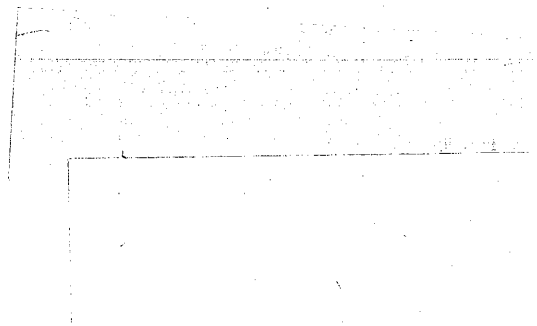
MATHEMATICAL LOGIC AND COMPUTABILITY

H. Jerome Keisler

Joel Robbin

Contributors:
Arnold Miller
Kenneth Kunen
Terrence Millar
Paul Corazza

The Wisconsin Logic Group
University of Wisconsin, Madison



The McGraw-Hill Companies, Inc

New York St. Louis San Francisco Auckland Bogotá Caracas Lisbon
London Madrid Mexico City Milan Montreal New Delhi
San Juan Singapore Sydney Tokyo Toronto

McGraw-Hill
A Division of The McGraw-Hill Companies

MATHEMATICAL LOGIC AND COMPUTABILITY

This book is printed on acid-free paper.

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

P/N 033939-2
PART OF
ISBN 0-07-912931-5

The editor was Jack Shira;
the production supervisor was Paula Keller.
R. R. Donnelley & Sons Company was printer and binder.

Library of Congress Catalog Card Number: 95-80649

Contents

Preface

This course is concerned with the two broad topics of logic and computability and the relationship between them. Classical propositional and predicate logic is the topic for the first three chapters of the text, the theory of computable functions occupies the fourth chapter, and the relationship between the two, as embodied in the Incompleteness Theorems of Gödel, comprises the fifth chapter.

A package of computer programs called **Logiclab** is included with this book. The package contains both DOS and Windows versions of four programs. The DOS versions are TABLEAU, COMPLETE, PREDCALC, and GNUMBER, and the Windows versions are TABWIN, COMPWIN, PREDWIN, and GNUMWIN. These programs are keyed to the book and are designed to be used for problems, student experimentation, and classroom demonstrations. They work on an IBM PC or compatible personal computer. Many of the problem sets in this book use the Logiclab programs. The Windows versions work with Windows 3.0 or later, and with Windows 95, and have built-in tutorials which will quickly show you how to use the programs. Complete instructions for the programs are included in the appendices at the end of the book.

While there are no specific mathematical prerequisites for the book, some experience with abstract mathematical proof is crucial. Some basic mathematical concepts used in this text are explained in Appendix A. The material of Chapters 2 and 3 will be more meaningful to the student who has had a course in linear algebra or abstract algebra.

About the Authors

Jerome Keisler, Kenneth Kunen, Terrence Millar, Arnold Miller, and Joel Robbin are Professors of Mathematics at the University of Wisconsin in Madison. Kenneth Kunen is also a Professor of Computer Science, and Terrence Millar is also a Dean in the Graduate School. Paul Corazza has taught at the University of Wisconsin and is a Professor of Mathematics at Maharishi International University. The authors have published numerous textbooks and research articles in mathematical logic and other areas of mathematics. Jerome Keisler received his Ph.D. degree from the University of California at Berkeley, Paul Corazza from Auburn University, Kenneth Kunen from Stanford University, Terrence Millar from Cornell University, Arnold Miller from the University of California at Berkeley, and Joel Robbin from Princeton University.

MATHEMATICAL LOGIC AND COMPUTABILITY

Chapter 1

Propositional Logic

This book is about *formal languages* which are powerful enough for the development of mathematics. Unlike natural languages such as English, formal languages have a precise set of rules for forming sentences. This set of rules is called the **syntax** of the language.

In this chapter we study a very simple formal language called **propositional logic**. The main topics will be well formed formulas (or wffs), formal tableau proofs, and models. These concepts will be tied together at the end of the chapter with the Completeness Theorem. At the end of the chapter there are two problem sets. One problem set uses the TABLEAU program and is done on a computer. It gives the student a set of examples of formal tableau proofs, and some experience in building such proofs. The other problem set is a collection of pencil and paper problems.

1.1 Introduction

In propositional logic one can build new statements out of old statements using **propositional connectives**. These connectives are *not*, *and*, *or*, *if*, and *if and only if*. We are only concerned with the common mathematical meanings of these connectives. In some cases the mathematical meaning is slightly different from the meaning in everyday English. We now explain these meanings.

NEGATION. A sentence of form *'not p'* is true when p is false, and is false when p is true. The symbol used in mathematical logic for *not* is $\neg$. Of the two sentences

$$\neg\ 2+2=4$$

$$\neg\ 2+2=5$$

the first is false while the second is true. The sentence $\neg p$ is called the **negation** of p.

CONJUNCTION. A sentence of form *'p and q'* is true exactly when both p and q are true. The mathematical symbol for *and* is $\wedge$ (or sometimes &). Of the four sentences

$$2+2=4 \ \wedge\ 2+3=5$$
$$2+2=4 \ \wedge\ 2+3=7$$
$$2+2=6 \ \wedge\ 2+3=5$$
$$2+2=6 \ \wedge\ 2+3=7$$

the first is true and the last three are false. The sentence $p \wedge q$ is called the **conjunction** of p and q.

The words *and* and *but* have the same meaning for the mathematician. For example, the statement

"$\pi > 3$ but $\pi < 3.2$"

has the same mathematical meaning as the statement

"$\pi > 3$ and $\pi < 3.2$."

DISJUNCTION. A sentence of form *'p or q'* is true exactly when at least one of the sentences p, q is true.

The symbol used in mathematical logic for *or* is $\vee$. Of the four sentences

$$2+2=4 \ \vee\ 2+3=5$$
$$2+2=4 \ \vee\ 2+3=7$$
$$2+2=6 \ \vee\ 2+3=5$$
$$2+2=6 \ \vee\ 2+3=7$$

the first three are true while the last is false. The sentence $p \vee q$ is called the **disjunction** of p and q.

In everyday usage, the phrase *soup or salad included* in a restaurant menu means that the customer can have either soup or salad with his/her dinner at no extra cost but not both. This usage of the word *or* is called **exclusive** (because it excludes the case where both components are true). On the other hand, the question *Do you want cream or sugar with your coffee?* means cream or sugar or both. This is the **inclusive** meaning of the word *or*, and is sometimes written *and/or* in English. Mathematicians always use the inclusive meaning; when they intend the exclusive meaning they say so explicitly as in *p or q but not both.*

IMPLICATION. *'p implies q'* is false exactly when p is true but q is false. The mathematical symbol for "implies" is $\Rightarrow$. Of the four sentences

$$2+2=4 \ \Rightarrow\ 2+3=5$$
$$2+2=4 \ \Rightarrow\ 2+3=7$$
$$2+2=6 \ \Rightarrow\ 2+3=5$$
$$2+2=6 \ \Rightarrow\ 2+3=7$$

the second is false and the first, third and fourth are true.

The forms *'p implies q'*, *'if p, then q'*, *'q, if p'*, *'p only if q'*, and *'q whenever p'* all have the same meaning for the mathematician.

This usage is in sharp contrast to the usage in everyday language. In common discourse a sentence of form *if p then q* or *p implies q* suggests that there is a causal relationship between p and q. Consider for example the sentence

If Columbus discovered America, then Aristotle was a Greek.

Since Aristotle was indeed a Greek this sentence either has form *If true then true* or *If false then true* and is thus true according to the meaning of *implies* we have adopted. However, common usage would judge this sentence either false or nonsensical because there is no causal relation between Columbus's voyage and Aristotle's nationality.

The mathematical usage of p implies q is much simpler than the everyday usage. The main advantage of the mathematical usage is that the truth value of p implies q depends only on the truth values of p and q, and not on other aspects of p and of q.

EQUIVALENCE. The forms *'p if and only if q'*, *'p is equivalent to q'*, and *'p exactly when q'* all have the same meaning for the mathematician: they are true when p and q have the same truth value and false when p and q have different truth values.

Sometimes *iff* is used as an abbreviation for *if and only if*. The mathematical symbol for *if and only if* is $\Leftrightarrow$. Equivalence is the *equality* of propositional logic, because $p \Leftrightarrow q$ says that the truth values of p and q are equal to each other.

Of the four sentences

$$\begin{aligned} 2+2=4 &\Leftrightarrow 2+3=5 \\ 2+2=4 &\Leftrightarrow 2+3=7 \\ 2+2=6 &\Leftrightarrow 2+3=5 \\ 2+2=6 &\Leftrightarrow 2+3=7 \end{aligned}$$

the first and last are true while the other two are false.

The statement *p if and only if q* has the same meaning as *if p then q and if q then p*.

For each of the connectives which we have introduced, the truth value of the new sentence depends in a simple way on the truth values of the original sentences. The rules for truth values are summarized in the following tables.

A	¬A
T	F
F	T

A	B	A ∧ B	A ∨ B	A ⇒ B	A ⇔ B
T	T	T	T	T	T
T	F	F	T	F	F
F	T	F	T	T	F
F	F	F	F	T	T

1.2 Syntax of Propositional Logic

In this section we give the grammatical rules for propositional logic.

A **vocabulary for propositional logic** is a non-empty set $\mathcal{P}_0$ of **proposition symbols**, which are denoted by lower case letters $p, q, r, s, p_1, q_1, \ldots$. The proposition symbols will stand for propositions, which are simple statements which may be combined to form other statements. Propositional logic is not concerned with any *internal* structure these propositions may have; indeed, for us the only meaning a proposition symbol may take is a truth value – either *true* or *false*.

We start our development of propositional logic by giving a list of primitive symbols which includes the vocabulary, the connectives, and two brackets which will be used in the same way that parentheses are used in algebra.

The primitive symbols of the **propositional logic** are:

- proposition symbols $p, q, r, \ldots$ from $\mathcal{P}_0$
- the **negation sign** $\neg$
- the **conjunction sign** $\wedge$
- the **disjunction sign** $\vee$
- the **implication sign** $\Rightarrow$
- the **equivalence sign** $\Leftrightarrow$
- the **left bracket** [
- the **right bracket**].

Any finite sequence of these symbols is called a **string**. Here are some examples of strings:

$$[p \wedge q] \qquad p \wedge q]] \qquad [p\wedge \qquad]] \wedge \wedge.$$

Our first task is to specify the **syntax** of propositional logic: we must say which strings are grammatically correct. These strings are

called **well-formed formulas**, or more briefly, **wffs**. If we wish to be specific about exactly which proposition symbols may appear in a wff **A** we say it is a **wff in the vocabulary** $\mathcal{P}_0$.

Definition 1.2.1 Let $\mathcal{P}_0$ be a set of proposition symbols. A **wff of propositional logic** with the vocabulary $\mathcal{P}_0$ is a string which can be obtained by finitely many applications of the following **rules of formation**:

(W:$\mathcal{P}_0$) If $p \in \mathcal{P}_0$, then p is a wff;

(W:$\neg$) If **A** is a wff, then $\neg$**A** is a wff;

(W:$\wedge$) if **A** is a wff and **B** is a wff, then [**A** $\wedge$ **B**] is a wff;

(W:$\vee$) if **A** is a wff and **B** is a wff, then [**A** $\vee$ **B**] is a wff;

(W:$\Rightarrow$) if **A** is a wff and **B** is a wff, then [**A** $\Rightarrow$ **B**] is a wff;

(W:$\Leftrightarrow$) if **A** is a wff and **B** is a wff, then [**A** $\Leftrightarrow$ **B**] is a wff.

For example the string $[p \vee q]$ can be built using the rules of formation and hence is a wff.

However, the strings $p \vee q$, $[p] \vee [q]$, $\vee pq$ (which correspond to $[p \vee q]$ in other treatments of propositional logic) cannot be built up in this way and are not wffs.

We can show that a particular string **A** is a wff by using the rules of formation repeatedly in a step by step manner. When we do this we get a sequence of strings, called a **parsing sequence** for **A**. A string which is not a wff cannot have a parsing sequence.

For example, we show that the string $[\neg p \Rightarrow [q \wedge p]]$ is a wff by giving a parsing sequence.

(1) p is a wff by (W:$\mathcal{P}_0$).

(2) q is a wff by (W:$\mathcal{P}_0$).

(3) $[q \wedge p]$ is a wff by (1), (2), and (W:$\wedge$).

(4) $\neg p$ is a wff by (1) and (W:$\neg$).

(5) $[\neg p \Rightarrow [q \wedge p]]$ is a wff by (3), (4), and (W:$\Rightarrow$).

Most wffs have several different parsing sequences. We must always start with one of the proposition letters, build up in some order from simpler to more complex wffs, and end with the string which we want to show is a wff. Here is another parsing sequence for the wff $[\neg p \Rightarrow [q \wedge p]]$.

(1) q is a wff by (W:$\mathcal{P}_0$).

(2) p is a wff by (W:$\mathcal{P}_0$).

(3) $\neg p$ is a wff by (2) and (W:$\neg$).

(4) $[q \wedge p]$ is a wff by (1), (2), and (W:$\wedge$).

(5) $[\neg p \Rightarrow [q \wedge p]]$ is a wff by (3), (4), and (W:$\Rightarrow$).

As the example illustrates, a parsing sequence for a string **S** is a finite sequence of strings $\mathbf{S}_1, \ldots, \mathbf{S}_n$ such that the last string $\mathbf{S}_n$ is **S**, and each string $\mathbf{S}_i$ in the sequence is either a proposition symbol, is the negation of an earlier string in the sequence, or is built from two earlier strings in the sequence using a binary connective. By applying the definition of a wff at each step, we see that each string $\mathbf{S}_i$ in the sequence is a wff, and hence the final string **S** is a wff.

To **parse** a wff is to find a parsing sequence for the wff.

We shall use bold-face upper-case letters near the beginning of the alphabet like **A**, **B**, **C** to denote arbitrary wffs. Other bold-face upper-case letters like **S**, **U** will denote strings which might or might not be wffs.

1.3 Induction on Length of Wffs

Many times in this book we shall use the idea of the length of a wff. The **length** of a string of symbols

$$\mathbf{S} = s_1 \ldots s_m$$

is the number m. The empty string has length zero. The only wffs of length one are the propositional symbols.

Quite often we shall prove some fact about wffs by induction on the length of wffs. We illustrate this method with a simple example. It will be useful to use an asterisk $*$ to stand for one of the four binary connectives $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$.

Proposition 1.3.1 *Every wff has the same number of left brackets as right brackets.*

Proof: Let us call a wff *balanced* if it has the same number of left as right brackets. Every wff of length one is balanced because the only wffs of length one are propositional symbols, which have no brackets. Assume that every wff of length at most n is balanced. Let $\mathbf{A}$ be a wff of length $n+1$. There are two cases:

Case 1: $\mathbf{A} = \neg\mathbf{B}$. $\mathbf{B}$ is a wff of length at most n and hence is balanced. $\mathbf{A}$ has the same brackets as $\mathbf{B}$, so $\mathbf{A}$ is also balanced.

Case 2: $\mathbf{A} = [\mathbf{B} * \mathbf{C}]$ where $*$ is a binary connective. $\mathbf{B}$ and $\mathbf{C}$ are wffs of length at most n and hence are balanced. The number of left brackets in $\mathbf{A}$ is equal to the number of left brackets in $\mathbf{B}$ plus the number of left brackets in $\mathbf{C}$ plus one, and the number of right brackets in $\mathbf{A}$ is the same, so $\mathbf{A}$ is balanced.

We have assumed that all wffs of length at most n are balanced, and proved that all wffs of length at most $n+1$ are balanced. By induction, all wffs are balanced. **End of Proof.**

The following fact turns out to be very useful and will be proved by a somewhat harder induction on the length of a wff.

Proposition 1.3.2 *If* $\mathbf{C}$ *is a wff of propositional logic, then no string which is obtained by removing one or more symbols at the end of* $\mathbf{C}$ *is a wff.*

Before giving the proof, we shall rephrase the proposition and give an example.

A string $\mathbf{T}$ is said to be an **initial part** of a string $\mathbf{S}$ if $\mathbf{T}$ is formed by removing one or more symbols at the end of $\mathbf{S}$.

We shall often use the notation $\mathbf{TU}$ to mean the string $\mathbf{T}$ followed by the string $\mathbf{U}$. If $\mathbf{T}$ is a string of length m and $\mathbf{U}$ is a string of length n, then $\mathbf{TU}$ will be a string of length $m+n$.

Thus $\mathbf{T}$ is an initial part of $\mathbf{S}$ if $\mathbf{S} = \mathbf{TU}$ for some string $\mathbf{U}$ which is not empty.

Proposition 1.3.2 says that: *no initial part of a wff of propositional logic is a wff.*

Here is an example. The initial parts of the wff

$$[[p \Rightarrow [q \wedge p]] \Rightarrow q]$$

are the empty string and the strings

$$[,\quad [[,\quad [[p,\quad [[p \Rightarrow,\quad [[p \Rightarrow [,\quad [[p \Rightarrow [q,\quad [[p \Rightarrow [q\wedge,$$
$$[[p \Rightarrow [q \wedge p,\quad [[p \Rightarrow [q \wedge p],\quad [[p \Rightarrow [q \wedge p]],\ [[p \Rightarrow [q \wedge p]] \Rightarrow,$$
$$[[p \Rightarrow [q \wedge p]] \Rightarrow q.$$

None of these initial parts are wffs. The whole wff has length 13, and the initial parts have lengths 0 through 12.

Proof of Proposition 1.3.2: We prove by induction on n that no initial part of a wff of length at most n is a wff. This is true for $n = 1$ because the only initial part of a wff of length 1 is the empty string, which is not a wff. Assume that no initial part of a wff of length at most n is a wff. Let $\mathbf{A}$ be a wff of length $n+1$. We must prove that no initial part of $\mathbf{A}$ is a wff. There are two cases:

Case 1: $\mathbf{A}$ is $\neg\mathbf{B}$. We assume that an initial part $\mathbf{D}$ of $\mathbf{A}$ is a wff and get a contradiction. We have $\mathbf{A} = \mathbf{DT}$ where $\mathbf{T}$ is not empty. $\mathbf{D}$ is a wff starting with $\neg$, so $\mathbf{D} = \neg\mathbf{E}$ where $\mathbf{E}$ is a wff. Removing the initial $\neg$ symbols from $\mathbf{A} = \mathbf{DT}$, we get $\mathbf{B} = \mathbf{ET}$. But then $\mathbf{B}$ is a wff of length at most n which has a wff $\mathbf{E}$ as an initial part. This contradicts our inductive hypothesis. Therefore no initial part of $\mathbf{A}$ is a wff.

Case 2: $\mathbf{A}$ is $[\mathbf{B}*\mathbf{C}]$ where $*$ is a binary connective. We assume that an initial part $\mathbf{D}$ of $\mathbf{A}$ is a wff and get a contradiction. $\mathbf{A} = \mathbf{DT}$ where $\mathbf{T}$ is nonempty. $\mathbf{D}$ is a wff starting with $[$, so $\mathbf{D} = [\mathbf{E} \circ \mathbf{F}]$ for some binary connective $\circ$ and some wffs $\mathbf{E}$ and $\mathbf{F}$. Then $\mathbf{B} * \mathbf{C}] = \mathbf{E} \circ \mathbf{F}]\mathbf{T}$. Both $\mathbf{B}$ and $\mathbf{E}$ are wffs of length at most n. By our inductive hypothesis, neither of $\mathbf{B}$, $\mathbf{E}$ can be an initial part of the other. Since $\mathbf{B}$ and $\mathbf{E}$ start at the same place within $\mathbf{A}$, they must be the same, $\mathbf{B} = \mathbf{E}$.

Therefore $\mathbf{B} * \mathbf{C}] = \mathbf{B} \circ \mathbf{F}]\mathbf{T}$, so $* = \circ$ and $\mathbf{C}] = \mathbf{F}]\mathbf{T}$. But then the wff $\mathbf{F}$ is proper initial part of the wff $\mathbf{C}$ of length at most n. This contradicts our inductive hypothesis. Therefore no initial part of $\mathbf{A}$ is a wff. **End of Proof.**

1.4 Main Connective

In order to assign meanings to wffs we need to know that each wff can be read in exactly one way. This will be shown by the **Unique Readability Theorem**, which will be proved rather easily from the preceding proposition.

Each wff is either a propositional symbol, starts with a negation symbol, or starts with a left bracket. A wff $\neg\mathbf{A}$ is the negation of the shorter wff $\mathbf{A}$. Wffs which start with a left bracket are more complicated, but they are also built up from shorter wffs. We shall see that every wff which is not already a proposition symbol can be broken down into shorter wffs in a unique way.

Consider a wff $\mathbf{C}$ which starts with a left bracket. $\mathbf{C}$ must have been built from two other wffs using a binary connective. This binary connective must be introduced in the last step of a parsing sequence, and is called the **main connective** of $\mathbf{C}$. It is clear that $\mathbf{C}$ has a main connective. The Unique Readability Theorem will show that $\mathbf{C}$ has only one main connective. This is the key fact we need in order to break each wff down into simpler wffs in a unique way.

For example, the main connective of the wff

$$[[p \Rightarrow [q \wedge p]] \Rightarrow q]$$

is the second occurrence of $\Rightarrow$. The given wff is built from the two shorter wffs

$$[p \Rightarrow [q \wedge p]], \qquad q$$

using the connective $\Rightarrow$.

In this example, the connective $\Rightarrow$ occurs twice in the wff, but only the second occurrence counts as the main connective.

Definition 1.4.1 We say that an occurrence of a binary connective $*$ is a **main connective** of a wff $\mathbf{C}$ if $\mathbf{C} = [\mathbf{A} * \mathbf{B}]$ where $\mathbf{A}$ and $\mathbf{B}$ are wffs.

Theorem 1.4.2 (Unique Readability) *Each propositional wff $\mathbf{C}$ which begins with a left bracket has exactly one main connective.*

Proof: We consider the case where $\mathbf{A}$ is a wff of the form $[\mathbf{B} * \mathbf{C}]$ for some wffs $\mathbf{B}$ and $\mathbf{C}$ and binary connective $*$. Suppose that $\mathbf{A}$ is also equal to $[\mathbf{D} \circ \mathbf{E}]$ where $\mathbf{D}$ and $\mathbf{E}$ are wffs and $\circ$ is a binary connective. The wffs $\mathbf{B}$ and $\mathbf{D}$ are strings which both start at the same place, right after the first left bracket in $\mathbf{A}$. By Proposition 1.3.2, one of $\mathbf{B}$, $\mathbf{D}$ cannot be an initial part of the other. Therefore $\mathbf{B} = \mathbf{D}$. It follows that $* = \circ$ and $\mathbf{C} = \mathbf{E}$. **End of Proof.**

Exercise 4 gives a useful rule for finding the main connective of a wff $\mathbf{C}$: An occurrence of a connective $*$ is the main connective of $\mathbf{C}$ if and only if $\mathbf{C}$ has the form $[\mathbf{S} * \mathbf{T}]$ where $\mathbf{S}$ has the same number of left brackets as right brackets.

To make our wffs more readable, we shall introduce abbreviated wffs. These are strings which are not wffs according to our definition, but are usually shorter and easier for people to read, and can always be translated into a full wff.

Rules for Abbreviating Wffs

- The outermost brackets of a wff need not be written. For example, we may write $p \vee q$ as an abbreviation for the wff $[p \vee q]$, and write $p \Leftrightarrow [q \vee r]$ as an abbreviation for the wff $[p \Leftrightarrow [q \vee r]]$.
- We define the **precedence** of the binary connectives by the list

  $$\wedge, \vee, \Rightarrow, \Leftrightarrow,$$

  with $\wedge$ being of highest precedence and $\Leftrightarrow$ lowest. If $*$ and $\circ$ are two binary connectives with $*$ having higher precedence than $\circ$, and $\mathbf{A}, \mathbf{B}, \mathbf{C}$ are wffs, then $\mathbf{A} * \mathbf{B} \circ \mathbf{C}$ means $[[\mathbf{A} * \mathbf{B}] \circ \mathbf{C}]$, and $\mathbf{A} \circ \mathbf{B} * \mathbf{C}$ means $[\mathbf{A} \circ [\mathbf{B} * \mathbf{C}]]$.

For example, $p \wedge q \vee r$ is an abbreviation for $[[p \wedge q] \vee r]$ rather than for $[p \wedge [q \vee r]]$, since $\wedge$ has a higher precedence than $\vee$.

The string which is obtained from a wff **C** by using the preceding rules whenever possible is called the **standard abbreviation** of **C**. The standard abbreviation of a wff is usually easier to read than the full wff. For this reason, the TABLEAU program always displays the standard abbreviation of a wff.

Proposition 1.3.2 is not true for abbreviated wffs. Abbreviated wffs frequently have initial parts which are abbreviated wffs, or even full wffs. For example, consider the string $\mathbf{S} = p \vee q \wedge r$. **S** is not a wff, but it is an abbreviation for the wff $[p \vee [q \wedge r]]$. The wff p is an initial part of **A**. The string $p \vee q$, which is an abbreviation for the wff $[p \vee q]$, is another initial part of **S**.

Given the standard abbreviation **C**′ of a wff **C**, it is always possible to recover the original wff **C** . Exercise 9 gives an easy way to do this, by finding which symbol of **C**′ corresponds to the main connective of the original wff **C**.

In defining the standard abbreviation, we have not changed our notion of a wff. We shall always use full wffs in the original sense when proving theorems about wffs, but will often use the abbreviated form when discussing particular examples.

There are two other conventions which we shall sometimes use to improve readability.

The first of these conventions involves repeated $\wedge$ or repeated $\vee$ connectives. We may write $\mathbf{A} \wedge \mathbf{B} \wedge \mathbf{C}$ instead of $[\mathbf{A} \wedge \mathbf{B}] \wedge \mathbf{C}$. Similarly, we may write $\mathbf{A} \vee \mathbf{B} \vee \mathbf{C}$ instead of $[\mathbf{A} \vee \mathbf{B}] \vee \mathbf{C}$. Note that $[[p \wedge q] \wedge r]$ and $[p \wedge [q \wedge r]]$ are two different wffs. The string $p \wedge q \wedge r$ is an abbreviation for the first wff $[[p \wedge q] \wedge r]$, but not for the second wff $[p \wedge [q \wedge r]]$. This convention is particularly useful when we wish to write a conjunction or disjunction of a finite number of wffs, for example, $\mathbf{A}_1 \wedge \mathbf{A}_2 \wedge \mathbf{A}_3 \wedge \mathbf{A}_4 \wedge \mathbf{A}_5$, or $\mathbf{A}_1 \wedge \cdots \wedge \mathbf{A}_n$.

Our second convention is that we may insert an extra pair of brackets around a wff to make it easier to read.

Notice that in the rules of formation of wffs, no new brackets are required in forming the negation $\neg\mathbf{A}$ of a wff **A**. Instead of the rule (W:$\neg$), we could have used the rule that if **A** is a wff, then $[\neg\mathbf{A}]$ is a wff. This was not done because it would only add an unnecessary extra pair of brackets.

According to the rules, $\neg p \wedge q$ means $[\neg p \wedge q]$, and does not mean $\neg[p \wedge q]$. To remind us of this fact, we might write $[\neg p] \wedge q$ instead of $\neg p \wedge q$.

A string obtained from a wff **C** using some combination of the conventions in this section will be called an **abbreviation** of **C**. Thus each wff has many abbreviations, but only one standard abbreviation. The TABLEAU program accepts as input any abbreviation of a wff. But after you finish typing the abbreviated wff at the keyboard, the program will display only its standard abbreviated form.

1.5 Semantics of Propositional Logic

In this section we shall assign truth values to wffs of propositional logic. We start with the notion of a model, which assigns a truth value to each propositional symbol. Given a model, we can then compute the truth value of any wff by a step by step process which parallels the rules for building wffs.

There are two truth values in propositional logic, **T** and **F**. A **model $\mathcal{M}$ for propositional logic of type $\mathcal{P}_0$** is a function which assigns to each proposition symbol $p \in \mathcal{P}_0$ a truth value which we denote by $p_{\mathcal{M}}$.

This is the first of many times in this text when we shall use the mathematical concept of a **function**. In general, a function f from a set A to a set B is a mathematical object which assigns an element $f(a) \in B$ to each element $a \in A$. We sometimes use the notation $f : A \to B$ to indicate that f is a function from A to B. Thus a model for propositional logic is just a function $\mathcal{M} : \mathcal{P}_0 \to \{\mathbf{T}, \mathbf{F}\}$.

For example, if the vocabulary contains two propositional symbols, $\mathcal{P}_0 = \{p, q\}$, there are 4 different models of type $\mathcal{P}_0$, which we may call $\mathcal{M}_0 \ldots \mathcal{M}_3$:

$$p_{\mathcal{M}_0} = \mathbf{T}, q_{\mathcal{M}_0} = \mathbf{T},$$

$$p_{\mathcal{M}_1} = \mathbf{T}, q_{\mathcal{M}_1} = \mathbf{F},$$

$$p_{\mathcal{M}_2} = \mathbf{F}, q_{\mathcal{M}_2} = \mathbf{T},$$

$$p_{\mathcal{M}_3} = \mathbf{F}, q_{\mathcal{M}_3} = \mathbf{F}.$$

If $\mathbf{P}_0$ has n propositional symbols where n is finite, there are 2^n different models of type $\mathcal{P}_0$. If $\mathcal{P}_0$ is infinite then there are infinitely many models of type $\mathcal{P}_0$.

Figure 1.5 lists the rules for computing the truth value $\mathbf{A}_\mathcal{M}$ of a wff $\mathbf{A}$ in model $\mathcal{M}$.

Truth Value Rules

(M:$\mathcal{P}_0$)	If $\mathbf{A}$ is a propositional symbol p, $\mathbf{A}_\mathcal{M} = p_\mathcal{M}$;	
(M:$\neg$)	$[\neg\mathbf{A}]_\mathcal{M} = \mathbf{T}$	if $\mathbf{A}_\mathcal{M} = \mathbf{F}$;
	$[\neg\mathbf{A}]_\mathcal{M} = \mathbf{F}$	if $\mathbf{A}_\mathcal{M} = \mathbf{T}$.
(M:$\wedge$)	$[\mathbf{A} \wedge \mathbf{B}]_\mathcal{M} = \mathbf{T}$	if $\mathbf{A}_\mathcal{M} = \mathbf{T}$ and $\mathbf{B}_\mathcal{M} = \mathbf{T}$;
	$[\mathbf{A} \wedge \mathbf{B}]_\mathcal{M} = \mathbf{F}$	otherwise.
(M:$\vee$)	$[\mathbf{A} \vee \mathbf{B}]_\mathcal{M} = \mathbf{T}$	if $\mathbf{A}_\mathcal{M} = \mathbf{T}$ or $\mathbf{B}_\mathcal{M} = \mathbf{T}$;
	$[\mathbf{A} \vee \mathbf{B}]_\mathcal{M} = \mathbf{F}$	otherwise.
(M:$\Rightarrow$)	$[\mathbf{A} \Rightarrow \mathbf{B}]_\mathcal{M} = \mathbf{T}$	if $\mathbf{A}_\mathcal{M} = \mathbf{F}$ or $\mathbf{B}_\mathcal{M} = \mathbf{T}$;
	$[\mathbf{A} \Rightarrow \mathbf{B}]_\mathcal{M} = \mathbf{F}$	otherwise.
(M:$\Leftrightarrow$)	$[\mathbf{A} \Leftrightarrow \mathbf{B}]_\mathcal{M} = \mathbf{T}$	if $\mathbf{A}_\mathcal{M} = \mathbf{B}_\mathcal{M}$;
	$[\mathbf{A} \Leftrightarrow \mathbf{B}]_\mathcal{M} = \mathbf{F}$	otherwise.

Figure 1.1: Truth Value Rules for Propositional Logic.

Using these rules, the truth value of each wff in each model can be computed by choosing a parsing sequence for the wff and applying one of the rules at each step.

For example, let us compute the value of $[p \Rightarrow \neg q] \Rightarrow [q \vee p]$ for a model $\mathcal{M}$ with $p_\mathcal{M} = \mathbf{T}$ and $q_\mathcal{M} = \mathbf{F}$. We first parse the wff.

(1) p is a wff by (W:$\mathcal{P}_0$).

(2) q is a wff by (W:$\mathcal{P}_0$).

(3) $\neg q$ is a wff by (2) and (W:$\neg$).

(4) $[p \Rightarrow \neg q]$ is a wff by (1), (3), and (W:$\Rightarrow$).

(5) $[q \vee p]$ is a wff by (1), (2), and (W:$\vee$).

(6) $[[p \Rightarrow \neg q] \Rightarrow [q \vee p]]$ is a wff by (4), (5), and (W:$\Rightarrow$).

Now we apply the rules for $\mathbf{A}_\mathcal{M}$:

(1) $p_\mathcal{M} = \mathbf{T}$.

(2) $q_\mathcal{M} = \mathbf{F}$.

(3) $[\neg q]_\mathcal{M} = \mathbf{T}$ by (2) and (M:$\neg$).

(4) $[p \Rightarrow \neg q]_\mathcal{M} = \mathbf{T}$ by (1), (3), and (M:$\Rightarrow$).

(5) $[q \vee p]_\mathcal{M} = \mathbf{T}$ by (1),(2), and (M:$\vee$).

(6) $[[p \Rightarrow \neg q] \Rightarrow [q \vee p]]_\mathcal{M} = \mathbf{T}$ by (4),(5), and (M:$\Rightarrow$).

The next theorem states a vitally important fact about truth values: Although a wff can have many different parsing sequences, the truth value depends only on the model and the wff, and does not depend on the particular parsing sequence which was used to construct the wff.

Theorem 1.5.1 *Given a model $\mathcal{M}$ and a wff $\mathbf{A}$, the truth value $\mathbf{A}_\mathcal{M}$ is the same for all parsing sequences of $\mathbf{A}$.*

This theorem shows that given a model $\mathcal{M}$ for $\mathcal{P}_0$, there is a unique function which assigns a truth value $\mathbf{A}_\mathcal{M}$ to each wff $\mathbf{A}$ and satisfies all the rules in Figure 1.5.

We leave the proof of this theorem as an exercise at the end of the chapter. Hint: the proof uses ideas that we have already developed in this book, the Unique Readability Theorem and induction on the length of wffs.

There are several different ways of saying that a wff is true in a model, which call attention to the model, the wff, or the truth value.

We shall often write the equation $\mathbf{A}_\mathcal{M} = \mathbf{T}$ in the alternate form $\mathcal{M} \models \mathbf{A}$. This alternate form uses the useful "turnstile symbol" $\models$, which is read "models," or "is a model of." The following five expressions all mean the same thing:

$\mathbf{A}_\mathcal{M} = \mathbf{T}$
$\mathbf{A}$ is true in $\mathcal{M}$
$\mathbf{A}$ holds in $\mathcal{M}$.
$\mathcal{M} \models \mathbf{A}$
$\mathcal{M}$ is a model of $\mathbf{A}$

Similarly, the following are the same:

$\mathbf{A}_\mathcal{M} = \mathbf{F}$
$\mathbf{A}$ is false in $\mathcal{M}$.
$\mathcal{M} \not\models \mathbf{A}$

In the next proposition we write down rules for truth values which are similar to the rules for tableau proofs in propositional logic which will be given later on in this chapter.

Proposition 1.5.2 *Let $\mathcal{M}$ be a model for propositional logic and $\mathbf{A}$ and $\mathbf{B}$ be wffs. Then:*

$\boxed{\neg\neg}$ *If $\mathcal{M} \models \neg\neg\mathbf{A}$, then $\mathcal{M} \models \mathbf{A}$.*

$\boxed{\wedge}$ *If $\mathcal{M} \models [\mathbf{A} \wedge \mathbf{B}]$, then $\mathcal{M} \models \mathbf{A}$ and $\mathcal{M} \models \mathbf{B}$.*

$\boxed{\neg\wedge}$ *If $\mathcal{M} \models \neg[\mathbf{A} \wedge \mathbf{B}]$, then $\mathcal{M} \models \neg\mathbf{A}$ or $\mathcal{M} \models \neg\mathbf{B}$.*

$\boxed{\vee}$ *If $\mathcal{M} \models [\mathbf{A} \vee \mathbf{B}]$, then either $\mathcal{M} \models \mathbf{A}$ or $\mathcal{M} \models \mathbf{B}$.*

$\boxed{\neg\vee}$ *If $\mathcal{M} \models \neg[\mathbf{A} \vee \mathbf{B}]$, then $\mathcal{M} \models \neg\mathbf{A}$ and $\mathcal{M} \models \neg\mathbf{B}$.*

$\boxed{\Rightarrow}$ *If $\mathcal{M} \models [\mathbf{A} \Rightarrow \mathbf{B}]$, then either $\mathcal{M} \models \neg\mathbf{A}$ or $\mathcal{M} \models \mathbf{B}$.*

$\boxed{\neg\Rightarrow}$ *If $\mathcal{M} \models \neg[\mathbf{A} \Rightarrow \mathbf{B}]$, then $\mathcal{M} \models \mathbf{A}$ and $\mathcal{M} \models \neg\mathbf{B}$.*

$\boxed{\Leftrightarrow}$ *If $\mathcal{M} \models [\mathbf{A} \Leftrightarrow \mathbf{B}]$, then either both $\mathcal{M} \models \mathbf{A}$ and $\mathcal{M} \models \mathbf{B}$ or else both $\mathcal{M} \models \neg\mathbf{A}$ and $\mathcal{M} \models \neg\mathbf{B}$.*

$\boxed{\neg\Leftrightarrow}$ *If $\mathcal{M} \models \neg[\mathbf{A} \Leftrightarrow \mathbf{B}]$, then either both $\mathcal{M} \models \mathbf{A}$ and $\mathcal{M} \models \neg\mathbf{B}$ or else both $\mathcal{M} \models \neg\mathbf{A}$ and $\mathcal{M} \models \mathbf{B}$.*

1.6 Truth Tables and Tautologies

The evaluation of the truth value $\mathbf{A}_\mathcal{M}$ of a wff $\mathbf{A}$ in a model $\mathcal{M}$ is so mechanical that we can arrange the work in a table. We first review our semantical rules in tabular form:

A	¬A
T	F
F	T

and

A	B	A ∧ B	A ∨ B	A ⇒ B	A ⇔ B
T	T	T	T	T	T
T	F	F	T	F	F
F	T	F	T	T	F
F	F	F	F	T	T

Now we can evaluate $\mathbf{A}_\mathcal{M}$ by the following strategy.

We first write the wff $\mathbf{A}$, and then underneath each occurrence of a proposition symbol we write the symbol's value:

$$\begin{array}{ccccccccc} [p & \Rightarrow & \neg & q] & \Rightarrow & [q & \vee & p] \\ \mathbf{T} & & & \mathbf{F} & & \mathbf{F} & & \mathbf{T} \end{array}$$

Then we fill in the value of each wff on the parsing sequence under its main connective:

$[p$	$\Rightarrow$	$\neg$	$q]$	$\Rightarrow$	$[q$	$\vee$	$p]$
T			**F**		**F**		**T**
		T				**T**	
	T						
				T			

To save space we may write all the truth values on the same line:

$[p$	$\Rightarrow$	$\neg$	$q]$	$\Rightarrow$	$[q$	$\vee$	$p]$
T	**T**	**T**	**F**	**T**	**F**	**T**	**T**

A wff **A** is called a **tautology** if it is true in *every* model: $\mathcal{M} \models \mathbf{A}$ for every model $\mathcal{M}$. To check if **A** is a tautology, we can make a **truth table** which computes the value of **A** in every possible model.

Take for the vocabulary $\mathcal{P}_0$ a finite set of propositional symbols which contains at least every propositional symbol in **A**. The *rows* of the truth table will correspond to the *models* $\mathcal{M}$ of type $\mathcal{P}_0$. The *columns* of the truth table will correspond to the proposition symbols and connectives in the string **A**. For example,

$[p$	$\Rightarrow$	$\neg$	$q]$	$\Rightarrow$	$[q$	$\vee$	$p]$
T	**F**	**F**	**T**	**T**	**T**	**T**	**T**
T	**T**	**T**	**F**	**T**	**F**	**T**	**T**
F	**T**	**F**	**T**	**T**	**T**	**T**	**F**
F	**T**	**T**	**F**	**F**	**F**	**F**	**F**

The entries in the column under the main connective (the fifth column in this example) give the values for the whole wff. Since the last of these values is **F**, the wff is *not* a tautology.

Here is a tautology:

$\neg$	p	$\Rightarrow$	$[p$	$\Rightarrow$	$q]$
F	**T**	**T**	**T**	**T**	**T**
F	**T**	**T**	**T**	**F**	**F**
T	**F**	**T**	**F**	**T**	**T**
T	**F**	**T**	**F**	**T**	**F**

Note that the same table shows that $\neg\mathbf{A} \Rightarrow [\mathbf{A} \Rightarrow \mathbf{B}]$ is a tautology for any wffs **A** and **B** (not just proposition symbols):

$\neg$	**A**	$\Rightarrow$	$[$**A**	$\Rightarrow$	**B**$]$
F	**T**	**T**	**T**	**T**	**T**
F	**T**	**T**	**T**	**F**	**F**
T	**F**	**T**	**F**	**T**	**T**
T	**F**	**T**	**F**	**T**	**F**

This is because the wffs **A** and **B** can only take the values **T** and **F** just like the proposition symbols p and q.

Suppose we have a tautology **C** built from two proposition symbols p and q. We will then get another tautology **D** by replacing each p in **C** by a wff **A** and replacing each q in **C** by a wff **B**. (A similar remark holds for wffs with more than two proposition symbols).

1.7 Tableaus

In ordinary discourse, a wff **A** is said to **follow from** another wff **B** if, assuming **B** is true, one can show that **A** is true by purely logical reasoning. Similarly, **A** follows from a list of other wffs $\mathbf{B}_1, \ldots, \mathbf{B}_n$ if one can show that **A** is true assuming that each of the wffs $\mathbf{B}_1, \ldots, \mathbf{B}_n$ is true. Truth tables give us one method of showing that one wff follows from others. In this section we shall introduce a second and more practical method for doing this, the method of tableau proofs. Tableau proofs have two major advantages over truth tables. First, a tableau proof will usually be much shorter than the corresponding truth table computation. Second, the method of tableau proofs carries over to the more important predicate logic, while the method of truth tables does not.

