,x,y) = \begin{cases} \Phi_{r(i)}(x) & \text{if } \Phi_{\varphi_i(r(i))}(x) = y \\ 0 & \text{otherwise.} \end{cases}$$

Note that if $\Phi_{\varphi_i(r(i))}(x) = y$ then $\varphi_{\varphi_i(r(i))}(x)$ is defined and consequently, $\varphi_{r(i)}(x)$ is defined. Hence k is recursive. Let

$$h(x,y) = \max_{i \leq x} \max_{z \leq y} k(i,x,z).$$

The function h is clearly monotone nondecreasing. Now, suppose $x \in$ domain $\varphi_{r(i)}$ and $x \geq i$. Then:

$$\Phi_{r(i)}(x) = k(i,x,\Phi_{\varphi_i(r(i))}(x)) \leq h(x, \Phi_{\varphi_i(r(i))}(x)).$$

Exercise: 3.46.

Given an i and j such that $\varphi_j(i) = 1$ and ($\varphi_j(x) = 1$ implies $\varphi_x = \varphi_i$), construct via the parametric recursion theorem the following function p:

$$\varphi_{p(k)}(x) = \begin{cases} \varphi_i(x) & \text{if } \varphi_j(p(k)) = 0, \\ \varphi_i(x) + 1 & \text{otherwise.} \end{cases}$$

We claim that $\varphi_j(p(j)) = 0$. Suppose by way of contradiction $\varphi_j(p(j)) = 1$. From the definition of j we have that $\varphi_{p(j)} = \varphi_i$. However, from the definition of p, we have that $\varphi_{p(j)} = \lambda x[\varphi_i(x) + 1]$, ($\Rightarrow\Leftarrow$)

Exercise: 4.9.

By an implicit application of the *s-m-n* theorem, there is a recursive function f such that:

$$\varphi_{f(x)} = \lambda y \left[\sum_{z=0}^{y} \varphi_x(z) \right].$$

If φ_x is total, then so is $\varphi_{f(x)}$. If $\varphi_x(y)$ is undefined, then $W_{f(x)} \subset \{z | z < y\}$.

Exercise: 4.23.

Yes. Program $f(\langle i,j \rangle)$, on input x, runs program i, on input x, for $j \cdot x^j$ steps, outputting the answer if the simulation converges and zero otherwise. Clearly, $\varphi_{f(\langle i,j \rangle)} \in P$, for all values of i and j. Suppose $g \in P$ as witnessed by program i and polynomial p. Choose j such that $j \cdot x^j \geq p(x)$ for all x. Then $\varphi_{f(\langle i,j \rangle)} = g$.

Exercise: 4.32.

Let Φ_0, Φ_1, be the complexity measure formed by counting steps in Turing machine computations. Let $\tilde{x}$ denote the lexicographically least input of size x. Define a recursive function f as follows:

$$f(x) = \begin{cases} 1 & \text{if } \Phi_x(\tilde{x}) = 0, \\ 0 & \text{otherwise.} \end{cases}$$

Suppose by way of contradiction that f is time constructible. Then there is an i such that, for any x, $f(x) = \Phi_i(\tilde{x})$. A contradiction is obtained by noting that $f(i) \neq \Phi_i(\tilde{i})$.

Exercise: 4.33.

Let r be given. Consider a RAM program that, on input x, first computes $z = \max\{r(y) \mid |y| = |x|\}$. Then the program decrements z to 0. The running time of this RAM program, on input x, is z plus the time taken to compute the max, a quantity that is larger than $r(x)$. Furthermore, given any two inputs of the same size, the running time of the program will be the same. The running time of this program is time constructible by definition.

Exercise: 4.34.

No. Suppose by way of contradiction that f is a recursive function such that $\{\varphi_{f(i)} \mid i \in \mathbb{N}\}$ is precisely the time constructible functions. Define a recursive function r as follows:

$$r(x) = 1 + \sum_{i=0}^{x} \varphi_{f(i)}(x).$$

Since all the time constructible functions are total, r is recursive. By exercise 4.33, there is a time constructible function r' such that $r'(x) \geq r(x)$, for all x. Hence, there is an i such that $\varphi_{f(i)} = r'$. For all $x \geq i$, $\varphi_{f(i)}(x) = r'(x) \geq r(x) > \varphi_{f(i)}(x)$, $(\Rightarrow\Leftarrow)$

Exercise: 4.60.

R may be as large as $c^{S(n)}$. Each of these configurations takes up to $S(n)$ space to store. $c^{S(n)} \cdot S(n) \neq O(S(n))$.

List of Symbols

Index