Often one can see very quickly (without computing the full truth table) whether some particular wff is a tautology by using an indirect argument. As an example we show that the wff $p \Rightarrow [q \Rightarrow [p \wedge q]]$ is a tautology. If not, there is a model $\mathcal{M}$ for its negation, i.e. (1) $\mathcal{M} \models \neg[p \Rightarrow [q \Rightarrow [p \wedge q]]$. From (1) we obtain (2) $\mathcal{M} \models p$ and (3) $\mathcal{M} \models \neg[q \Rightarrow [p \wedge q]$. From (3) we obtain (4) $\mathcal{M} \models q$ and (5) $\mathcal{M} \models \neg[p \wedge q]$. From (5) we conclude that either (6) $\mathcal{M} \models \neg p$ or else

(7) $\mathcal{M} \models \neg q$. But (6) contradicts (2) and (7) contradicts (4). Thus no such model $\mathcal{M}$ exists; i.e. the wff $p \Rightarrow [q \Rightarrow [p \wedge q]]$ is a tautology as claimed.

We can arrange this argument in a diagram, Figure 1.2, called a *tableau.*

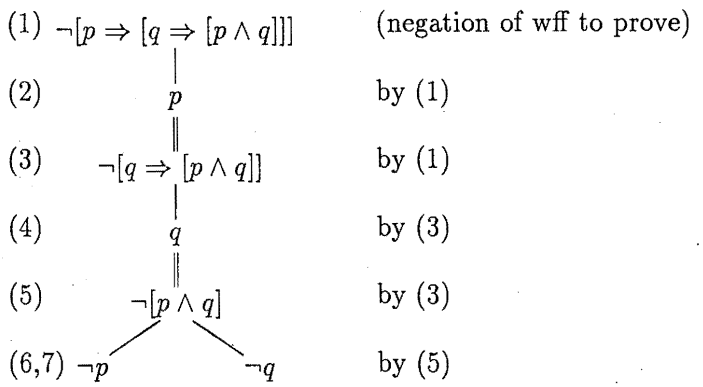


Figure 1.2: A Tableau Proof.

The steps in the original argument appear at "nodes" of the tableau. The number to the left of a wff is its step number in the argument; the number to the right is the number of the earlier step which justified the given step. The nodes are connected by lines. (Later on we shall explain why some of these lines are double). The two branches at the bottom of the tree correspond to the two possibilities in the case analysis. There are two ways to move from wff (1) down to the bottom of the diagram:

(1)-(2)-(3)-(4)-(5)-(6) and (1)-(2)-(3)-(4)-(5)-(7);

Along each of these two branches there is a wff and its negation: namely (2) and (6) for the former branch and (4) and (7) for the latter.

The method of tableaus can also be used to show that one wff (called the conclusion) follows from one or more other wffs (called the hypotheses). The tableau in Figure 1.3 shows that the wff $p \Rightarrow r$ follows from the set of hypotheses $p \Rightarrow q$ and $q \Rightarrow r$. The first node

is the negation of the conclusion, $\neg[p \Rightarrow r]$, and the second and third nodes contain the two hypotheses. On each branch of the tableau there is a wff and its negation. This shows that it is impossible for both hypotheses to be true and the conclusion to be false. Thus in any model in which both hypotheses are true, the conclusion is also true.

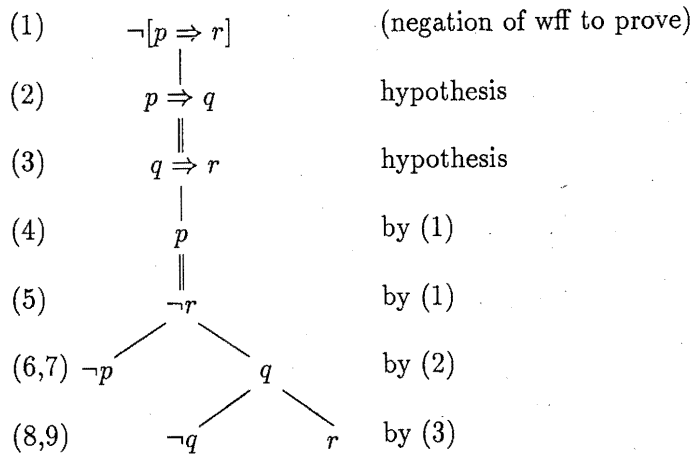


Figure 1.3: A Tableau Proof with Two Hypotheses.

We shall now extend the "turnstile" notation to apply to sets of wffs as well as single wffs. This will make it easier to discuss the case where one wff follows from a set of hypotheses. After that we will be ready to explain the tableau method in general.

A **finite set** is a set of the form $S = \{s_0, \ldots, s_n\}$ where n is a natural number. A **countable set** is an infinite set of the form $S = \{s_0, \ldots, s_n, \ldots\}$ where n runs over all natural numbers. The empty set is also considered to be a finite set.

Let us consider sets whose elements are wffs. In this book we shall confine our attention to sets of wffs which are either finite or countable. If $\mathbf{H}$ is a set of wffs and $\mathcal{M}$ is a model we shall say $\mathcal{M}$ **models** $\mathbf{H}$ (or $\mathcal{M}$ is a model of $\mathbf{H}$, or $\mathcal{M}$ **satisfies** $\mathbf{H}$) and write $\mathcal{M} \models \mathbf{H}$ if $\mathcal{M}$ models

every element $\mathbf{A}$ of $\mathbf{H}$:

$$\mathcal{M} \models \mathbf{H} \text{ iff } \mathcal{M} \models \mathbf{A} \text{ for all } \mathbf{A} \in \mathbf{H}.$$

Of course, when $\mathbf{H}$ is a finite set, say $\mathbf{H} = \{\mathbf{A}_1, \mathbf{A}_2, \ldots, \mathbf{A}_n\}$, then the notations

$$\mathcal{M} \models \mathbf{H}$$

and

$$\mathcal{M} \models \mathbf{A}_1 \wedge \mathbf{A}_2 \wedge \ldots \wedge \mathbf{A}_n$$

are synonymous. However, the new notation $\mathcal{M} \models \mathbf{H}$ is handy, especially when $\mathbf{H}$ is an infinite set. A wff $\mathbf{A}$ is a tautology if and only if the set $\{\neg\mathbf{A}\}$ consisting of the single wff $\neg\mathbf{A}$ has no models. Instead of trying to show that a given wff is a tautology, the tableau method tries to show that a given set of wffs has no models.

We now introduce yet another use of the "turnstile" symbol. A wff $\mathbf{A}$ is called a **semantic consequence** of the set of wffs $\mathbf{H}$, in symbols $\mathbf{H} \models \mathbf{A}$, if every model of $\mathbf{H}$ is a model of $\mathbf{A}$. Evidently, $\mathbf{A}$ is a semantic consequence of $\mathbf{H}$ if and only if the set $\mathbf{H} \cup \{\neg\mathbf{A}\}$ has no models. The notation "$\mathbf{H} \models \mathbf{A}$" is a formal description of the intuitive idea "$\mathbf{A}$ follows from $\mathbf{H}$."

To sum up, we have introduced three ways to use the "turnstile" notation. $\mathcal{M} \models \mathbf{A}$ means that $\mathcal{M}$ is a model of the wff $\mathbf{A}$. $\mathcal{M} \models \mathbf{H}$ means that $\mathcal{M}$ is a model of the set of wffs $\mathbf{H}$. $\mathbf{H} \models \mathbf{A}$ means that every model of $\mathbf{H}$ is a model of $\mathbf{A}$.

The tableau method which we now describe makes the task of deciding whether $\mathbf{H} \models \mathbf{A}$ holds more manageable, particularly in the case of first order logic in the next chapter.

As a stepping stone to the mathematical definition of a tableau, we first introduce the concept of a tree. A **tree** $\mathbf{T}$ is a system consisting of a finite or countable set of points called the **nodes** of the tree, a distinguished node $r_\mathbf{T}$ called the **root** of the tree, and a function π, or $\pi_\mathbf{T}$, which assigns to each node t distinct from the root another node $\pi(t)$ called the **parent** of t; it is further required that if we repeatedly take parents starting from any node t, forming the sequence of nodes

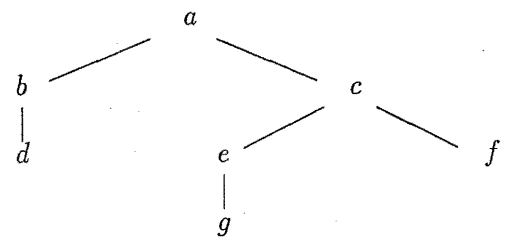
$$\pi^0(t) = t, \pi^1(t) = \pi(t), \pi^2(t) = \pi(\pi(t)), \pi^3(t) = \pi(\pi(\pi(t))), \ldots$$

we will reach the root node

$$\pi^n(t) = r_\mathbf{T}.$$

in finitely many steps. The nodes $\pi^1(t), \pi^2(t), \pi^3(t), \ldots$ are called the **proper ancestors** of t; a node t' is an **ancestor** of t if it is either t itself or is a proper ancestor of t. Thus the root is an ancestor of every node, including itself. Conversely, each node s whose parent is t is called a **child** of t. A node of the tree which has no children is called a **terminal node**.

It is customary to draw a tree upside down with the root at the top, because it is natural to start at the top of a piece of paper and work down when building a tableau. Each node is connected to its parent by a line. For example, in the tree



the root is a; the parent function is defined by $\pi(b) = \pi(c) = a$, $\pi(d) = b$, $\pi(e) = \pi(f) = c$, $\pi(g) = e$; the terminal nodes are d, f, g.

A tree with finitely many nodes, such as the preceding example, is called a **finite tree**, and a tree with infinitely many nodes is called an infinite tree. Infinite trees are possible because, although we required that a node has only finitely many ancestors, a node can have infinitely many descendants (children, grandchildren, etc.)

The simplest example of an infinite tree is the tree of natural numbers, with the set of nodes $\mathbf{T} = \{0, 1, 2, \ldots\}$, the root node $r_\mathbf{T} = 0$, and the parent function $\pi(n) = n - 1$. This tree has no terminal nodes, and every node has exactly one child. Here is a picture.

A subset Γ of a tree $\mathbf{T}$ is called a **branch** of $\mathbf{T}$ if the root node $r_{\mathbf{T}}$ belongs to Γ, the parent of each nonroot node in Γ is in Γ, and each node in Γ is either a terminal node of $\mathbf{T}$ or has exactly one child in Γ. We say that a node t is on the branch Γ if t is an element of the set Γ.

By successively taking parents, we see that for every node t on a branch Γ, every ancestor of t is also on Γ. By successively choosing children, we see that each node of a tree is on at least one branch of the tree. A terminal node t will be on exactly one branch Γ, which is equal to the set of all ancestors of t and is finite. On the other hand, a node with more than one child will be on more than one branch.

A branch Γ will either have exactly one terminal node t, in which case Γ is finite, or will have no terminal nodes, in which case Γ is infinite. The number of nodes on a finite branch Γ is called the **length** of Γ.

All the branches of a finite tree must be finite. In the above example of a finite tree, the branches are (d, b, a), (f, c, a), (g, e, c, a).

The infinite tree of natural numbers has just one branch, which is the whole tree.

Figure 1.2 at the beginning of this section is a tree with a wff attached to each node. This is an example of a labeled tree. By a **labeled tree for propositional logic** we shall mean a system consisting of a tree $\mathbf{T}$, a finite or countable set of wffs $\mathbf{H}$ which is called the set of **hypotheses**, and a wff $\Phi(t)$ attached to each nonroot node t. We shall say that the wff "$\mathbf{A}$ occurs at t" or that "$\mathbf{A}$ is t," when $\mathbf{A} = \Phi(t)$. All the wffs in the hypothesis set $\mathbf{H}$ are considered to occur at the root node.

A wff which occurs at a child of a node t will be called a **child wff** (or simply **child**) of t, and we shall use similar terminology for grandchildren, ancestors, etc. Thus a hypothesis wff is an ancestor of every node of $\mathbf{T}$.

We are now ready to define tableaus. An example of a tableau is shown in Figure 1.2 at the start of this section. You will see hundreds of additional examples of tableaus as you work the problems using the TABLEAU computer program. The idea is that tableaus are labeled trees which are built up step by step according to a particular set of rules, called the tableau extension rules. In this process, we start with just the root node labeled by the hypothesis set, and at each step we form a new tableau by adding one or more new nodes with attached wffs. During this process we form a sequence of larger and larger tableaus, called a tableau chain.

Definition 1.7.1 A **propositional tableau chain** is a finite or infinite sequence of finite labeled trees $\mathbf{T}_0, \ldots, \mathbf{T}_n, \ldots$ such that $\mathbf{T}_0$ consists only of a root node with the set of hypotheses $\mathbf{H}$, and each $\mathbf{T}_{k+1}$ in the sequence is obtained from $\mathbf{T}_k$ by applying one of the following **tableau extension rules** at a terminal node t of $\mathbf{T}_k$:

$\boxed{\neg\neg}$ If t has an ancestor $\neg\neg\mathbf{A}$, extend $\mathbf{T}_k$ by adding the child $\mathbf{A}$ of t.

$\boxed{\wedge}$ If t has an ancestor $\mathbf{A} \wedge \mathbf{B}$, extend by adding a child $\mathbf{A}$ and grandchild $\mathbf{B}$ of t.

$\boxed{\neg\wedge}$ If t has an ancestor $\neg[\mathbf{A} \wedge \mathbf{B}]$, extend by adding two children $\neg\mathbf{A}$ and $\neg\mathbf{B}$ of t.

$\boxed{\vee}$ If t has an ancestor $\mathbf{A} \vee \mathbf{B}$, extend by adding two children $\mathbf{A}$ and $\mathbf{B}$ of t.

$\boxed{\neg\vee}$ If t has an ancestor $\neg[\mathbf{A} \vee \mathbf{B}]$, extend by adding a child $\neg\mathbf{A}$ and grandchild $\neg\mathbf{B}$ of t.

$\boxed{\Rightarrow}$ If t has an ancestor $\mathbf{A} \Rightarrow \mathbf{B}$, extend by adding two children $\neg\mathbf{A}$ and $\mathbf{B}$ of t.

$\boxed{\neg \Rightarrow}$ If t has an ancestor $\neg[\mathbf{A} \Rightarrow \mathbf{B}]$, extend by adding a child $\mathbf{A}$ and a grandchild $\neg\mathbf{B}$ of t.

$\boxed{\Leftrightarrow}$ If t has an ancestor $[\mathbf{A} \Leftrightarrow \mathbf{B}]$, extend by adding two children $\mathbf{A}$ and $\neg\mathbf{A}$ of t, a child $\mathbf{B}$ of $\mathbf{A}$, and a child $\neg\mathbf{B}$ of $\neg\mathbf{A}$.

$\boxed{\neg \Leftrightarrow}$ If t has an ancestor $\neg[\mathbf{A} \Leftrightarrow \mathbf{B}]$, extend by adding two children $\mathbf{A}$ and $\neg\mathbf{A}$ of t, a child $\neg\mathbf{B}$ of $\mathbf{A}$, and a child $\mathbf{B}$ of $\neg\mathbf{A}$.

In each case, the ancestor wff is said to be **used** at t and the other wffs mentioned are said to be **added** at t.

Definition 1.7.2 A **finite propositional tableau** is a labeled tree $\mathbf{T}$ which is the last term $\mathbf{T}_n$ of some finite propositional tableau chain $\mathbf{T}_0, \ldots, \mathbf{T}_n$.

Thus a finite propositional tableau has finitely many nodes, but its hypothesis set $\mathbf{H}$ may be either finite or countable.

Definition 1.7.3 An **infinite propositional tableau** is a labeled tree $\mathbf{T}$ which is the **union** of some infinite propositional tableau chain

$$\mathbf{T}_0, \ldots, \mathbf{T}_k, \ldots,$$

in symbols, $\mathbf{T} = \bigcup_{k=0}^{\infty} \mathbf{T}_k$.

That is, $\mathbf{T}$ is the infinite labeled tree such that t is a node of $\mathbf{T}$ if and only if t is a node of $\mathbf{T}_k$ for some $k \in \mathbf{N}$, and whenever $t \in \mathbf{T}_k$, the parent $\pi(t)$ and wff $\Phi(t)$ are the same in $\mathbf{T}$ as in $\mathbf{T}_k$.

By a **propositional tableau** with root $\mathbf{H}$ we shall mean either a finite or an infinite propositional tableau whose set of hypotheses is $\mathbf{H}$.

The role of a tableau chain in building a tableau is analogous to the role of a parsing sequence in building a wff [1]. To build a propositional tableau, start with a tree $\mathbf{T}_0$ consisting of a single node (its root) and a set $\mathbf{H}$ of hypotheses at the root node. Then extend the tableau $\mathbf{T}_0$

[1]The TABLEAU program makes it easy to build a finite tableau. The program starts with a tableau $\mathbf{T}_0$ with only a root node, and forms a new tableau each time the Extend command is used.

to a tableau $\mathbf{T}_1$, and extend $\mathbf{T}_1$ to $\mathbf{T}_2$, and so on. Each extension uses one of the set of nine rules for extending a finite propositional tableau $\mathbf{T}_n$. At each stage we choose a terminal node t of $\mathbf{T}_n$ and a wff $\mathbf{C}$ which appears on the branch through t, and build $\mathbf{T}_{n+1}$ by adjoining one, two, or four nodes below t according to the rule determined by the form of $\mathbf{C}$.

At each stage of the process of building a tableau, we will have a finite propositional tableau $\mathbf{T}_k$. If the process continues through all k, the union of the chain of finite tableaus $\mathbf{T}_k$ will be an infinite propositional tableau $\mathbf{T}$.

For reference we have summarized the nine extension rules in Figure 1.4. This figure shows the node t and a wff $\mathbf{C}$ above it; the vertical dots indicate the branch of the tableau through t so the figure shows $\mathbf{C}$ on this branch. (It is not precluded that $\mathbf{C}$ be at t itself.) Below t in the figure are the wffs at the children of t, and when appropriate the grandchildren of t. When both child and grandchild are added together in a single rule, they are connected by a double line.

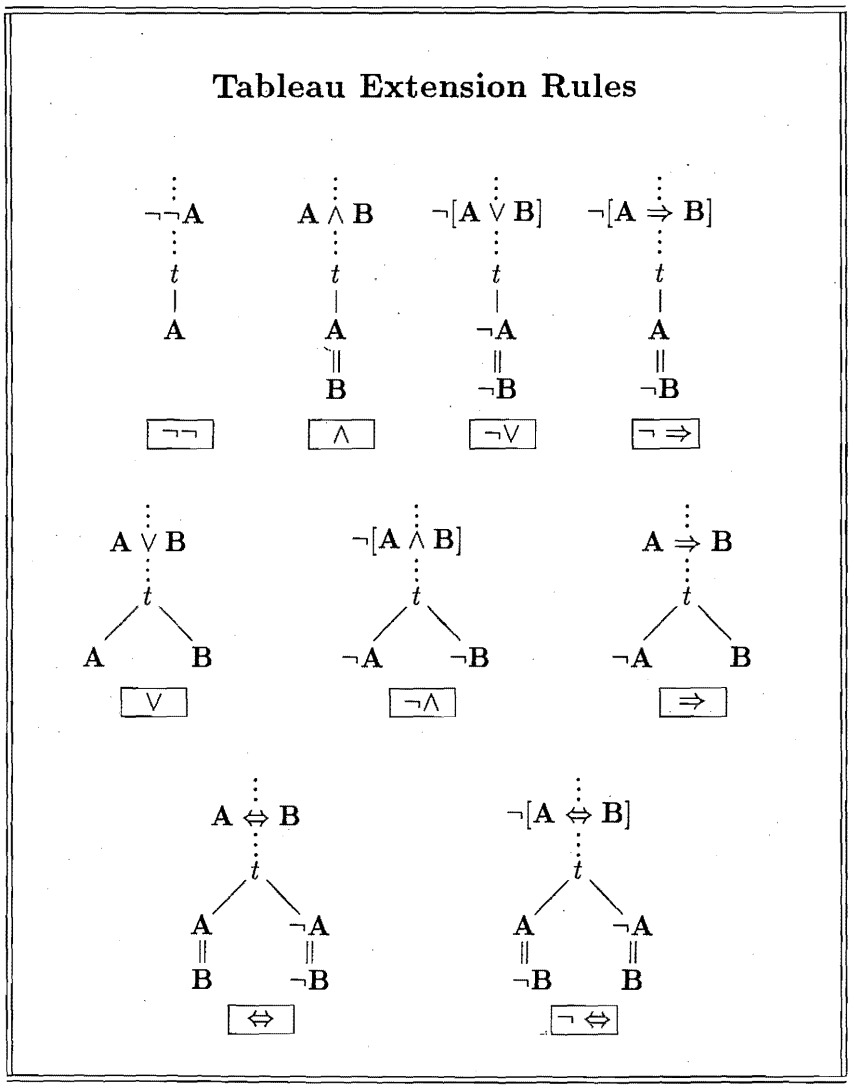


Figure 1.4: Propositional Tableau Extension Rules.

Tableaus will be used in two ways: to build a formal proof of a wff **A** from a hypothesis set **H**, and to build a model of a set of wffs **H**. Formal proofs will be finite tableaus, while both finite and infinite tableaus will be used to build models.

We are now ready to define the notion of a tableau proof. The tableau in Figure 1.2 at the beginning of this section is an example of a tableau proof. You will see other examples of tableau proofs when you solve the problems using the TABLEAU program. Going along with the idea of proving a wff by showing that its negation has no models, we shall first define a tableau confutation of a set of hypotheses, and then define a proof of a wff to be a confutation of the negation of the wff.

We say that a wff **A** occurs **along**, or *on*, a branch Γ if **A** is either a hypothesis (hence attached to the root node) or is attached to a nonroot node of Γ. We call a branch Γ of a tableau **contradictory** if for some wff **A**, both **A** and $\neg$**A** occur along the branch.

Definition 1.7.4 By a **confutation** of a hypothesis set **H** in propositional logic we mean a finite propositional tableau **T** with root **H** such that every branch of **T** is contradictory [2]. By a confutation of a wff **A** we mean a confutation of the one-element set $\{\mathbf{A}\}$. By a **tableau proof** of a wff **A** from a hypothesis set **H** we mean a tableau confutation of $\mathbf{H} \cup \{\neg\mathbf{A}\}$.

The case that **H** is the empty set is of particular interest. By a tableau proof of **A** we mean a tableau confutation of $\{\neg\mathbf{A}\}$. This is the same thing as a tableau proof of **A** from the empty set of hypotheses.

The "single turnstile" symbol $\vdash$ is useful when discussing whether or not a wff has a tableau proof. The notation

$$\mathbf{H} \vdash \mathbf{A}$$

means that there is a tableau proof of **A** from **H**. The notation $\vdash$ **A** means that there is a tableau proof of **A**.

[2] In the TABLEAU program, one can see at a glance whether or not a finite tableau with a finite root is a confutation. A node is colored red if every branch through the node is contradictory. In a confutation every node is colored red.

Since tableau confutations are by definition finite tableaus, all tableau proofs have only finitely many nodes, even when the hypothesis set is infinite.

In the next few sections we shall prove the Soundness and Completeness Theorems, which will clarify the relationship between tableau proofs and semantic consequences.

1.8 Soundness

In this section we will prove the

Soundness Theorem

If a propositional wff has a tableau proof, then it is a tautology.

The main step is the following

Lemma 1.8.1 *Let* $\mathbf{T}$ *be a finite propositional tableau with root* $\mathbf{H}$. *Let* $\mathcal{M}$ *be a propositional model of the hypothesis set* $\mathbf{H}$. *Then there is a branch* Γ *such that* $\mathcal{M} \models \Gamma$, *that is,* $\mathcal{M} \models \mathbf{A}$ *for every wff* $\mathbf{A}$ *on* Γ.

Proof: By Definition 1.7.1 there is a finite propositional tableau chain $\mathbf{T}_0, \mathbf{T}_1, \ldots$ such that $\mathbf{T}$ is the last term $\mathbf{T}_n$. We must show that there is a branch Γ of $\mathbf{T}$ such that every wff $\mathbf{A}$ which occurs on Γ holds in $\mathcal{M}$. To do this, we shall find a sequence of branches Γ_k of $\mathbf{T}_k$, $k \leq n$, such that for each $k < n$, $\Gamma_k \subset \Gamma_{k+1}$, and every wff $\mathbf{A}$ which occurs on Γ_{k+1} holds in $\mathcal{M}$ (in symbols, $\mathcal{M} \models \Gamma_{k+1}$). Then Γ_n is a branch of $\mathbf{T}$ and $\mathcal{M} \models \Gamma_n$ as required.

When $k = 0$ we take Γ_0 to be the set whose only element is the root node, so that the wffs $\mathbf{A}$ on Γ_0 are simply those of $\mathbf{H}$. Thus the assumption $\mathcal{M} \models \mathbf{H}$ shows that $\mathcal{M} \models \Gamma_0$. If $\mathbf{T}_{k+1}$ is obtained from $\mathbf{T}_k$ by extending at some node other than the terminal node of Γ_k we simply take $\Gamma_k = \Gamma_{k+1}$ and there is nothing to prove. Hence assume that $\mathbf{T}_{k+1}$ is obtained from $\mathbf{T}_k$ by extending at the terminal node of Γ_k by applying one of the nine tableau extension rules to some wff $\mathbf{A}_j$ in the list. We use a case analysis and Proposition 1.5.2.

(1) If $\mathbf{A}_j$ is $\neg\neg\mathbf{A}$ then Γ_{k+1} is obtained from Γ_k by adjoining $\mathbf{A}$.

(2) If $\mathbf{A}_j$ is $[\mathbf{A} \wedge \mathbf{B}]$ then Γ_{k+1} is obtained from Γ_k by adjoining $\mathbf{A}$ and $\mathbf{B}$.

(3) If $\mathbf{A}_j$ is $\neg[\mathbf{A} \wedge \mathbf{B}]$, then Γ_{k+1} is obtained from Γ_k by adjoining either $\neg\mathbf{A}$ (if $\mathcal{M} \models \neg\mathbf{A}$) or $\neg\mathbf{B}$ (if $\mathcal{M} \models \neg\mathbf{B}$).

(4) If $\mathbf{A}_j$ is $[\mathbf{A} \vee \mathbf{B}]$, then Γ_{k+1} is obtained from Γ_k by adjoining either $\mathbf{A}$ (if $\mathcal{M} \models \mathbf{A}$) or $\mathbf{B}$ (if $\mathcal{M} \models \mathbf{B}$).

(5) If $\mathbf{A}_j$ is $\neg[\mathbf{A} \vee \mathbf{B}]$, then Γ_{k+1} is obtained from Γ_k by adjoining $\neg\mathbf{A}$ and $\neg\mathbf{B}$.

(6) If $\mathbf{A}_j$ is $[\mathbf{A} \Rightarrow \mathbf{B}]$, then then Γ_{k+1} is obtained from Γ_k by adjoining either $\neg\mathbf{A}$ (if $\mathcal{M} \models \neg\mathbf{A}$) or $\mathbf{B}$ (if $\mathcal{M} \models \mathbf{B}$).

(7) If $\mathbf{A}_j$ is $\neg[\mathbf{A} \Rightarrow \mathbf{B}]$, then then Γ_{k+1} is obtained from Γ_k by adjoining $\mathbf{A}$ and $\neg\mathbf{B}$.

(8) If $\mathbf{A}_j$ is $[\mathbf{A} \Leftrightarrow \mathbf{B}]$, then Γ_{k+1} is obtained from Γ_k by adjoining either both $\mathbf{A}$ and $\mathbf{B}$ (if $\mathcal{M} \models \mathbf{A}$ and $\mathcal{M} \models \mathbf{B}$) or else both $\neg\mathbf{A}$ and $\neg\mathbf{B}$ (if $\mathcal{M} \models \neg\mathbf{A}$ and $\mathcal{M} \models \neg\mathbf{B}$).

(9) If $\mathbf{A}_j$ is $\neg[\mathbf{A} \Leftrightarrow \mathbf{B}]$, then then Γ_{k+1} is obtained from Γ_k by adjoining either both $\mathbf{A}$ and $\neg\mathbf{B}$ (if $\mathcal{M} \models \mathbf{A}$ and $\mathcal{M} \models \neg\mathbf{B}$) or else both $\neg\mathbf{A}$ and $\mathbf{B}$ (if $\mathcal{M} \models \neg\mathbf{A}$ and $\mathcal{M} \models \mathbf{B}$).

In cases (1), (2), (5), and (7) the branch Γ_{k+1} is the unique branch of $\mathbf{T}_{k+1}$ which extends Γ_k; in the remaining cases Γ_{k+1} is one of the two branches of $\mathbf{T}_{k+1}$ which extend Γ_k. **End of Proof.**

The above lemma actually holds for infinite tableaus as well as finite tableaus (Exercise 20), but we shall only use the lemma in the finite case.

Lemma 1.8.2 *If a finite or countable set* $\mathbf{H}$ *of propositional wffs has a tableau confutation, then* $\mathbf{H}$ *has no model.*

Proof: Suppose $\mathbf{H}$ is a hypothesis set and $\mathbf{T}$ is a tableau confutation of $\mathbf{H}$; if $\mathbf{H}$ has a model $\mathcal{M}$, then by the previous lemma, there is a branch Γ of $\mathbf{T}$ each of whose wffs holds in $\mathcal{M}$. Since every branch of $\mathbf{T}$ is contradictory, there is a wff $\mathbf{A}$ such that both $\mathbf{A}$ and $\neg\mathbf{A}$ are on Γ. But this is impossible since by Definition 1.5, no model satisfies a wff and its negation. **End of Proof.**

Theorem 1.8.3 (Extended Soundness Theorem) *Suppose* $\mathbf{H}$ *is a finite or countable set of propositional wffs and* $\mathbf{A}$ *is a propositional wff. If* $\mathbf{H} \vdash \mathbf{A}$ *then* $\mathbf{H} \models \mathbf{A}$*; in other words, if there is a tableau proof of* $\mathbf{A}$ *from* $\mathbf{H}$*, then* $\mathbf{A}$ *is a semantic consequence of* $\mathbf{H}$*.*

Proof: Given $\mathbf{H}$ and $\mathbf{A}$ and a tableau confutation $\mathbf{T}$ of $\mathbf{H} \cup \{\neg\mathbf{A}\}$, we note that by the previous lemma, $\mathbf{H} \cup \{\neg\mathbf{A}\}$ has no model, that is, no model of $\mathbf{H}$ is also a model of $\neg\mathbf{A}$. Thus, if $\mathcal{M}$ is a model of $\mathbf{H}$, $\mathcal{M}$ is a model of $\mathbf{A}$. It follows that $\mathbf{H} \models \mathbf{A}$. **End of Proof.**

A tableau confutation can be used to show that a propositional wff is a tautology. Remember that a propositional wff $\mathbf{A}$ is a tautology if and only if it is true in every model, and also if and only if $\neg\mathbf{A}$ is false in every model. Thus if $\neg\mathbf{A}$ has a confutation, then $\mathbf{A}$ is a tautology. Therefore the Soundness Theorem in the box at the beginning of this section is a corollary of the Extended Soundness Theorem.

1.9 Finished Sets

In this section we introduce the concept of a *finished set of wffs*. It will be used in the proof of the Completeness Theorem in the next section. The concept will be refined in the next chapter to handle predicate logic.

By a **basic wff** we shall mean a propositional symbol or a negation of a propositional symbol. The basic wffs are the ones which cannot be broken down into simpler wffs by the rules for extending tableaus. A set $\mathbf{\Delta}$ of wffs is called **contradictory** iff it contains some wff $\mathbf{A}$ together with the negation $\neg\mathbf{A}$ of that wff. A set $\mathbf{\Delta}$ of wffs is called **finished** iff it is not contradictory and for each wff $\mathbf{C} \in \mathbf{\Delta}$ either $\mathbf{C}$ is basic or one of the following is true:

$[\neg\neg]$ $\mathbf{C}$ has form $\neg\neg\mathbf{A}$ where $\mathbf{A} \in \mathbf{\Delta}$;

$[\wedge]$ $\mathbf{C}$ has form $[\mathbf{A} \wedge \mathbf{B}]$ where both $\mathbf{A} \in \mathbf{\Delta}$ and $\mathbf{B} \in \mathbf{\Delta}$;

$[\neg\wedge]$ $\mathbf{C}$ has form $\neg[\mathbf{A} \wedge \mathbf{B}]$ where either $\neg\mathbf{A} \in \mathbf{\Delta}$ or $\neg\mathbf{B} \in \mathbf{\Delta}$;

$[\vee]$ $\mathbf{C}$ has form $[\mathbf{A} \vee \mathbf{B}]$ where either $\mathbf{A} \in \mathbf{\Delta}$ or $\mathbf{B} \in \mathbf{\Delta}$;

$[\neg\vee]$ $\mathbf{C}$ has form $\neg[\mathbf{A} \vee \mathbf{B}]$ where both $\neg\mathbf{A} \in \mathbf{\Delta}$ and $\neg\mathbf{B} \in \mathbf{\Delta}$;

$[\Rightarrow]$ $\mathbf{C}$ has form $[\mathbf{A} \Rightarrow \mathbf{B}]$ where either $\neg\mathbf{A} \in \mathbf{\Delta}$ or $\mathbf{B} \in \mathbf{\Delta}$;

$[\neg\Rightarrow]$ $\mathbf{C}$ has form $\neg[\mathbf{A} \Rightarrow \mathbf{B}]$ where both $\mathbf{A} \in \mathbf{\Delta}$ and $\neg\mathbf{B} \in \mathbf{\Delta}$;

$[\Leftrightarrow]$ $\mathbf{C}$ has form $[\mathbf{A} \Leftrightarrow \mathbf{B}]$ where either both $\mathbf{A} \in \mathbf{\Delta}$ and $\mathbf{B} \in \mathbf{\Delta}$ or else both $\neg\mathbf{A} \in \mathbf{\Delta}$ and $\neg\mathbf{B} \in \mathbf{\Delta}$;

$[\neg\Leftrightarrow]$ $\mathbf{C}$ has form $\neg[\mathbf{A} \Leftrightarrow \mathbf{B}]$ where either both $\mathbf{A} \in \mathbf{\Delta}$ and $\neg\mathbf{B} \in \mathbf{\Delta}$ or else both $\neg\mathbf{A} \in \mathbf{\Delta}$ and $\mathbf{B} \in \mathbf{\Delta}$.

Notice the similarity between this definition and the tableau extension rules of Definition 1.7.1. Notice also that in each of these rules, the new wffs have smaller length than the original wff $\mathbf{C}$.

Here is an example of a finished set of wffs:

$$p \wedge q, p \Rightarrow [s \vee p], s \vee p, p, q.$$

The set

$$p \wedge q, p \Rightarrow [s \vee p], p, q$$

is not finished because it does not satisfy rule $[\Rightarrow]$. The set

$$p \wedge q, p \Rightarrow [s \vee p], s \vee p, p$$

is not finished because it does not satisfy rule $[\wedge]$. The set

$$p \wedge q, p \Rightarrow [s \vee p], \neg p, p, q$$

is not finished because it is contradictory.

Lemma 1.9.1 (Finished Set Lemma) *Let Δ be a finished set of wffs. Then Δ has a model. In fact, any model of the set of basic wffs in Δ is a model of all the wffs in Δ.*

Proof: Let us first note that the set of basic wffs in Δ has at least one model. Let us define $\mathcal{N}$ by $p_{\mathcal{N}} = \mathbf{T}$ if p is in Δ and $p_{\mathcal{N}} = \mathbf{F}$ in p is not in Δ. Then (because Δ is not contradictory) $p_{\mathcal{M}} = \mathbf{F}$ if $\neg p$ is in Δ. Indeed, any model $\mathcal{M}$ in which each p which occurs in Δ is true, and each p such that $\neg p$ occurs in Δ is false, is a model of the set of basic wffs in Δ. Given one model of the set of basic wffs in Δ, another model of the set of basic wffs in Δ can be obtained by changing the truth values of any propositional symbols q such that neither q nor $\neg q$ occur on Δ.

Let $\mathcal{M}$ be a model of all basic wffs in Δ. We must show that

$$\mathcal{M} \models \Delta,$$

that is, that $\mathcal{M} \models \mathbf{C}$ for each wff $\mathbf{C} \in \Delta$. Now let $R(n)$ be the following property of a natural number n: For every wff $\mathbf{C}$, if $\mathbf{C}$ belongs to Δ and $\mathbf{C}$ has length at most n, then $\mathcal{M}$ models $\mathbf{C}$.

$R(0)$, $R(1)$, and $R(2)$ are true because every wff of length ≤ 2 is basic, and $\mathcal{M}$ models every basic wff in Δ. Assume $R(n)$. Suppose that $\mathbf{C}$ has length at most $n+1$ and belongs to Δ. By examining each of the nine cases listed above, we see that since $\mathcal{M}$ models every wff in Δ of length at most n, $\mathcal{M}$ also models $\mathbf{C}$. This proves $R(n+1)$. We conclude by induction that $R(n)$ holds for all n, and thus $\mathcal{M}$ models every wff in Δ as required. **End of Proof.**

1.10 Completeness

In this section we will prove the

Completeness Theorem

If a propositional wff is a tautology, then it has a tableau proof.

The next lemma is the main fact which we shall prove in order to get the Completeness Theorem.

Lemma 1.10.1 (Finite Main Lemma) *Let $\mathbf{H}$ be a finite set of propositional wffs. Either $\mathbf{H}$ has a tableau confutation or $\mathbf{H}$ has a model.*

We have already shown in Lemma 1.8.2 that $\mathbf{H}$ cannot have both a tableau confutation and a model. This, combined with the Finite Main Lemma above, shows that $\mathbf{H}$ has a tableau confutation if and only if $\mathbf{H}$ does not have a model.

Here is the basic idea in proving the Finite Main Lemma. First make a systematic attempt to find a tableau confutation of $\mathbf{H}$ by building a very rich finite tableau, called a finished tableau. Then show that this finished tableau is either a tableau confutation of $\mathbf{H}$, or else has a branch whose wffs form a finished set which gives us a model of $\mathbf{H}$.

To carry out this basic idea, we first give a careful definition of the notion of a finished tableau. Then a finished tableau will be built in the proof of the Tableau Extension Lemma. After that, near the end of this section, we prove the Finite Main Lemma.

A branch Γ of a tableau is said to be **finished** if Γ is not contradictory and every nonbasic wff on Γ is used at some node of Γ[3]. In other words, a branch Γ is finished if and only if the set Δ of wffs which occur along Γ is a finished set in the sense of the previous section. A propositional tableau $\mathbf{T}$ is said to be **finished** if every branch of $\mathbf{T}$ is either finished or is finite and contradictory.

A confutation is automatically a finished tableau because every branch is finite and contradictory. A finite finished tableau either has at least one finished branch or is a confutation. Figure 1.5 is an example of a finished tableau which is not a confutation. It has two contradictory branches and one finished branch.

Finished tableaus can be either finite or infinite. In this section we shall construct a finite finished tableau on the way to proving the

[3] In the TABLEAU program, a branch Γ is finished if its terminal node is yellow and each node of Γ is either a basic wff or is shown by the **Why command** to be invoked at some other node of Γ.

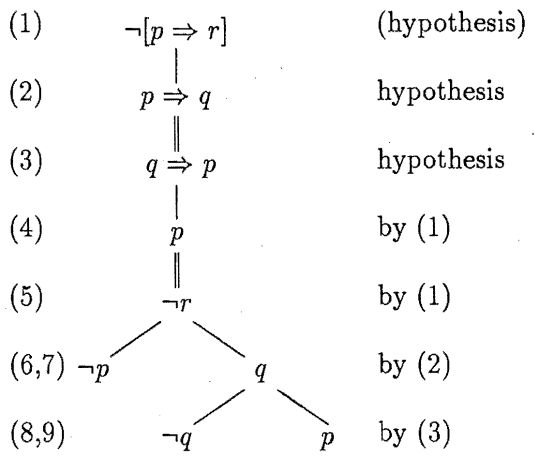


Figure 1.5: A Finished Tableau.

Completeness Theorem. In the next section we shall use infinite finished tableaus to establish the connection between proofs and semantic consequences of an infinite set of hypotheses.

A tableau $\mathbf{T}'$ is said to be an **extension** of a finite tableau $\mathbf{T}$ if $\mathbf{T}'$ can be obtained from $\mathbf{T}$ by repeatedly adding nodes at the ends of branches.

Lemma 1.10.2 (Tableau Extension Lemma) *Every finite propositional tableau with a finite root* $\mathbf{H}$ *can be extended to a finite finished tableau (with the same hypothesis set)*[4].

Proof: We shall call a wff $\mathbf{A}$ at a node t in a tableau **unused** if $\mathbf{A}$ is not a basic wff and there is a noncontradictory branch through t on which $\mathbf{A}$ is not used[5]. Note that a tableau is finished if and only if there are no unused wffs in the tableau.

[4]An algorithm for doing this is illustrated by the computer program COMPLETE, which is included with this book.

[5]In the COMPLETE program, unused wffs are colored yellow, wffs through which every branch is contradictory are colored red, and other wffs are colored blue.

Let $\mathbf{H}$ be a finite hypothesis set which remains fixed throughout our proof. Given a finite tableau $\mathbf{T}$ with root $\mathbf{H}$, let $u(\mathbf{T})$ be the length of the longest unused wff in $\mathbf{T}$, with the provision that $u(\mathbf{T}) = 0$ if there are no unused wffs, that is if $\mathbf{T}$ is finished. Since there are only finitely many wffs occurring anywhere in $\mathbf{T}$, the number $u(\mathbf{T})$ exists. We prove the lemma by induction on $u(\mathbf{T})$.

Let $R(n)$ be the statement that every finite propositional tableau $\mathbf{T}$ with root $\mathbf{H}$ and with $u(\mathbf{T}) < n$ can be extended to a finite finished tableau. $R(n)$ asserts that the lemma is true whenever $u(\mathbf{T}) < n$. The statement $R(1)$ is true, because a tableau $\mathbf{T}$ with $u(\mathbf{T}) < 1$ is already finished. Assume $R(n)$. Choose a finite tableau $\mathbf{T}$ with root $\mathbf{H}$ and $u(\mathbf{T}) < n+1$. Extend $\mathbf{T}$ to a new finite tableau $\mathbf{T}'$ by using every unused wff $\mathbf{A}$ in $\mathbf{T}$ once on every noncontradictory branch through $\mathbf{A}$. Each of the unused wffs in the original tableau $\mathbf{T}$ is used in the new tableau $\mathbf{T}'$. Moreover, each new wff which was added in forming $\mathbf{T}'$ has length less than $u(\mathbf{T})$, because the added wffs always have smaller length than the used wffs. Therefore $u(\mathbf{T}') < u(\mathbf{T}) < n + 1$, so $u(\mathbf{T}') < n$. By the induction hypothesis $R(n)$, there is a finite finished extension $\mathbf{T}''$ of $\mathbf{T}'$. $\mathbf{T}''$ is also a finished extension of $\mathbf{T}$. This proves $R(n+1)$ and completes the induction. **End of Proof.**

Proof of the Finite Main Lemma: Let $\mathbf{H}$ be a finite set of wffs which does not have a tableau confutation. By the Tableau Extension Lemma, the tableau consisting of only a root node with hypothesis set $\mathbf{H}$ can be extended to a finite finished tableau $\mathbf{T}$. This tableau still has root $\mathbf{H}$. Since $\mathbf{T}$ is not a confutation, it has a finished branch Γ. By the Finished Set Lemma 1.9.1, the set Δ of all wffs on Γ has a model $\mathcal{M}$. In particular, $\mathcal{M}$ is a model of $\mathbf{H}$ as required. **End of Proof.**

Theorem 1.10.3 (Extended Completeness Theorem) *If a wff* $\mathbf{A}$ *is a semantic consequence of a finite set of wffs* $\mathbf{H}$*, then there is a tableau proof of* $\mathbf{A}$ *from* $\mathbf{H}$*. In other words,*

$$\mathbf{H} \models \mathbf{A} \quad \textit{implies} \quad \mathbf{H} \vdash \mathbf{A}.$$

Proof: Suppose that $\mathbf{A}$ is a semantic consequence of $\mathbf{H}$. Then the set formed by adding the negation of $\mathbf{A}$ to $\mathbf{H}$ has no models. By the Finite

Main Lemma, this set has a tableau confutation, which is a tableau proof of **A** from **H**. (The special case where the hypothesis set **H** is empty is the Completeness Theorem in the box at the beginning of this section.) **End of Proof.**

We reiterate that tableau proofs are finite. Thus in the Extended Completeness Theorem, if $\mathbf{H} \models \mathbf{A}$ then there is a finite tableau proof of **A** from **H**. In the next section we see that this still works when the hypothesis set **H** is infinite.

1.11 Compactness

In this section we shall show that the Extended Completeness Theorem and other results of the last section hold for a countable set of hypotheses. We are studying countable sets of hypotheses in this chapter to prepare the way for predicate logic, where they are of great importance. Most of contemporary of mathematics is based on two particular countable sets of hypotheses in predicate logic, Zermelo Fraenkel set theory, to be introduced in Chapter 3, and Peano arithmetic, to be introduced in Chapter 4.

The key result in this section is the following infinite form of the Main Lemma.

Lemma 1.11.1 (Main Lemma) *Let* **H** *be a countable set of propositional wffs. Either* **H** *has a tableau confutation or* **H** *has a model.*

We first show that each countable hypothesis set has a finished tableau.

Lemma 1.11.2 *For every finite or countable set* **H** *of propositional wffs, there is a finished tableau with root* **H**.

Proof: The Tableau Extension Lemma shows that each finite hypothesis set **H** is the root of a finished tableau. It remains to give the proof in the case that **H** is a countable set

$$\mathbf{H} = \{\mathbf{A}_1, \ldots, \mathbf{A}_n, \ldots\}.$$

Let $\mathbf{H}_n$ be the finite subset

$$\mathbf{H}_n = \{\mathbf{A}_1, \ldots, \mathbf{A}_n\}$$

composed of the first n elements of **H**. We shall say that a finite tableau $\mathbf{T}_n$ with root **H** is finished for $\mathbf{H}_n$ if the tableau $\mathbf{T}'_n$ which is the same as $\mathbf{T}_n$ except that it has root $\mathbf{H}_n$ instead of **H** is a finished tableau. Using the Tableau Extension Lemma countably many times, we obtain a sequence of finite tableaus $\mathbf{T}_0, \ldots, \mathbf{T}_n, \ldots$ with root **H** such that $\mathbf{T}_0$ has only a root node, and for each $n > 0$, $\mathbf{T}_n$ is an extension of $\mathbf{T}_{n-1}$ which is finished for $\mathbf{H}_n$. We can also take the $\mathbf{T}_n$ to have the additional property that no contradictory branch Γ of $\mathbf{T}_{n-1}$ gets extended in forming $\mathbf{T}_n$, that is, the terminal node of Γ in $\mathbf{T}_{n-1}$ is still a terminal node of $\mathbf{T}_n$.

Let **T** be the union $\mathbf{T} = \bigcup_{k=0}^{\infty} \mathbf{T}_k$. Let Γ be a branch of **T**. If Γ is contradictory, with a contradictory pair $\mathbf{A}, \neg\mathbf{A}$, then there is an n such that both of the nodes **A** and $\neg\mathbf{A}$ belong to $\mathbf{T}_n$. Then $\Gamma \cap \mathbf{T}_n$ is already a contradictory branch of $\mathbf{T}_n$. By our construction, the contradictory branch $\Gamma \cap \mathbf{T}_n$ never gets extended after stage n, so $\Gamma = \Gamma \cap \mathbf{T}_n$ and Γ is finite.

On the other hand, if Γ is noncontradictory, then our construction insures that Γ is a finished branch. Therefore **T** is a finished tableau with root **H**. **End of Proof.**

Our next lemma is a general mathemathematical principle which is useful in a variety of circumstances. We shall use it here to show that if all the branches of a tableau are finite and contradictory, then the tableau itself is finite and hence is a confutation.

Theorem 1.11.3 (König Tree Theorem) *If a tree has infinitely many nodes and each node has finitely many children, then the tree has an infinite branch.*

Proof: To prove this, choose an infinite sequence of nodes $t_0, t_1, t_2, \ldots$ with the properties

1. t_0 is the root node;
2. t_{n+1} is a child of t_n; and

3. each t_n has infinitely many nodes beneath it;

Given a node t_n with infinitely many nodes beneath it, one of its children must also have infinitely many nodes beneath it. This is because t_n has finitely many children and an infinite set cannot be the union of finitely many finite sets. Let t_{n+1} be any child of t_n which has infinitely many nodes beneath it. The set of nodes $\{t_n : n = 0, 1, 2, \ldots\}$ is an infinite branch. **End of Proof.**

The König Tree Theorem fails if we omit the requirement that each node have only finitely many children; see Exercise 30.

Corollary 1.11.4 *Let* $\mathbf{T}$ *be a finished tableau. Then either* $\mathbf{T}$ *has a finished branch or* $\mathbf{T}$ *is a tableau confutation.*

Proof: Suppose $\mathbf{T}$ has no finished branch. Then every branch of $\mathbf{T}$ is finite and contradictory. Since every branch of $\mathbf{T}$ is finite, $\mathbf{T}$ is a finite tableau by the König Tree Theorem. Since $\mathbf{T}$ is finite and every branch of $\mathbf{T}$ is contradictory, $\mathbf{T}$ is a tableau confutation. **End of Proof.**

Proof of the Main Lemma: Suppose that $\mathbf{H}$ does not have a tableau confutation. By Lemma 1.11.2, there is a finished tableau $\mathbf{T}$ with root $\mathbf{H}$. $\mathbf{T}$ is not a tableau confutation by assumption, so by the preceding corollary, $\mathbf{T}$ has a finished branch Γ. By the Finished Set Lemma, the set of wffs on Γ has a model $\mathcal{M}$. Finally, since all the wffs in the hypothesis set $\mathbf{H}$ occur on Γ, $\mathcal{M}$ is a model of $\mathbf{H}$. **End of Proof.**

We now give several consequences of the Main Lemma. Our first consequence is the Compactness Theorem

Theorem 1.11.5 (Compactness Theorem) *Let* $\mathbf{H}$ *be a countable set of propositional wffs. Suppose that every finite subset of* $\mathbf{H}$ *has a model. Then* $\mathbf{H}$ *has a model* [6].

Proof: Suppose that $\mathbf{H}$ does not have a model. By the Main Lemma, $\mathbf{H}$ has a tableau confutation $\mathbf{T}$. Since each tableau confutation is a finite tableau, the set $\mathbf{H}'$ of all wffs in $\mathbf{H}$ which are used somewhere in $\mathbf{T}$ is finite. Now let $\mathbf{T}'$ be the labeled tree which is the same as $\mathbf{T}$ but with root $\mathbf{H}_0$ instead of $\mathbf{H}$. Then $\mathbf{T}'$ is a tableau confutation of $\mathbf{H}'$. By the Extended Soundness Theorem, $\mathbf{H}'$ has no models. But this contradicts the assumption that every finite subset of $\mathbf{H}$ has a model. Therefore $\mathbf{H}$ does have a model. **End of Proof.**

[6] The Compactness Theorem is actually true even when $\mathbf{H}$ is an uncountable set of wffs. The proof in the general case requires *transfinite induction* which is beyond the scope of this book.

As we mentioned at the beginning of this section, the Extended Completeness Theorem holds for *countable* as well as finite hypothesis sets $\mathbf{H}$. The Soundness Theorem also holds for such hypothesis sets. We can therefore combine the Extended Soundness, Extended Completeness, and the Compactness Theorems together into one concise statement.

Corollary 1.11.6 *Suppose* $\mathbf{H}$ *is a finite or countable set of wffs and* $\mathbf{A}$ *is a wff. Then*

$$\mathbf{H} \vdash \mathbf{A} \text{ if and only if } \mathbf{H} \models \mathbf{A}.$$

Proof: The Extended Soundness Theorem says that if $\mathbf{H} \vdash \mathbf{A}$ then $\mathbf{H} \models \mathbf{A}$. Suppose that $\mathbf{H} \models \mathbf{A}$. Then $\mathbf{H} \cup \{\neg\mathbf{A}\}$ has no models. By the Compactness Theorem, there is a finite subset $\mathbf{H}_0 \subset \mathbf{H}$ such that $\mathbf{H}_0 \cup \{\neg\mathbf{A}\}$ has no models. Then $\mathbf{H}_0 \models \mathbf{A}$. By the Extended Completeness Theorem, $\mathbf{H}_0 \vdash \mathbf{A}$. Therefore $\mathbf{H} \vdash \mathbf{A}$. **End of Proof.**

Let us say that a set $\mathbf{H}$ of wffs is **logically consistent** if there is no wff $\mathbf{A}$ for which $\mathbf{H} \vdash [\mathbf{A} \wedge \neg\mathbf{A}]$. From the last corollary, we have the following:

Corollary 1.11.7 *Suppose* $\mathbf{H}$ *is a finite or countable set of wffs. Then the following are equivalent:*

1. $\mathbf{H}$ *has a model.*
2. $\mathbf{H}$ *is logically consistent.*
3. $\mathbf{H}$ *has no tableau confutation.*

Proof: Exercise 31. **End of Proof.**

One application of the Propositional Compactness Theorem is that the Four Color Theorem for finite maps implies the Four Color Theorem for infinite maps. That is:

If every finite map in the plane can be colored with four colors so that no two adjacent countries have the same color, then the same is true for every infinite map in the plane.

Suppose that C is a set of countries on some given map. Introduce four proposition symbols

$$p_c^1, p_c^2, p_c^3, p_c^4$$

for each country c/inC. The proposition symbol p_c^i is meant to express the fact that the color of country c is i. We thus define the vocabulary $\mathcal{P}_0$ to be the set

$$\mathcal{P}_0 = \{p_c^1, p_c^2, p_c^3, p_c^4 \ : \ c \in C\}.$$

Let **H** be the set of all sentences of the following forms:

1. $p_c^1 \vee p_c^2 \vee p_c^3 \vee p_c^4$ for each c;
2. $p_c^i \Rightarrow \neg p_c^j$ for each c and for each $i \neq j$; and
3. $\neg[p_c^i \wedge p_{c'}^i]$ for each i and for each pair of distinct countries c and c' which are next to each other.

Now a model $\mathcal{M}$ for **H** corresponds to a coloring of the countries by the four colors $\{1, 2, 3, 4\}$ such that adjacent countries are colored differently. If every finite submap of the given map has a four coloring, then every finite subset of **H** has a model. By the Compactness Theorem **H** has a model, hence the entire map can be four colored.

For another application of the Compactness Theorem of this kind, see Exercise 32.

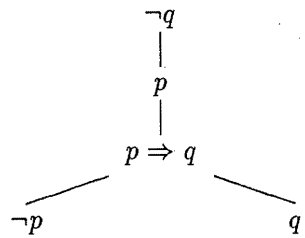
1.12 Valid Arguments

In this section we shall use tableaus to obtain some valid consequence patterns which arise frequently in mathematical proofs. Here is a first example.

Modus Ponens. From p and $p \Rightarrow q$ we may conclude q.

$$p, p \Rightarrow q \models q$$

In view of the Soundness Theorem, we need only give a tableau proof of q from the hypotheses p and $p \Rightarrow q$. Here it is.



If, in developing a mathematical proof, we happen to know that certain statements $\mathbf{A}_1, \ldots, \mathbf{A}_k$ are all true (they may have already been proved or they may be assumed as hypotheses) and we know that another statement **B** is a semantic consequence of $\mathbf{A}_1, \ldots, \mathbf{A}_k$, then we can conclude that **B** is also true. Thus, taking Modus Ponens as an example, if we can establish the truth of p and $p \Rightarrow q$, then we may conclude that q is also true.

Laws such as Modus Ponens are called **valid argument forms.** They are often used without being mentioned in ordinary mathematical proofs, and are helpful in understanding the plan of the proof. Here is a typical example of a mathematical proof which makes use of Modus Ponens.

Proposition 1.12.1 *There is an x in the interval $(1, \pi)$ such that $\ln(x) = \sin(x)$, where $\ln(x)$ is the natural logarithm of x.*

Proof: Let $f(x) = \ln(x) - \sin(x)$. We must show that there is an $x \in (1, \pi)$ such that $f(x) = 0$. The Intermediate Value Theorem states that if f is continuous on a closed interval $[a, b]$ and $f(a) < 0 < f(b)$, then there exists $x \in (a, b)$ such that $f(x) = 0$. We note that f is

continuous on $[1, \pi]$ and that $f(1) < 0 < f(\pi)$. Therefore there exists $x \in (1, \pi)$ such that $f(x) = 0$. **End of Proof.**

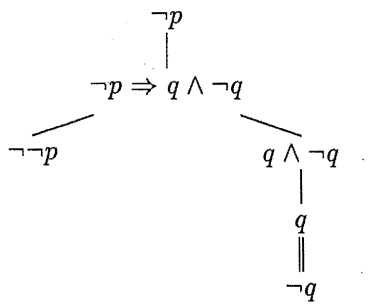
The above proof does not explicitly mention the Law of Modus Ponens. In fact, Modus Ponens is so familiar that it is rarely mentioned in a proof and should be understood as *implicit* in the argument. To see where Modus Ponens was used, let p be the statement "f is continuous on $[1, \pi]$ and $f(1) < 0 < f(\pi)$," and let q be the statement to be proved, "there exists $x \in (1, \pi)$ such that $f(x) = 0$." We know that p is true, and the Intermediate Value Theorem gives us $p \Rightarrow q$. The statement q follows from p and $p \Rightarrow q$ by Modus Ponens.

We shall now use tableaus to find two more valid argument forms, and illustrate them in actual mathematical proofs.

Indirect Proof. From $\neg p \Rightarrow [q \wedge \neg q]$ we may conclude p.

$$\neg p \Rightarrow [q \wedge \neg q] \models p$$

Verbally, the Indirect Proof Law says that in order to prove p, we may show that $\neg p$ leads to a contradiction. Here is a tableau proof.



The proof of Euclid's famous theorem that there are infinitely many prime numbers can by analyzed as an Indirect Proof.

Proposition 1.12.2 *There is no largest prime.*

Proof: Suppose there is a largest prime a. Let $b = a! + 1$. Let c be a prime number which divides b. Since a is the largest prime, $c \leq a$. However, no number $d \leq a$ divides b, so $\neg c \leq a$. We conclude that there is no largest prime. **End of Proof.**

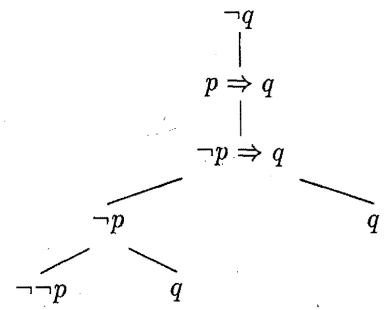
To see where the Indirect Proof Law was used, let p denote the sentence to be proved, in this case "a is not the largest prime." Let q be the statement "$c \leq a$." In the course of the proof we have shown that $\neg p \Rightarrow q \wedge \neg q$. Using the Indirect Proof Law, it follows that the desired conclusion p is true.

Here is another commonly used valid argument form and its tableau proof.

Proof by Cases. From $p \Rightarrow q$ and $\neg p \Rightarrow q$ we may conclude q.

$$p \Rightarrow q, \neg p \Rightarrow q \models q$$

Verbally, the Proof by Cases Law says that in order to prove q, we may prove that q holds in each of the two cases p and $\neg p$. Tableau proof:



We give an example of the Proof by Cases Law from the calculus.

Proposition 1.12.3 (Rolle's Theorem) *If a, b are real numbers with $a < b$, the function f is continuous on $[a, b]$ and differentiable on (a, b), and $f(a) = f(b) = 0$, then there is a number c in (a, b) such that $f'(c) = 0$.*

Proof: We find the desired number c in (a, b) as follows. If for all x in (a, b), $f(x) = 0$, then let c be any such x. If, on the other hand, there exists x in (a, b) such that $f(x) \neq 0$, then if $f(x) > 0$ we may take $(c, f(c))$ to be a maximum for f by continuity on $[a, b]$, and if $f(x) < 0$ we may take $(c, f(c))$ to be a minimum for f again by continuity. We have shown in each case that $a < c < b$ and $f'(c) = 0$. **End of Proof.**

Let us analyze the above proof and see where the Proof by Cases Law was used. Let q be the sentence to be proved, namely, "There is a number c in (a, b) such that $f'(c) = 0$," and let p be the sentence "For all x in (a, b), $f(x) = 0$." We have proved that $p \Rightarrow q$ and $\neg p \Rightarrow q$. It then follows using Proof by Cases that the desired conclusion q is true.

In Exercise 17 we consider some other valid argument forms which are commonly used in mathematics.

1.13 Tableau Problems (TAB1)

This is the first of three problem sets using the TABLEAU or TABWIN program. In this assignment you will construct tableau proofs in propositional logic. The problems are located in directory TAB1 on the distribution diskette, and the SETUPDOS or SETUPWIN program will put them in a subdirectory called TAB1 on your hard disk. This directory contains an assignment of seven problems, called CASES, CONTR, CYCLE, EQUIV, PIGEON, PENT, SQUARE. It also has the extra files SAMPLE, ASAMPLE, RAMSEY. SAMPLE is a sample problem and ASAMPLE is its solution. The problem RAMSEY is very difficult and is described below.

Use the TABLEAU or TABWIN program commands to load each problem, do your work, and then save your answer on your diskette or hard drive. Each problem consists of a list of hypotheses and/or a wff to be proved. Your solution should be a tableau proof, with every node colored red. The file name of your answer should be the letter A followed by the name of the problem. (For example, your answer to the CYCLE problem should be called ACYCLE). Be sure your name is on your diskette label.

The solutions to these computer problems will be similar to the two "hand" examples of tableau proofs given at the beginning of Section 1.7.

EXAMPLE 1 A rule for conjunctions of wffs.

Hypotheses: none

To prove: $p \Rightarrow [q \Rightarrow [p \wedge q]]$

The solution is given in Figure 1.2 and has 6 nonroot nodes.

EXAMPLE 2 The Transitivity Law.

Hypotheses: $p \Rightarrow q$, $q \Rightarrow r$

To prove: $p \Rightarrow r$

The solution is given in Figure 1.3 and also has 6 nonroot nodes.

At the end of this paragraph we list the set of problems in order of difficulty, with attached comments. For each TABLEAU problem in this book, an approximate value is given for the number of nodes in the solution: its *par value*. There will always be at least one solution with the suggested number of nodes, and in many cases there are solutions which use even fewer nodes. You are not required to find a solution with the suggested number of nodes. The par value is included only as a guide to the difficulty of the problem. We also list the number of entries in the truth table for the problem. This number is equal to $2^n * m$ where n is the number of distinct propositional symbols and m is the number of occurrences of propositional symbols and connectives. You are not required to build the truth table. Its size is given only so you can compare it with the size of the tableau proof.

CASES (8 nodes) (88 truth table entries) The rule of proof by cases.

Hypotheses: $a \Rightarrow c$, $b \Rightarrow c$

To prove: $[a \vee b] \Rightarrow c$

CONTR (12 nodes) (36 truth table entries) The law of contraposition.

Hypotheses: none

To prove: $[p \Rightarrow q] \Leftrightarrow [\neg q \Rightarrow \neg p]$

EQUIV (20 nodes) (72 truth table entries) Two wffs which are equivalent to a third wff are equivalent to each other.

Hypotheses: $p \Leftrightarrow q$, $q \Leftrightarrow r$

To prove: $p \Leftrightarrow r$

PIGEON (24 nodes) (88 truth table entries) The pigeonhole principle: Among any three propositions there must be a pair with the same truth value.

Hypotheses: None

To prove: $[p \Leftrightarrow q] \vee [p \Leftrightarrow r] \vee [q \Leftrightarrow r]$

CYCLE (26 nodes) (416 truth table entries) Given that four wffs imply each other around a cycle and at least one of them is true, prove that all of them are true.

Hypotheses: $p \Rightarrow q$, $q \Rightarrow r$, $r \Rightarrow s$, $s \Rightarrow p$, $p \vee q \vee r \vee s$;

To prove: $p \wedge q \wedge r \wedge s$

PENT (38 nodes) (55,320 truth table entries) It is not possible to color each side of a pentagon red or blue in such a way that adjacent sides are of different colors.

Hypotheses: $b1 \vee r1$, $b2 \vee r2$, $b3 \vee r3$, $b4 \vee r4$, $b5 \vee r5$, $\neg[b1 \wedge b2]$, $\neg[b2 \wedge b3]$, $\neg[b3 \wedge b4]$, $\neg[b4 \wedge b5]$, $\neg[b5 \wedge b1]$, $\neg[r1 \wedge r2]$, $\neg[r2 \wedge r3]$, $\neg[r3 \wedge r4]$, $\neg[r4 \wedge r5]$, $\neg[r5 \wedge r1]$

To prove: A tableau confutation.

SQUARE (58 nodes) (17,408 truth table entries) There are nine propositional symbols which can be arranged in a square:

$$\begin{array}{ccc} a1 & a2 & a3 \\ b1 & b2 & b3 \\ c1 & c2 & c3 \end{array}$$

Assume that there is a letter such that for every number the proposition is true (that is, there is a row of true propositions). Prove that for every number there is a letter for which the proposition is true (that is, each column contains a true proposition).

Hypothesis: $[a1 \wedge a2 \wedge a3] \vee [b1 \wedge b2 \wedge b3] \vee [c1 \wedge c2 \wedge c3]$

To prove: $[a1 \vee b1 \vee c1] \wedge [a2 \vee b2 \vee c2] \wedge [a3 \vee b3 \vee c3]$

RAMSEY (1140 nodes) (8,060,928 truth table entries) The simplest case of Ramsey's Theorem can be stated as follows. Out of any six people, there are either three people who all know each other or three people none of whom know each other. This problem has 15 proposition symbols $ab, ac, \ldots, ef$, which may be interpreted as meaning "a knows b," etc. The problem has a list of hypotheses which state that for any three people among a, b, c, d, e, f, there is at least one pair who know each other and one pair who do not know each other. Ramsey's Theorem says that these hypotheses are inconsistent and so must have a tableau confutation.

Here is an informal proof of Ramsey's Theorem in the case at hand. Select one of the people, say a. The five remaining people may be divided into two sets: those who know a and those who do not. At least one of these sets must have three people in it. Hence there are essentially two cases:

1. a knows all the people b, c, d. If none of b, c, d know each other, then $\{b, c, d\}$ is a set of three people none of whom know each other. If two of b, c, d know each other, say b knows c, then $\{a, b, c\}$ is a set of three people all of whom know each other.

2. a does not know any of the people b, c, d. If b, c, d know each other, then $\{b, c, d\}$ is a set of three people all of whom know each other. If two of b, c, d do not know each other, say b does not know c, then $\{a, b, c\}$ is a set of three people none of whom know each other.

The tableau confutation is very long since the rules of propositional tableaus do not allow us to rename the people as we have done in the informal proof. This problem is optional, and is included mainly to illustrate the power of the tableau method.

1.14 Exercises

1. For a wff $\mathbf{A}$ define $s(\mathbf{A})$ to be the number of occurrences of proposition symbols in $\mathbf{A}$, and $b(\mathbf{A})$ to be the number of occurrences of binary connectives ($\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$) in $\mathbf{A}$. Prove by induction on the length of wffs that for every wff $\mathbf{A}$,

$$s(\mathbf{A}) = b(\mathbf{A}) + 1.$$

2. Prove by induction on the length of wffs that every wff has the same number of left brackets as right brackets.

3. Prove by induction on the length of wffs that an initial part of a wff is either a string of negation symbols or has more left brackets than right brackets.

4. Let $\mathbf{C}$ be a wff which has the form $\mathbf{C} = [\mathbf{S} * \mathbf{T}]$, where $\mathbf{S}$ and $\mathbf{T}$ are strings. Prove that $*$ is the main connective of $\mathbf{C}$ if and only if $\mathbf{S}$ has the same number of left brackets as right brackets.

5. Show that there is a unique function c from the set of wffs on the vocabulary $\mathcal{P}_0$ to the set $\mathbf{N}$ of natural numbers such that

(basis) $c(p) = 0$ for any $p \in \mathcal{P}_0$.

(negation) $c(\neg\mathbf{A}) = c(\mathbf{A}) + 1$.

(binary) $c([\mathbf{A} * \mathbf{B}]) = c(\mathbf{A}) + c(\mathbf{B}) + 1$
for any binary connective $*$.

Prove that for any wff $\mathbf{A}$ the number $c(\mathbf{A})$ is the number of occurrences of connectives in $\mathbf{A}$. (A connective is one of the symbols $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$.)

6. Show that there is a unique function L from the set $WFF(\mathcal{P}_0)$ of wffs to the set $\mathbf{N}$ of natural numbers as follows:

(Basis) $L(p) = 1$ for $p \in \mathcal{P}_0$.

(Negation) $L(\neg\mathbf{A}) = L(\mathbf{A}) + 1$.

(Binary connective) $L([\mathbf{A} * \mathbf{B}]) = L(\mathbf{A}) + L(\mathbf{B}) + 3$

What information does $L(\mathbf{A})$ give about the wff $\mathbf{A}$?

7. Write the standard abbreviations of the following wffs. (You can use the TABLEAU program to check your answers).

1. $[[p \Rightarrow q] \Leftrightarrow \neg[r \wedge s]]$
2. $[p \Rightarrow [q \Leftrightarrow \neg[r \wedge s]]]$
3. $\neg[[p \Rightarrow q] \Leftrightarrow [r \wedge s]]$
4. $\neg[p \Rightarrow [q \Leftrightarrow [r \wedge s]]]$

8. Write the wffs with the following standard abbreviations.

1. $p \wedge q \vee r \Rightarrow s$
2. $p \wedge [q \vee r] \Rightarrow s$
3. $p \wedge q \vee [r \Rightarrow s]$
4. $[p \wedge q \vee r] \Rightarrow s$
5. $p \wedge [q \vee r \Rightarrow s]$

9. Prove the following rule for finding the main connective of a wff $\mathbf{C}$ given only the standard abbreviation $\mathbf{C}'$. If there is an occurrence of a binary connective $*$ in $\mathbf{C}'$ which is preceded by the same number of left brackets as right brackets, then $*$ is the main connective of $\mathbf{C}$. Otherwise, $\mathbf{C}$ is either a proposition symbol or $\mathbf{C}$ is the negation of a wff.

(This proof requires a more difficult induction on the length of wffs.)

10. The purpose of this exercise is to show that bad things could happen without the Unique Readability Theorem. Let P be a subset

of the set of integers $\mathbf{Z}$. Define the set of *well formed integers* with the vocabulary P to be the set of all integers which can be obtained by finitely many applications of the following *rules of formation*:

- If $p \in P$, then p is a well formed integer.

- If a and b are well formed integers then so is their product ab.

The set of all well formed integers with the vocabulary P is denoted by $W(P)$. In the following take $P = \{-1, 2, 5\}$.

(i) Find all well formed integers a such that $-31 \leq a \leq 31$.

(ii) Show that the analog of the Unique Readability Theorem fails by exhibiting well-formed integers a_1, a_2, b_1, b_2 such that $a_1 b_1 = a_2 b_2$ but $a_1 \neq a_2$.

(iii) Show that for any function $g : P \to \mathbf{Z}$ and any function $f : \mathbf{Z} \times \mathbf{Z} \to \mathbf{Z}$ there is at most one function $\phi : W(P) \to \mathbf{Z}$ such that

- If $p \in P$, then $\phi(p) = g(p)$.
- If $a, b \in W(P)$ then $\phi(ab) = f(\phi(a), \phi(b))$.

(iv) Show that no such function ϕ exists when $g(p) = p$ and $f(a, b) = a$.

(v) Show that there is such a function ϕ when $g(-1) = -1$, $g(2) = g(5) = 1$, and $f(a, b) = ab$. What is it?

11. Prove Theorem 1.5.1, that for a given a model $\mathcal{M}$ and wff $\mathbf{A}$, the truth value $\mathbf{A}_{\mathcal{M}}$ is the same for all parsing sequences of $\mathbf{A}$. (Hint: Use the Unique Readability Theorem and an induction on the length of wffs.)

12. Show that the following are tautologies, first by using truth tables and then using tableaus.

(1)	$\neg\neg p \Leftrightarrow p$	*(Double Negation Law)*
(2)	$[p \wedge q] \wedge r \Leftrightarrow p \wedge [q \wedge r]$	*(Associative Law)*
(3)	$[p \vee q] \vee r \Leftrightarrow p \vee [q \vee r]$	*(Associative Law)*
(4)	$p \wedge q \Leftrightarrow q \wedge p$	*(Commutative Law)*
(5)	$p \vee q \Leftrightarrow q \vee p$	*(Commutative Law)*
(6)	$p \wedge [q \vee r] \Leftrightarrow [p \wedge q] \vee [p \wedge r]$	*(Distributive Law)*
(7)	$p \vee [q \wedge r] \Leftrightarrow [p \vee q] \wedge [p \vee r]$	*(Distributive Law)*
(8)	$p \Rightarrow [q \Rightarrow r] \Leftrightarrow [p \Rightarrow q] \Rightarrow [p \Rightarrow r]$	*(Self-Distributive Law)*
(9)	$\neg[p \vee q] \Leftrightarrow \neg p \wedge \neg q$	*(DeMorgan's Law)*
(10)	$\neg[p \wedge q] \Leftrightarrow \neg p \vee \neg q$	*(DeMorgan's Law)*
(11)	$[[p \Rightarrow q] \Rightarrow p] \Rightarrow p$	*(Peirce's Law)*

13. Let $\mathcal{M}$ be the model for propositional logic such that $p_{\mathcal{M}} = \mathbf{T}$ for every proposition symbol p. Prove by induction on length that for every wff $\mathbf{A}$: Either the $\neg$ symbol occurs in $\mathbf{A}$, or $\mathcal{M} \models \mathbf{A}$.

14. Show that $[\mathbf{A} \Rightarrow \mathbf{B}] \Rightarrow \mathbf{A}$ is a tautology if $\mathbf{A}$ is $p \Rightarrow p$ and $\mathbf{B}$ is q, but is *not* a tautology if $\mathbf{A}$ and $\mathbf{B}$ are both $p \Rightarrow q$. (The aim of this exercise is to make sure you distinguish between a proposition symbol p and a variable $\mathbf{A}$ used to stand for a wff which may have more complicated structure.)

15. We say that two wffs $\mathbf{A}$ and $\mathbf{B}$ are **logically equivalent** if the wff $\mathbf{A} \Leftrightarrow \mathbf{B}$ is a tautology. Show that for any wff $\mathbf{A}$ there is a wff $\mathbf{B}$ such that $\mathbf{A}$ and $\mathbf{B}$ are logically equivalent and the only connectives which occur in $\mathbf{B}$ are $\neg$ and $\wedge$. Do the same for the connectives $\neg$ and $\Rightarrow$.

16. If p is a proposition symbol and $\mathbf{C}$ is a propositional wff, then for each propositional wff $\mathbf{A}$, the wff $\mathbf{A}(p//\mathbf{C})$ formed by substituting $\mathbf{C}$ for p in $\mathbf{A}$ is defined inductively by:

(a) $p(p//\mathbf{C}) = \mathbf{C}$.

(b) If q is a proposition symbol different from p, then $q(p//\mathbf{C}) = q$.

(c) $(\neg\mathbf{A})(p//\mathbf{C}) = \neg(\mathbf{A}(p//\mathbf{C}))$.

(d) For each binary connective $*$

$$[\mathbf{A} * \mathbf{B}](p//\mathbf{C}) = [\mathbf{A}(p//\mathbf{C}) * \mathbf{B}(p//\mathbf{C})].$$

For example,

$$[[p \Leftrightarrow r] \Rightarrow p](p//q \wedge p) \text{ is } [[q \wedge p] \Leftrightarrow r] \Rightarrow [q \wedge p].$$

Prove that for any proposition symbol p and wffs $\mathbf{A}, \mathbf{B}$, and $\mathbf{C}$,

$$[\mathbf{B} \Leftrightarrow \mathbf{C}] \Rightarrow [\mathbf{A}(p//\mathbf{B}) \Leftrightarrow \mathbf{A}(p//\mathbf{C})]$$

is a tautology. (Show by induction on the the length of the wff $\mathbf{A}$ that in every model of

$$\mathbf{B} \Leftrightarrow \mathbf{C},$$

the two wffs

$$\mathbf{A}(p//\mathbf{B}) \quad , \quad \mathbf{A}(p//\mathbf{C})$$

have the same truth value.)

17. Here are some additional valid argument forms which are frequently used in mathematical proofs. Give a tableau proof for each one.

(i) $p \Rightarrow q, \neg q \models \neg p$

(ii) $p \models q \Rightarrow p$

(iii) $p \vee q, \neg p \models q$

(iv (**Contraposition Law**) $\neg q \Rightarrow \neg p \models p \Rightarrow q$

(v) (**Transitive Law**) $p \Rightarrow q, q \Rightarrow r \models p \Rightarrow r$

(vi) $p \Rightarrow [q \vee r], q \Rightarrow t, r \Rightarrow t \models p \Rightarrow t.$

18. In this exercise you are asked to provide a proof of the given statement using the given argument form.

(1) "The square root of 2 is irrational."

Use the Indirect Proof Law. (Hint: Assume there is a number m/n, with m and n integers, whose square is 2 and arrive at a contradiction.)

(2) "Between any two rational numbers there is an irrational number."

Use the Proof by Cases Law. (Hint: You may first wish to prove that for any integer k and any prime p, $k + (1/\sqrt{p})$ is irrational; see part (1) above.)

(3) "If n is an odd integer, then n^2 is odd."

Use the Contraposition Law.

(4) "If x, y are real numbers, then $x \neq y$ implies $e^x \neq e^y$."

Use the law $p \Rightarrow [q \vee r], q \Rightarrow t, r \Rightarrow t \models p \Rightarrow t$.

(5) "If $2^n - 1$ is a prime number, so is n."

Use the Contraposition Law.

19. In this exercise we present several well-known theorems and their proofs. In each proof, find a valid argument form that is used.

(a) **Definition.** A function f with domain A is *one-one* if for all $x, y \in A$, $f(x) = f(y)$ implies $x = y$ (see also Section A.5 in the Appendix). A function f is *left cancellable* if for all sets A and all $g_1 : A \to B$, $g_2 : A \to B$, if $f \circ g_1 = f \circ g_2$ then $g_1 = g_2$. (See the Appendix for the definition of $f \circ g$.)

Theorem. If f is left cancellable, then f is one-one.

Proof. Suppose f with domain B is not one-one. Then there are $x \neq y$ with $f(x) = f(y)$. Define $g_1 : \{0\} \to B$, $g_2 : \{0\} \to B$ by $g_1(0) = x$, $g_2(0) = y$. Now $f \circ g_1 = f \circ g_2$ but $g_1 \neq g_2$. Thus f is not left cancellable.

(b) **Theorem.** (Subgroups of cyclic groups are cyclic.) Suppose A is a set of integers (recall the set $\mathbf{Z}$ of integers consists of the numbers $\ldots -2, -1, 0, 1, 2, \ldots$) and A is closed under subtraction (i.e. for all $x, y \in A$, $x - y \in A$ as well). Then there is an $n \in \mathbf{N}$ such that every $m \in A$ is a multiple of n.

Proof. Let n be the least positive integer in A. Given $m \in A$, use long division to write $m = nq + r$ for $q, r \in \mathbf{Z}$ and $r \geq 0, r < n$. Now m, nq are in A (why?). Since A is closed under subtraction, $r = m - nq \in A$. Since n is the least *positive* integer in A, $r \geq 0$, and $r < n$, it follows that $r = 0$. Hence, $r = 0$ and $m = nq$, as required.

(c) **Theorem.** (Fundamental Theorem of Arithmetic) Every composite positive integer (i.e. an integer greater than one which is not prime) is a product of primes.

Proof. Suppose not, i.e., suppose there is a composite number c which has no prime factorization. Let $k \in \mathbf{N}$ be such that $2^k > c$. Since c is composite but unfactorable into primes, we can write $c = c_1 d_1$ where c_1 is composite and also unfactorable into primes. Similarly, write $c_1 = c_2 d_2$ where c_2 is composite and unfactorable into primes. Continuing in this way, obtain $c_{k-1} = c_k d_k$. Now $c = c_1 d_1 = c_2 d_2 d_1 = \cdots = c_k d_k d_{k-1} \cdots d_1 \geq c_k \cdot 2^k > c$, which is impossible.

(d) Examine Cantor's Theorem given in Appendix A.6. What is the argument form?

20. Prove that Lemma 1.8.1 holds for infinite tableaus.

21. The Kill command in the TABLEAU program works as follows when it is invoked with the cursor at a node t. If there is a double line below t, (i.e. t and its child were added together) then every node below the child of t is removed from the tableau. Otherwise, every node below t is removed from the tableau. Using the definition of propositional tableau, prove that if you have a tableau before invoking the Kill command, then you have a tableau after using the Kill command.

22. Prove: If $\mathbf{A}$ has a tableau proof then $\mathbf{A}(p//\mathbf{C})$ has a tableau proof with the same number of nodes (in fact, with the same tree but different wffs assigned to the nodes).

23. Let $\mathbf{H}$ be a finite set of propositional wffs. By a **strict confutation** of $\mathbf{H}$ we mean a tableau $\mathbf{T}$ with root $\mathbf{H}$ such that every branch of $\mathbf{T}$ has a contradictory pair of the form $\{s, \neg s\}$ where s is a *propositional symbol.*

(a) Give a strict confutation of the set

$$\mathbf{H} = \{[\neg p \vee [q \wedge r]], \neg[\neg p \vee [q \wedge r]]\}.$$

(b) Prove by induction of the length of wffs that for every wff $\mathbf{A}$, the set $\mathbf{H} = \{\mathbf{A}, \neg\mathbf{A}\}$ has a strict confutation.

(c) Using part (b), prove that every finite set $\mathbf{H}$ of wffs which has a tableau confutation has a strict confutation.

24. Use the Soundness and Completeness Theorems for propositional logic to prove that if $\mathbf{A}$ has a tableau proof from $\mathbf{H}$ and $\mathbf{B}$ has a tableau proof from $\mathbf{A}$, then $\mathbf{B}$ has a tableau proof from $\mathbf{H}$.

25. Use the Soundness and Completeness Theorems to prove that if $[\mathbf{A} \vee \mathbf{B}]$ has a tableau proof from $\mathbf{H}$, $\mathbf{C}$ has a tableau proof from $\mathbf{A}$, and $\mathbf{C}$ has a tableau proof from $\mathbf{B}$, then $\mathbf{C}$ has a tableau proof from $\mathbf{H}$.

26.

(a) Make a finished tableau with the single hypothesis

$$[q \Rightarrow p \wedge \neg r] \wedge [t \vee r].$$

(b) Choose one of the finished branches, Γ, and circle the terminal node of Γ.

(c) Using the Finished Set Lemma, find a wff $\mathbf{A}$ such that:

1. **A** has exactly the same models as the set of wffs on the branch Γ which you chose, and
2. The only connectives occurring in **A** are $\wedge$ and $\neg$.

27. Let **T** be a finished tableau with finite hypothesis set **H** in which every wff is used at most once on each branch. Prove that each branch of **T** has at most $2n+1$ nodes, where n is the total number of connectives occurring in wffs in the set **H**.

28. In this exercise we describe an extremely simple language to give the reader an easy example of the Soundness and Completeness Theorems.

The vocabulary for "baby logic" is a nonempty set $\mathcal{P}_0$ of proposition symbols. The primitive symbols are the proposition symbols from $\mathcal{P}_0$ together with the connective $\neg$. A string in this language is a wff if it is obtained from finitely many applications of the following rules of formation.

Each p in $\mathcal{P}_0$ is a wff.

If **A** is a wff, then $\neg$**A** is a wff.

Given a model $\mathcal{M}$ of type $\mathcal{P}_0$, we obtain, as in the text, a uniquely defined function which assigns a truth value $\mathbf{A}_\mathcal{M}$ to each wff **A** of baby logic according to the rules

If **A** is a propositional symbol p, $\mathbf{A}_\mathcal{M} = p_\mathcal{M}$.

$\neg\mathbf{A}_\mathcal{M} = \mathbf{T}$ iff $\mathbf{A}_\mathcal{M} = \mathbf{F}$.

Tableaus are also defined as before, but now every tableau has only one branch.

Without using the Soundness and Completeness Theorems for Propositional Logic, prove these theorems for baby logic; i.e., prove

(Soundness) If there is a tableau proof of **A** from **H**, then $\mathbf{H} \models \mathbf{A}$.

(Completeness) If $\mathbf{H} \models \mathbf{A}$, there is a tableau proof of **A** from **H**.

(Hint: One approach is to mimic the lemmas used to prove these theorems for Propositional Logic in the text. This approach will provide the student with easy special cases of these lemmas. Another approach is as follows. For any $p \in \mathcal{P}_0$ and natural number n, define $\neg^n p$ by induction with the rules: $\neg^0 p = p$, $\neg^{n+1} p = \neg\neg^n p$. As a main lemma, show that there is a tableau confutation of a hypothesis set **H** if and only if there are $p \in \mathcal{P}_0$ and natural numbers m, n such that m is even, n is odd, $\neg^m p \in \mathbf{H}$, and $\neg^n p \in \mathbf{H}$.

29. Let X and Y be sets and R be a binary relation between X and Y, i.e. $R \subset X \times Y$. For each $x \in X$ define

$$R_x = \{y \in Y : (x, y) \in R\}$$

Assume

(1) for every finite $S \subset X$ there exists a one-one function $f : S \to Y$ such that $f(x) \in R_x$ for $x \in S$;

(2) for every $x \in X$ the set R_x is finite.

(a) Show that there exists a one-one function $F : X \to Y$ such that $F(x) \in R_x$ for all $x \in X$. Hint: For each $a \in X$, and $b \in Y$ introduce a proposition symbol p_{ab} whose intended interpretation is $F(a) = b$. Use the Compactness Theorem.

(b) Give an example which shows that hypothesis (2) cannot be dropped. Hint: The negation of (2) asserts that at least one R_x is not finite. In the example, there should be no *one-one* function $F : X \to Y$ such that $F(x) \in R_x$ for all $x \in X$.

30. Give an example of a tree with infinitely many nodes that has no infinite branch. Why does this not contradict the König Tree Theorem?

31. Prove Corollary 1.11.7

32. Given a countable set of students and a countable set of classes, suppose each student wants one of a finite set of classes, and each class

has a finite enrollment limit. Prove that *if each finite set of students can be accommodated, then the whole set can.* Hint: Use the Compactness Theorem. Let your basic proposition symbols consist of p_{sc} where s is a student and c is a class: p_{sc} is intended to mean *student s will take class c.*

Polish notation for propositional logic is defined as follows. The logical symbols are $\{\wedge, \vee, \neg, \Leftrightarrow, \Rightarrow\}$, and the nonlogical symbols or proposition symbols are the elements of an arbitrary set $\mathcal{P}_0$. The well-formed formulas in Polish notation (wffpn) are the members of the smallest set of strings which satisfy:

1. Each $p \in \mathcal{P}_0$ is wffpn;
2. If $\mathbf{A}$ is wffpn, then so is $\neg\mathbf{A}$;
3. If $\mathbf{A}$ is wffpn and $\mathbf{B}$ is wffpn, then $\wedge\mathbf{AB}$ is wffpn, $\vee\mathbf{AB}$ is wffpn, $\Leftrightarrow\mathbf{AB}$ is wffpn, and $\Rightarrow\mathbf{AB}$ is wffpn.

Note that no parentheses or brackets are needed for Polish notation.

33. Put the wff $[p \Leftrightarrow q] \Rightarrow [\neg q \vee r]$ into Polish notation.

34. Construct a parsing sequence for the wffpn

$$\vee\neg \Rightarrow pq \Leftrightarrow rp$$

to verify that it is wffpn. Write this wff in regular notation.

35. Prove using induction on length that for any wffpn $\mathbf{A}$, the number of occurrences of logical symbols of the kind $\{\wedge, \vee, \Leftrightarrow, \Rightarrow\}$ in $\mathbf{A}$ is always exactly one less than the number of occurrences of proposition symbols.

36. Using induction on length, prove that for any wffpn $\mathbf{A}$ and any occurrence of a proposition symbol p in $\mathbf{A}$ except the last, the number of logical symbols of the kind $\{\wedge, \vee, \Leftrightarrow, \Rightarrow\}$ to the left of p is strictly greater than the number of proposition symbols to the left of p.

37. State and prove a Unique Readability Theorem for wffs in Polish notation.

Chapter 2

Pure Predicate Logic

In this chapter we study the family of languages known as **first-order languages** or **predicate logics**. These languages have the quantifiers *for all* and *there exists*. Instead of propositional symbols they have **predicates**. As in the first chapter, we shall develop the concepts of a wff, a formal proof, and a model, and prove a Completeness Theorem which ties them together. Predicate logic is rich enough to express the statements and prove the theorems which arise in ordinary mathematical practice.

2.1 Introduction

A **predicate** is a word or phrase like *is a man*, *is less than*, *belongs to*, or even *is* which can be combined with one or more names of individuals to yield meaningful sentences. For example, *Socrates is a man*, *Two is less than four*, *This hat belongs to me*, *He is her partner*. Names of specific individuals are called **parameters**. Symbols called **variables** stand for arbitrary individuals. If the variables in an expression are replaced by parameters the result acquires a meaning. For example, in the assertion

$$P(x) : x \text{ is less than } 4$$

we understand that the variable x stands for any number in the particular class of numbers we are studying (e.g. the natural numbers, the real numbers, etc.). For instance, if x is understood to stand for

a natural number in this example, and we replace x by the number 1, the assertion

$$P(1) : 1 \text{ is less than } 4$$

is true, whereas replacing x by 5 yields the false statement

$$P(5) : 5 \text{ is less than } 4.$$

The number of variables associated with a predicate is called the **arity** of the predicate. Hence, the predicate

$$P(x) : x \text{ is less than } 4$$

is a 1-ary, or **unary** predicate;

$$Q(x, y) : x \text{ is less than } y$$

is a 2-ary or **binary** predicate; and

$$R(x, y, z) : x \text{ is between } y \text{ and } z$$

is a 3-ary, or **ternary** predicate.

If $P(x_1, \ldots, x_n)$ is an n-ary predicate and if $a_1, \ldots, a_n$ are values such that $P(a_1, \ldots, a_n)$ is true we say that $(a_1, \ldots, a_n)$ **satisfies** P. Thus in the above examples, 1 satisfies P, $(1, 2)$ satisfies Q, but $(1, 2, 3)$ does *not* satisfy R.

The predicate logic developed here will be called **pure predicate logic** to distinguish it from the **full predicate logic** of the next chapter. (Full predicate logic will add to pure predicate logic the expressive power of constants, functions, and equality).

A unary predicate determines a set of things; namely those things for which it is true. Similarly, a binary predicate determines a set of pairs of things – a **binary relation** – and in general an n-ary predicate determines an n-**ary relation**. For example, the predicate *is a man* determines the set of men and the predicate *is west of* (when applied to American cities) determines the set of pairs (a, b) of American cities such that a is west of b. (For example, the relation holds between Chicago and New York and does not hold between New York and Chicago.) Different predicates may determine the same relation (for example, x *is west of* y and y *is east of* x.)

The phrase **for all** is called the **universal quantifier** and is denoted symbolically by $\forall$. The phrases **there exists**, **there is a**, and **for some** all have the same meaning: **there exists** is called the **existential quantifier** and is denoted symbolically by $\exists$.

The universal quantifier is like an iterated conjunction and the existential quantifier is like an iterated disjunction. To understand this, suppose that there are only finitely many individuals; that is the variable x takes on only the values $a_1, a_2, \ldots, a_n$. Then the sentence $\forall x\, P(x)$ means the same as the sentence $P(a_1) \wedge P(a_2) \wedge \ldots \wedge P(a_n)$ and the sentence $\exists x\, P(x)$ means the same as the sentence $P(a_1) \vee P(a_2) \vee \ldots \vee P(a_n)$. In other words, if

$$\forall x[x = a_1 \vee x = a_2 \vee \ldots \vee x = a_n]$$

then

$$[\forall x P(x)] \Leftrightarrow [P(a_1) \wedge P(a_2) \wedge \ldots \wedge P(a_n)]$$

and

$$[\exists x P(x)] \Leftrightarrow [P(a_1) \vee P(a_2) \vee \ldots \vee P(a_n)].$$

Of course, if the number of distinct individuals is infinite, such an interpretation of quantifiers is not possible since infinitely long sentences are not allowed in predicate logic.

The similarity between $\forall$ and $\wedge$ and between $\exists$ and $\vee$ suggests many logical laws. For example, **DeMorgan's laws**

$$\neg[p \vee q] \Leftrightarrow [\neg p \wedge \neg q], \qquad \neg[p \wedge q] \Leftrightarrow [\neg p \vee \neg q],$$

have the following versions in predicate logic:

$$\neg\exists x P(x) \Leftrightarrow \forall x \neg P(x), \qquad \neg\forall x P(x) \Leftrightarrow \exists x \neg P(x).$$

In sentences of form $\forall x P(x)$ or $\exists x P(x)$, the variable x is called a **dummy variable** or a **bound variable**. The meaning of the sentence is unchanged if the variable x is replaced everywhere by another variable. Thus the sentences

$$\forall x P(x) \Leftrightarrow \forall y P(y), \qquad \exists x P(x) \Leftrightarrow \exists y P(y),$$

are both true. For example, the sentence *there is an* x *satisfying* $x + 7 = 5$ has exactly the same meaning as the sentence *there is a* y *satisfying*

$y + 7 = 5$. We say that the second sentence arises from the first by **alphabetic change of a bound variable.**

In mathematics, universal quantifiers are not always explicitly inserted in a text but must be understood by the reader. For example, when an algebra textbook contains the equation

$$x + y = y + x$$

the author means

$$\forall x \forall y \quad x + y = y + x.$$

(The former equation is called an **identity**, since it is true for all values of the variables, as opposed to an **equation to be solved** where the object is to find those values of the variables which make the equation true.)

A precise notation for predicate logic is important because natural language is ambiguous in certain situations. Particularly troublesome in English is the word *any* which sometimes means *for all* and sometimes *there exists*, depending on the context.

2.2 Syntax of Predicate Logic

A **vocabulary** $\mathcal{P}$ for pure predicate logic consists of a set $\mathcal{P}_n$ of n-ary **predicate symbols** for each natural number $n = 0, 1, \ldots$, where at least one of the sets $\mathcal{P}_n$ is nonempty. The 0-ary predicate symbols are just propositional symbols as in propositional logic. The words *unary, binary, ternary* mean respectively 1-ary, 2-ary, 3-ary. In the intended interpretation of predicate logic the predicate symbols denote relations such as $x < y$ or $x + y = z$.

In addition to the primitive symbols of propositional logic the following are **primitive symbols** of pure predicate logic:

- the *predicate symbols* from $\mathcal{P}_0, \mathcal{P}_1, \mathcal{P}_2, \ldots$;
- an infinite set

  $$VAR = \{x, y, z, x_0, y_0, z_0, x_1, y_1, \ldots\}$$

  of symbols which are called *variables*;

- a set $\mathcal{K}$, possibly empty, of symbols which are called *parameters*;
- the *right and left parenthesis and comma* (,);
- the *universal quantifier* $\forall$;
- the *existential quantifier* $\exists$.

For the syntax, the only difference between a variable and a parameter is that the latter may not appear immediately after a quantifier in a wff. The reason for having parameters is that they will make it much easier to develop the semantics for predicate logic, beginning in Section 2.4. (The parameters will denote particular elements of a model and the variables will stand for arbitrary members of a model).

Definition 2.2.1 A symbol which is either a variable or a parameter is called an **individual symbol**. When we wish to emphasize the similarity between them, we will sometimes call variables **individual variables**, and call parameters **individual parameters.**

Any finite sequence of symbols of any kind is called a **string**. Our first task is to specify the **syntax** of pure predicate logic; i.e. to specify which strings are grammatically correct. These strings are called **well-formed formulas**. The phrase *well-formed formula* is often abbreviated to **wff**.

Definition 2.2.2 A **wff of pure predicate logic** is a string which can be obtained by finitely many applications of the following **rules of formation**:

(W:$\mathcal{P}_0$) Any proposition symbol from $\mathcal{P}_0$ is a wff;

(W:$\mathcal{P}_n$) If $u_1, u_2, \ldots, u_n$ are individual symbols (variables or parameters), and $p \in \mathcal{P}_n$ is an n-ary predicate symbol, then $p(u_1, u_2, \ldots, u_n)$ is a wff;

(W:$\neg$) If **A** is a wff, the $\neg$**A** is a wff;

(W:$\wedge, \vee, \Rightarrow, \Leftrightarrow$) If **A** and **B** are wffs, then [**A** $\wedge$ **B**], [**A** $\vee$ **B**], [**A** $\Rightarrow$ **B**], and [**A** $\Leftrightarrow$ **B**] are wffs;

(W:∀,∃) If **A** is a wff and x is a variable, then the strings $\forall x\mathbf{A}$ and $\exists x\mathbf{A}$ are wffs.

If we wish to emphasize that the predicate symbols appearing in a wff **A** come from a specific vocabulary $\mathcal{P}$, and that the parameters come from a set $\mathcal{K}$, we say that the wff is formed from the vocabulary $\mathcal{P}$ with parameters from $\mathcal{K}$. The set of all wffs formed from the vocabulary $\mathcal{P}$ with parameters from $\mathcal{K}$ will be denoted by $WFF(\mathcal{P},\mathcal{K})$.

The wffs obtained from the basic rules (W:$\mathcal{P}_0$) and (W:$\mathcal{P}_n$) are called **atomic** wffs. Thus the atomic wffs are precisely those wffs in which no connectives or quantifiers occur.

To show that a particular string of symbols is a wff we construct a sequence of wffs using this definition. This is called **parsing** the wff and the sequence is called a **parsing sequence**. Although it is never difficult to tell if a short string is a wff, the parsing sequence is important for theoretical reasons.

As an example, let us assume that $\mathcal{P}_0$ contains a propositional symbol q, and that $\mathcal{P}_1$ contains a unary predicate symbol P. We first parse the wff $\forall x[P(x) \Rightarrow q]$.

(1) $P(x)$ is a wff by (W:$\mathcal{P}_1$).

(2) q is a wff by (W:$\mathcal{P}_0$).

(3) $[P(x) \Rightarrow q]$ is a wff by (1), (2), and (W:⇒).

(4) $\forall x[P(x) \Rightarrow q]$ is a wff by (3) and (W:∀).

Now we parse the wff $[\forall xP(x) \Rightarrow q]$.

(1) $P(x)$ is a wff by (W:$\mathcal{P}_1$).

(2) $\forall xP(x)$ is a wff by (1) and (W:∀).

(3) q is a wff by (W:$\mathcal{P}_0$).

(4) is a wff by (2), (3) and (W:⇒).

The two wffs are alike except for the location of the brackets. In the parsing sequence for the first wff, $\forall x[P(x) \Rightarrow q]$, the ⇒ must be introduced before the ∀, but in the parsing sequence for the second wff, $[\forall xP(x) \Rightarrow q]$, the ∀ must be introduced before the ⇒.

We continue using the abbreviations and conventions introduced in the propositional logic chapter and in addition add a few more.

- We shall use $\doteq$ rather than $=$ for a predicate symbol corresponding to equality in our formal language, to avoid confusion with the ordinary equality symbol used outside of predicate logic.
- Certain well-known binary predicates like $\doteq$ and $<$ are traditionally written between the variables (for example $x < y$) rather than before the variables (for example $<(x,y)$), and we continue this practice. Expressions such as $x < y$ are said to be written in **infix notation**.
- The three rules (W:¬), (W:∀), and (W:∃) put brackets around wffs in the same way. Thus $\neg P(x) \Rightarrow q$ means $[\neg P(x) \Rightarrow q]$ rather than $\neg[P(x) \Rightarrow q]$. Likewise $\forall xP(x) \Rightarrow q$ means $[\forall xP(x) \Rightarrow q]$ and not $\forall x[P(x) \Rightarrow q]$. Since it is easy to confuse these two, we may insert extra brackets and write $[\forall xP(x)] \Rightarrow q$ for $\forall xP(x) \Rightarrow q$. Thus, an abbreviated wff can actually contain more brackets than an unabbreviated wff.

The following lemma is proved in the same way as the corresponding lemma in propositional logic, by induction on the length of wffs.

Lemma 2.2.3 *In pure predicate logic, no initial part of a wff is a wff.*

Each wff of pure predicate logic is either an atomic wff, starts with a negation symbol or quantifier, or starts with a left bracket. As before, the wffs which start with a left bracket are formed by combining two other wffs with a binary connective called the main connective.

Theorem 2.2.4 (Unique Readability) *Each wff* **C** *of pure predicate logic which starts with a left bracket has exactly one main connective* $*$ *such that* $\mathbf{C} = [\mathbf{A} * \mathbf{B}]$ *where* **A** *and* **B** *are wffs.*

The proof is an easy modification of the Unique Readability Theorem on page 11 and is left as an exercise.

2.3 Free and Bound Variables

In predicate logic, an individual symbol x may appear in several different places in the same wff **A**. We shall call each place where a symbol or string s appears in **A** an **occurrence** of s in **A**. It is important to distinguish between two kinds of occurrences of a variable in a wff, free and bound occurrences. Informally, the free occurrences of variables stand for elements of a universe set, and the truth value of a wff will depend on which element is assigned to the free occurrences of individual symbols. On the other hand, the bound occurrences of variables are dummy variables which appear within quantifiers.

We first declare that every occurrence of an individual parameter in a wff is free. For individual variables, we shall first define the notion of a bound occurrence and then declare that all other occurrences are free.

A wff **B** is said to be a **well-formed part** of a wff **A** if **A** is **SBT** for some strings **S** and **T**.

Let x be a variable and **Q** be a quantifier, either $\forall$ or $\exists$, such that $\mathbf{Q}x$ occurs in **A**. Suppose that **B** is a well formed part of **A**, so that $\mathbf{A} = \mathbf{SBT}$ for some strings **S** and **T**, and that **B** begins with $\mathbf{Q}x$. Thus **B** is a wff of the form $\mathbf{Q}x\,\mathbf{C}$. **B** is called the **scope** of that occurrence of the quantifier **Q** in **A**. We shall show later that the scope of a quantifier in a wff is unique. Every occurrence of x in the wff $\mathbf{B} = \mathbf{Q}x\,\mathbf{C}$ (including the occurrence immediately after the **Q**) is called a **bound occurrence** of x in **A**. Any occurrence of x in **A** which is not a bound occurrence is called a **free occurrence** of x in **A**.

For example, in the wff

$$P(x,y) \Rightarrow \forall x[\exists y R(x,y) \Rightarrow Q(x,y)],$$

the first occurrence of x is free, the three remaining occurrences of x are bound, the first and last occurrences of y are free, the second and third occurrences of y are bound, the wff

$$\forall x[\exists y R(x,y) \Rightarrow Q(x,y)]$$

is the scope of the quantifier $\forall x$ and the wff $\exists y R(x,y)$ is the scope of the quantifier $\exists y$. If we make a change of bound variable (say replacing

all bound occurrences of x by u and all bound occurrences of y by v) we obtain the wff

$$P(x,y) \Rightarrow \forall u[\exists v\, R(u,v) \Rightarrow Q(u,y)]$$

which has exactly the same meaning as the original wff.

Before going further, we shall prove that a quantifier occurrence has only one scope in a wff.

Theorem 2.3.1 (Unique Scope) *For each occurrence* **Q** *of a quantifier in a wff* **A**, *there is a unique well formed part of* **A** *which begins with* **Q**. *This unique well formed part of* **A** *is called the* **scope** *of that occurrence of* **Q**.

Proof: We first prove the existence of a scope by induction on the length of **A**. Let $P(n)$ be the property that for each wff **A** of length $\leq n$, each occurrence **Q** of a quantifier in **A** is the beginning of at least one well formed part of **A**. An easy proof by induction shows that $P(n)$ is true for all natural numbers n. Thus every occurrence of a quantifier has at least one scope.

The proof of the uniqueness of the scope uses the lemma that no initial part of a wff is a wff. Let **Q** be an occurrence of a quantifier in a wff **A**, and suppose **B** and **C** are two well formed parts of **A** which begin with **Q**. Since **B** and **C** both start at **Q** and neither one can be an initial part of the other, **B** and **C** are the same. Thus there is only one well formed part of **A** which begins with **Q**. **End of Proof.**

We shall denote by

$$\mathbf{C}(x//y)$$

the result of replacing all free occurrences of the variable x in **C** by the individual symbol y, which may be either a variable or a parameter. For example, if **C** is the wff $R(x) \vee [Q(x) \Rightarrow \exists x P(x,z)]$ then $\mathbf{C}(x//u)$ is the wff $R(u) \vee [Q(u) \Rightarrow \exists x P(x,z)]$.

There is a problem with this notation. We would like any wff of the form

$$\forall x\mathbf{C} \Rightarrow \mathbf{C}(x//y)$$

to be valid (i.e. true in any interpretation), because it says that if **C** is true for all x, then it is in particular true when x is y. But consider the case where **C** is $\exists y\, x < y$. In this case we would obtain

$$\forall x \exists y\, x < y \Rightarrow \exists y\, y < y$$

which is false for the natural numbers since $\forall x \exists y\, x < y$ is true (take $y = x + 1$) but $\exists y\, y < y$ is false. The problem is that the substitution of y for x in $\exists y\, x < y$ creates a bound occurrence of y at a position where there is a free occurrence of x; this problem is called **confusion of bound variables.**

We say that the individual symbol y is **freely substitutable for** the individual variable x in the wff **C** if no free occurrence of x in **C** occurs in a well-formed part of **C** which is of the form $\forall y\mathbf{B}$ or $\exists y\mathbf{B}$. Henceforth *we will use the notation* $\mathbf{C}(x//y)$ *only in the case that* y *is freely substitutable for* x *in* **C**. We use **free for** as an abbreviation for **freely substitutable for**, so y is free for x in **A** means that y is freely substitutable for x in **A**. By definition a parameter is always freely substitutable for a variable x in a wff **C**.

We shall see later that if y is free for x in **C**, then the wff

$$\forall x\mathbf{C} \Rightarrow \mathbf{C}(x//y)$$

is true in all interpretations, which is what we wanted.

By a **plain wff** we shall mean a wff which has no parameter symbols. Thus $WFF(\mathcal{P}, \emptyset)$ is the set of all plain wffs formed from $\mathcal{P}$.

A plain wff with no free variables is called a **sentence.** The set of all sentences in the vocabulary $\mathcal{P}$ is $SENT(\mathcal{P}, \emptyset)$.

A sentence has a meaning (truth value) once we specify (1) the meanings of all the propositional symbols and predicate symbols which appear in it, and (2) the range of values which the bound variables assume. For example, the sentence $\exists x \forall y\, x \leq y$ is true if $\leq$ has its usual meaning and the variables x and y range over the natural numbers (since $\forall y\ 0 \leq y$) but is false if the variables x and y range over the integers. By contrast the truth value of a wff which has one or more free variables depends on the values of the free variables. For example, the wff $x \doteq y$ is true if $x = 2$ and $y = 2$ but is false if $x = 2$ and $y = 3$.

A wff $\mathbf{A} \in WFF(\mathcal{P}, \mathcal{K})$ with parameters from $\mathcal{K}$ but no free variables is called a **sentence with parameters from** $\mathcal{K}$. The set of all sentences with parameters from $\mathcal{K}$ is denoted by $SENT(\mathcal{P}, \mathcal{K})$.

A sentence with parameters from $\mathcal{K}$ has a meaning once we specify (1) and (2) above, and (3) the meanings of all parameter symbols which appear in it. For example, the sentence $\forall y\ 0 \leq y$ is true if $\leq$ and 0 have their usual meaning, and the variable y ranges over the natural numbers.

2.4 Semantics of Predicate Logic

In this section we shall introduce models of pure predicate logic, and then define what is meant by the truth value of a sentence in a model.

Given a natural number n and a set X, an **n-ary relation** on X is a subset of the set X^n of all length n sequences $(x_1, x_2, \ldots, x_n)$ of elements from X. The set of all n-ary relations on X will be written $REL_n(X)$.

The set X^1 is the same as X, and a 1-ary relation, or **unary relation**, on X is just a subset of X. Similarly, X^2 is the same as $X \times X$, and 2-ary relations are also called **binary relations.**

The 0-ary relations on X correspond in a natural way to truth values. The only sequence of length 0 is the empty sequence (). The set X^0 has only one element, the empty sequence (); in symbols, $X^0 = \{()\}$. There are two 0-ary relations on X, the empty set $\emptyset$ which corresponds to the truth value **F**, and the set X^0 which corresponds to the truth value **T**.

A **model for pure predicate logic of type** $\mathcal{P}$ is a system $\mathcal{M}$ consisting of a non-empty set M called the **universe of the model** $\mathcal{M}$, and a function which assigns an n-ary relation $q^{\mathcal{M}}$ to each n-ary predicate symbol q of $\mathcal{P}$.

We emphasize that only the universe set M of a model $\mathcal{M}$ is required to be nonempty. A unary relation $p^{\mathcal{M}}$ may be any subset of M at all, empty or nonempty. After we define the notion of a truth value of a sentence in a model, we will be able to use sentences of predicate logic to express properties of relations. As a simple example, the sentence $\exists x\, p(x)$ will be true in a model $\mathcal{M}$ if and only if $p^{\mathcal{M}}$ is a nonempty

subset of M, and the sentence $\forall x\, p(x)$ will be true in a model $\mathcal{M}$ if and only if $p^{\mathcal{M}} = M$. As an even simpler example, a propositional symbol q will be true in $\mathcal{M}$ if and only if $q^{\mathcal{M}} = M^0$, i.e. $q^{\mathcal{M}}$ contains the empty sequence.

To illustrate the concept of a model, suppose the vocabulary $\mathcal{P}$ has only a single unary predicate symbol p. Then a model $\mathcal{M}$ of type $\mathcal{P}$ consists of a nonempty set M and a subset $p^{\mathcal{M}}$ (which may or may not be empty) of M. Given a nonempty finite set M with n elements, there are 2^n different models of type $\mathcal{P}$ with universe M, one for each subset $p^{\mathcal{M}}$ of M. Given an infinite set M, there are infinitely many different models of type $\mathcal{P}$ with universe M.

As a second example, suppose the vocabulary $\mathcal{P}$ has two unary predicate symbols p and q. In this case a model $\mathcal{M}$ of type $\mathcal{P}$ consists of a nonempty universe set M and two subsets $p^{\mathcal{M}}$ and $q^{\mathcal{M}}$ of M. Given a nonempty finite set M of size n, there will be $(2^n)^2$ different models of type $\mathcal{P}$ with universe M.

Finally, suppose the vocabulary $\mathcal{P}$ has one binary predicate symbol p. In this case a model $\mathcal{M}$ of type $\mathcal{P}$ consists of a nonempty universe set M and a subset $p^{\mathcal{M}}$ of the set $M \times M$. Given a nonempty finite set M of size n, there will be 2^{n^2} different models of type $\mathcal{P}$ with universe M.

Recall that a **plain wff** is a wff in which no parameters occur. A wff **with parameters from** $\mathcal{K}$ is a wff all of whose parameters (if any) are in the set $\mathcal{K}$ – in other words, a wff which has no parameters outside of $\mathcal{K}$. Thus a plain wff is a wff with parameters from $\mathcal{K}$ for every set $\mathcal{K}$.

Our next goal will be to assign an appropriate truth value to each plain sentence in every model for a vocabulary $\mathcal{P}$. The easiest way to do this is to do even more: given a model $\mathcal{M}$, we shall assign a truth value to each sentence with parameters from M. Since every plain sentence is also a sentence with parameters from M, this will accomplish our goal.

Given a model $\mathcal{M}$, we shall work with the predicate logic whose set of parameters $\mathcal{K}$ is the universe set M of $\mathcal{M}$. $SENT(\mathcal{P}, M)$ is the set of all sentences with parameters from M.

If $\mathbf{C}$ is a plain wff of pure predicate logic and $x_1, \ldots, x_n$ are the free variables of $\mathbf{C}$, we may form a sentence with parameters from M by choosing $a_1, \ldots, a_n \in M$ and replacing all free occurrences of x_k in $\mathbf{C}$ by a_k for $k = 1, \ldots, n$. The resulting sentence, called an **instance** of $\mathbf{C}$ in M, is denoted by

$$\mathbf{C}(x_1, \ldots, x_n // a_1, \ldots, a_n).$$

As a particular case, if $\mathbf{C}$ is a plain sentence then no parameters are needed, and $\mathbf{C}$ already an instance of itself.

Now we define $\mathcal{M} \models \mathbf{A}$ where $\mathbf{A} \in SENT(\mathcal{P}, M)$. Figure 2.1 gives the rules which determine the truth value $\mathbf{A}_{\mathcal{M}}$ of a sentence $\mathbf{A}$ with parameters from M. As in propositional logic we sometimes write $\mathcal{M} \models \mathbf{A}$ instead of $\mathbf{A}_{\mathcal{M}} = \mathbf{T}$, and $\mathcal{M} \not\models \mathbf{A}$ instead of $\mathbf{A}_{\mathcal{M}} = \mathbf{F}$.

Truth Value Rules

(M:$\mathcal{P}_0$) If $p \in \mathcal{P}_0$, $p_{\mathcal{M}} = \mathbf{T}$ iff $() \in p^{\mathcal{M}}$;

(M:$\mathcal{P}_n$) $p(a_1, a_2, \ldots, a_n)_{\mathcal{M}} = \mathbf{T}$ iff $(a_1, a_2, \ldots, a_n) \in p^{\mathcal{M}}$;

(M:$\neg$) $[\neg \mathbf{A}]_{\mathcal{M}} = \mathbf{T}$ iff $\mathbf{A}_{\mathcal{M}} = \mathbf{F}$;

(M:$\wedge$) $[\mathbf{A} \wedge \mathbf{B}]_{\mathcal{M}} = \mathbf{T}$ iff $\mathbf{A}_{\mathcal{M}} = \mathbf{T}$ and $\mathbf{B}_{\mathcal{M}} = \mathbf{T}$;

(M:$\vee$) $[\mathbf{A} \vee \mathbf{B}]_{\mathcal{M}} = \mathbf{T}$ iff $\mathbf{A}_{\mathcal{M}} = \mathbf{T}$ or $\mathbf{B}_{\mathcal{M}} = \mathbf{T}$;

(M:$\Rightarrow$) $[\mathbf{A} \Rightarrow \mathbf{B}]_{\mathcal{M}} = \mathbf{T}$ iff $\mathbf{A}_{\mathcal{M}} = \mathbf{F}$ or $\mathbf{B}_{\mathcal{M}} = \mathbf{T}$;

(M:$\Leftrightarrow$) $[\mathbf{A} \Leftrightarrow \mathbf{B}]_{\mathcal{M}} = \mathbf{T}$ iff $\mathbf{A}_{\mathcal{M}} = \mathbf{B}_{\mathcal{M}}$;

(M:$\forall$) $[\forall x \mathbf{A}]_{\mathcal{M}} = \mathbf{T}$ iff $\mathbf{A}(x//a)_{\mathcal{M}} = \mathbf{T}$ for every $a \in M$;

(M:[$\exists$) $\exists x \mathbf{A}]_{\mathcal{M}} = \mathbf{T}$ iff $\mathbf{A}(x//b)_{\mathcal{M}} = \mathbf{T}$ for some $b \in M$.

Figure 2.1: Truth Value Rules for Predicate Logic.

The following theorem is important for the semantics for pure predicate logic because it shows that the rules unambiguously determine a truth value for each sentence in a model. It is the analog of Theorem 1.5.1 for propositional logic.

Theorem 2.4.1 *For any model $\mathcal{M}$ (of type $\mathcal{P}$) with universe M there is a unique function which assigns a truth value $\mathbf{A}_{\mathcal{M}}$ to each sentence $\mathbf{A} \in SENT(\mathcal{P}, M)$ and satisfies the rules of Figure 2.1.*

We shall skip the proof of Theorem 2.4.1, which is again by induction on the length of wffs using the Unique Readability Theorem.

In the next few examples we illustrate our definition of the truth value of a sentence in a model with some detailed computations. In each example, we go step by step through a parsing sequence for the sentence. Because of the quantifier rules, we shall compute the truth value of every instance of the wff at each step of the parsing sequence.

Example 2.4.2 We compute the truth value of the sentence

$$\forall x P(x) \Rightarrow q$$

in a model $\mathcal{M}$ whose universe is a two element set $M = \{0, 1\}$, with $q^{\mathcal{M}} = \emptyset$ and $P^{\mathcal{M}} \in REL_1(M)$ given by

$$P^{\mathcal{M}} = \{0\}.$$

We first parse the sentence.

(1) $P(x)$ is a wff by (W:$\mathcal{P}_1$).

(2) $\forall x P(x)$ is a wff by (1) and (W:$\forall$).

(3) q is a wff by (W:$\mathcal{P}_0$).

(4) $\forall x P(x) \Rightarrow q$ is a wff by (2), (3) and (W:$\Rightarrow$).

Now we apply the definition.

(1) $P(0)_{\mathcal{M}} = \mathbf{T}$ and $P(1)_{\mathcal{M}} = \mathbf{F}$ by (M:$\mathcal{P}_1$).

(2) $[\forall x P(x)]_{\mathcal{M}} = \mathbf{F}$ by (1) and (M:$\forall$).

(3) $q_{\mathcal{M}} = \mathbf{F}$ by (M:$\mathcal{P}_0$).

(4) $[\forall x P(x) \Rightarrow q]_{\mathcal{M}} = \mathbf{T}$ by (2), (3), and (M:$\Rightarrow$).

Example 2.4.3 We compute the truth value of sentence

$$\forall x[P(x) \Rightarrow q]$$

in the model $\mathcal{M}$ of the previous example.

We first parse the sentence.

(1) $P(x)$ is a wff by (W:$\mathcal{P}_1$).

(2) q is a wff by (W:$\mathcal{P}_0$).

(3) $P(x) \Rightarrow q$ is a wff by (1), (2), and (W:$\Rightarrow$).

(4) $\forall x[P(x) \Rightarrow q]$ is a wff by (3) and (W:$\forall$).

Now we apply the definition.

(1) $\mathcal{M} \models P(0)$ and $\mathcal{M} \not\models P(1)$ by (M:$\mathcal{P}_0$).

(2) $\mathcal{M} \not\models q$ by (M:$\mathcal{P}_0$) because $q_{\mathcal{M}} = \mathbf{F}$.

(3) $\mathcal{M} \not\models P(0) \Rightarrow q$ and $\mathcal{M} \models P(1) \Rightarrow q$ by (1), (2), and (M:$\Rightarrow$).

(4) $\mathcal{M} \not\models \forall x[P(x) \Rightarrow q]$ by (3) and (M:$\forall$).

Example 2.4.4 We compute the truth value of

$$\forall y \exists x\ x \leq y \Rightarrow \exists x \forall y\ x \leq y$$

for a model $\mathcal{M}$ whose universe set is the set $M = \mathbf{N}$ of natural numbers, and $\leq^{\mathcal{M}}$ is the usual order relation on $\mathbf{N}$:

$$\leq^{\mathcal{M}} \quad = \quad \{(a, b) \in \mathbf{N}^2 : a \leq b\}.$$

We first parse the wff.

(1) $x \leq y$ is a wff by (W:$\mathcal{P}_2$).

(2) $\exists x\ x \leq y$ is a wff by (1) and (W:$\exists$).

(3) $\forall y \exists x\ x \leq y$ is a wff by (2) and (W:$\forall$).

(4) $\forall y\ x \leq y$ is a wff by (1) and (W:$\forall$).

(5) $\exists x \forall y\ x \leq y$ is a wff by (4) and (W:$\exists$).

(6) $[\forall y \exists x\ x \leq y \Rightarrow \exists x \forall y\ x \leq y]$ is a wff by (3), (5), and (W:$\Rightarrow$).

Now we apply the definition of $\mathcal{M} \models \mathbf{A}$ to this parsing sequence.

(1) $\mathcal{M} \models\ c \leq d$ iff $c \leq^{\mathcal{M}} d$.

(2) $\mathcal{M} \models \exists x\ x \leq d$ for every d since $\mathcal{M} \models\ 0 \leq d$ for every d.

(3) $\mathcal{M} \models \forall y \exists x\ x \leq y$ by (2).

(4) $\mathcal{M} \models \forall y\ c \leq y$ iff $c = 0$.

(5) $\mathcal{M} \models \exists x \forall y\ x \leq y$ by (4).

(6) $\mathcal{M} \models \forall y \exists x\ x \leq y \Rightarrow \exists x \forall y\ x \leq y$ by (3) and (5).

Example 2.4.5 We compute the truth value of the wff of the preceding example for a different model. Take $M = \mathbf{Z}$, the set of integers, with $\leq^{\mathcal{M}}$ the usual order relation on $\mathbf{Z}$:

$$\leq^{\mathcal{M}} \quad = \quad \{(a, b) \in \mathbf{Z}^2 : a \leq b\}.$$

(1) $\mathcal{M} \models\ c \leq d$ iff $c \leq^{\mathcal{M}} d$.

(2) $\mathcal{M} \models \exists x\ x \leq d$ for every d, since $\mathcal{M} \models\ c \leq d$ if $c = d$.

(3) $\mathcal{M} \models \forall y \exists x\ x \leq y$ by (2).

(4) $\mathcal{M} \not\models \forall y\ c \leq y$ for every c, since $\mathcal{M} \not\models\ c \leq d$ if $d = c - 1$.

(5) $\mathcal{M} \not\models \exists x \forall y\ x \leq y$ by (4).

(6) $\mathcal{M} \not\models \forall y \exists x\ x \leq y \Rightarrow \exists x \forall y\ x \leq y$ by (3) and (5).

We can now define the notion of semantic consequence as before. The sentence $\mathbf{A}$ is said to be a **semantic consequence** of a set $\mathbf{H}$ of sentences, and we write $\mathbf{H} \models \mathbf{A}$, if every model of $\mathbf{H}$ is also a model of $\mathbf{A}$. If $\mathbf{H} \models \mathbf{A}$ and $\mathbf{H}$ is the empty set, we say that $\mathbf{A}$ is a **valid sentence**. In other words, a valid sentence is one which holds in every model; it is the analog for predicate logic of a tautology in propositional logic. In Section 2.6 we will again encounter the tableau method for establishing semantic consequence and validity of sentences. To motivate the new tableau rules we give the following extension of Proposition 1.5.2 from page 16.

Proposition 2.4.6 *Suppose $\mathcal{M}$ is a model with universe M.*

$\boxed{\forall}$ *If $\mathcal{M} \models \forall x\mathbf{A}$ and $a \in M$ then $\mathcal{M} \models \mathbf{A}(x//a)$.*

$\boxed{\neg\forall}$ *If $\mathcal{M} \models \neg\forall x\mathbf{A}$ then $\mathcal{M} \models \neg\mathbf{A}(x//b)$ for some $b \in M$.*

$\boxed{\exists}$ *If $\mathcal{M} \models \exists x\mathbf{A}$ then $\mathcal{M} \models \mathbf{A}(x//b)$ for some $b \in M$.*

$\boxed{\neg\exists}$ *If $\mathcal{M} \models \neg\exists x\mathbf{A}$ and $a \in M$ then $\mathcal{M} \models \neg\mathbf{A}(x//a)$.*

2.5 Graphs

The semantics for wffs with three or fewer variables can be represented graphically. Let $\mathbf{A}$ be a wff with at most the free variables x, y. The (x, y) **graph** of $\mathbf{A}$ in $\mathcal{M}$ is the set of all pairs of elements of M for which $\mathbf{A}$ is true, that is,

$$GRAPH_{x,y}(\mathbf{A}, \mathcal{M}) = \{(a, b) \in M^2 : \mathcal{M} \models \mathbf{A}(x, y//a, b)\}.$$

If $\mathbf{A}$ is a sentence, the (x, y) graph of $\mathbf{A}$ in $\mathcal{M}$ is either the whole plane M^2 or the empty set. This is because $\mathbf{A}$ has no free variables, so $\mathbf{A}(x, y//a, b)$ is just the original sentence $\mathbf{A}$ for every pair a, b. If $\mathcal{M} \models \mathbf{A}$ then the (x, y) graph of $\mathbf{A}$ is the whole plain M^2, and otherwise the graph is the empty set.

If the x axis is horizontal and $\mathbf{A}$ is a wff with only x free, then the (x, y) graph of $\mathbf{A}$ in $\mathcal{M}$ will be a union of vertical columns in the $M \times M$ plane. This is because the graph of $\mathbf{A}$ is the set

$$\{(a, b) \in M^2 : \mathcal{M} \models \mathbf{A}(x//a)\},$$

and two pairs (a, b) and (a, c) in the same column go with the same instance $\mathbf{A}(x//a)$ of $\mathbf{A}$. Similarly, if $\mathbf{A}$ has only y free, its (x, y) graph will be a union of horizontal rows in the $M \times M$ plane.

A wff with n free variables can be represented by an n-dimensional graph. The PREDCALC program gives a graphical representation of wffs all of whose variables, both free and bound, are among x, y, and z. A finite universe of the form $0, 1, \ldots, n-1$ of size n must first be chosen, where n is between 1 and 8. The (x, y, z) graph of a wff is a subset of a cube with n points on each side. The model in the program has three binary relations

$$x = y, \quad x < y, \quad x > y,$$

and nine ternary relations

$$x = y + z, \quad x = y - z, \quad x = y * z,$$

$$x < y + z, \quad x < y - z, \quad x < y * z,$$

$$x > y + z, \quad x > y - z, \quad x > y * z,$$

which can be entered using the button for atomic formulas. Here the addition, subtraction, and multiplication are performed **modulo** n. (To add or multiply two numbers modulo n, add or multiply them in the usual way and then take the remainder after division by n. To subtract two numbers modulo n, subtract in the usual way and then add n if the result is negative). There is also a provision for adding "random" unary, binary, or ternary relations to the vocabulary. By experimenting with the program, you can see what happens to the graphs when you combine wffs with connectives and quantifiers.

2.6 Tableaus

Recall that a sentence $\mathbf{A}$ of predicate logic is said to be valid if $\mathbf{A}$ is true in *every* model. In propositional logic it is possible to test whether a wff is valid in a finite number of steps by constructing a truth table. This cannot be done in predicate logic. In predicate logic there are infinitely many models to consider, even when the vocabulary

of predicate symbols is finite. Since we cannot physically make a table of all models, we need another method of showing that a sentence is valid. To this end, we shall generalize the notion of tableau proof from propositional logic to predicate logic. As before, a formal proof of a sentence **A** will be represented as a tableau confutation of the negation of **A**.

Tableaus in predicate logic are defined in the same way as tableaus in propositional logic except that there are four additional rules for extending them. The new rules are the $\boxed{\forall}$ and $\boxed{\exists}$ rules for wffs which begin with quantifiers and the $\boxed{\neg\forall}$ and $\boxed{\neg\exists}$ rules for the negations of wffs which begin with quantifiers. As in the case of propositional logic, our objective will be to prove the Soundness Theorem and the Completeness Theorem. The Soundness Theorem will show that every sentence which has a tableau proof is valid, and the Completeness Theorem will show that every valid sentence has a tableau proof. The tableau rules are chosen in such a way that if $\mathcal{M}$ is a model of the set of hypotheses of the tableau, then there is at least one branch of the tableau such that every wff on the branch is true for $\mathcal{M}$.[1]

A **labeled tree for pure predicate logic** is a system $(\mathbf{T}, \mathbf{H}, \Phi)$ where $\mathbf{T}$ is a tree, $\mathbf{H}$ is a set of wffs and Φ is a function which assigns to each nonroot node t of $\mathbf{T}$ a wff $\Phi(t)$ of pure predicate logic. The definition is exactly the same as for propositional logic, except that the wffs are now wffs of predicate logic. As in propositional logic, "the wff **A** is at the node t" means that "**A** is $\Phi(t)$." The wffs of $\mathbf{H}$ are said to be "at the root." We shall use the same terminology (ancestor, child, parent, etc.) as we did for propositional logic.

Definition 2.6.1 A **tableau chain** for pure predicate logic is a finite or infinite sequence of finite labeled trees which is formed using the nine tableau extension rules for propositional logic (see section 1.7.1) and the following additional **tableau extension rules:**

$\boxed{\forall}$ If t has an ancestor $\forall x\mathbf{A}$, extend by adding a child $\mathbf{A}(x//a)$ of t, where a is an individual symbol which is free for x in **A**.

[1] For a precise statement, see Lemma 2.7.2 on page 86 below.

$\boxed{\neg\forall}$ if t has an ancestor $\neg\forall x\mathbf{A}$, extend by adding a child $\neg\mathbf{A}(x//b)$ of t, where b is an individual symbol which does not occur in any ancestor of t.

$\boxed{\exists}$ If t has an ancestor $\exists x\mathbf{A}$, extend by adding a child $\mathbf{A}(x//b)$ of t, where b is an individual symbol which does not occur in any ancestor of t.

$\boxed{\neg\exists}$ If t has an ancestor $\neg\exists x\mathbf{A}$, extend by adding a child $\neg\mathbf{A}(x//a)$ of t where a is an individual symbol which is free for x in **A**.

The four new rules are summarized in Figure 2.2, which should be viewed as an extension of Figure 1.4 on page 28.

Definition 2.6.2 A **tableau for predicate logic** is a labeled tree which is either the last term of a finite tableau chain, or the union of an infinite tableau chain.

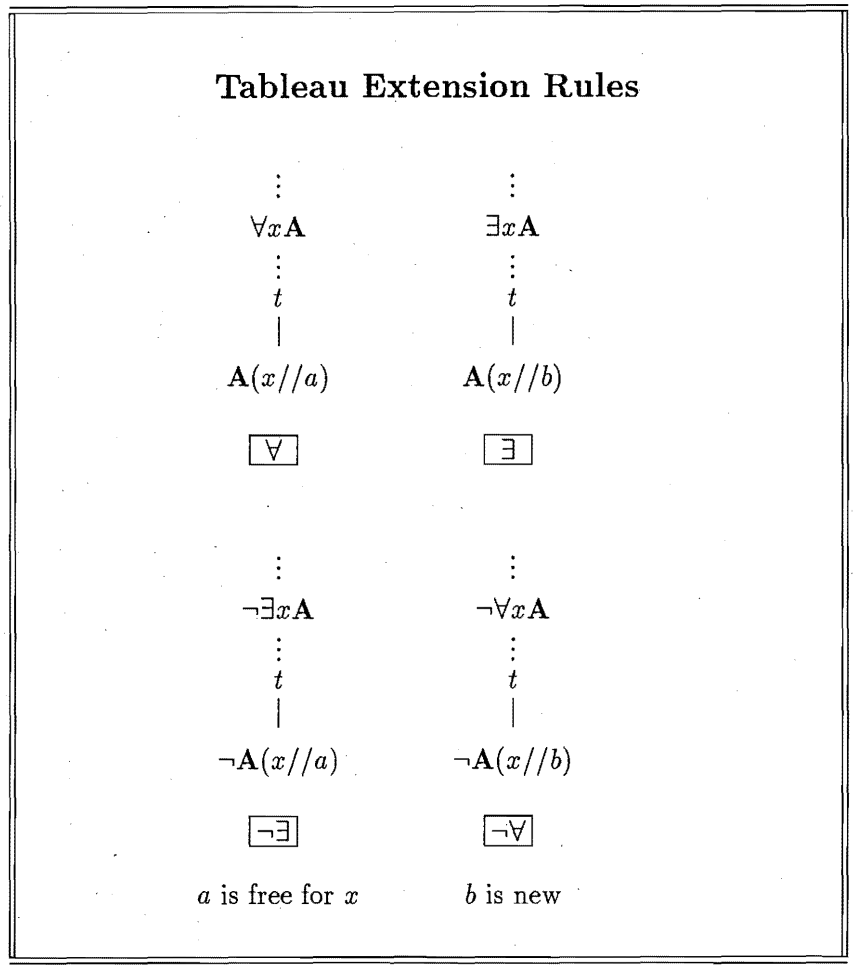


Figure 2.2: Tableau Extension Rules for Pure Predicate Logic.

Notice that the $\boxed{\forall}$ and $\boxed{\neg\exists}$ rules are similar to each other, and the $\boxed{\exists}$ and $\boxed{\neg\forall}$ rules are similar to each other. The $\boxed{\forall}$ and $\boxed{\neg\exists}$ rules allow any substitution at all as long as there is no confusion of free and bound variables. On the other hand, the $\boxed{\exists}$ and $\boxed{\neg\forall}$ rules are very restricted, and only allow us to substitute a completely new symbol b for x. In an informal mathematical proof, if we know that $\exists x\mathbf{A}$ is true we may introduce a new symbol b to name the element for which $\mathbf{A}(x//b)$ is true. It would be incorrect to use a symbol which has already been used for something else. This informal step corresponds to the $\exists$ rule for extending a tableau. A similar remark applies to the $\neg\forall$ rule.

A **tableau confutation** of a set $\mathbf{H}$ of wffs in predicate logic is a tableau $\mathbf{T}$ with root $\mathbf{H}$ such that each branch is **contradictory**, that is, each branch has a pair of wffs $\mathbf{A}$ and $\neg\mathbf{A}$. A **tableau proof** of a wff $\mathbf{A}$ is a tableau confutation of the set $\{\neg\mathbf{A}\}$, and a **tableau proof of A from the hypotheses H** is a tableau confutation of the set $\mathbf{H}\cup\{\neg\mathbf{A}\}$. If there is a tableau proof of $\mathbf{A}$ from $\mathbf{H}$, we say that $\mathbf{A}$ is **provable** from $\mathbf{H}$ and write $\mathbf{H}\vdash\mathbf{A}$.

The main purpose of tableaus is to give a method for showing that a sentence is valid, or that one sentence is a semantic consequence of a set of other sentences. For this reason, We shall usually work with tableaus whose hypothesis set $\mathbf{H}$ is a set of sentences, rather than merely a set of wffs.

We shall see later that if a set of sentences $\mathbf{H}$ has a tableau confutation, it has one such that every individual symbol which occurs freely on the tableau is a parameter rather than a variable. We shall always follow the practice of building tableaus with no free variables, because then we never have to worry about a variable being both free and bound in a wff. This is done by using individual parameters rather than individual variables in the quantifier extension rules.

It is usually much more difficult to find formal proofs in predicate logic than in propositional logic, because if one is careless, the tableau will keep growing forever. One useful rule of thumb is to try to use the $\boxed{\exists}$ and $\boxed{\neg\forall}$ rules, which introduce new individuals, as early as possible. Quite often, these new individuals will appear in substitutions in the $\boxed{\forall}$ or $\boxed{\neg\exists}$ rules later on. This rule of thumb is illustrated in the two simple examples in Figure 2.3.

(1)	$\neg\exists y\ P(y)$	$\neg$ to be proved
	|	
(2)	$\exists x\ P(x)$	hypothesis
	|	
(3)	$P(a)$	by (2)
	|	
(4)	$\neg P(a)$	by (1)

A tableau proof of $\exists y\ P(y)$ from $\exists x\ P(x)$.

(1)	$\neg\forall y\exists x\ P(x,y)$	$\neg$ to be proved
	|	
(2)	$\exists x\forall y\ P(x,y)$	hypothesis
	|	
(3)	$\forall y\ P(a,y)$	by (2)
	|	
(4)	$\neg\exists x\ P(x,b)$	by (1)
	|	
(5)	$P(a,b)$	by (3)
	|	
(6)	$\neg P(a,b)$	by (4)

A tableau proof of $\forall y\exists x\ P(x,y)$ from $\exists x\forall y\ P(x,y)$.

Figure 2.3: Two Tableau Proofs in Predicate Logic.

2.7 Soundness

In this section we will prove the

Soundness Theorem

If a sentence of pure predicate logic has a tableau proof, then it is valid.

The proof of the Soundness Theorem for predicate logic is similar to the proof of the Soundness Theorem for propositional logic, but with extra steps for the quantifiers. Recall that Lemma 1.8.1 for propositional logic asserted that if $\mathbf{T}$ is a finite tableau with a set $\mathbf{H}$ of wffs at the root and if $\mathcal{M}$ is a model for $\mathbf{H}$ then there is a branch $\mathbf{\Gamma}$ such that $\mathcal{M} \models \mathbf{\Gamma}$. Without some qualification this will not be true in predicate logic since the wffs in the tableau proof may have free variables or parameter symbols which are not elements of M. To make it correct we must replace the free variables or parameter symbols which occur in the tableau with suitable parameters from M.

To this end define a **valuation** in the set M to be a list of pairs

$$v = \{(x_1, a_1), (x_2, a_2), \ldots, (x_\ell, a_\ell)\}$$

where $x_1, x_2, \ldots x_\ell$ are distinct individual symbols (variables or parameters) and $a_1, a_2, \ldots a_\ell$ are elements of M. For any wff $\mathbf{A}$ we write $\mathbf{A}(v)$ in place of the more cumbersome

$$\mathbf{A}(x_1, x_2, \ldots x_\ell // a_1, a_2, \ldots, a_\ell)$$

If the list $x_1, x_2, \ldots x_\ell$ contains all the individual symbols occurring freely in the wff $\mathbf{A}$ then $\mathbf{A}(v)$ is a sentence with parameters from M. If $\mathcal{M}$ is a model with universe M and $\mathbf{\Gamma}$ is a set of wffs, then the notation

$$\mathcal{M} \models \mathbf{\Gamma}(v)$$

means that $\mathcal{M} \models \mathbf{A}(v)$ for each wff $\mathbf{A}$ in $\mathbf{\Gamma}$. The notation is used only when the list $x_1, x_2, \ldots x_\ell$ contains all the individual symbols which occur freely in some wff of $\mathbf{\Gamma}$.

Recall that a sentence of pure predicate logic is a wff with no free individual symbols, that is, no free variables and no parameters. To keep things simple, in this section we shall consider only finite tableaus $\mathbf{T}$ whose hypothesis set $\mathbf{H}$ is a finite or countable set of sentences. If $\mathbf{T}$ is such a tableau, then each branch $\mathbf{\Gamma}$ of $\mathbf{T}$ will have only finitely many wffs in addition to the hypotheses. Since no individual symbols occur freely in $\mathbf{H}$, only finitely many individual symbols occur freely in $\mathbf{\Gamma}$. In this case, $\mathcal{M} \models \mathbf{\Gamma}(v)$ is meaningful and says that $\mathcal{M} \models \mathbf{A}(v)$ for each wff $\mathbf{A}$ which occurs along $\mathbf{\Gamma}$; i.e. the same notation is used for the branch and the set of wffs which occur along the branch.

In the exercises we shall see that the results in this section can be extended to all tableaus by using infinite valuations.

Definition 2.7.1 A wff $\mathbf{A}$ is called **satisfiable** in a model $\mathcal{M}$ iff there is a valuation v in the universe of $\mathcal{M}$ such that $\mathcal{M} \models \mathbf{A}(v)$. A set $\mathbf{\Gamma}$ of wffs in which only finitely many individual symbols occur freely is called **simultaneously satisfiable** in a model $\mathcal{M}$ iff there is a valuation v in the universe of $\mathcal{M}$ such that $\mathcal{M} \models \mathbf{\Gamma}(v)$.

Lemma 2.7.2 *Let $\mathbf{T}$ be a finite tableau in predicate logic whose hypotheses set $\mathbf{H}$ is a finite or countable set of sentences, and let $\mathcal{M}$ be a model for $\mathbf{H}$, that is,*

$$\mathcal{M} \models \mathbf{H}.$$

Then there is a branch $\mathbf{\Gamma}$ of $\mathbf{T}$ which is simultaneously satisfiable in $\mathcal{M}$.

Proof: The proof of this lemma is similar to the proof of Lemma 1.8.1 which is the corresponding lemma for propositional logic. The idea is to carefully choose individual symbols from the model at each step where the $\boxed{\exists}$ rule or the $\boxed{\forall}$ rule is used in extending the tableau $\mathbf{T}$.

By definition there is a finite tableau chain $\mathbf{T}_0, \mathbf{T}_1, \ldots, \mathbf{T}_n$ with $\mathbf{T} = \mathbf{T}_n$. We will construct inductively a branch $\mathbf{\Gamma}_k$ of $\mathbf{T}_k$ and a valuation

$$v_k = \{(x_1, a_1), (x_2, a_2), \ldots, (x_{\ell_k}, a_{\ell_k})\}$$

such that the wffs

$$\mathbf{A}_1, \mathbf{A}_2, \ldots, \mathbf{A}_m$$

which occur along this branch satisfy $\mathcal{M} \models \mathbf{A}_j(v_k)$ for $j = 1, 2, \ldots k$. The first coordinates $x_1, x_2, \ldots, x_{\ell_k}$ of the pairs in the list v will be precisely the individual symbols which occur free along $\mathbf{\Gamma}_k$. The branch $\mathbf{\Gamma}_{k+1}$ will extend the branch $\mathbf{\Gamma}_k$ and the valuation v_{k+1} will extend v_k.

When $k = 0$ the wffs $\mathbf{A}_j$ are simply those of $\mathbf{H}$ so we take v_0 to be empty and the result is the hypothesis $\mathcal{M} \models \mathbf{H}$. If $\mathbf{T}_{k+1}$ is obtained from $\mathbf{T}_k$ by extending at some node other than the terminal node of $\mathbf{\Gamma}_k$ we simply take $\mathbf{\Gamma}_k = \mathbf{\Gamma}_{k+1}$ and $v_k = v_{k+1}$ and there is nothing to prove. Hence assume $\mathbf{T}_{k+1}$ is obtained from $\mathbf{T}_k$ by extending at the terminal node of $\mathbf{\Gamma}_k$ by applying one of the thirteen tableau extension rules to some wff $\mathbf{A}_j$ in the list. We use a case analysis and Proposition 1.5.2 (page 16).

In case the $\mathbf{T}_k$ is extended to $\mathbf{T}_{k+1}$ via one of the nine propositional tableau extension rules we take $v_{k+1} = v_k$ and argue as in Proposition 1.8.1. In the remaining cases we argue as follows.

(10) Suppose $\mathbf{A}_j$ is $\forall x \mathbf{A}$ and the tableau is extended by adjoining $\mathbf{A}(x//y)$. Take $v_{k+1} = v_k$ if the individual symbol y appears in the list $x_1, \ldots x_\ell$ of first coordinates in v_k; if not, extend v_k to v_{k+1} by adjoining the pair (y, a) where a is any element of M. By the induction hypothesis $\mathcal{M} \models \forall x \mathbf{A}(v_k)$ so $\mathcal{M} \models \mathbf{A}(v_{k+1})$.

(11) Suppose $\mathbf{A}_j$ is $\neg\forall x \mathbf{A}$ and the tableau is extended by adjoining $\neg\mathbf{A}(x//y)$. In this case the individual symbol y does not occur in the list of first coordinates in v_k and by the induction hypothesis $\mathcal{M} \models \neg\forall x \mathbf{A}(v_k)$. Choose $b \in M$ so that $\mathcal{M} \models \neg\mathbf{A}(v_{k+1})$ where v_{k+1} is defined by adjoining (y, b) to v_k.

(12) Suppose $\mathbf{A}_j$ is $\exists x \mathbf{A}$ and the tableau is extended by adjoining $\mathbf{A}(x//z)$. Proceed as in (11).

(13) Suppose $\mathbf{A}_j$ is $\neg\exists x \mathbf{A}$ and the tableau is extended by adjoining $\neg\mathbf{A}(x//y)$. Proceed as in (10).

End of Proof.

As in propositional logic, we have the following lemma which is proved in essentially the same way:

Lemma 2.7.3 *If a finite or countable set* **H** *of sentences has a tableau confutation, then* **H** *has no model.*

Proof: Suppose **H** is a hypothesis set and **T** is a tableau confutation of **H**; if **H** has a model $\mathcal{M}$, then by the previous lemma, there is a branch Γ in **T** and a valuation v in $\mathcal{M}$ such that $\mathcal{M} \models \Gamma(v)$. But this is impossible since every branch of **T** is contradictory. **End of Proof.**

This lemma gives us the Extended Soundness Theorem just as with propositional logic. Since the proof carries over without change, we omit the details. The Soundness Theorem in the above box is the special case where the hypothesis set **H** is empty.

Theorem 2.7.4 (Extended Soundness Theorem) *Suppose that* $\mathbf{H} \cup \{\mathbf{A}\}$ *is a finite or countable set of sentences. If* $\mathbf{H} \vdash \mathbf{A}$ *then* $\mathbf{H} \models \mathbf{A}$*; in other words, if there is a tableau proof of* **A** *from* **H**, *then* **A** *is a semantic consequence of* **H**.

As in propositional logic, a tableau confutation can be used to show that a sentence is valid. This is the special case of the Extended Soundness Theorem in which the hypothesis set **H** is *empty*. Thus, if $\vdash \mathbf{A}$, then every model (of the empty set of hypotheses) is a model of **A**; hence **A** is valid.

2.8 Finished Sets

By an **atomic wff** we mean either a propositional symbol alone or a wff of form $p(x_1, x_2, \ldots, x_n)$ where p is an n-ary predicate symbol and $x_1, x_2, \ldots, x_n$ are individual symbols. By a **basic wff** in pure predicate logic we mean a wff which is either an atomic wff or the negation of an atomic wff. We call a set Δ of wffs **contradictory** if it contains some wff **A**, and its negation $\neg\mathbf{A}$. A set Δ of sentences with parameters from M is a **finished set** on M if Δ is not contradictory, and for each $\mathbf{C} \in \Delta$, either **C** is a basic wff, **C** satisfies one of the conditions $[\neg\neg]$ to $[\neg \Leftrightarrow]$ from Section 1.9 on page 33, or else one of the following is true:

$[\forall]$ **C** has form $\forall x\mathbf{A}$ where $\mathbf{A}(x//a) \in \Delta$ for every $a \in M$;

$[\neg\forall]$ **C** has form $\neg\forall x\mathbf{A}$ where $\neg\mathbf{A}(x//b) \in \Delta$ for some $b \in M$;

$[\exists]$ **C** has form $\exists x\mathbf{A}$ where $\mathbf{A}(x//b) \in \Delta$ for some $b \in M$;

$[\neg\exists]$ **C** has form $\neg\exists x\mathbf{A}$ where $\neg\mathbf{A}(x//a) \in \Delta$ for every $a \in M$.

The definition of a finished set is parallel to the definition of a tableau. It should be noted, however, that the $\boxed{\forall}$ and $\boxed{\neg\exists}$ tableau extension rules differ markedly from the $[\forall]$ and$[\neg\exists]$ clauses in the definition of a finished set. The latter two rules say that *every possible* substitution instance must lie in the finished set, whereas the former two rules say that the tableau is extended by one substitution.

Lemma 2.8.1 (Finished Set Lemma) *Suppose M is a non-empty set and that Δ is a set of sentences with parameters from M. Assume that Δ is finished set on M. Define a model $\mathcal{M}$ for pure predicate logic as follows:*

- *The universe set of the model $\mathcal{M}$ is the set M.*
- *For each propositional symbol $p \in \mathcal{P}_0$, $p_{\mathcal{M}} = \mathbf{T}$ if and only if $p \in \Delta$.*
- *For each n-ary predicate symbol $p \in \mathcal{P}_n$*

$$p^{\mathcal{M}} = \{(b_1, b_2, \ldots, b_n) \in M^n : p(b_1, b_2, \ldots, b_n) \in \Delta\}.$$

Then $\mathcal{M} \models \Delta$.

Proof: We shall prove that

$$(*) \qquad \mathcal{M} \models \mathbf{C} \text{ if } \mathbf{C} \in \Delta$$

by induction of the length of **C**. The pattern of proof is as follows. First we prove $(*)$ in case **C** is a basic wff. Then we choose $\mathbf{C} \in \Delta$, assume that $(*)$ is true for all wffs **A** which are shorter than **C**, and prove that $\mathcal{M} \models \mathbf{C}$. (This shows that if $(*)$ is true for ali wffs **A** shorter than **C**, then $(*)$ is also true when **A** is **C**.)

First consider the case where **C** is basic. If **C** is $p(b_1, b_2, \ldots, b_n)$ and $p(b_1, b_2, \ldots, b_n) \in \Delta$, then $\mathcal{M} \models \mathbf{C}$ by the definition of $\mathcal{M}$. If **C** is

$\neg p(b_1, b_2, \ldots, b_n)$ and $\mathbf{C} \in \mathbf{\Delta}$, then $p(b_1, b_2, \ldots, b_n) \notin \mathbf{\Delta}$ for otherwise the set $\mathbf{\Delta}$ would be contradictory and hence not finished. Hence in this case as well $\mathcal{M} \models \mathbf{C}$ by the definition of $\mathcal{M}$.

Now choose $\mathbf{C} \in \mathbf{\Delta}$ and assume inductively that $(*)$ is true for all wffs shorter than $\mathbf{C}$. We have just handled the case where $\mathbf{C}$ is basic so we may assume that $\mathbf{C}$ is not basic. Hence $\mathbf{C}$ has one of the forms $[\neg\neg]$, $[\wedge]$,..., $[\neg\exists]$ as in the definition of finished set given above. There are thirteen cases, one for each part of the definition. They are all similar so we will only prove five of them and leave the rest to the reader.

$[\neg\neg]$ In this case $\mathbf{C}$ has the form $\neg\neg\mathbf{A}$. As we have assumed that $\mathbf{C} \in \mathbf{\Delta}$ the definition of finished set tells us that $\mathbf{A} \in \mathbf{\Delta}$. By the induction hypothesis, $\mathcal{M} \models \mathbf{A}$. Hence $\mathcal{M} \models \mathbf{C}$.

$[\vee]$ In this case $\mathbf{C}$ has the form $\mathbf{A} \vee \mathbf{B}$. As we have assumed that $\mathbf{C} \in \mathbf{\Delta}$ the definition of finished set tells us that either $\mathbf{A} \in \mathbf{\Delta}$ or $\mathbf{B} \in \mathbf{\Delta}$. By the induction hypothesis, either $\mathcal{M} \models \mathbf{A}$ or $\mathcal{M} \models \mathbf{B}$. Hence $\mathcal{M} \models \mathbf{C}$.

$[\neg\vee]$ In this case $\mathbf{C}$ has the form $\neg[\mathbf{A} \vee \mathbf{B}]$. As we have assumed that $\mathbf{C} \in \mathbf{\Delta}$ the definition of finished set tells us that $\neg\mathbf{A} \in \mathbf{\Delta}$ and $\neg\mathbf{B} \in \mathbf{\Delta}$. By the induction hypothesis, $\mathcal{M} \models \neg\mathbf{A}$ and $\mathcal{M} \models \neg\mathbf{B}$. Hence $\mathcal{M} \models \mathbf{C}$.

$[\forall]$ In this case $\mathbf{C}$ has the form $\forall x\mathbf{A}$. As we have assumed that $\mathbf{C} \in \mathbf{\Delta}$ the definition of finished set tells us that

$$\mathbf{A}(x//a) \in \mathbf{\Delta} \text{ for every } a \in M.$$

The induction hypothesis tells us that $\mathcal{M} \models \mathbf{A}(x//a)$ for every $a \in M$. Hence $\mathcal{M} \models \mathbf{C}$.

$[\neg\forall]$ In this case $\mathbf{C}$ has the form $\neg\forall x\mathbf{A}$. As we have assumed that $\mathbf{C} \in \mathbf{\Delta}$, the definition of finished set tells us that

$$\neg\mathbf{A}(x//b) \in \mathbf{\Delta} \text{ for some } b \in M.$$

The induction hypothesis tells us that $\mathcal{M} \models \mathbf{A}(x//b)$. Hence $\mathcal{M} \models \mathbf{C}$.

End of Proof.

2.9 Completeness

In this section we will prove the

Completeness Theorem

If a sentence of pure predicate logic is valid, then it has a tableau proof.

The Completeness Theorem for pure predicate logic uses many of the ideas introduced in connection with the Completeness Theorem for propositional logic. One important difference is that infinite tableaus are needed even when the set of hypotheses is finite. As with propositional logic, our main task is to prove the following Main Lemma.

Lemma 2.9.1 (Main Lemma) *Suppose* $\mathbf{H}$ *is a finite or countable set of sentences in pure predicate logic. Either* $\mathbf{H}$ *has a tableau confutation in which no free variables occur, or* $\mathbf{H}$ *has a model.*

As before, the Extended Soundness Theorem shows that $\mathbf{H}$ cannot have both a tableau confutation (with or without free variables) and a model. To prove the Main Lemma we shall construct a tableau $\mathbf{T}$ in which every branch is either finished or finite and contradictory. The tableau $\mathbf{T}$ will also have the property that no free variables occur on $\mathbf{T}$.

The formulation of the Completeness Theorem in the box at the beginning of this section is a special case of the following. (Take the hypothesis set $\mathbf{H}$ to be empty.)

Theorem 2.9.2 (Extended Completeness Theorem) *Suppose* $\mathbf{H}$ *is a finite or countable set of sentences and* $\mathbf{A}$ *is a sentence in pure predicate logic. If every model of* $\mathbf{H}$ *is a model of* $\mathbf{A}$*, then there is a tableau proof of* $\mathbf{A}$ *from* $\mathbf{H}$ *in which no free variables occur. Thus if* $\mathbf{H} \models \mathbf{A}$*, then* $\mathbf{H} \vdash \mathbf{A}$.

The proof of the Completeness Theorem from the Main Lemma carries over from propositional logic without change, and so we omit it here. Following the pattern which we used for propositional logic, we shall now state and prove a Tableau Extension Lemma for predicate logic, and then prove the Main Lemma.

We fix a countable set M of new individual parameters which occur nowhere in $\mathbf{H}$. A branch of a tableau is said to be **finished** on M if the set of wffs on the branch is finished on M. Define a tableau $\mathbf{T}$ to be **finished** on M if every branch of $\mathbf{T}$ is either finished on M or else both finite and contradictory. In a finished tableau, the finished branches, if any, may be either finite or infinite. (A branch will have to be infinite if M is infinite and a wff of form $\forall x\mathbf{A}$ or $\neg\exists x\mathbf{A}$ appears on the branch.) If all the branches of a tableau are finite and contradictory, then by the König Tree Theorem from Chapter 1, the tableau will have finitely many nodes and hence will be a confutation. Tableau proofs and tableau confutations are always required to be finite, but finished tableaus which are not confutations are allowed to be infinite.

Lemma 2.9.3 (Tableau Extension Lemma) *Let M be a countable set (to be used as a set of parameter symbols) and let $\mathbf{H}$ be a finite or countable set of sentences in pure predicate logic. Then there exists a finished tableau $\mathbf{T}$ on M with root $\mathbf{H}$, such that no free variables occur on $\mathbf{T}$.*

Proof: We construct a sequence of finite tableaus

$$\mathbf{T}_0 \subset \mathbf{T}_1 \subset \mathbf{T}_2 \subset \ldots$$

such that $\mathbf{T}_{n+1}$ is an extension of $\mathbf{T}_n$ for each $n \in \mathbf{N}$. The finished tableau will be the union of the tableaus in this sequence. The tableau $\mathbf{T}_0$ is just the trivial tableau with only the root node and the given set of sentences $\mathbf{H}$ attached to it. Since the set M is countable we may list its elements:

$$M = \{a_1, a_2, a_3, \ldots\}.$$

We also list the elements of the finite or countable set $\mathbf{H}$,

$$\mathbf{H} = \{\mathbf{C}_1, \mathbf{C}_2, \mathbf{C}_3, \ldots\}.$$

Let $\mathbf{H}_0$ be the empty set and $\mathbf{H}_n = \{\mathbf{C}_1, \ldots, \mathbf{C}_n\}$, with the understanding that if $\mathbf{H}$ is finite with n elements then we instead take $\mathbf{H}_m = \mathbf{H}$ when $m \geq n$.

We shall construct the sequence $\mathbf{T}_n$ of tableaus using only the parameter symbols from M in the quantifier rules. Since each tableau $\mathbf{T}_n$ will have only finitely many nodes, $\mathbf{T}$ will contain only finitely many sentences outside $\mathbf{H}$ and hence only finitely many parameters from M occur in $\mathbf{T}_n$. (The finished tableau, however, may well use all the parameters from M.)

Given the finite tableau $\mathbf{T}_n$ we form a finite extension $\mathbf{T}_{n+1}$ with the following properties. For any noncontradictory branch $\mathbf{\Gamma}$ of $\mathbf{T}_{n+1}$ and wff $\mathbf{A}$ on $\mathbf{\Gamma}$ such that either $\mathbf{A} \in \mathbf{H}_n$ or $\mathbf{A}$ is a nonroot wff $\mathbf{T}_n$:

1. If $\mathbf{A}$ is of the form $\forall x\mathbf{B}$ then for every $i = 1, 2, \ldots, n+1$ the wff $\mathbf{B}(x//a_i)$ is on $\mathbf{\Gamma}$.

2. If $\mathbf{A}$ is of the form $\neg\exists x\mathbf{B}$ then for every $i = 1, 2, \ldots, n+1$, the wff $\neg\mathbf{B}(x//a_i)$ is on $\mathbf{\Gamma}$.

3. If $\mathbf{A}$ is of any other form, then $\mathbf{A}$ is used (as the hypothesis of a tableau extension rule) at least once along $\mathbf{\Gamma}$. For example, if $\mathbf{A}$ is of the form $\exists x\mathbf{B}$ then for some integer k, possibly much bigger than n, the wff $\mathbf{B}(x//a_k)$ is on $\mathbf{\Gamma}$. As a second example, if $\mathbf{A}$ is of the form $\mathbf{B} \vee \mathbf{C}$ then either $\mathbf{B}$ is on $\mathbf{\Gamma}$ or $\mathbf{C}$ is on $\mathbf{\Gamma}$.

Furthermore, no contradictory branch of $\mathbf{T}_n$ is extended in forming $\mathbf{T}_{n+1}$.

The tableau $\mathbf{T}_{n+1}$ is constructed in finitely many stages by taking care of all wffs in $\mathbf{H}_n$ and all nonroot wffs of $\mathbf{T}_n$ one at a time. Now we claim that the union $\mathbf{T} = \bigcup_n \mathbf{T}_n$ is a finished tableau on M. Let $\mathbf{\Gamma}$ be any branch of $\mathbf{T}$. If $\mathbf{\Gamma}$ is contradictory then $\mathbf{\Gamma}$ is finite as before.

If $\mathbf{\Gamma}$ is not contradictory we must show that $\mathbf{\Delta}$, the set of all wffs on $\mathbf{\Gamma}$, is a finished set. Suppose that $\mathbf{A} \in \mathbf{\Delta}$. Then for some n, $\mathbf{A}$ is either in $\mathbf{H}_n$ or is a nonroot wff of $\mathbf{T}_n$. Since $\mathbf{\Gamma} \cap \mathbf{T}_{n+1}$ is a branch of $\mathbf{T}_{n+1}$, by the construction $\mathbf{A}$ has been used on $\mathbf{\Gamma} \cap \mathbf{T}_{n+1}$ and hence on $\mathbf{\Gamma}$. Now suppose that $\mathbf{A}$ has the form $\forall x\mathbf{B}$. Then for every $m > n$ and $i \leq m$, the wff $\mathbf{B}(x//a_i)$ is on $\mathbf{\Gamma} \cap \mathbf{T}_m$. Hence for every $i = 1, 2, \ldots$ the wff $\mathbf{B}(x//a_i)$ is on $\mathbf{\Gamma}$. Similarly if $\mathbf{A}$ has the form $\neg\exists x\mathbf{B}$, then for every

$i = 1, 2, \ldots$ the wff $\neg\mathbf{B}(x//a_i)$ is on $\mathbf{\Gamma}$. The other cases for the wff $\mathbf{A}$ may be dealt with in a similar manner to complete the proof that $\mathbf{\Delta}$ is a finished set. It follows that $\mathbf{T}$ is a finished tableau with hypothesis set $\mathbf{H}$. **End of Proof.**

Proof of the Main Lemma: The Main Lemma for the Completeness Theorem can now be deduced as follows. Let $\mathbf{H}$ be a finite or countable set of sentences in pure predicate logic. By the Tableau Extension Lemma, there is a finished tableau $\mathbf{T}$ on M with root $\mathbf{H}$ and no free variables. By the König Tree Theorem, $\mathbf{T}$ is either a tableau confutation of $\mathbf{H}$ or $\mathbf{T}$ has a noncontradictory branch $\mathbf{\Gamma}$. In the latter case, the set of wffs on $\mathbf{\Gamma}$ is a finished set on M, so by the Finished Set Lemma, $\mathbf{H}$ has a model. **End of Proof.**

Note that this proof shows that any finite or countable set of sentences of pure predicate logic which has a model has an infinite model, i.e., one with an infinite universe. This will not be the case for the full predicate logic (at least if we require our model to respect equality in the sense explained in the next chapter).

We conclude this section by stating the Compactness Theorem for pure predicate logic. It is proved from the Main Lemma exactly as in the propositional logic case.

Theorem 2.9.4 (Compactness Theorem) *Let* $\mathbf{H}$ *be any countable set of sentences of pure predicate logic. If every finite subset of* $\mathbf{H}$ *has a model, then* $\mathbf{H}$ *has a model.*

2.10 Equivalence Relations

The full predicate logic studied in the next chapter introduces some rules of logic which deal with equality. The pure predicate logic studied in this chapter treats the equality symbol like any other binary relation symbol. However, by adding certain axioms to the hypothesis set of any tableau, we can assure (without adding any additional logical rules) that the equality symbol essentially represents true equality. We shall explain how to do this in this section.

To make it easier to distinguish an equality symbol in our vocabulary $\mathcal{P}$ of predicate logic from the ordinary uses of equality outside

of predicate logic, we shall use the symbol $\doteq$ as an equality symbol in predicate logic. There is nothing in our definition of model which says that the value $\doteq^{\mathcal{M}}$ of the equality symbol $\doteq$ has to be the equality relation between elements of M. We say that a model $\mathcal{M}$ of type $\mathcal{P}$ **respects equality** iff for all $a, b \in M$, the universe of $\mathcal{M}$, we have

$$\mathcal{M} \models a \doteq b \text{ if and only if } a = b.$$

In the next chapter we introduce the term **pre-model** for a model which may or may not respect equality, and reserve the term *model* for models which do respect equality.

Equality Axioms

(1) $\forall x\, x \doteq x$

(2) $\forall x \forall y [x \doteq y \Rightarrow y \doteq x]$

(3) $\forall x \forall y \forall z [x \doteq y \wedge y \doteq z \Rightarrow x \doteq z]$

(4) $\forall \vec{x} \forall \vec{y} [\vec{x} \doteq \vec{y} \Rightarrow [p(\vec{x}) \Leftrightarrow p(\vec{y})]]$

Definition 2.10.1 The sentences in the box comprise the set $\mathbf{E}(\mathcal{P})$ of **equality axioms** for the vocabulary $\mathcal{P}$. There is one instance of (4) for each predicate symbol p. In (4) p denotes an n-ary predicate symbol and we have used the following abbreviations:

$$\begin{array}{rcl} \forall \vec{x} & \text{for} & \forall x_1 \forall x_2 \cdots \forall x_n \\ \forall \vec{y} & \text{for} & \forall y_1 \forall y_2 \cdots \forall y_n \\ \vec{x} \doteq \vec{y} & \text{for} & x_1 \doteq y_1 \wedge x_2 \doteq y_2 \wedge \cdots \wedge x_n \doteq y_n \\ p(\vec{x}) & \text{for} & p(x_1, x_2, \ldots, x_n) \\ p(\vec{y}) & \text{for} & p(y_1, y_2, \ldots, y_n). \end{array}$$

In this section we shall prove the following Soundness and Completeness Theorem for models which respect equality.

Soundness & Completeness with Equality

A sentence **B** in the vocabulary $\mathcal{P}$ is true in every model for $\mathcal{P}$ which respects equality if and only if **B** is tableau provable from the hypothesis set $\mathbf{E}(\mathcal{P})$.

This is a special case of the following theorem:

Theorem 2.10.2 *Let* **H** *be a set of sentences and* **A** *a sentence in the vocabulary* $\mathcal{P}$. *Every model of* **H** *which respects equality is a model of* **A** *if and only if there is a tableau proof of* **A** *from the hypothesis set* $\mathbf{H} \cup \mathbf{E}(\mathcal{P})$.

To prove this theorem we need to develop the theory of equivalence relations. We shall use this theory again in Chapter 3. A binary relation $\equiv$ on a set X is called an **equivalence relation** iff the equality axioms (1), (2), and (3) above hold in the model $\mathcal{M}$ whose universe is X and where the value $\doteq^{\mathcal{M}}$ assigned the equality symbol is $\equiv$. The equivalence relation is called a **congruence relation** for the relation $R \in REL_n(X)$ iff in addition $\mathcal{M}$ models equality axiom (4) when $p^{\mathcal{M}} = R$. In other words an equivalence relation on X is a binary relation on X which satisfies the following three laws:

Reflexive Law	$x \equiv x$
Symmetric Law	$x \equiv y$ implies $y \equiv x$
Transitive Law	$x \equiv y$ and $y \equiv z$ implies $x \equiv z$

for $x, y, z \in X$. An equivalence relation $\equiv$ is a congruence relation for $R \in REL_n(X)$ iff in addition

$$(x_1, \ldots, x_n) \in R \text{ and } x_1 \equiv y_1, \cdots, x_n \equiv y_n \text{ implies } (y_1, \ldots, y_n) \in R$$

for $x_1, \ldots, y_n \in X$.

The equality relation on any set is an equivalence relation. The equality relation is a congruence relation for any relation R: equals may be substituted for equals without changing the meaning. Another important equivalence relation is *equality modulo* m. Each positive integer m determines an equivalence relation on $\mathbf{Z}$ denoted $\equiv_m$. The definition is[2]

$$x \equiv_m y \Longleftrightarrow m|(y - x).$$

The notation $m|b$ is read m *divides* b and means that $m = ab$ for some integer a. For example $3 \equiv_7 24$ while $7 \not\equiv_3 2$. Equality mod m is a congruence relation for each of the ternary relations $x + y = z$ and $xy = z$ but not for the binary relation $x < y$. (See Exercises 34 on page 135 and 10 on page 183.)

Any function π from X to $\bar{X}$ determines an equivalence relation on X via the definition

$$x \equiv_\pi y \Longleftrightarrow \pi(x) = \pi(y).$$

For example, define π from $\mathbf{Z}$ to $\{0, 1, \ldots, m-1\}$ by taking $\pi(x)$ to be the remainder when x is divided by m:

$$r = \pi(x) \Longleftrightarrow x = qm + r, \qquad 0 \leq r < m.$$

Then $x \equiv_m y$ iff $\pi(x) = \pi(y)$. The following lemma reverses this process.

Lemma 2.10.3 *Let* $\equiv$ *be an equivalence relation on a set* X *and for each* $x \in X$ *define the* **equivalence class** *of* x *by*

$$[x] = \{y \in X : x \equiv y\}.$$

Then for all $x, y \in X$ *the following are equivalent:*

(i) $x \equiv y$;

(ii) $[x] = [y]$;

[2] Other commonly used conventional notations for this are $x \equiv y\,(m)$ and $x \equiv y \pmod m$.

(iii) $[x] \cap [y] \neq \emptyset$.

Proof: Assume (i). Choose $z \in [x]$. Then $z \equiv x$ and $x \equiv y$ so $z \equiv x$ by the Transitive Law. Hence $[x] \subset [y]$. Choose $z \in [y]$. Then $z \equiv y$ so $z \equiv x$ by the Transitive and Symmetric Laws. Hence $[y] \subset [x]$. Hence $[x] = [y]$. We have proved (ii).

Assume (ii). Then $x \in [x] = [y]$ by the Reflexive Law $[x] \cap [y] \neq \emptyset$. We have proved (iii).

Assume (iii). Then there is a $z \in [x] \cap [y]$. Hence $z \equiv x$ abd $z \equiv y$ so $x \equiv y$ by the Transitive and Symmetric laws. We have proved (i). **End of Proof.**

Lemma 2.10.4 *Suppose that $\equiv$ is a congruence relation for a relation $R \in REL_n(X)$. Let $\bar{X}$ denote the set of equivalence classes of $\equiv$. Then there is a unique relation $\bar{R} \in REL_n(\bar{X})$ such that*

$$(x_1, x_2, \ldots, x_n) \in R \Longleftrightarrow ([x_1], [x_2], \ldots, [x_n]) \in \bar{R}.$$

The relation $\bar{R}$ is called the relation **induced** *by R on the set of equivalence classes $\bar{X}$.*

Proof: Define $\bar{R}$ by

$$\bar{R} = \{([x_1], [x_2], \ldots, [x_n]) : (x_1, x_2, \ldots, x_n) \in R\}.$$

Then $(x_1, x_2, \ldots, x_n) \in R$ implies $([x_1], [x_2], \ldots, [x_n]) \in \bar{R}$ by definition. If $([x_1], [x_2], \ldots, [x_n]) \in \bar{R}$ then (again by definition) there exist $y_1, y_2, \ldots, y_n$ with $[x_i] = [y_i]$ and $(y_1, y_2, \ldots, y_n) \in R$. But then $(x_1, x_2, \ldots, x_n) \in R$ by the definition of congruence relation. Uniqueness is an immediate consequence of the definition of equality of sets. (Exercise 35 relates to this construction.) **End of Proof.**

Now assume that $\mathcal{P}$ is a vocabulary which contains the equality symbol and let $\mathcal{M}$ be a model for sentences (1), (2), and (3) and all the sentences (4) where $p \in \mathcal{P}$. Let M be the universe of $\mathcal{M}$. Let $\equiv$ be the binary relation $\doteq^{\mathcal{M}}$ which represents the equality predicate symbol $\doteq$ in the model $\mathcal{M}$. By definition $\equiv$ is a congruence relation for each of the relations $p^{\mathcal{M}}$. Let $\bar{M}$ be the set of equivalence classes and for each $p \in \mathcal{P}$ let $p^{\bar{\mathcal{M}}}$ be the relation induced by $p^{\mathcal{M}}$ and let $\bar{\mathcal{M}}$ be the model thus defined.

Theorem 2.10.5 (Equality Construction) *The model $\bar{\mathcal{M}}$ respects equality. Moreover for any sentence* **A** *we have*

$$\mathcal{M} \models \mathbf{A} \Longleftrightarrow \bar{\mathcal{M}} \models \mathbf{A}.$$

Proof: The proof is by induction on the length of **A**. To make the induction work it is necessary to prove a stronger statement, namely that

$$\mathcal{M} \models \mathbf{A}(v) \Longleftrightarrow \bar{\mathcal{M}} \models \mathbf{A}(\bar{v})$$

for every valuation

$$v = \{(x_1, a_1), (x_2, a_2), \ldots, (x_\ell, a_\ell)\}$$

where $x_1, x_2, \ldots x_\ell$ are distinct individuals, $a_1, a_2, \ldots, a_\ell \in M$, and

$$\bar{v} = \{(x_1, [a_1]), (x_2, [a_2]), \ldots, (x_\ell, [a_\ell])\}.$$

We omit the details.

Proof of Soundness in 2.10.2: Suppose that $\mathbf{H} \cup \mathbf{E}(\mathcal{P}) \vdash \mathbf{A}$. Let $\mathcal{M}$ be a model of **H** which respects equality. Then $\mathcal{M}$ is also a model for the set $\mathbf{E}(\mathcal{P})$ of equality axioms. Now by the ordinary Soundness Theorem 2.7.4, $\mathcal{M}$ is a model of **A**. **End of Proof.**

Proof of Completeness in 2.10.2: Suppose there is no tableau proof of **A** from $\mathbf{H} \cup \mathbf{E}(\mathcal{P})$. Then by the ordinary Completeness Theorem 2.9.2 there is a model $\mathcal{M}$ of $\mathbf{H} \cup \mathbf{E}(\mathcal{P})$ in which **A** is false. By the Equality Construction the model $\bar{\mathcal{M}}$ respects equality, and is a model of **H** in which **A** is false. **End of Proof.**

Henceforth we assume that all models mentioned in this book respect equality.

2.11 Order Relations

By an **order relation** mathematicians usually mean a transitive binary relation, that is a binary relation $\leq$ satisfying the transitive law below. As usual write $x \leq y$ instead of $(x,y) \in \leq$, and $x < y$ instead of $[x \leq y \wedge \neg x \doteq y]$.

Order Axioms

(1) Reflexive Law	$\forall x\ x \leq x$
(2) Transitive Law	$\forall x \forall y \forall z [x \leq y \wedge y \leq z \Rightarrow x \leq z]$
(3) Anti-symmetric Law	$\forall x \forall y [x \leq y \wedge y \leq x \Rightarrow x \doteq y]$
(4) Comparability Law	$\forall x \forall y [x \leq y \vee y \leq x]$
(5) No First Element	$\neg \exists z \forall x\ z \leq x$
(6) No Last Element	$\neg \exists w \forall x\ x \leq w$
(7) Density Law	$\forall x \forall y [x < y \Rightarrow \exists z [x < z \wedge z < y]]$.

A model for axioms (1)-(2) is called a **pre-order**. A model for axioms (1)-(3) is called a **partial order**. A model for axioms (1)-(4) is called a **linear order**. An order which satisfies (5) is said to have **no first element**; an order which doesn't is said to have a **first element**. Similarly for (6). A model for axioms (1)-(7) is called a **dense linear order** without first or last element.

Some familiar linear orders include the set $\mathbf{R}$ of real numbers, the set $\mathbf{Q}$ of rational numbers, the set $\mathbf{Z}$ of integers, and the set $\mathbf{N}$ of natural numbers, all with the usual $\leq$ relation. Of these $\mathbf{R}$ and $\mathbf{Q}$ are dense linear orders without first or last element, $\mathbf{Z}$ and $\mathbf{N}$ are not dense, $\mathbf{Z}$ has no first or last element, and $\mathbf{N}$ has a first element but no last element. Each $a, b \in \mathbf{R}$ determines four **intervals**

$$[a,b] = \{x \in \mathbf{R} : a \leq x \leq b\}, \qquad]a,b[= \{x \in \mathbf{R} : a < x < b\},$$

$$[a,b[= \{x \in \mathbf{R} : a \leq x < b\}, \qquad]a,b] = \{x \in \mathbf{R} : a < x \leq b\},$$

called respectively the **closed interval, open interval**, and **half-open intervals** with endpoints a and b. If $a < b$ these are all dense linear orders, $[a,b]$ has first and last element, $]a,b[$ has neither first nor last element, $[a,b[$ has a first but no last element, and $]a,b]$ has a last but no first element. An example of a partial order which is not a linear order is the set $P(X)$ of all subsets of a set X having more than one element, where the relation symbol $\leq$ is interpreted as the subset relation $\subset$. Thus, for example, letting $X = \mathbf{N}$, the set of natural numbers, we can demonstrate that (4) fails for the model $\mathcal{M} = (P(X), \subset)$ by considering the two sets

$$O = \{n \in \mathbf{N} : n \text{ is odd}\}, \qquad E = \{n \in \mathbf{N} : n \text{ is even}\}.$$

Any binary relation R on a set X determines a preorder $\leq_R$ called the **transitive closure** of R. The definition is that for $x, y \in X$ we have $x \leq_R y$ iff there is a sequence $x_0, x_1, x_2, \ldots x_n$ of elements of X such that $x_0 = x$, $x_n = y$ and

$$(x_{k-1}, x_k) \in R \text{ for } k = 1, 2, \ldots, n.$$

The transitive closure is reflexive since sequences of length $n = 0$ are allowed. It is transitive since a sequence from x to y may be followed by a sequence from y to z to given a sequence from x to z. If the set X is finite we may represent the transitive closure as follows: Draw a dot for each element of X and an arrow from x to y if $(x,y) \in R$. Then $x \leq_R y$ iff x may be connected to y by a path of arrows.

2.12 Set Theory

In this section we give the axioms for ZST — **Zermelo set theory**, which were introduced by Zermelo in 1906 as a foundation for mathematics. Zermelo set theory is a part of a larger and more recent set of axioms called ZFC — **Zermelo-Fraenkel set theory** with the axiom of choice. The first-order language in which the sentences of Zermelo set theory are formulated has no proposition symbols and has just two predicate symbols: one for equality ($\doteq$) and one for membership($\in$). The equality axioms are tacitly included in ZST. Thus, when we say

that a sentence **C** is a **theorem** of ZST we mean that it is provable from the axioms of ZST and the equality axioms. While the vocabulary of ZST is very simple, it has been shown that the sentences of (virtually) every mathematical theory can be translated into sentences of ZST. Much of mathematics, including all the mathematics done in this book, can be carried within ZST, and (virtually) every theorem of mathematics can be treated as a theorem of the larger axiom set ZFC. See A. Levy's book on set theory for a complete list of axioms for the larger theory ZFC and interesting discussion.

Axioms of Zermelo Set Theory

(1) Pairing: $\forall x \forall y \exists z \forall u[u \in z \Leftrightarrow x \doteq u \vee y \doteq u]$

Translation: If x, y are sets, so is $z = \{x, y\}$.

(2) Extensionality: $\forall x \forall y[x \doteq y \Leftrightarrow \forall z[z \in x \Leftrightarrow z \in y]]$

Translation: Two sets are equal iff they have the same elements.

(3) Empty set: $\exists x \forall y[y \in x \Rightarrow y \neq y]$

Translation: There is a set which has no elements.

(4) Union: $\forall x \exists y \forall z[z \in y \Leftrightarrow \exists u[u \in x \wedge z \in u]]$

Translation: The union of a set of sets is a set.

(5) Power set: $\forall x \exists y \forall z[z \in y \Leftrightarrow \forall u[u \in z \Rightarrow u \in x]]$

Translation: The collection of all subsets of a set is also a set.

(6) Infinity: $\exists u[\emptyset \in u \wedge \forall x[x \in u \Rightarrow x \cup \{x\} \in u]]$

Translation: There is an infinite set.

($7_{\mathbf{A}}$) Comprehension: $\forall x \, \exists y \, \forall z \, [z \in y \Leftrightarrow [z \in x \wedge \mathbf{A}(z)]]$

Translation: There is a set $y = \{z \in x : \mathbf{A}\}$.

The last item ($7_{\mathbf{A}}$) is an infinite list of axioms, one for each wff $\mathbf{A}(z)$ in which y does not occur. Together, this infinite list is called the *Comprehension scheme*. Given a set x and a wff $\mathbf{A}(z)$, the Comprehension scheme allows us to form the set of all $z \in x$ such that $\mathbf{A}(z)$. For example, once we have the set of natural numbers and a wff which expresses the property "z is even", we can use the Comprehension scheme to prove that the set of even natural numbers exists.

The remaining axioms of ZFC, which are not given here, are also sentences of pure predicate logic with the $\doteq$ and $\in$ symbols. Their names are the *Axiom of Regularity*, the *Axiom of Choice*, and an infinite list of axioms called the *Scheme of Replacement.*

The Axiom of Infinity deserves some comment. To make the axiom readable, we have expressed it using symbols which are not in the original vocabulary of Zermelo set theory: $\emptyset$ for the empty set, and $x \cup \{x\}$ for the union of x and the singleton $\{x\}$. These expressions are abbreviations for notions given to us by the other axioms. So, for example, $\emptyset \in u$ could be formally expressed by the wff

$$\exists z[z \in u \wedge \forall y \neg y \in z].$$

We leave as an exercise (Exercise 49) the verification that the entire Axiom of Infinity can be expressed in a formally correct way as a wff in the vocabulary of Zermelo set theory.

It is reasonable to translate the Axiom of Infinity as "there is an infinite set," because it says that there is a set u such that

$$\emptyset \in u, \{\emptyset\} \in u, \{\emptyset, \{\emptyset\}\} \in u, \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\} \in u, \ldots.$$

2.13 Tableaus and Mathematical Proofs

In composing a "real" mathematical proof, a mathematician is free to use not only the rules of tableau proofs, but any other rules which are known to be sound. By a **sound set of rules** we mean a set of rules such that any wff which is proved from a hypothesis set **H** using the rules is a semantic consequence of **H**. Real mathematical proofs are usually written in paragraph form rather than in tree form. However, they can be translated into tree form, and can be thought of as tableau proofs which use extra rules. When extra rules are allowed, proofs become easier to find and easier to understand. On the other hand, the concept of a proof is more complicated when more rules are allowed. When the aim is to study the concept of a proof, as in this book, one should keep the set of rules as small and simple as possible. But when the aim is to discover proofs in mathematics, one should make the set of rules as rich as possible.

In this section we shall make a short detour from our main path and discuss some of the extra rules of proof in pure predicate logic which are commonly used in mathematics. Each of these extra rules is easily seen to be sound. The Extended Completeness Theorem shows that any wff which can be proved from a hypothesis set **H** using the tableau rules and the extra rules can be proved from **H** using only the tableau rules. Often, however, the formal tableau proof will be considerably longer.

For the sake of simplicity, our presentation in this section will be less precise than in our main line of development. We shall deal with tableaus in a broader sense which are built using a variety of extra rules as well as the original tableau extension rules. In order to combine various rules together, we need to work with hypothesis sets which contain sentences with parameters from $\mathcal{K}$. The Extended Soundness and Completeness Theorems for sentences with parameters from $\mathcal{K}$ are given in the Exercises.

For each of the extra rules, we shall first display the rule in a box, and then prove a theorem which says that the extra rule is sound. In this section we shall always assume that **H** is a set of sentences with parameters from $\mathcal{K}$, and that all tableaus mentioned are finite.

> **Direct Proof Rule.** Color a node of a tableau red if every branch through it either contains the formula to be proved or is contradictory.

Theorem 2.13.1 (Direct Proof Theorem) *If there is a tableau* **T** *with root* **H** *such that the wff* **A** *occurs on every noncontradictory branch of* **T**, *then* **A** *is tableau provable from* **H**.

Proof: To get a tableau proof of **A** from **H**, simply add $\neg$**A** to the list of hypotheses **H**. This makes each branch of **T** contradictory, so that **T** is a tableau proof of **A** from **H**. **End of Proof.**

By a **direct proof** of **A** from **H** we mean a tableau with root **H** such that **A** occurs on every noncontradictory branch. The Direct

Proof Theorem shows that if **A** has a direct proof from **H**, then it has a tableau proof from **H**. Ordinary tableau proofs, which add $\neg\mathbf{A}$ to the list of hypotheses, are called **indirect proofs**. Sometimes there is an indirect proof of **A** from **H** but no direct proof. It is considered good form in mathematics to give a direct proof if you can find one, because direct proofs are often easier to follow than indirect proofs.

Learning Rule. If $\mathbf{\Gamma}$ is a branch in a tableau and **B** is tableau provable from some or all of the formulas in $\mathbf{\Gamma}$, then the tableau may be extended by adding **B** to the end of the branch $\mathbf{\Gamma}$.

Theorem 2.13.2 (Learning Theorem) *Suppose that a wff* **A** *has a proof from* **H** *which uses all the tableau rules plus the Learning Rule. Then* **A** *is tableau provable from* **H**.

Proof: Our plan is to prove that the Learning Rule is sound by imitating the proof of the Soundness Theorem, then to use the Extended Completeness Theorem to prove the Learning Theorem.

Let **T** be a labeled tree whose root is a set of sentences **H** which is built up using the tableau rules and the Leaning Rule. We may assume that all the wffs on **T** are sentences with parameters in some set $\mathcal{K}'$ which contains $\mathcal{K}$. As in the proof of Lemma 2.7.2, one can prove by induction on the number of nodes in **T** that for any model $\mathcal{M}$ of **H**, there is a branch $\mathbf{\Gamma}$ of **T** and a valuation v such that $\mathcal{M} \models \mathbf{\Gamma}(v)$. The induction step has one new case corresponding to the Learning Rule. Suppose **B** is added to the end of a branch $\mathbf{\Gamma}$ by the Learning Rule. Let $\mathcal{M} \models \mathbf{\Gamma}(v)$. By the Extended Soundness Theorem for hypotheses with extra parameters (Exercise 24), $\mathcal{M} \models \mathbf{B}(v)$. This completes the induction.

Exactly as in the proof of the Soundness Theorem, we see that if **A** is provable from **H** using the tableau rules and the Learning Rule, then **A** is a semantic consequence of **H**. Finally, by the Extended Completeness Theorem, if **A** is provable from **H** using the tableau rules and the Learning Rule, then **A** is tableau provable from **H**. **End of Proof.**

The Learning Rule is quite powerful. There are two ways to use it in a mathematical proof. One way is to invoke a previous theorem during the proof of a new theorem. This makes it possible to build up a body of knowledge by keeping a record of theorems which have been proved. The second way is to temporarily stop work on the original tableau, use a new a sheet of paper to write out a tableau proof of a wff **A** from the formulas on the branch, and then add **A** to the end of the branch in the original tableau. One can think of this method in terms of "windows" which can be opened and used to hold subordinate tableaus within the main tableau. To use the Learning Rule, "open a window" at the end of a branch in a tableau. Inside the window, put a tableau proof of a wff **A** from the formulas on the branch. Then return to the main tableau and add the new formula **A** right below the window. Sometimes there will be windows within windows.

In many cases a wff **A** easily follows from a branch $\mathbf{\Gamma}$ using only *propositional* logic, and one can add **A** to the end of $\mathbf{\Gamma}$ by the Learning Rule. For example, if **B** and **C** both occur on a branch, one can add $\mathbf{B} \wedge \mathbf{C}$ to the end of the branch. Similarly, if $\mathbf{A} \Rightarrow \mathbf{B}$ and $\mathbf{B} \Rightarrow \mathbf{A}$ both occur on a branch, one can add $\mathbf{A} \Leftrightarrow \mathbf{B}$ to the end of the branch. Another common example is modus ponens: if **B** and $\mathbf{B} \Rightarrow \mathbf{C}$ both occur on a branch, one can add **C** to the end of the branch. There is a similar consequence of the Learning Rule which uses an equivalence instead of an implication: If **B** and $\mathbf{C} \Leftrightarrow \mathbf{B}$ both occur on a branch, one can add **C** to the end of the branch. Some other examples are provided by the valid argument rules given in Chapter 1.

By the Learning Rule, any formula which has a tableau proof can be added at any time to the end of a branch; for example, $\mathbf{A} \vee \neg\mathbf{A}$ can always be added.

The Learning Rule may also be used to add the formula $\exists x\, \mathbf{A}$ to the end of a branch whenever a formula of the form $\mathbf{A}(x//c)$ occurs on the branch. In this way, one can often give a direct proof of a formula which starts with an existential quantifier.

Each of the next three rules is obtained by combining a theorem with the Learning Rule.

Deduction Rule. If Γ is a branch of a tableau and $\mathbf{B}$ is tableau provable from $\mathbf{A}$ and some or all of the formulas in Γ, then $\mathbf{A} \Rightarrow \mathbf{B}$ may be added to the end of the branch Γ.

Theorem 2.13.3 (Deduction Theorem) *If* $\mathbf{B}$ *is tableau provable from* $\mathbf{H}$ *and* $\mathbf{A}$*, then* $\mathbf{A} \Rightarrow \mathbf{B}$ *is tableau provable from* $\mathbf{H}$.

Proof: Suppose $\mathbf{H}, \mathbf{A} \vdash \mathbf{B}$. By the Extended Soundness Theorem, $\mathbf{H}, \mathbf{A} \models \mathbf{B}$. It follows from the truth table for $\Rightarrow$ that $\mathbf{H} \models \mathbf{A} \Rightarrow \mathbf{B}$. By the Extended Completeness Theorem, $\mathbf{H} \vdash \mathbf{A} \Rightarrow \mathbf{B}$. **End of Proof.**

To see that the Deduction Rule is sound, we note that it is obtained by combining the Deduction Theorem with the Learning Rule as follows. If $\mathbf{B}$ is tableau provable from $\mathbf{A}$ and a branch Γ, then $\mathbf{A} \Rightarrow \mathbf{B}$ is provable from Γ by the Deduction Theorem, so we may add $\mathbf{A} \Rightarrow \mathbf{B}$ to the end of Γ by the Learning Rule.

The Deduction Rule is often used in the following way. To add $\mathbf{A} \Rightarrow \mathbf{B}$ to the end of a branch Γ, open a window and prove $\mathbf{B}$ from $\mathbf{A}$ and formulas on Γ, then return to the main tableau and use the Deduction Rule. In a mathematical proof, this is usually expressed by saying that we temporarily assume $\mathbf{A}$ and prove $\mathbf{B}$, then conclude that $\mathbf{A} \Rightarrow \mathbf{B}$.

Generalization Rule. If Γ is a branch of a tableau and $\mathbf{A}(x//c)$ is tableau provable from a set of wffs on Γ in which the individual symbol c does not occur free, then $\forall x\, \mathbf{A}$ may be added to the end of the branch Γ.

Theorem 2.13.4 (Generalization Theorem) *Suppose that* $\forall x \mathbf{A}$ *is a sentence with parameters from* $\mathcal{K}$*. If an individual symbol* c *does not occur free in* $\mathbf{H}$ *and* $\mathbf{A}(x//c)$ *is tableau provable from* $\mathbf{H}$*, then* $\forall x\, \mathbf{A}$ *is tableau provable from* $\mathbf{H}$.

Proof: Let $\mathbf{T}$ be a tableau proof of $\mathbf{A}(x//c)$ from $\mathbf{H}$. By adding the additional hypothesis $\neg\forall x\, \mathbf{A}$ to $\mathbf{T}$ and inserting the wff $\neg\mathbf{A}(x//c)$ immediately below the root of the tableau, we obtain a tableau proof of $\forall x\, \mathbf{A}$ from $\mathbf{H}$. **End of Proof.**

We can see that the Generalization Rule is sound as follows. If $\mathbf{A}(x//c)$ is tableau provable from formulas on a branch Γ in which c does not occur free, then $\forall x\, \mathbf{A}$ is provable from Γ by the Generalization Theorem, so we may add $\forall x\, \mathbf{A}$ to the end of Γ by the Learning Rule.

The Generalization Rule is often used in the following way. To add $\forall x\, \mathbf{A}$ to the end of a branch Γ, open a window, choose a new individual symbol c, prove $\mathbf{A}(x//c)$ from formulas on Γ in which c does not occur free, then come back to the main tableau and use the Generalization Rule. In a mathematical proof, this is usually expressed by saying that we let c be arbitrary, prove $\mathbf{A}(x//c)$, then conclude that $\forall x\, \mathbf{A}$.

We shall discuss two more extra rules which are used very frequently in mathematical proofs, the Definition Rule and the Substitution Rule.

Definition Rule. If Γ is a branch of a tableau, $\mathbf{A}$ is a wff with the free variables $x_1, \ldots, x_n$, and r is an n-ary predicate symbol which does not occur on the branch Γ or in $\mathbf{A}$, then the formula

$$\forall x_1 \ldots \forall x_n [r(x_1, \ldots, x_n) \Leftrightarrow \mathbf{A}] \qquad (2.1)$$

may be added at the end of the branch.

Theorem 2.13.5 (Definition Theorem) *If* $\mathbf{A}$ *and* $\mathbf{B}$ *are wffs with parameters from* $\mathcal{K}$*,* $\mathbf{A}$ *has the free variables* $x_1, \ldots, x_n$*,* r *is an* n*-ary predicate symbol which does not occur in* $\mathbf{H}$*,* $\mathbf{A}$*, or* $\mathbf{B}$*, and* $\mathbf{B}$ *is tableau provable from* $\mathbf{H}$ *together with the formula 2.1, then* $\mathbf{B}$ *is tableau provable from* $\mathbf{H}$ *alone.*

The formula 2.1 is called a **definition** of the predicate r. An application of the Definition Rule can be easily recognized in a mathematical

proof because it is usually signaled by a word such as "define," "let," or "where." The purpose of this rule is to make a proof easier to understand by replacing a long formula which may appear several times by an atomic formula with a new predicate symbol. This is especially helpful if the name of the new symbol is chosen to remind the reader of its meaning.

There are many examples of the Definition Rule in the proofs in this book. For instance, during the proof of the Completeness Theorem for Propositional Logic, the predicates "basic wff," "unused node," "finished branch," and "finished tableau" were defined.

Proof of the Definition Theorem: Suppose that **B** is tableau provable from $\mathbf{H} \cup \{\mathbf{C}\}$ where **C** is the formula 2.1. Let $\mathcal{P}$ be the vocabulary of **H**, so that $\mathcal{P} \cup \{r\}$ is the vocabulary of $\mathbf{H} \cup \{\mathbf{C}\}$. Then any model of $\mathbf{H} \cup \{\mathbf{C}\}$ in the vocabulary $\mathcal{P} \cup \{r\}$ is a model of **B**. Now let $\mathcal{M}$ be a model of **H** in the original vocabulary $\mathcal{P}$. We may expand $\mathcal{M}$ to a model $\mathcal{N}$ of $\mathbf{H} \cup \{\mathbf{C}\}$ in the vocabulary $\mathcal{P} \cup \{r\}$ by taking $r_{\mathcal{N}}$ to be the set of n-tuples of elements of M which satisfy the wff **A** in $\mathcal{M}$. Thus $\mathcal{N} \models \mathbf{B}$. It can by shown by an induction on wffs that for every wff **D** in the original vocabulary $\mathcal{P}$ and any valuation v in M, $\mathcal{M} \models \mathbf{D}(v)$ if and only if $\mathcal{N} \models \mathbf{D}(v)$. Therefore $\mathcal{M} \models \mathbf{B}$, so $\mathbf{H} \models \mathbf{B}$. Finally, by the Extended Completeness Theorem, $\mathbf{H} \vdash \mathbf{B}$. **End of Proof.**

The following Substitution Rule is often used in combination with the Definition Rule.

Substitution Rule. Suppose **C** is a wff, **A** and **B** are wffs with at most the free variables $x_1, \ldots, x_n$, **A** is a well-formed part of **C**, and **D** is the wff obtained from **C** by replacing the string **A** by the string **B**. If Γ is a branch which contains the wffs **C** and

$$\forall x_1 \ldots \forall x_n \, [\mathbf{A} \Leftrightarrow \mathbf{B}], \tag{2.2}$$

then **D** may be added to the end of the branch.

Theorem 2.13.6 (Substitution Theorem) *Suppose* **C** *is a sentence with parameters from* $\mathcal{K}$, **A** *and* **B** *are wffs with at most the free variables* $x_1, \ldots, x_n$, **A** *is a well-formed part of* **C**, *and* **D** *is the wff obtained from* **C** *by replacing the string* **A** *by the string* **B**. *Then* **D** *is tableau provable from* **C** *and* $\forall x_1 \ldots \forall x_n \, [\mathbf{A} \Leftrightarrow \mathbf{B}]$.

The proof is by induction on the length of the wff **C**, and is left as Exercise 53.

The Substitution Rule is obtained from the Substitution Theorem and the Learning Rule as follows. Suppose that **C** and $\forall x_1 \ldots \forall x_n \, [\mathbf{A} \Leftrightarrow \mathbf{B}]$ occur on Γ. By the Substitution Theorem, **D** is tableau provable from Γ, so by the Learning Rule, **D** may be added to the end of the branch. Thus the Substitution Rule is sound.

The Substitution Rule is frequently used in the following way. Suppose a new predicate r is introduced by the definition

$$\forall x_1 \ldots \forall x_n \, [r(x_1, \ldots, x_n) \Leftrightarrow \mathbf{A}]$$

using the Definition Rule. Then the Substitution Rule may be used to replace a well-formed part **A** within a wff **C** by the new predicate $r(x_1, \ldots, x_n)$. It may also be used to "unravel" the definition by replacing a well-formed part $r(x_1, \ldots, x_n)$ within a wff **C** by the old wff **A**.

Recall that a set of rules is said to be sound if any wff which is proved from a hypothesis set **H** using the rules is a semantic consequence of **H**. In this section we have introduced several extra rules which are commonly used in real mathematical proofs. We showed that each of these extra rules is sound by proving that any tableau proof using an extra rule can be replaced by an ordinary tableau proof. What we really need in order to use these extra rules in mathematical proofs is one grand soundness theorem which says that the set of all the extra rules together, plus the original tableau rules, is sound.

Theorem 2.13.7 *The set of rules consisting of the original tableau extension rules and the Direct Proof, Learning, Deduction, Generalization, Definition, and Substitution Rules is sound.*

Proof: We prove that Lemma 2.7.2 is true for tableau proofs and direct tableau proofs which use the extra rules. That is, for every tableau $\mathbf{T}$ for $\mathbf{H}$ which uses the extra rules and any model $\mathcal{M}$ of $\mathbf{H}$, there is a branch Γ of $\mathbf{T}$ and a valuation v such that $\mathcal{M} \models \Gamma(v)$. Like the ordinary Soundness Theorem, the proof is by induction on the number of nodes in the tableau. The induction has one new case for each of the extra rules for extending the tableau. In each case, we need only repeat the argument used to show that the extra rule by itself is sound. The soundness of the set of all our extra rules now follows as before. **End of Proof.**

As we mentioned before, mathematical proofs are usually written in paragraph form rather than in tree form. When the proof is translated into tree form, a list of "cases" will translate into a node with two children, as in the $\vee$ rule and similar tableau rules. A temporary assumption in a mathematical proof will often begin an application of the Deduction Rule, and a phrase such as "consider an arbitrary c" will begin an application of the Generalization Rule.

Very simple steps in proofs are often omitted or grouped together. For example, if a hypothesis has the form $\forall x \forall y \forall z\, \mathbf{C}$, one usually substitutes for the variables x, y and z all at once rather than using the $\forall$ tableau rule three times.

Because of the extra rules, a real mathematical proof translated into tree form will usually be shorter and have fewer negations and branches than the corresponding full tableau proof.

Example 2.13.8 We conclude this section with an example of a mathematical proof in paragraph form which we shall analyze as a tableau proof with extra rules.

Hypotheses:

(1) $p \Rightarrow q \vee r$
(2) $q \Rightarrow \forall x\, s(x)$
(3) $r \Rightarrow \forall y\, t(y)$
(4) $\forall y\, [t(y) \Rightarrow s(y)]$

To Prove:

$p \Rightarrow \forall x\, s(x)$

Proof in paragraph form: Temporarily assume p. By (1), $q \vee r$.

Case 1: q. By (2), $\forall x\, s(x)$.

Case 2: r. By (3), $\forall y\, t(y)$. Consider an arbitrary a. Then $t(a)$. By (4), $t(a) \Rightarrow s(a)$. Therefore $s(a)$. Since a was arbitrary, $\forall x\, s(x)$.

Since $\forall x\, s(x)$ in all cases, $p \Rightarrow \forall x\, s(x)$ as required. **End of Proof.**

The above proof can be translated into a direct proof which uses the tableau rules together with the Deduction and Generalization Rules. The figure on page 114 shows the proof in tree form, skipping the simpler steps which use the ordinary tableau rules. The large window contains a direct proof of $\forall x\, s(x)$ from the original hypotheses and the temporary hypothesis p, and is used for the Deduction Rule. The small window contains a direct proof of $s(a)$, where a is new, from the wffs on the branch above the window, and is used for the Generalization Rule.

2.14 PREDCALC Problems (PRED2)

This set of problems uses the PREDCALC or PREDWIN program. Its purpose is to make the student more familiar with the behavior of truth values of wffs of predicate logic in a model. There are twelve problems. They are located in the directory PRED2 on the distribution diskette, and the SETUPDOS or SETUPWIN program will put them in a subdirectory called PRED2 on your hard disk. In each problem, a goal graph will appear on the screen and your task is to use the "calculator" keys to get an exact copy of the goal in position one of the stack. If the letters NC appear after the label GOAL, your answer must use none of the parameter (or constant) symbols 0,1,... in order to get full credit. (If you have a text only monitor, you will have to use the View command to see the goal graph.)

Suggestions: Think of a wff which has the required graph and write it down, then make a parsing sequence for the wff and build it up step by step. You always start out with atomic wffs involving $=, <, >, +, -,$ or $*$. To see the graph of the goal wff in detail, hit V for View. If you want to keep part of what you did and change the rest, hit R to Replay, go as far as you want by pressing the Enter key, then hit K to

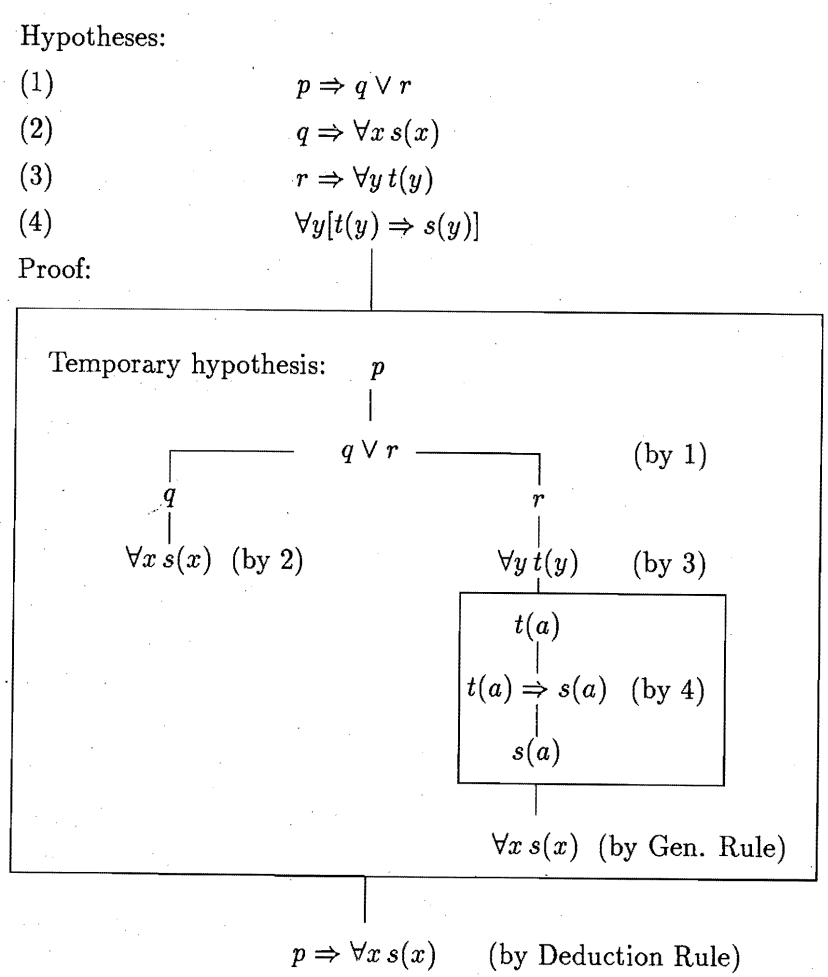


Figure 2.4: Example 2.13.8 in Tree Form.

Kill the remaining steps and make your changes. As in the previous problem set, you should give your solution the name of the problem preceded by the letter A. The approximate number of steps needed for a solution and other comments are given below. You do not have to find a solution with exactly the suggested number of steps. However, if you are using many more steps than suggested you are probably on the wrong track.

POINT. 5 steps. (A graph consisting of one point in the cube).

PLANES. 5 steps. (Three perpendicular planes).

SQRPLUS. 4 steps. (The graph of the equation $z = y^2 + 1$).

TOUGH. 3 steps. (A graph which has something to do with divisibility by 3).

DIAG. 3 steps. (The diagonal of the cube from lower front left to upper back right). No parameters allowed.

TWOLESS. 4 steps. (x is at least 2 less than z). No constants allowed.

FOUR. 5 steps. (z is divisible by 4). No parameters allowed.

XXX. 5 steps. (A stack of eight X's formed by two vertical planes). No parameters allowed.

TRUESUM. 6 steps. ($z = x + y$ in the usual way instead of modulo 8). No parameters allowed.

ALLEVEN. 8 steps. (All three variables are even). No parameters allowed.

SOMEVEN. 8 steps. (At least one variable is even). No parameters allowed.

SQRSUM. 9 steps. (z is the sum of two squares). No parameters allowed.

Here are some optional projects using the PREDCALC program.

1. The sentence

$$\forall z \forall x\, [0 < x \Rightarrow \exists y\, z = x * y]$$

is true for some universes of size between 1 and 8 but false for others. Find out when it is true and when it is false.

2. The $R(\ldots)$ key in the upper left corner of the PREDCALC keypad can be used to add extra predicate symbols with one, two, or three places to the vocabulary. The computer will randomly choose models for these predicates. Use this key to experiment with graphs of a wff in randomly chosen models. By using the "Replay" command, you can repeat a session with the same wffs but different randomly chosen models.

3. The . $= h(..)$ key also adds extra predicate symbols with two or three places to the vocabulary. The computer will randomly choose models in which the first variable is a function of the other one or two variables.

4. Find a single sentence **A** which uses only the variables x and y, $+$, connectives, and quantifiers, such that **A** is true in each of the PREDCALC models of even universe size $2, 4, 6, 8$ and false in each of the models of odd universe size $1, 3, 5, 7$.

5. Find eight different sentences $\mathbf{A}_1, \ldots, \mathbf{A}_8$ which use only the variables x and y, the predicate symbol $<$, connectives and quantifiers, such that for each n, $\mathbf{A}_n$ is true in the PREDCALC model with universe size n, but is false in every other universe size.

2.15 Tableau Problems (TAB3)

This assignment uses the TABLEAU or TABWIN program. In this assignment you will construct tableau proofs in predicate logic. The problems are located in directory TAB3 on the distribution diskette, and the SETUPDOS or SETUPWIN program will put them in a subdirectory called TAB3 on your hard disk. There are three groups of problems in this directory:

1. SHORT1, SHORT2, ..., SHORT8,

2. SET1, SET2, ..., SET6,

3. ORDER1, ORDER2, ..., ORDER6.

Use the TABLEAU or TABWIN program commands to load the problem, do your work, and then save your answer on your diskette or hard drive. The file name of your answer should be the letter A followed by the name of the problem.

As in the propositional problems, each problem is assigned a suggested number of nodes: its *par value*. The par value is given only as a guide; you are not expected to attain it exactly. You should try problems with smaller par value first.

The first group of problems, called

SHORT1.TBU through SHORT8.TBU,

develop some of the basic properties of quantifiers. You should do these problems first by hand on a piece of paper, and then do them on the computer to check your work. This will help you discover any misunderstandings you may have.

SHORT1 (3 nodes)

Hypothesis: $\exists x\ p(x,x)$

To prove: $\exists x\ \exists y\ p(x,y)$

SHORT2 (3 nodes)

Hypothesis: $\exists y\ p(y)$

To prove: $\exists y\ \forall x\ p(y)$

SHORT3 (4 nodes)

Hypothesis: $\forall y\ \forall x\ p(x,y)$

To prove: $\forall x\ \forall y\ p(x,y)$

SHORT4 (6 nodes)

Hypothesis: $p \wedge \exists x\ q(x)$

To prove: $\exists x\ [p \wedge q(x)]$

SHORT5 (7 nodes)

Hypothesis: $\exists x\ [p(x) \wedge q(x)]$

To prove: $\exists x\ p(x) \wedge \exists x\ q(x)$

SHORT6 (11 nodes)

Hypothesis: None

To prove: $\exists x\ p(x) \Leftrightarrow \neg\forall x\ \neg p(x)$

SHORT7 (9 nodes)

Hypothesis: $\forall x\ \exists y\ F(x,y)$

To prove: $\forall x\ \exists y\ \exists z\ [F(x,y) \wedge F(y,z)]$

SHORT8 (23 nodes)

Hypothesis: None

To prove: $\forall x\ p(x) \wedge \forall x\ q(x) \Leftrightarrow \forall x\ [p(x) \wedge q(x)]$

The remaining problems are more difficult, and you need an overall picture of your proof so that you will be able to choose useful substitutions for the quantifiers. Before doing the formal proof on the computer, you should make a sketch of the main steps of the proof with pencil and paper.

The next group of problems called

SET1.TBU through SET6.TBU

are about sets.

SET1 (10 nodes).

Hypothesis:

$$\forall x\ \forall y\ [\text{subset}(x,y) \Leftrightarrow \forall z\ [\text{in}(z,x) \Rightarrow \text{in}(z,y)]]$$

To prove:

$$\forall x\ \text{subset}(x,x)$$

The predicate $\text{in}(x,y)$ means that x is an element of y, and the predicate $\text{subset}(x,y)$ means that x is a subset of y. The hypothesis defines $\text{subset}(x,y)$ in terms of $\text{in}(x,y)$. The conclusion states that every set is a subset of itself.

SET2 (14 nodes)

Hypotheses:

$$\forall x\ \forall y\ [\text{subset}(x,y) \Leftrightarrow \forall z\ [\text{in}(z,x) \Rightarrow \text{in}(z,y)]],$$

$$\forall x\ [\text{empty}(x) \Leftrightarrow \neg\ \exists y\ \text{in}(y,x)].$$

To prove:

$$\forall x\ [\text{empty}(x) \Rightarrow \forall y\ \text{subset}(x,y)]$$

The predicate $\text{empty}(x)$ means that x is the empty set. The first hypothesis is the same as before. The second hypothesis defines $\text{empty}(x)$ in terms of $\text{in}(x,y)$. The conclusion states that the empty set is a subset of every set.

SET3 (28 nodes)

Hypotheses:

$$\forall x\ \forall y\ [\text{subset}(x,y) \Leftrightarrow \forall z\ [\text{in}(z,x) \Rightarrow \text{in}(z,y)]],$$

$$\forall x\ \forall y\ \forall z\ [\text{union}(x,y,z) \Leftrightarrow \forall t\ [\text{in}(t,z) \Leftrightarrow \text{in}(t,x) \vee \text{in}(t,y)]].$$

To prove:

$$\forall x\ \forall y\ \forall z\ [\text{union}(x,y,z) \Rightarrow \text{subset}(x,z)]$$

The predicate $\text{union}(x,y,z)$ means that z is the union of x and y. The hypotheses define $\text{subset}(x,y)$ and $\text{union}(x,y)$ in terms of $\text{in}(x,y)$. The conclusion states that x is a subset of the union of x and y.

SET4 (33 nodes)

Hypothesis:

$$\forall x\ \forall y\ [\text{subset}(x,y) \Leftrightarrow \forall z\ [\text{in}(z,x) \Rightarrow \text{in}(z,y)]]$$

To prove:

$$\forall x\ \forall y\ \forall z\ [\text{subset}(x,y) \wedge \text{subset}(y,z) \Rightarrow \text{subset}(x,z)]$$

The hypothesis is again the definition of subset(x,y) in terms of in(x,y). The conclusion is the transitivity law for subsets, that if x is a subset of y and y is a subset of z, then x is a subset of z.

SET5 (42 nodes)

Hypotheses:

$$\forall x\ \forall y\ [\text{subset}(x,y) \Leftrightarrow \forall z\ [\text{in}(z,x) \Rightarrow \text{in}(z,y)]],$$

$$\forall x\ \forall y\ [\text{eq}(x,y) \Leftrightarrow \forall z\ [\text{in}(x,z) \Rightarrow \text{in}(y,z)]],$$

$$\forall x\ [\text{single}(x) \Leftrightarrow \exists y\ \text{in}(y,x) \wedge \forall y\ \forall z\ [\text{in}(y,x) \wedge \text{in}(z,x) \Rightarrow \text{eq}(y,z)]].$$

To prove:

$$\forall x\ [\text{single}(x) \Rightarrow \forall y\ [\exists z\ [\text{in}(z,x) \wedge \text{in}(z,y)] \Rightarrow \text{subset}(x,y)]]$$

The predicate eq(x,y) means that x and y are elements of the same sets. The predicate single(x) means that x has exactly one element. The hypotheses define the predicates subset(x,y), eq(x,y), and single(x). The conclusion states that if single(x) and y contains some element of x, then x is a subset of y.

SET6 (51 nodes)

Hypotheses:

$$\forall x\ \forall y\ [\text{subset}(x,y) \Leftrightarrow \forall z\ [\text{in}(z,x) \Rightarrow \text{in}(z,y)]],$$

$$\forall x\ \forall y\ \forall z\ [\text{union}(x,y,z) \Leftrightarrow \forall t\ [\text{in}(t,z) \Leftrightarrow \text{in}(t,x) \vee \text{in}(t,y)]].$$

To prove:

$$\forall x\ \forall y\ \forall z\ [\text{union}(x,y,z) \Rightarrow \forall u\ [\text{subset}(x,u) \wedge \text{subset}(y,u) \Rightarrow \text{subset}($$

The hypotheses define the predicates subset(x,y) and union(x,y,z). The conclusion states that if both x and y are subsets of u, then the union of x and y is a subset of u.

The next group of problems called

ORDER1.TBU through ORDER6.TBU

concern partial orders.

ORDER1 (19 nodes)

Hypotheses:

$$\forall x\ x \leq x,$$

$$\forall x\ \forall y\ \forall z\ [x \leq y \wedge y \leq z \Rightarrow x \leq z].$$

To prove:

$$\forall w\ \forall x\ \forall y\ \forall z\ \ [[w \leq x \wedge x \leq y] \wedge y \leq z \Rightarrow w \leq z]$$

The hypotheses state that $\leq$ is a partial ordering. The conclusion states that if $w \leq x \leq y \leq z$, then $w \leq z$.

ORDER2 (21 nodes)

Hypotheses:

$$\forall x\ x \leq x,$$

$$\forall x\ \forall y\ \forall z\ [x \leq y \wedge y \leq z \Rightarrow x \leq z],$$

$$\forall x \forall y \forall z[\text{glb}(x,y,z) \Leftrightarrow [z \leq x \wedge z \leq y] \wedge \forall t[t \leq x \wedge t \leq y \Rightarrow t \leq z]].$$

To prove:

$$\forall x\ \forall y\ [x \leq y \Rightarrow \text{glb}(x,y,x)]$$

The predicate $\text{glb}(x,y,z)$ means that z is the greatest lower bound of x and y in the partial ordering $\leq$, that is, z is the greatest element which is $\leq$ both x and y. The conclusion states that if $x \leq y$, then x is the greatest lower bound of x and y.

ORDER3 (27 nodes)

Hypotheses:

$$\forall x\ x \leq x,$$

$$\forall x\ \forall y\ \forall z\ [x \leq y \wedge y \leq z \Rightarrow x \leq z],$$

$$\forall x \forall y \forall z[\text{glb}(x,y,z) \Leftrightarrow [z \leq x \wedge z \leq y] \wedge \forall t[t \leq x \wedge t \leq y \Rightarrow t \leq z]].$$

To prove:

$$\forall x\ \forall y\ \forall z\ \forall t\ [\text{glb}(x,y,z) \wedge \text{glb}(x,y,t) \Rightarrow z \leq t]$$

The hypotheses are the same as for ORDER2. The conclusion states that if z and t are both greatest lower bounds of x and y, then $z \leq t$. (Since the same reasoning gives $t \leq z$, this shows that any two greatest lower bounds of x and y are equal).

ORDER4 (32 nodes)

Hypotheses:

$$\forall x\ x \leq x,$$

$$\forall x\ \forall y\ \forall z\ [x \leq y \wedge y \leq z \Rightarrow x \leq z],$$

$$\forall x\ \forall y\ \exists t\ [x \leq t \wedge y \leq t].$$

To prove:

$$\forall x\ \forall y\ \forall z\ \exists t\ [[x \leq t \wedge y \leq t] \wedge z \leq t]$$

The hypotheses state that $\leq$ is a partial ordering, and that for any two elements x, y there is an element t such that $x \leq t$ and $y \leq t$. The conclusion states that for any three elements x, y, z there is an element t such that $x \leq t$, $y \leq t$, and $z \leq t$.

ORDER5 (36 nodes)

Hypotheses:

$$\forall x\ x \leq x,$$

$$\forall x\ \forall y\ \forall z\ [x \leq y \wedge y \leq z \Rightarrow x \leq z],$$

$$\forall x\ \forall y\ [x < y \Leftrightarrow x \leq y \wedge \neg y \leq x].$$

To prove:

$$\forall x\ \forall y\ \forall z\ [x < y \wedge y \leq z \Rightarrow x < z]$$

The predicate $x < y$ means that $x \leq y$ but not $y \leq x$. The hypotheses state that $\leq$ is a partial ordering and define the predicate $x < y$ in terms of $x \leq y$. The conclusion states that if $x < y \leq z$ then $x < z$.

ORDER6 (104 nodes)

Hypotheses:

$$\forall x\ x \leq x,$$

$$\forall x\ \forall y\ \forall z\ [x \leq y \wedge y \leq z \Rightarrow x \leq z],$$

$$\forall x \forall y \forall z[\text{glb}(x,y,z) \Leftrightarrow [z \leq x \wedge z \leq y] \wedge \forall t[t \leq x \wedge t \leq y \Rightarrow t \leq z]],$$

$$\forall x\ \forall y\ [\text{eq}(x,y) \Leftrightarrow x \leq y \wedge y \leq x].$$

To prove:

$$[[\text{glb}(a,b,c) \wedge \text{glb}(b,c,e)] \wedge \text{glb}(d,c,f)] \wedge \text{glb}(a,e,g) \Rightarrow \text{eq}(f,g)$$

The predicate $\text{eq}(x,y)$ means that $x \leq y$ and $y \leq x$. The hypotheses state that $\leq$ is a partial ordering and define the predicates $\text{glb}(x,y,z)$ and $\text{eq}(x,y)$ in terms of $x \leq y$. The conclusion is an associative law for greatest lower bounds. If we write $x \sqcup y$ for the greatest lower bound of x and y, and $x = y$ for $\text{eq}(x,y)$, the conclusion states that

$$(a \sqcup b) \sqcup c = a \sqcup (b \sqcup c).$$

2.16 Exercises

1. The string

$$\exists x\,[\forall y\; p(x,y) \Rightarrow \neg q(x) \vee r(y)]$$

is an abbreviation for a wff in predicate logic.

(a) Change the string into the wff which it abbreviates by inserting brackets in the correct places.

(b) Write down a parsing sequence for the wff.

(c) For each wff of your parsing sequence, circle every occurrence of a variable which is bound in that wff.

2. Give an example of a wff **A** in predicate logic with variables x, y, and z which satisfies each of the following four conditions at the same time:

- x is free for y but not for z,
- y is free for z but not for x,
- z is not free for x, and
- z is not free for y.

3. Prove that for each wff **A** in pure predicate logic, if **B** and **C** are well formed parts of **A** and the first symbol of **C** is within **B**, then the last symbol of **C** is within **B**.

Hint: Use induction on the length of **B** and Lemma 2.2.3.

4. Which of the following sentences **A** are valid? For those which are, give a tableau proof. For those which are not, give a counter-model (i.e. a model $\mathcal{M}$ such that $\mathcal{M} \not\models \mathbf{A}$).

(1) $[p_{11} \vee p_{12}] \wedge [p_{21} \vee p_{22}] \Rightarrow [p_{11} \wedge p_{21}] \vee [p_{12} \wedge p_{22}]$

(2) $[p_{11} \wedge p_{21}] \vee [p_{12} \wedge p_{22}] \Rightarrow [p_{11} \vee p_{12}] \wedge [p_{21} \vee p_{22}]$

(3) $\forall x \exists y\, p(x,y) \Rightarrow \exists y \forall x\, p(x,y)$

(4) $\exists y \forall x\, p(x,y) \Rightarrow \forall x \exists y\, p(x,y)$

5. In the following, $\mathbf{N}$ denotes the set of **natural numbers**,

$$\mathbf{N} = \{0,1,2,3,\ldots\}$$

Let $\mathcal{N}$ be the model with universe $\mathbf{N}$ in which the predicate symbols $\doteq$, $\leq$, and $<$ and the expression $x + y = z$ all have their usual meanings. Which of the following are true in $\mathbf{N}$?

(1.a) $\forall x \forall y \forall z [x + y = z \Rightarrow y + x = z]$.

(1.b) $\forall x \exists y\, x + y = x$.

(2) $\exists y \forall x\, x + y = x$.

(3.a) $\forall x \forall z \exists y\, x + y = z$.

(3.b) $\forall x \forall z\, [x \leq z \Rightarrow \exists y\, x + y = z]$.

(4.a) $\forall x \exists y\, x < y$.

(4.b) $\forall x \exists y\, x \leq y$.

(5.a) $\forall x \exists y\, y < x$.

(5.b) $\forall x \exists y\, y \leq x$.

(6.a) $\exists y \forall x\, y < x$.

(6.b) $\exists y \forall x\, y \leq x$.

(7) $\forall x [\forall y\, x \leq y \Rightarrow x \doteq 0]$.

(8) $\forall x \forall y [[x \leq y \wedge y \leq x] \Rightarrow x \doteq y]$.

(9) $\forall x \forall y [[x < y \wedge y < x] \Rightarrow \neg x \doteq y]$.

(10) $\forall x \exists y\, x < y \Rightarrow \exists y\, 3 < y$.

(11) $\forall x \exists y\, x < y \Rightarrow \exists y\, y < y$.

6. Let $\mathbf{Z}$ denote the set of **integers**:

$$\mathbf{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\},$$

and let $\mathcal{Z}$ be the model with universe $\mathbf{Z}$ and the usual meaning for the predicate symbols. Which of the sentences of Exercise 5 remain true when $\mathcal{N}$ is replaced by $\mathcal{Z}$?

7. In this problem you are to find a model $\mathcal{M}$ for predicate logic with one binary predicate symbol p. The universe of $\mathcal{M}$ is the set $\{0, 1, 2\}$ and the relation $p^{\mathcal{M}}$ is a subset of the set of pairs (i, j) with i and j from $\{0, 1, 2\}$. Your answer will be counted as correct if and only if the wff

$$\forall x \exists y\, p(x, y) \wedge \exists x \forall y\, p(x, y) \wedge \neg \exists y \forall x\, p(x, y)$$

is true in your model $\mathcal{M}$. You may specify your model by drawing a 3 by 3 matrix of truth values to indicate the graph of $p_{\mathcal{M}}$.

8. For each positive integer n construct a model $\mathcal{M} = (M, p^{\mathcal{M}})$ as follows:

$$M = \{1, 2, 3, \ldots, n-1\},$$
$$p^{\mathcal{M}} = \{(i, j) \in M \times M : ij \equiv 1 \pmod{n+1}\}.$$

(Note: $x \equiv y \pmod{k}$ iff $x - y$ is divisible by k.)

Show that $\mathcal{M} \models \forall x \exists y\, p(x, y)$ when $n = 6$ but not when $n = 5$.

9. In pure predicate logic, let x and y be variables and let $\mathbf{A}(x/y)$ (with only one slash) be the wff formed by replacing each **bound** occurrence of x in $\mathbf{A}$ by y, leaving the free occurrences alone. For example, $(\forall z\, \exists x\, p(x, z))(x/y)$ is $\forall z\, \exists y\, p(y, z)$. Prove by induction on wffs that if $\mathbf{A}$ is a wff and y does not occur in $\mathbf{A}$, then the wff $[\mathbf{A} \Leftrightarrow \mathbf{A}(x/y)]$ is valid. Hint: Let $R(n)$ be the following property: For every wff $\mathbf{C}$ of length $\leq n$, every model $\mathcal{M}$, and every instance $\mathbf{A}$ of $\mathbf{C}$ in M, $\mathcal{M} \models \mathbf{A}(x/y)$ whenever y does not occur in $\mathbf{A}$.

10. Give a tableau proof of each of the following:

(1) $\neg \forall x\, p(x) \Leftrightarrow \exists x\, \neg p(x)$

(2) $\neg \exists x\, p(x) \Leftrightarrow \forall x\, \neg p(x)$

(3) $\forall x\, p(x) \Leftrightarrow \neg \exists x\, \neg p(x)$

(4) $\exists x\, p(x) \Leftrightarrow \neg \forall x\, \neg p(x)$

(5) $\forall x\, p(x) \Leftrightarrow \forall y\, p(y)$

(6) $\exists x\, p(x) \Leftrightarrow \exists y\, p(y)$

(7) $\forall x[p(x) \wedge q(x)] \Leftrightarrow [\forall x\, p(x) \wedge \forall x\, q(x)]$

(8) $\exists x[p(x) \vee q(x)] \Leftrightarrow [\exists x\, p(x) \vee \exists x\, q(x)]$

11. In this exercise $[p \equiv q]$ is to be understood as an abbreviation for the sentence $\forall x[p(x) \Leftrightarrow q(x)]$. Give a tableau proof of each of the following:

(1) $[p \equiv q] \Rightarrow [\forall x\, p(x) \Leftrightarrow \forall x\, q(x)]$

(2) $[p \equiv q] \Rightarrow [\exists x\, p(x) \Leftrightarrow \exists x\, q(x)]$

(3) $[p \equiv q] \Rightarrow \forall x[\neg p(x) \Leftrightarrow \neg q(x)]$

(4) $[p_1 \equiv q_1] \wedge [p_2 \equiv q_2] \Rightarrow \forall x[[p_1(x) \wedge p_2(x)] \Leftrightarrow [q_1(x) \wedge q_2(x)]]$

(5) $[p_1 \equiv q_1] \wedge [p_2 \equiv q_2] \Rightarrow \forall x[[p_1(x) \vee p_2(x)] \Leftrightarrow [q_1(x) \vee q_2(x)]]$

(6) $[p_1 \equiv q_1] \wedge [p_2 \equiv q_2] \Rightarrow \forall x[[p_1(x) \Rightarrow p_2(x)] \Leftrightarrow [q_1(x) \Rightarrow q_2(x)]]$

(7) $[p_1 \equiv q_1] \wedge [p_2 \equiv q_2] \Rightarrow \forall x[[p_1(x) \Leftrightarrow p_2(x)] \Leftrightarrow [q_1(x) \Leftrightarrow q_2(x)]]$

12. Which of the following sentences $\mathbf{A}$ are valid? For those which are, give a tableau proof. For those which are not, give a counter-model. You may specify your counter-model $\mathcal{M}$ by writing down the universe set $\mathcal{M}$ and one or two subsets $p^{\mathcal{M}}$ and $q^{\mathcal{M}}$.

(1.a) $\exists x[p(x) \wedge q(x)] \Rightarrow [\exists x\, p(x) \wedge \exists x\, q(x)]$

(1.b) $[\exists x\, p(x) \wedge \exists x\, q(x)] \Rightarrow \exists x[p(x) \wedge q(x)]$

(2.a) $\forall x[p(x) \vee q(x)] \Rightarrow [\forall x\, p(x) \vee \forall x\, q(x)]$

(2.b) $[\forall x\, p(x) \vee \forall x\, q(x)] \Rightarrow \forall x[p(x) \vee q(x)]$

(3.a) $\forall x[p(x) \Rightarrow q(x)] \Rightarrow [\forall x\, p(x) \Rightarrow \forall x\, q(x)]$

(3.b) $[\forall x\, p(x) \Rightarrow \forall x\, q(x)] \Rightarrow \forall x[p(x) \Rightarrow q(x)]$

(4.a) $\exists x[p(x) \Rightarrow q(x)] \Rightarrow [\exists x\, p(x) \Rightarrow \exists x\, q(x)]$

(4.b) $[\exists x\, p(x) \Rightarrow \exists x\, q(x)] \Rightarrow \exists x[p(x) \Rightarrow q(x)]$

(5.a) $\forall x[p(x) \Leftrightarrow q(x)] \Rightarrow [\forall x\, p(x) \Leftrightarrow \forall x\, q(x)]$

(5.b) $[\forall x\, p(x) \Leftrightarrow \forall x\, q(x)] \Rightarrow \forall x[p(x) \Leftrightarrow q(x)]$

(6.a) $\exists x[p(x) \Leftrightarrow q(x)] \Rightarrow [\exists x\, p(x) \Leftrightarrow \exists x\, q(x)]$

(6.b) $[\exists x\, p(x) \Leftrightarrow \exists x\, q(x)] \Rightarrow \exists x[p(x) \Leftrightarrow q(x)]$

(7.a) $\forall x\, \neg p(x) \Rightarrow \neg \forall x\, p(x)$

(7.b) $\neg \forall x\, p(x) \Rightarrow \forall x\, \neg p(x)$

(8.a) $\exists x\, \neg p(x) \Rightarrow \neg \exists x\, p(x)$

(8.b) $\neg \exists x\, p(x) \Rightarrow \exists x\, \neg p(x)$

13. Give a tableau proof of each of the following:

(1) $\forall x\, p \Leftrightarrow p$

(2) $\exists x\, p \Leftrightarrow p$

(3) $\forall x[p \wedge q(x)] \Leftrightarrow [p \wedge \forall x\, q(x)]$

(4) $\exists x[p \wedge q(x)] \Leftrightarrow [p \wedge \exists x\, q(x)]$

(5) $\forall x[p \vee q(x)] \Leftrightarrow [p \vee \forall x\, q(x)]$

(6) $\exists x[p \vee q(x)] \Leftrightarrow [p \vee \exists x\, q(x)]$

(7) $\forall x[p \Rightarrow q(x)] \Leftrightarrow [p \Rightarrow \forall x\, q(x)]$

(8) $\exists x[p \Rightarrow q(x)] \Leftrightarrow [p \Rightarrow \exists x\, q(x)]$

(9) $\exists x[q(x) \Rightarrow p] \Leftrightarrow [\forall x\, q(x) \Rightarrow p]$

(10) $\forall x[q(x) \Rightarrow p] \Leftrightarrow [\exists x\, q(x) \Rightarrow p]$

14. For each pair of wffs (a,b) below, give a tableau proof of one of the wffs and a countermodel of the other.

(1.a) $\forall x[p \Rightarrow q(x)] \Rightarrow [p \Rightarrow \exists x\, q(x)]$

(1.b) $[p \Rightarrow \exists x\, q(x)] \Rightarrow \forall x[p \Rightarrow q(x)]$

(2.a) $\exists x[p \Rightarrow q(x)] \Rightarrow [p \Rightarrow \forall x\, q(x)]$

(2.b) $[p \Rightarrow \forall x\, q(x)] \Rightarrow \exists x[p \Rightarrow q(x)]$

(3.a) $\exists x[q(x) \Rightarrow p] \Rightarrow [\exists x\, q(x) \Rightarrow p]$

(3.b) $[\exists x\, q(x) \Rightarrow p] \Rightarrow \exists x[q(x) \Rightarrow p]$

(4.a) $\forall x[q(x) \Rightarrow p] \Rightarrow [\forall x\, q(x) \Rightarrow p]$

(4.b) $[\forall x\, q(x) \Rightarrow p] \Rightarrow \forall x[q(x) \Rightarrow p]$

(5.a) $\forall x[q(x) \Leftrightarrow p] \Rightarrow [\forall x\, q(x) \Leftrightarrow p]$

(5.b) $[\forall x\, q(x) \Leftrightarrow p] \Rightarrow \forall x[q(x) \Leftrightarrow p]$

(6.a) $\exists x[q(x) \Leftrightarrow p] \Rightarrow [\exists x\, q(x) \Leftrightarrow p]$

(6.b) $[\exists x\, q(x) \Leftrightarrow p] \Rightarrow \exists x[q(x) \Leftrightarrow p]$

15. Let Γ be the set consisting of the two wffs $x \doteq y$ and $\neg x \doteq y$. Construct a model $\mathcal{M}$ such that each of these two wffs is satisfiable in $\mathcal{M}$ but the set Γ is not simultaneously satisfiable in $\mathcal{M}$.

16. Find a finished tableau with the hypothesis set

$$\forall x\, p(x,x), \qquad \exists x \forall y\, p(x,y), \qquad \exists x \forall y\, p(y,x),$$

$$\exists x \exists y\ [\neg p(x,y) \wedge \neg p(y,x)]$$

and the set of parameters $M = \{a, b, c, d\}$.

17. This exercise gives a formal proof of Problem 24 from Chapter 1. Consider the following four statements.

(1) There exists a tableau proof of **A** from **D**.

(2) There exists a tableau proof of **B** from **A**.

(3) For all **A** and all **B**, there exists a tableau proof of **B** from **A** if and only if for all $\mathcal{M}$, if $\mathcal{M}$ models **A** then $\mathcal{M}$ models **B**.

(4) There exists a tableau proof of **B** from **D**.

Statements (1)–(3) are the hypotheses, and statement (4) is the formula to be proved. Statement (3) combines the Soundness and Completeness Theorems for propositional logic.

Consider the following vocabulary for pure predicate logic.

$$\mathcal{P}_2 = \{MO\}, \qquad \mathcal{P}_3 = \{TP\}.$$

Let $MO(x,y)$ be interpreted as "x models y," and $TP(x,y,z)$ as "x is a tableau proof of z from y." Let the individual parameters a, b, and d be interpreted as the wffs **A**, **B**, and **D**.

Write out the above hypothesis set and formula to prove as sentences of pure predicate logic with individual parameters a, b, and d. Then give a tableau proof.

18. Give a formal proof of Problem 25 from Chapter 1 analogous to the preceding exercise. In addition to the predicate symbols MO, TP,

another ternary predicate symbol OR is needed, where $OR(x,y,z)$ is interpreted as "$z = [x \vee y]$." One of the hypotheses should correspond to the statement:

(0) For all **A**, **B**, **C** and $\mathcal{M}$, if **C** = **A** $\vee$ **B** then $\mathcal{M}$ models **C** if and only if $\mathcal{M}$ models **A** or $\mathcal{M}$ models **B**.

19. This exercise gives a formal proof of the Main Lemma for the Completeness Theorem for propositional logic. Here is a list of five statements from Chapter 1.

(1) For all **H** and for all **T**, **T** is a finished tableau with hypothesis set **H** if and only if **T** is a tableau with hypothesis set **H** and for every Γ, if Γ is a branch of **T** then either Γ is finished or Γ is contradictory and finite.

(2) For all **H** and for all **T**, **T** is a confutation of **H** if and only if **T** is a tableau with hypothesis set **H** and for every Γ, if Γ is a branch of **T** then Γ is contradictory and finite.

(3) For all **H** ,**T** and Γ, if **T** is a tableau with hypothesis set **H** and Γ is a branch of **T** and Γ is finished, then there exists $\mathcal{M}$ such that $\mathcal{M}$ models **H**.

(4) For every **H** there exists **T** such that **T** is a finished tableau with hypothesis set **H**.

(5) For all **H**, either there exists $\mathcal{M}$ such that $\mathcal{M}$ models **H**, or there exists **T** such that **T** is a confutation of **H**.

The hypotheses (1-4) are versions of the definitions of a finished tableau and a confutation, and of the Finished Set and Tableau Extension lemmas. Statement (5) is the formula to be proved, the Main Lemma for the Completeness Theorem.

Consider the following vocabulary for pure predicate logic:

$$\mathcal{P}_1 = \{FB, CB\}, \qquad \mathcal{P}_2 = \{F, B, M, T, C\}$$

Let $FB(x)$ be interpreted as "x is a finished branch", and $CB(x)$ as "x is a contradictory finite branch." Let $F(x,y)$ be interpreted as "x is a finished tableau with hypothesis set y," $B(x,y)$ as "x a branch of y," $M(x,y)$ as "x is a model of hypothesis set y," $T(x,y)$ as "x is a tableau with the the hypothesis set y," and $C(x,y)$ as "x is a confutation of hypothesis set y."

Write out the above hypothesis set and sentence to prove in pure predicate logic with this vocabulary. Give a tableau proof.

20. Suppose that $\mathbf{T}$ is a finite tableau in predicate logic, that $\mathbf{H}$ is the set of hypotheses of $\mathbf{T}$, that $\mathbf{A}$ is a wff whose only free variable is x, that b is a variable which is free for x in $\mathbf{A}$, and that every branch of $\mathbf{T}$ is either contradictory or contains the wff $\mathbf{A}(x//b)$. Describe a simple way to change $\mathbf{T}$ into a tableau proof of $\exists x\ \mathbf{A}$ from $\mathbf{H}$.

21. Suppose that $\mathbf{H}$ is a finite set of sentences of pure predicate logic, that $\mathbf{H}$ has at least one model, and that $\mathbf{H}$ has a finished tableau with fewer than 100 nodes. Prove that $\mathbf{H}$ has a model whose universe has fewer than 100 elements.

The next four problems need assignments of infinite sets of individual symbols. By an assignment of a set S of individual symbols in M we mean a function v from S into M. Let $\mathbf{\Gamma}$ be a set of wffs and let S be the (possibly infinite) set of individual symbols which occur freely in $\mathbf{\Gamma}$. $\mathbf{\Gamma}(v)$ is the set of sentences with parameters from M obtained by replacing each free occurrence of an individual symbol x by $v(x)$. $\mathbf{\Gamma}$ is said to be simultaneously satisfiable in a model $\mathcal{M}$ if $\mathcal{M} \models \mathbf{\Gamma}(v)$ for some assignment v of S in M.

22. Prove the analogue of Lemma 2.7.2 for infinite tableaus: If $\mathbf{T}$ is an infinite tableau whose hypothesis set $\mathbf{H}$ is a set of sentences and $\mathcal{M} \models \mathbf{H}$, then some branch of $\mathbf{T}$ is simultaneously satisfiable in $\mathcal{M}$.

23. Let $\mathbf{T}$ be an infinite tableau whose hypothesis set $\mathbf{H}$ is a set of sentences with parameters from $\mathcal{K}$. Prove that if $\mathbf{H}$ is simultaneously satisfiable in a model $\mathcal{M}$, then some branch of $\mathbf{T}$ is simultaneously satisfiable in $\mathcal{M}$.

24. Prove the following Extended Soundness Theorem for sentences with parameters from $\mathcal{K}$: Let $\mathbf{H} \cup \{\mathbf{A}\}$ be a finite or countable set of sentences with parameters from $\mathcal{K}$. If $\mathbf{H} \vdash \mathbf{A}$ then $\mathbf{H} \models \mathbf{A}$, that is, for every model $\mathcal{M}$ and assignment v of $\mathcal{K}$ in $\mathcal{M}$, if $\mathcal{M} \models \mathbf{H}(v)$ then $\mathcal{M} \models \mathbf{A}(v)$.

25. Prove the following Extended Completeness Theorem for sentences with parameters from $\mathcal{K}$. Let $\mathbf{H} \cup \{\mathbf{A}\}$ be a finite or countable set of sentences with parameters from $\mathcal{K}$. If $\mathbf{H} \models \mathbf{A}$ then $\mathbf{H} \vdash \mathbf{A}$. Hint: To prove the Main Lemma for sentences with parameters from $\mathcal{K}$, introduce an infinite set of new parameter symbols M and use the set $\mathcal{K} \cup M$ as the universe of the model being constructed.

26. This exercise indicates why we need to assume that the universe set of any model of (pure or full) predicate logic is nonempty. Assume we are working in a logic which has at least one binary predicate symbol P. (We will see that, by assumption, every full predicate logic has such a symbol.)

(a) Show that each of the following sentences is valid by giving a tableau proof of each using an empty set of hypotheses:

$$\mathbf{A} \ : \ \forall x \forall y \ [P(x,y) \vee \neg P(x,y)]$$

$$\mathbf{B} \ : \ \mathbf{A} \Rightarrow [\exists x \exists y \ [P(x,y) \vee \neg P(x,y)]]$$

(b) Conclude from (a) that for any model $\mathcal{M}$ for pure predicate logic,

$$\mathcal{M} \models \mathbf{A} \qquad \text{and} \qquad \mathcal{M} \models \mathbf{B}.$$

(c) Conclude from (b) that for any model $\mathcal{M}$ for pure predicate logic,

$$\mathcal{M} \models \exists x \exists y \ [P(x,y) \vee \neg P(x,y)].$$

(d) Conclude from (c) that a model of predicate logic must have a nonempty universe.

It should be mentioned that there are other treatments of logic in which the universe of a model is allowed to be empty; such treatments generally require a more restricted definition of "proof" than we have given in this text.

27. Let A be a finite linearly ordered set (for example the 26 letters of the Latin alphabet) and A^* denote set of all finite sequences (words) of elements of A. Given two words $w, w' \in A^*$ we write $w \preceq w'$ iff w precedes w' in alphabetical order. Define this order relation precisely and prove that it is a linear order. (This order is often called the **lexicographic order** on A^*.) Hint: The empty sequence comes first, and ac precedes acb but not $abaaaa$.

28. Show that the theory of linear orders with no last element has infinite models but has no finite models.

29. Let $X = \{1,2,3,4\}$. Compute the transitive closure $\leq_R$ of each of the following relations $R \in REL_2(X)$:

1. $R = \{(1,2), (2,3), (1,4)\}$.
2. $R = \{(1,2), (2,3), (3,1), (1,4)\}$.
3. $R = \{(1,2), (2,3), (3,4)\}$.

30. Give an example of

(1) a binary relation R_1 which is not a pre-order and whose transitive closure is a pre-order but not a partial order;

(2) a binary relation R_2 which is not a pre-order and whose transitive closure is a partial order but not a linear order;

(3) a binary relation R_3 which is not a pre-order and whose transitive closure is a linear order.

31. For each of the first three order axioms in Section 2.11, give a model in which it fails but the other two axioms hold.

32. Let H denote the following three hypotheses:

$$\neg\exists x\, \exists y\, [x < y \wedge y < x]$$

$$\forall x\, \forall y\, [x < y \Rightarrow \forall z\, [x < z \vee z < y]]$$

$$\forall x\, \forall y\, [x \leq y \Leftrightarrow x < y \vee x \doteq y]$$

Must $\leq$ be represented by a linear order in any model for H? Give tableau proofs or a counter-model which respects equality.

33. Show that every equivalence relation is a congruence relation for itself.

34. Show that the relation $x \equiv_m y$ (on $\mathbf{Z}$) is an equivalence relation, that it is a congruence relation for each of the ternary relations $x+y \equiv_m z$ and $xy \equiv_m z$, but that it is not a congruence relation for the binary relation $x < y$.

35. Let π be a function from X to $\bar{X}$ and $\bar{R}$ be an n-ary relation on $\bar{X}$. In the text we observed that π determines an equivalence relation $\equiv_\pi$ on X via the definition

$$x \equiv_\pi y \Longleftrightarrow \pi(x) = \pi(y).$$

In Lemma 2.10.3 we saw that every equivalence relation could be defined this way: if an equivalence relation $\equiv$ is given on X and $\bar{X}$ denotes the set of equivalence classes $[x]$ and $\pi(x) = [x]$ then $\equiv$ and $\equiv_\pi$ are the same. The n-ary relation $\bar{R}$ on $\bar{X}$. determines an n-ary relation $\pi^*\bar{R}$ on X via

$$(x_1, x_2, \ldots, x_n) \in \pi^*\bar{R} \Longleftrightarrow (\pi(x_1), \pi(x_2), \ldots, \pi(x_n)) \in \bar{R}.$$

(a) Show that the relation $\equiv_\pi$ is a congruence relation for $\pi^*\bar{R}$.

(b) Show that if $\bar{X}$ happens to be the space of equivalence classes of some equivalence relation $\equiv$ and $\pi(x) = [x]$, then $\bar{R}$ is the relation induced on $\bar{X}$ by π^*R in the sense of Lemma 2.10.4.

36. Let $\leq$ be a pre-order on a set X and define a binary relation $\equiv$ on X by the rule

$$x \equiv y \iff x \leq y \text{ and } y \leq x.$$

Show that $\equiv$ is an equivalence relation, that it is a congruence relation for $\leq$, and that the induced relation on the set of equivalence classes is a partial order.

37. Enumerate the eight subsets $X_0, \ldots, X_7$ of $\{1,2,3\}$ in such a way that

$$X_i \subset X_j \text{ implies } i \leq j.$$

38. Show that for any finite partial order $(X, \leq)$ (i.e. X is finite) there is a linear order $(X, \leq^*)$ which extends $\leq$, i.e. for every $a, b \in P$ if $a \leq b \Rightarrow a \leq^* b$. Hint: By induction we may assume that

$$X = \{a_1, a_2, \ldots, a_n, b\}$$

where $a_i \leq a_j \Rightarrow i \leq j$. Let

$$L = \{x \in P : x \leq b,\ x \neq b\}, \qquad R = \{y \in P : b \leq y,\ x \neq b\}.$$

Argue that there must be an integer k with

$$L \subset \{a_1, a_2, \ldots, a_k\}, \qquad R \subset \{a_{k+1}, \ldots, a_n\}.$$

39. Show that every partial order on a countable set can be extended to a linear order.

Hint: Use the previous problem and the Compactness Theorem.

40. Let $\mathbf{A}_n$ be the sentence

$$\exists x_1 \cdots \exists x_n \left[\left[\bigwedge_{i \neq j} x_i \neq x_j \right] \wedge \forall y \left[\bigvee_{i=1}^{n} y \doteq x_i \right] \right]$$

where we have used the abbreviations $\left[\bigwedge_{i \neq j} x_i \neq x_j\right]$ for

$$x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge \cdots \wedge x_{n-1} \neq x_n$$

and $\left[\bigvee_{i=1}^{n} y \doteq x_i\right]$ for

$$y \doteq x_1 \vee y \doteq x_2 \vee \cdots \vee y \doteq x_n.$$

Let $\mathbf{H}_n$ consist of the four sentences: the sentence $\mathbf{A}_n$ and axioms (1-3) from Definition 2.10.1 on page 95. The set $\mathbf{H}_n$ has an obvious model $\mathcal{M}_n$ which respects equality: its universe consists of the first n positive integers $\{1, 2, \ldots, n\}$. Show that for any sentence $\mathbf{B}$ containing equality as its only predicate symbol we have $\mathbf{H}_n \vdash \mathbf{B}$ if and only if $\mathcal{M}_n \models \mathbf{B}$.

41. Two orders $(X, \leq)$ and $(X', \leq')$ are said to be **isomorphic** iff there is a one-one onto function $f : X \to X'$ such that for all $x, y \in X$ we have

$$x \leq y \iff f(x) \leq' f(y).$$

Such a function f is called an **order isomorphism** between the two orders.

(a) Show that the tangent function is an isomorphism between the open interval $]-\pi/2, \pi/2[$ and the set $\mathbf{R}$ of all real numbers (each with the usual linear order).

(b) Find real numbers m and c such that the formula

$$f(x) = mx + c$$

defines an order isomorphism from the interval $[a_1, a_2]$ to the interval $[b_1, b_2]$.

42. Show that any two countable dense linear orders without first or last element are isomorphic.[3] Deduce (using the Completeness Theorem) that if $\mathbf{A}$ is any sentence with no parameters and containing only the relation symbols $\leq$ and $\doteq$, then $\mathbf{Q} \models \mathbf{A}$ if and only if $\mathbf{R} \models \mathbf{A}$.

43. A **directed set** is a pair $(D, <)$ consisting of a set D and a binary relation $<$ on D which models the following axioms:

[3] See *Studies in Model Theory*, ed. by M.D. Morley, MAA Studies in Math, page 6 if you get stuck.

(anti-reflexive law) $\forall x \forall y [\neg x < y \vee \neg y < x]$

(transitive law) $\forall x \forall y \forall z [x < y \wedge y < z \Rightarrow x < z]$

(maximum law) $\forall x \forall y \exists z [x < z \wedge y < z]$

Which of the following sentences are true for all directed sets?

1. $\forall x \neg x < x$
2. $\forall x \forall y \forall z \exists w [x < w \wedge y < w \wedge z < w]$
3. $\forall x \exists y\ x < y$
4. $\exists y \forall x\ x < y$
5. $\forall x \exists y\ y < x$

For those that are true for all directed sets give a tableau proof with the three axioms and the negation of the wff to be proved at the root. For those that are not true for all directed sets give a counterexample. Can a directed set be finite?

44.

(a) Give a proof of the problem SET2 from the TABLEAU problem set in paragraph form, and analyze it as a proof using the tableau rules together with the Direct Proof, Learning, Deduction, and Generalization Rules.

(b) Do the same for the problem SET4.

45. Show that the first three equality laws (viz. the Reflexive, Symmetric, and Transitive Laws) follow from the Axiom of Extensionality. (You must give three tableau proofs.)

46.

(a) Show that there is no set T such that for all sets x we have

$$x \in T \iff x \notin x.$$

(b) Give a tableau confutation of the wff

$$\exists y \forall z [z \in y \Leftrightarrow \neg z \in z].$$

This wff has form

$$\exists y \forall z [z \in y \Leftrightarrow \mathbf{A}(z)].$$

and hence is *not* a case of the Comprehension scheme

$$\forall x \exists y \forall z [z \in y \Leftrightarrow [z \in x \wedge \mathbf{A}(z)]].$$

The proof of a contradiction is called **Russell's paradox.**

(c) Russell gave the following analogue of the above paradox:

> "Among the citizens of the town of Kenilworth there is a barber who shaves all and only those citizens of Kenilworth who do not shave themselves. Who shaves the barber?"

Note that the question as stated is impossible to answer. Can you think of a way to resolve the paradox?

47. Give a tableau confutation of the following two sentences:

$$\exists x \forall y\ y \in x$$

$$\forall x \exists y \forall z [z \in y \Leftrightarrow z \in x \wedge \neg z \in z]$$

The second hypothesis is a case of the Comprehension scheme of ZST. This gives a proof in ZST that the set of all sets does not exist.

48. Let $X_0, X_1, X_2, \ldots$ be subsets of $\mathbf{N} = \{0, 1, 2, \ldots\}$. Define a subset Y such that $Y \neq X_n$ for all n. Conclude that the set of subsets of $\mathbf{N}$ is not countable.

49. Let $W(u)$ be the sentence

$$\emptyset \in u \wedge \forall x [x \in u \Rightarrow x \cup \{x\} \in u].$$

The Axiom of Infinity from Section 2.12 is the wff $\exists u\, W(u)$.

(a) Write the Axiom of Infinity in a formally correct way, i.e. without using abbreviations like $\emptyset$ or $x \cup \{x\}$.

(b) Show any u satisfying $W(u)$ really is infinite. (Describe an infinite list of elements that the set u must contain). Hint: The sets $\emptyset$ and $\{\emptyset\}$ are different.

50. Let $\Omega(\omega)$ be the wff

$$\forall x\,[x \in \omega \Leftrightarrow \forall u\,[W(u) \Rightarrow x \in u]]$$

where $W(u)$ is the wff of the previous exercise and let $\mathbf{H}$ be the axioms (1-7) of ZST in the text. Recall that the notation $\mathbf{H} \vdash \mathbf{A}$ means that there is a tableau proof of $\mathbf{A}$ from the hypotheses $\mathbf{H}$. Prove the following:

- $\mathbf{H} \vdash \exists\omega\,\Omega(\omega)$.
- $\mathbf{H} \vdash \forall\omega\forall\omega'\,[\Omega(\omega) \wedge \Omega(\omega') \Rightarrow \omega \doteq \omega']$
- $\mathbf{H} \vdash \forall u\forall\omega\,[W(u) \wedge \Omega(\omega) \Rightarrow \omega \subset u]$

(The expression $\omega \subset u$ abbreviates $\forall x[x \in \omega \Rightarrow x \in u]$.) This exercise says that there is a unique set ω satisfying $\Omega(\omega)$ and that it satisfies an analog of the axiom of induction.

51. In this exercise we describe a model $\mathcal{M}_0 = (M_0, \doteq, \in)$ for Axioms (1-5,7) of ZST, given in Section 2.12. Axiom (6), the Axiom of Infinity, is false in this model.

(a) List the elements of the three sets $P(\emptyset)$, $P(P(\emptyset))$, and $P(P(P(\emptyset)))$, where $\emptyset$ denotes the empty set and for any set X, $P(X)$ denotes the power set

$$P(X) = \{Y : Y \subset X\}$$

of all subsets of X.

(b) Define sets

$$V_0, V_1, \ldots, V_n, \ldots$$

and natural numbers

$$k_0, k_1, \ldots, k_n, \ldots$$

as follows:

$$V_0 = \emptyset, k_0 = 0.$$

$$V_{n+1} = P(V_n), k_{n+1} = 2^{k_n}.$$

Prove that for all n, V_n has exactly k_n elements. (Intuitively, $V_n = P(P(\ldots(\emptyset)\ldots))$ where P is repeated n times).

(c) We now define a model $\mathcal{M}_0$ for pure predicate logic with two relation symbols which will be $\doteq$ and $\in$ to suggest equality and set membership. The universe M_0 for $\mathcal{M}_0$ is the set

$$M_0 = \bigcup_{n \in \mathbf{N}} V_n$$

where V_n is defined in part (b). Now let $\mathcal{M}_0 = (M_0, \doteq, \in)$ where $\doteq$ and $\in$ are the equality and membership relations among the elements of M_0. Prove that $\mathcal{M}_0$ is a model of Axioms (1)-(5) of ZST.

(d) Prove that the Axiom of Infinity is false in $\mathcal{M}_0$.

52. In this exercise we build on the preceding exercise to describe a model of Axioms (1)-(7) of ZST. The idea is to repeat the construction used in the preceding exercise, but starting with the set M_0 instead of the empty set.

Define a model $\mathcal{M}$ for pure predicate logic with the two predicate symbols $\doteq$ and $\in$ as follows. The universe M of $\mathcal{M}$ is defined to be the union of a sequence of sets

$$M_0, M_1, \ldots, M_n, \ldots$$

where M_0 is the set of the previous exercise and M_{n+1} is defined inductively by:

$$M_{n+1} = P(M_n) = \{X : X \subset M_n\}$$

Now, let $M = \bigcup_n M_n$ and interpret $\doteq$ by equality and $\in$ by membership among elements of M. Prove that $\mathcal{M}$ is a model of each of Axioms (1)-(6) of ZST.

53. Prove the Substitution Theorem (Theorem 2.13.6). Hint: The proof is by induction on the formula **C**. The Unique Scope Theorem is needed at the quantifier step, and Exercise 3 is needed at the binary connective step.

Chapter 3

Full Predicate Logic

In this chapter we enrich predicate logic by adding function symbols and a special symbol for equality. We shall call this enriched language **full predicate logic** to distinguish it from the simpler **pure predicate logic** developed in the last chapter. Full predicate logic is closer to the usual language of mathematics. Although it is in principle possible to express everything in the pure predicate logic of the previous chapter, in practice it is usually more convenient to develop mathematics in full predicate logic.

3.1 Syntax

A **vocabulary** $(\mathcal{P}, \mathcal{F})$ for full predicate logic consists of a list of sets $\mathcal{P}_n$ of n-ary **predicate symbols**, and sets $\mathcal{F}_n$ of n-ary **function symbols**, where $n = 0, 1, \ldots$. These sets may or may not be empty, but $\mathcal{P}_2$ always contains the equality symbol $\doteq$. The 0-ary predicate symbols in $\mathcal{P}_0$ are also called **proposition symbols**, and the 0-ary function symbols in $\mathcal{F}_0$ are also called **constant symbols**.

In addition to the vocabulary symbols $(\mathcal{P}, \mathcal{F})$, full predicate logic has all the primitive symbols of pure predicate logic, including the set VAR of variables, a set $\mathcal{K}$ of parameters, and the universal and existential quantifiers. As before, the elements of the set $VAR \cup \mathcal{K}$ are called **individual symbols**. The vocabulary constants from $\mathcal{F}_0$ are distinct from the individual parameters from $\mathcal{K}$, and will play a

different role in the semantics of full predicate logic.

The equality symbol $\doteq$, which always belongs to $\mathcal{P}_2$ in full predicate logic, plays a special role. Like the propositional connectives and quantifiers, it will be interpreted in a fixed way in all models. We always write $\tau \doteq \sigma$ in place of the more cumbersome $\doteq(\sigma, \tau)$.

Variables, parameter symbols, constant symbols, and function symbols may be combined to form terms. A **term** is a string which can be obtained by finitely many applications of the following **rules of formation**:

(**T:***VAR*) Any variable is a term.

(**T:**$\mathcal{K}$) Any element of $\mathcal{K}$ is a term.

(**T:**$\mathcal{F}_0$) Any constant symbol from $\mathcal{F}_0$ is a term.

(**T:**$\mathcal{F}_n$) If $\mathrm{f} \in \mathcal{F}_n$ is a function symbol, where $n > 0$, and $\tau_1, \tau_2, \ldots, \tau_n$ are terms, then $\mathrm{f}(\tau_1, \tau_2, \ldots, \tau_n)$ is a term.

These rules are used repeatedly. For example, if y is a variable, c is a constant, f is binary, and g is unary, then $g(f(c, g(y)))$ is a term. Terms, like wffs, have parsing sequences. The above example is parsed as follows:

(1) c is a term by (T:$\mathcal{F}_0$).

(2) y is a term by (T:*VAR*).

(3) $g(y)$ is a term by (2) and (T:$\mathcal{F}_1$).

(4) $f(c, g(y))$ is a term, by (1), (3), and (T:$\mathcal{F}_2$).

(5) $g(f(c, g(y)))$ is a term by (4) and (T:$\mathcal{F}_1$).

The set $TERM(\mathcal{F}, \mathcal{K})$ of **variable free terms** of type $\mathcal{F}$ with parameters from $\mathcal{K}$ consists of those terms which contain no elements of *VAR*, that is, which are built without using the (T:*VAR*) rule.

We continue using the abbreviations and notational conventions introduced earlier and in addition add the the usual mathematical conventions regarding infix notation and parentheses.

- The familiar binary function symbols $+$, $-$, and $*$ are written in infix notation so that $(x + y)$ is written instead of $+(x, y)$.
- The outer parentheses may be suppressed, so that $x + y$ means $(x + y)$.
- Multiplication has a higher precedence than addition or subtraction, so that $x + y * z$ means $x + (y * z)$ and not $(x + y) * z$.
- Operations of equal precedence associate to the left in the absence of explicit parentheses, so that $x - y - z$ means $(x - y) - z$ and not $x - (y - z)$.

The set of wffs is defined as before except that the argument places in the predicate symbols may be filled by terms. Here are the rules of formation.

(**W:**$\mathcal{P}_0$) Any propositional symbol is a wff.

(**W:**$\mathcal{P}_n$) If $p \in \mathcal{P}_n$ is a predicate symbol and $\tau_1, \tau_2, \ldots, \tau_n$ are terms, then $p(\tau_1, \tau_2, \ldots, \tau_n)$ is a wff.

(**W:**$\neg$) If **A** is a wff, then $\neg$**A** is a wff.

(**W:**$\wedge, \vee, \Rightarrow, \Leftrightarrow$) If **A** and **B** are wffs, then [**A** $\wedge$ **B**], [**A** $\vee$ **B**], [**A** $\Rightarrow$ **B**], and [**A** $\Leftrightarrow$ **B**] are wffs.

(**W:**$\forall, \exists$) If **A** is a wff, and x is a variable, then $\forall x$**A** and $\exists x$**A** are wffs.

(If it is necessary to explicitly specify the vocabulary $(\mathcal{P}, \mathcal{F})$ used in the definition of the set of wffs, we shall refer to the wffs defined here as **built using the vocabulary** $(\mathcal{P}, \mathcal{F})$.)

Atomic wffs and basic wffs are defined as before except that now arbitrary terms may occupy the argument positions. Thus **atomic wffs** are those constructed by rules (W:$\mathcal{P}_0$) and (W:$\mathcal{P}_n$) above, while a **basic wff** is a wff which is either an atomic wff or the negation of an atomic wff.

The Unique Readability Theorem generalizes to full predicate logic. As in the case of pure predicate logic, an occurrence of a variable x in a wff **A** is a **bound occurrence** if it is in the scope of a quantifier on

x; all other occurrences of individual symbols are called **free**. As in pure predicate logic all occurrences of a variable in a basic wff are free, because a basic wff has no quantifiers.

In full predicate logic, the notion of an individual being free for a variable in a wff is replaced by the notion of a **term** being free for a variable in a wff. A term τ is said to be **freely substitutable for**, or **free for**, the variable x in a wff $\mathbf{A}$ if every variable which occurs in τ is free for x in $\mathbf{A}$. Given a wff $\mathbf{A}$, a variable x, and a term τ which is free for x in $\mathbf{A}$, $\mathbf{A}(x//\tau)$ is the wff obtained by replacing each free occurrence of x in $\mathbf{A}$ by τ.

3.2 Semantics

In this section we define the notion of a model for full predicate logic, and then give the rules which determine the truth value of a sentence in a model. As in the case of pure predicate logic, the n-ary predicate symbols will stand for relations on the universe set of the model. The n-ary function symbols will stand for functions of n variables on the universe set.

Recall that for each natural number $n > 0$, an n-ary relation on a set X is a subset of X^n, and a 0-ary relation on X is just a truth value. $REL_n(X)$ is the set of all n-ary relations on X. We now introduce n-ary functions on a set X. When $n > 0$, an n-ary function on X is a function $f : X^n \to X$ from the set X^n of n-tuples to the set X. A 0-ary function on X is just an element of X. $FUN_n(X)$ will denote the set of all n-ary functions on X.

A **premodel for full predicate logic of type** $(\mathcal{P}, \mathcal{F})$ is a system $\mathcal{M}$ consisting of a non-empty set M called the **universe set of** $\mathcal{M}$, and for each $n \geq 0$ a function which assigns to each n-ary predicate (or propositional) symbol p an n-ary relation $p^{\mathcal{M}}$ on M, and another function which assigns to each n-ary function (or constant) symbol $\mathbf{f}$ an n-ary function (or constant) $\mathbf{f}^{\mathcal{M}}$ on M. We say that the premodel $\mathcal{M}$ **respects equality** if the equality relation of the premodel $\mathcal{M}$ is true equality, that is,

$$\doteq^{\mathcal{M}} \quad \text{is} \quad \{(a,b) \in M^2 : a = b\}.$$

A **model for full predicate logic of type** $(\mathcal{P}, \mathcal{F})$ is a premodel which respects equality.

In mathematics, models are more important than premodels. Premodels are a convenient tool which allows us to begin proving results which do not involve the special properties of the equality relation. Since every model is a premodel, all of our results for premodels will hold for models as well.

In the next theorem we assign an element of the universe set M as a value for each variable free term from $TERM(\mathcal{F}, M)$.

Theorem 3.2.1 *For each premodel $\mathcal{M}$ of type $(\mathcal{P}, \mathcal{F})$, there is a unique function which assigns an element $\tau_{\mathcal{M}} \in M$ to each variable free term $\tau \in TERM(\mathcal{F}, M)$ such that the following formation rules hold:*

(M:M) *If $u \in M$, then $u_{\mathcal{M}} = u$.*

(M:$\mathcal{F}_0$) *If $c \in \mathcal{F}_0$, then $c_{\mathcal{M}} = c^{\mathcal{M}}$.*

(M:$\mathcal{F}_n$) *If $\tau_1, \tau_2, \ldots, \tau_n$ are terms and $\mathbf{f} \in \mathcal{F}_n$ is a function symbol, then*

$$\mathbf{f}(\tau_1, \tau_2, \ldots, \tau_n)_{\mathcal{M}} = \mathbf{f}^{\mathcal{M}}(\tau_{1\mathcal{M}}, \tau_{2\mathcal{M}}, \ldots, \tau_{n\mathcal{M}}).$$

Proof: To justify this definition we need a Unique Readability Theorem for terms: Every term in $TERM(\mathcal{F}, \mathcal{K})$ is either an individual symbol, a constant symbol from $\mathcal{F}_0$, or can be uniquely read in the form

$$\mathbf{f}(\tau_1, \tau_2, \ldots, \tau_n)$$

where $\mathbf{f} \in \mathcal{F}_n$ and $\tau_1, \ldots, \tau_n$ are terms. We omit the remaining details of the proof. **End of Proof.**

We define the set $WFF(\mathcal{P}, \mathcal{F}, \mathcal{K})$ of wffs based on the vocabulary $(\mathcal{P}, \mathcal{F})$ with additional parameters from the set $\mathcal{K}$ as in pure predicate logic except that the rule (W:$\mathcal{P}_n$) is modified to allow terms:

(W:$\mathcal{P}_n$) If $p \in \mathcal{P}_n$ and $\tau_1, \tau_2, \ldots, \tau_n$ are terms then $p(\tau_1, \tau_2, \ldots, \tau_n) \in WFF(\mathcal{P}, \mathcal{F}, \mathcal{K})$.

As in pure predicate logic, $SENT(\mathcal{P},\mathcal{F},\mathcal{K})$ is the subset of $WFF(\mathcal{P},\mathcal{F},\mathcal{K})$ consisting of those wffs with no free variables: it is the set of all **sentences** built from the vocabulary $(\mathcal{P},\mathcal{F})$ with additional parameters from the set $\mathcal{K}$. The following is proved in the same way as the analogous result for pure predicate logic.

Theorem 3.2.2 *Given a premodel $\mathcal{M}$ of type $(\mathcal{P},\mathcal{F})$ there is a unique function which assigns a truth value $\mathbf{A}_{\mathcal{M}}$ to each sentence $\mathbf{A}$ with parameters from M which satisfies the conditions of Theorem 2.4.1, but with the condition (M:$\mathcal{P}_n$) modified to read*

(M:$\mathcal{P}_n$) $\mathcal{M} \models p(\tau_1,\tau_2,\ldots,\tau_n)$ *iff* $(\tau_{1\mathcal{M}},\tau_{2\mathcal{M}},\ldots,\tau_{n\mathcal{M}}) \in p^{\mathcal{M}}$.

As usual we have written $\mathcal{M} \models \mathbf{A}$ in place of the more cumbersome phrase $\mathbf{A}_{\mathcal{M}} = \mathbf{T}$.

Remark 3.2.3 If the premodel $\mathcal{M}$ respects equality, then for all terms

$$\tau,\sigma \in TERM(\mathcal{F},M)$$

we have

$$\mathcal{M} \models \tau \doteq \sigma \text{ if and only if } \tau_{\mathcal{M}} = \sigma_{\mathcal{M}}.$$

3.3 Tableaus

In full predicate logic, a tableau may be formed using all the rules for tableaus in propositional logic (see Figure 1.4) plus additional rules for handling terms and the equality relation. A **labeled tree for full predicate logic** is defined as for propositional logic, except that now the wffs are those of full predicate logic.

Definition 3.3.1 A **tableau for full predicate logic** is defined as before except that two of the four quantifier rules allow the substitution of terms, and there are three new equality rules. The new rules are:

$\boxed{\forall}$ If t has an ancestor $\forall x\mathbf{A}$, extend by adding a child $\mathbf{A}(x//\tau)$ of t, where τ is a term which is free for x in $\mathbf{A}$;

$\boxed{\neg\forall}$ If t has an ancestor $\neg\forall x\mathbf{A}$, extend by adding a child $\neg\mathbf{A}(x//b)$ of t, where b is an individual symbol which does not occur in any ancestor of t;

$\boxed{\exists}$ If t has an ancestor $\exists x\mathbf{A}$, extend by adding a child $\mathbf{A}(x//b)$ of t, where b is an individual symbol which does not occur in any ancestor of t;

$\boxed{\neg\exists}$ If t has an ancestor $\neg\exists x\mathbf{A}$, extend by adding a child $\neg\mathbf{A}(x//\tau)$ of t, where τ is a term which is free for x in $\mathbf{A}$.

$\boxed{=1}$ If t has an ancestor $[\neg]p(\ldots\tau\ldots)$, and another ancestor of form $\tau \doteq \sigma$, extend by adding a child $[\neg]p(\ldots\sigma\ldots)$ of t.

$\boxed{=2}$ If t has an ancestor $[\neg]p(\ldots\tau\ldots)$, and another ancestor of form $\sigma \doteq \tau$, extend by adding a child $[\neg]p(\ldots\sigma\ldots)$ of t.

$\boxed{=3}$ Extend by adding a child $\sigma \doteq \sigma$ of t.

In these rules t denotes the terminal node at which the tableau is extended.

Diagrams for the three equality rules [1] are shown in Figure 3.1. In the first two equality rules, $[\neg]p(\ldots\tau\ldots)$ and $[\neg]p(\ldots\sigma\ldots)$ denote basic wffs (i.e. atomic wffs or negations of atomic wffs) such that $[\neg]p(\ldots\sigma\ldots)$ results from $[\neg]p(\ldots\tau\ldots)$ by replacing one occurrence of the term τ by the term σ. The occurrence of τ may be a part of some longer term within the wff $[\neg]p(\ldots\tau\ldots)$.

For example, if τ is $f(a)$ and σ is b, and we take

$$p(g(f(a)),a,f(a)) \text{ for } p(\ldots f(a)\ldots),$$

then there are two possibilities for $p(\ldots b\ldots)$ (one for each occurrence of $f(a)$). We can either take

$$p(g(b),a,f(a)) \text{ for } p(\ldots b\ldots),$$

[1] In the TABLEAU program, the first two equality rules are invoked by typing the G key at the node **A** to put **A** in the **Get box**, typing the S key at the node $\tau \doteq \sigma$ to put either $\tau \doteq \sigma$ or $\sigma \doteq \tau$ into the **Sub box** (pressing the right arrow key toggles between these two), then going to the end of the branch and typing the E key to extend the tableau. The third equality rule is invoked by typing the = key.

or

$$p(g(f(a)), a, b) \text{ for } p(\ldots b \ldots).$$

In order to be sure that the string $[\neg]p(\ldots\sigma\ldots)$ is a wff, one must prove that whenever τ occurs within a term, the string formed by replacing one occurrence of τ by σ is also a term. This is left as an exercise, with a hint, at the end of this chapter.

The rules $\boxed{=1}$ and $\boxed{=2}$ differ only in that in the former the equality ancestor is $\tau \doteq \sigma$ while in the latter it is $\sigma \doteq \tau$.

The equality rules are justified by the fact that sentences

$$\mathcal{M} \models [\neg]p(\ldots\tau\ldots) \wedge [\tau \doteq \sigma] \Rightarrow [\neg]p(\ldots\sigma\ldots)$$

$$\mathcal{M} \models [\neg]p(\ldots\tau\ldots) \wedge [\sigma \doteq \tau] \Rightarrow [\neg]p(\ldots\sigma\ldots)$$

$$\mathcal{M} \models \sigma \doteq \sigma$$

will be valid in any model $\mathcal{M}$ which respects equality.

The basic definitions are the same as before except for the addition of the new tableau rules. A branch $\mathbf{\Gamma}$ of a tableau is said to be **contradictory** if $\mathbf{\Gamma}$ contains some wff and its negation.

The notions of a tableau confutation and a tableau proof are defined as before. A tableau $\mathbf{T}$ is said to be a **confutation** of a set of sentences $\mathbf{H}$ if $\mathbf{T}$ is a finite tableau with hypothesis set $\mathbf{H}$ and every branch of $\mathbf{T}$ is contradictory. A **tableau proof** of $\mathbf{A}$ from $\mathbf{H}$ is a tableau confutation of $\mathbf{H} \cup \{\neg\mathbf{A}\}$.

Tableau Extension Rules

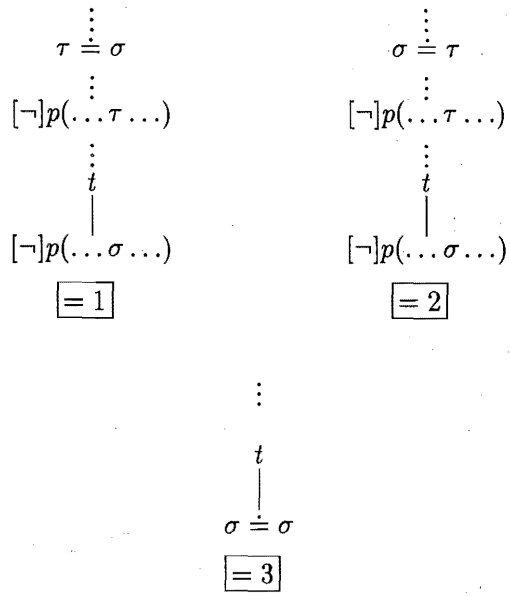


Figure 3.1: Equality Rules for Full Predicate Logic.

Here are two simple examples of tableau proofs in full predicate logic. The second example is one of the Equality Axioms from Section 2.10. In Exercise 7 you are asked to give tableau proofs of the remaining Equality Axioms.

Example 3.3.2 A tableau proof of

$$\forall x \forall y[x \doteq y \Rightarrow f(x) \doteq f(y)].$$

(1)	$\neg\forall x\forall y[x \doteq y \Rightarrow f(x) \doteq f(y)]$	$\neg$ to be proved
	|	
(2)	$\neg\forall y[a \doteq y \Rightarrow f(a) \doteq f(y)]$	by (1)
	|	
(3)	$\neg[a \doteq b \Rightarrow f(a) \doteq f(b)]$	by (2)
	|	
(4)	$a \doteq b$	by (3)
	$\|\|$	
(5)	$\neg f(a) \doteq f(b)$	by (3)
	|	
(6)	$\neg f(b) \doteq f(b)$	by (4) and (5)
	|	
(7)	$f(b) \doteq f(b)$	by equality rule 3

Example 3.3.3 A tableau proof of

$$\forall x \forall y \forall z[x \doteq y \wedge y \doteq z \Rightarrow x \doteq z].$$

(1)	$\neg\forall x\forall y\forall z[x \doteq y \wedge y \doteq z \Rightarrow x \doteq z].$	$\neg$ to be proved
	|	
(2)	$\neg\forall y\forall z[a \doteq y \wedge y \doteq z \Rightarrow a \doteq z]$	by (1)
	|	
(3)	$\neg\forall z[a \doteq b \wedge b \doteq z \Rightarrow a \doteq z]$	by (2)
	|	
(4)	$\neg[a \doteq b \wedge b \doteq c \Rightarrow a \doteq c]$	by (3)
	|	
(5)	$a \doteq b \wedge b \doteq c$	by (4)
	$\|\|$	
(6)	$\neg a \doteq c$	by (4)
	|	
(7)	$a \doteq b$	by (5)
	$\|\|$	
(8)	$b \doteq c$	by (5)
	|	
(9)	$a \doteq c$	by (7) and (8)

3.4 Soundness

The proof of the Soundness Theorem for full predicate logic is much as before. The definition of **valuation** in M (which assigns elements of M to finitely many individual symbols), **satisfiable**, and **simultaneously satisfiable** are the same as for pure predicate logic (see Definition 2.7.1).

Lemma 3.4.1 *Let* $\mathbf{H}$ *be a set of sentences of full predicate logic of type* $(\mathcal{P}, \mathcal{F})$. *Let* $\mathbf{T}$ *be a tableau in predicate logic with hypothesis set* $\mathbf{H}$. *Let* $\mathcal{M}$ *be a model of* $\mathbf{H}$. *Then there is a branch* $\mathbf{\Gamma}$ *of* $\mathbf{T}$ *such that the wffs on* $\mathbf{\Gamma}$ *are simultaneously satisfiable in* $\mathcal{M}$.

Proof: The proof is like that of Lemma 2.7.2 except that we must deal with the three equality rules in the step where we build a branch $\mathbf{\Gamma}_{k+1}$ on $\mathbf{T}_{k+1}$ from a given branch $\mathbf{\Gamma}_k$ of a smaller tableau $\mathbf{T}_k$. We have to check that if any of the equality rules were used to extend $\mathbf{\Gamma}_k$, the valuation v_k given by the induction hypothesis satisfies the new wff given by the equality rule. This follows from the fact that the model respects equality. **End of Proof.**

Theorem 3.4.2 (Soundness Theorem) *Suppose* $\mathbf{H}$ *is a set of sentences in full predicate logic and* $\mathbf{A}$ *is a sentence. If* $\mathbf{H} \vdash \mathbf{A}$, *then* $\mathbf{H} \models \mathbf{A}$, *that is, every model of* $\mathbf{H}$ *is a model of* $\mathbf{A}$. *In particular, if there is a tableau proof (without hypotheses) of a sentence* $\mathbf{A}$, *then* $\mathbf{A}$ *is valid.*

This is proved as before: see Theorem 2.7.4. Both Lemma 3.4.1 and the Soundness Theorem require that $\mathcal{M}$ respect equality. They are true for all models but not for all premodels.

3.5 Completeness

The Completeness Theorem for full predicate logic is similar to the one for predicate logic, but with some additional twists. As before, we begin with a main lemma which easily implies the Completeness Theorem.

Lemma 3.5.1 (Main Lemma) *Let* $\mathbf{H}$ *be a finite or countable set of sentences of full predicate logic. Either* $\mathbf{H}$ *has a tableau confutation or* $\mathbf{H}$ *has a model which respects equality.*

Theorem 3.5.2 (Extended Completeness Theorem) *Suppose* $\mathbf{H}$ *is a finite or countable set of sentences and* $\mathbf{A}$ *is a sentence of full predicate logic. If* $\mathbf{H} \models \mathbf{A}$ *then* $\mathbf{H} \vdash \mathbf{A}$; *that is, if every model of* $\mathbf{H}$ *is a model of* $\mathbf{A}$, *then there is a tableau proof of* $\mathbf{A}$ *from* $\mathbf{H}$. *In particular, a valid sentence has a tableau proof.*

As in the Completeness Theorem for pure predicate logic we fix an infinite set M of new parameters. The set $TERM(\mathcal{F}, M)$ will be used as the universe set of a model.

We call a set $\mathbf{\Delta}$ of wffs **closed under the equality rules** if any basic wff obtained from two wffs of $\mathbf{\Delta}$ by an equality substitution is again a element of $\mathbf{\Delta}$; in other words, if for all terms τ and σ in $TERM(\mathcal{F}, M)$ and all basic wffs $[\neg]p(\ldots\tau\ldots)$, the following conditions hold:

[= 1] if $[\tau \doteq \sigma], [\neg]p(\ldots\tau\ldots) \in \mathbf{\Delta}$ then $[\neg]p(\ldots\sigma\ldots) \in \mathbf{\Delta}$.

[= 2] if $[\sigma \doteq \tau], [\neg]p(\ldots\tau\ldots) \in \mathbf{\Delta}$ then $[\neg]p(\ldots\sigma\ldots) \in \mathbf{\Delta}$.

[= 3] $[\sigma \doteq \sigma] \in \mathbf{\Delta}$

A set $\mathbf{\Delta}$ of wffs is called **contradictory** if it contains some wff and its negation.

The definition of a finished set for full predicate logic on a set M is verbatim the same as the definition of a finished set of wffs for pure predicate logic given before except that now

- In the $[\forall]$ and $[\neg\exists]$ rules the set $TERM(\mathcal{F}, M)$ is used in place of M.
- the set $\mathbf{\Delta}$ must be closed under equality rules.

In particular, if a wff $\forall x\mathbf{A}$ is in a finished set $\mathbf{\Delta}$ the new version of the $[\forall]$ rule requires that every wff $\mathbf{A}(x//\tau)$ with $\tau \in TERM(\mathcal{F}, M)$ be an element of $\mathbf{\Delta}$, not just those where $\tau \in M$.

As in pure predicate logic, a branch of a tableau is **finished** on M if the set of all wffs on the branch is finished on M, and a tableau in

full predicate logic is **finished** on M if every branch is either finished on M or else both finite and contradictory.

For the Tableau Extension Lemma we require that the set $\mathcal{F}$ of function symbols be finite or countable. In this case, the set $TERM(\mathcal{F}, M)$ is countable (see Exercise 8) and we are able to build a finished tableau on a countable set of new parameters M as in Chapter 2. We will not need the assumption that $\mathcal{F}$ is finite or countable for the Main Lemma or its consequences.

Lemma 3.5.3 (Tableau Extension Lemma) *Suppose that the set $\mathcal{F}$ of function symbols is finite or countable. Let M be a countable set, and suppose $\mathbf{H}$ is a finite or countable set of sentences. Then $\mathbf{H}$ is the hypothesis set of a finished tableau on M.*

Proof: The proof is basically the same as in pure predicate logic except that now we must use terms to extend the tableau at nodes with universal quantifiers and we must make sure that the final tableau is closed under the equality rules.

As before, we let $\mathbf{H} = \{\mathbf{C}_1, \mathbf{C}_2, \ldots\}$ and $\mathbf{H}_n = \{\mathbf{A}_1, \ldots, \mathbf{A}_n\}$.

Since the set $TERM(\mathcal{F}, M)$ is countable, it may be arranged in a list

$$TERM(\mathcal{F}, M) = \{\tau_1, \tau_2, \ldots\}.$$

We build finite tableaus $\mathbf{T}_0 \subset \mathbf{T}_1 \subset \ldots$ with hypothesis set $\mathbf{H}$ as before, and our final tableau $\mathbf{T}$ will be the union of the tableaus $\mathbf{T}_n$. We extend $\mathbf{T}_n$ to a finite tableau $\mathbf{T}_{n+1}$ as in the proof of the Tableau Extension Lemma for pure predicate logic with the following additional features.

If $\mathbf{A}$ is either in $\mathbf{H}_n$ or at a nonroot node of $\mathbf{T}_n$ and $\mathbf{A}$ is of the form $\forall x\mathbf{B}$ or $\neg\exists x\mathbf{B}$, then each noncontradictory branch in $\mathbf{T}_{n+1}$ through $\mathbf{A}$ must have the $n+1$ formulas $[\neg]\mathbf{B}(x//\tau_i)$ for $i = 1, \ldots, n+1$.

To make progress toward closure under the equality rules, each noncontradictory branch of $\mathbf{T}_{n+1}$ must have a basic wff $p(\ldots\tau\ldots)$ whenever required by the equality rules [= 1] or [= 2] using wffs in $\mathbf{H}_n$ and/or wffs at nonroot nodes of $\mathbf{T}_n$. Finally, each noncontradictory branch of $\mathbf{T}_{n+1}$ must have the wff $\tau_{n+1} \doteq \tau_{n+1}$ so that condition [= 3] will be satisfied.

We leave the straightforward proof that $\mathbf{T}$ is a finished tableau on M to the reader. **End of Proof.**

Lemma 3.5.4 (Finished Set Lemma for Premodels) *Suppose $\mathbf{\Delta}$ is a finished set of wffs on a set M. Define a premodel $\mathcal{M}$ for full predicate logic as follows:*

- *the universe set of the premodel is $TERM(\mathcal{F}, M)$;*
- *for each propositional symbol $p \in \mathcal{P}_0$, $p_{\mathcal{M}} = \mathbf{T}$ iff $p \in \mathbf{\Delta}$;*
- *for each n-ary predicate symbol $p \in \mathcal{P}_n$ and all $\tau_1, \ldots, \tau_n \in TERM(\mathcal{F}, M)$ $(\tau_1, \ldots, \tau_n) \in p^{\mathcal{M}}$ iff $p(\tau_1, \ldots, \tau_n) \in \mathbf{\Delta}$.*

Then $\mathcal{M} \models \mathbf{\Delta}$.

Proof: The proof proceeds as in the Finished Set Lemma for pure predicate logic except that we need to use induction on the **height** rather than the length of wffs. This is because if $\mathbf{C} \in \mathbf{\Delta}$ and $\mathbf{C}$ is of the form $\forall x\mathbf{A}$ then $\mathbf{A}(x//\tau)$ may be longer than $\forall x\mathbf{A}$. We define the height $h(\mathbf{A})$ to be the number of occurrences of quantifiers and connectives in $\mathbf{A}$. Thus atomic wffs have height zero. Now proceed as in the proof of the Finished Set Lemma for pure predicate logic, replacing length by height. For example, if $\mathbf{C} \in \mathbf{\Delta}$ and $\mathbf{C}$ is of the form $\forall x\mathbf{A}$, then $\mathbf{A}(x//\tau) \in \mathbf{\Delta}$ for all τ. Since $\mathbf{A}(x//\tau)$ is of lower height than $\forall x\mathbf{A}$, we have $\mathcal{M} \models \mathbf{A}(x//\tau)$ for all τ, and hence $\mathcal{M} \models \forall x\mathbf{A}$. **End of Proof.**

The Finished Set Lemma for Premodels gives us a premodel which need not respect equality. To get a model, we need three more lemmas. In all three lemmas we assume that $\mathbf{\Delta} \subset SENT(\mathcal{P}, \mathcal{F}, M)$ is a finished set of wffs in the parameters M. We shall call terms τ and σ in $TERM(\mathcal{F}, M)$ **equivalent** (abbreviated $\tau \equiv \sigma$) if the sentence $\tau \doteq \sigma$ is an element of the finished set $\mathbf{\Delta}$. Thus

$$\tau \equiv \sigma \text{ iff } [\tau \doteq \sigma] \in \mathbf{\Delta}.$$

Lemma 3.5.5 *Let $\mathbf{\Delta}$ be a finished set of wffs on M. Then $\equiv$ is an equivalence relation on the set of terms in $TERM(\mathcal{F}, M)$. That is, for $\tau, \sigma, \rho \in TERM(\mathcal{F}, M)$:*

(reflexivity) $\tau \equiv \tau$;

(symmetry) *if* $\tau \equiv \sigma$ *then* $\sigma \equiv \tau$;

(transitivity) *if* $\tau \equiv \sigma$ *and* $\sigma \equiv \rho$ *then* $\tau \equiv \rho$.

Lemma 3.5.6 *Let* Δ *be a finished set of wffs on* M. *Let*

$$\tau_1, \tau_2, \ldots, \tau_n, \sigma_1, \sigma_2, \ldots, \sigma_n \in TERM(\mathcal{F}, M)$$

and $\mathrm{f} \in \mathcal{F}_n$. *If*

$$\tau_1 \equiv \sigma_1,\ \tau_2 \equiv \sigma_2,\ \ldots,\ \tau_n \equiv \sigma_n,$$

then

$$\mathrm{f}(\tau_1, \tau_2, \ldots, \tau_n) \equiv \mathrm{f}(\sigma_1, \sigma_2, \ldots, \sigma_n).$$

Lemma 3.5.7 *Let* Δ *be a finished set of wffs on* M. *Suppose*

$$\tau_1, \tau_2, \ldots, \tau_n, \sigma_1, \sigma_2, \ldots, \sigma_n \in TERM(\mathcal{F}, M)$$

and $p \in \mathcal{P}_n$. *If*

$$\tau_1 \equiv \sigma_1,\ \tau_2 \equiv \sigma_2,\ \ldots,\ \tau_n \equiv \sigma_n,$$

then

$$p(\tau_1, \ldots, \tau_n) \in \Delta \text{ iff } p(\sigma_1, \ldots, \sigma_n) \in \Delta.$$

Proof: The proofs of these three lemmas are easy consequences of what it means for the set Δ to be closed under the equality rules. For example, the reflexive law in Lemma 3.5.5 follows from part [= 3] in the definition. To prove the symmetry law assume $[\tau \doteq \sigma] \in \Delta$. By [= 3] we have $[\tau \doteq \tau] \in \Delta$ so we may use [= 1] with the first occurrence of τ in the basic wff $\tau \doteq \tau$ to conclude that $[\sigma \doteq \tau] \in \Delta$. To prove the transitive law assume $[\tau \doteq \sigma], [\sigma \doteq \rho] \in \Delta$. Apply [= 2] to replace the occurrence of σ in the basic wff $\sigma \doteq \rho$ by τ and conclude that $[\tau \doteq \rho] \in \Delta$. Lemma 3.5.6 follows by applying part [= 1] n times to the basic wff

$$\mathrm{f}(\tau_1, \tau_2, \ldots, \tau_n) \doteq \mathrm{f}(\tau_1, \tau_2, \ldots, \tau_n),$$

(this is an element of Δ by part [= 3]) to obtain that the wff

$$\mathrm{f}(\tau_1, \tau_2, \ldots, \tau_n) \doteq \mathrm{f}(\sigma_1, \sigma_2, \ldots, \sigma_n)$$

is an element of Δ. Finally, Lemma 3.5.7 simply follows by repeated application of [= 1] and [= 2]. **End of Proof.**

Lemma 3.5.8 (Finished Set Lemma) *Let* Δ *be a finished set of wffs on the nonempty set* M. *Then there is a model* $\mathcal{N}$ *with an interpretation for each element* $a \in M$ *such that* $\mathcal{N} \models \Delta$.

Proof: For each $\tau \in TERM(\mathcal{F}, M)$ let $[\tau]$ denote the equivalence class of τ:

$$[\tau] \doteq \{\sigma \in TERM(\mathcal{F}, M) : \tau \equiv \sigma\}.$$

By Lemma 3.5.5 we have

$$[\tau] \doteq [\sigma] \text{ iff } \tau \equiv \sigma.$$

We define the universe N of our model $\mathcal{N}$ to be the set of equivalence classes:

$$N = \{[\tau] : \tau \in TERM(\mathcal{F}, M)\}.$$

Now by Lemma 3.5.6 each function symbol $\mathrm{f} \in \mathcal{F}_n$ determines a function $\mathrm{f}^{\mathcal{N}} \in FUN_n(N)$ by the condition

$$\mathrm{f}^{\mathcal{N}}([\tau_1], [\tau_2], \ldots, [\tau_n]) \doteq [\mathrm{f}(\tau_1, \tau_2, \ldots, \tau_n)].$$

In the case $n = 0$, if $c \in \mathcal{F}_0$ then $c^{\mathcal{N}} = [c]$.

This gives the universe set and the operations of a model $\mathcal{N}$.

It follows by induction on lengths of terms that for each term $\tau \in TERM(\mathcal{F}, M)$, the element $[\tau] \in \mathcal{N}$ is named by τ, that is,

$$[\tau] = \tau_{\mathcal{N}}.$$

By Lemma 3.5.7 each predicate symbol $p \in \mathcal{P}_n$ determines a relation $p^{\mathcal{N}} \in REL_n(N)$ by the condition

$$([\tau_1], [\tau_2], \ldots, [\tau_n]) \in p^{\mathcal{N}} \text{ iff } p(\tau_1, \tau_2, \ldots, \tau_n) \in \Delta.$$

For propositional symbols $p \in \mathcal{P}_0$, $p_{\mathcal{N}} = \mathbf{T}$ if and only if $p \in \Delta$.

This gives the predicates and completes the definition of the model $\mathcal{N}$. Let $\mathcal{M}$ be the premodel defined in the Finished Set Lemma for Premodels. It can be shown by induction on the height of sentences $\mathbf{B}$ over M that $\mathcal{N} \models \mathbf{B}$ if and only if $\mathcal{M} \models \mathbf{B}$. The details are left as an exercise. Since $\mathcal{M} \models \Delta$, we have $\mathcal{N} \models \Delta$ as required. **End of Proof.**

Proof of the Main Lemma: Let $\mathbf{H}$ be a finite or countable set of sentences with no tableau confutation. Let $\mathcal{F}'$ be a finite or countable subset of $\mathcal{F}$ which contains all the function symbols occurring in $\mathbf{H}$. We may apply the Tableau Extension Lemma to get a finished tableau with hypothesis set $\mathbf{H}$ on a countable set M. Since there is no confutation of $\mathbf{H}$, at least one branch Γ of the tableau is finished, and so by the Finished Set Lemma there is a model $\mathcal{M}$ of $\mathbf{H}$ of type $(\mathcal{P}, \mathcal{F}')$. The remaining function symbols which are in $\mathcal{F}_n$ but not in $\mathcal{F}'$, if any, can now be interpreted by any n-ary function on M at all, making $\mathcal{M}$ into a model of $\mathbf{H}$ of the required type $(\mathcal{P}, \mathcal{F})$. **End of Proof.**

As before, we now easily get the Compactness Theorem and the Extended Completeness Theorem for full predicate logic.

Theorem 3.5.9 (Compactness Theorem) *Let* $\mathbf{H}$ *be a countable set of sentences of full predicate logic. If every finite subset of* $\mathbf{H}$ *has a model, then* $\mathbf{H}$ *has a model.*

3.6 Theory of Groups

A set of sentences in first order logic is sometimes called a **first order theory**. In this section we look at an important example of a first order theory in full predicate logic, the theory of groups. The vocabulary for our language will consist of one infix binary function symbol $*$ and one constant symbol $\mathbf{e}$. The axioms of group theory are as follows:

Axioms of Group Theory

(1) Associativity: $\forall x \forall y \forall z\, (x*y)*z \doteq x*(y*z)$

(2) Identity: $\forall x\, [x*\mathbf{e} \doteq x \wedge \mathbf{e}*x \doteq x]$

(3) Inverses: $\forall x \exists y\, [x*y \doteq \mathbf{e} \wedge y*x \doteq \mathbf{e}]$

These axioms will be collectively known as $\mathbf{GT}$. The first axiom says that the operation is associative; the second says that the constant symbol $\mathbf{e}$ is an identity for the operation; and the third says that every element of a group has an inverse relative to $*$.

A model $\mathcal{G}$ of these axioms is a **group** which consists of a universe G together with interpretations $*^{\mathcal{G}}$ and $\mathbf{e}^{\mathcal{G}}$ of the symbols $*$ and $\mathbf{e}$. Instead of writing the group as $\mathcal{G} = (G, *^{\mathcal{G}}, \mathbf{e}^{\mathcal{G}})$, most textbooks simply identify a group $\mathcal{G}$ with its universe G whenever the operation and identity are clear from the context.

Examples of groups include

(1) $(\mathbf{Z}, +, 0)$ (recall that $\mathbf{Z}$ denotes the set of integers);

(2) $(\mathbf{Q}^+, \cdot, 1)$ where $\mathbf{Q}^+$ denotes the positive rationals and "$\cdot$" denotes multiplication; and

(3) for any set X, the group $(S(X), \circ, I_X)$ defined as follows: $S(X)$ is the set of all permutations f of X. (Recall that a permutation of X is a one-one, onto function from X to X; see Appendix A.) The operation "$\circ$" is composition of functions (see page 372). Finally, I_X is the identity permutation on X (see Appendix A, Section A.5). The reader may wish to verify that the group axioms are satisfied by this model.

Example 3.6.1 Figure 3.2 gives a tableau proof that in every group, the identity is unique; in this example, we prove the following sentence $\mathbf{A}$:

$$\forall x\, [\forall y\, x*y \doteq y \Rightarrow x \doteq \mathbf{e}].$$

(The sentence actually says that every left identity equals $\mathbf{e}$.) We include in the hypothesis set only the second axiom since we do not need the others in the proof. In the tableau problems at the end of this section, other properties of groups are established.

The groups $\mathbf{Z}$ and $\mathbf{Q}^+$ mentioned above satisfy the additional property

$$\mathbf{C} : \forall x \forall y\, x*y \doteq y*x$$

called the **commutative law**. If we could prove $\mathbf{C}$ from $\mathbf{GT}$ then by the Soundness Theorem, $\mathbf{C}$ would hold in *every* group. This is not the case however, since for any set X with more than two elements, $\mathbf{S}_X$ does not satisfy the commutative law (see Exercise 11). A group in which $\mathbf{C}$ holds is called **abelian**.

(1)	$\forall x\,[\mathbf{e}{*}x \doteq x \wedge x{*}\mathbf{e} \doteq x]$	
(2)	$\neg\forall x\,[\forall y\, x{*}y \doteq y \Rightarrow x \doteq \mathbf{e}]$	$\neg$ to be proved
(3)	$\neg[\forall y\, t{*}y \doteq y \Rightarrow t \doteq \mathbf{e}]$	by (2)
(4)	$\forall y\, t{*}y \doteq y$	by (3)
(5)	$\neg t \doteq \mathbf{e}$	
(6)	$t{*}\mathbf{e} \doteq \mathbf{e}$	by (4)
(7)	$\mathbf{e}{*}t \doteq t \wedge t{*}\mathbf{e} \doteq t$	by (1)
	$\mathbf{e}{*}t \doteq t$	by (7)
(9)	$t{*}\mathbf{e} \doteq t$	
(10)	$t \doteq \mathbf{e}$	by (6) and (9)

Figure 3.2: Tableau proof that the identity is unique

3.7 Peano Arithmetic

We now turn to another first order theory, called Peano Arithmetic. Throughout this book, Mathematical Induction has been one of our most important methods in informal proofs. The axioms of Peano Arithmetic consist of a group of six basic axioms, and an infinite list of additional axioms called the First Order Induction Principle which is the formal counterpart of Mathematical Induction.

The vocabulary for the predicate logic we will use consists of two infix function symbols $+$ and $*$, one unary function symbol $\mathbf{s}$, and one constant symbol $\mathbf{0}$. The constant symbol $\mathbf{0}$ is a boldfaced zero to distinguish it from the usual mathematical symbol 0. (Recall that the relation $\doteq$ is automatically a relation symbol in the vocabulary.) The full predicate logic with this vocabulary will be called the **language of arithmetic**. We let $\mathcal{N}$ denote the model of this language which has universe $\mathbf{N}$, the set of natural numbers, and in which the function symbols $+$ and $*$ are interpreted as ordinary addition and multiplication, respectively, of natural numbers; $\mathbf{s}$ is interpreted as the successor function

$$\mathbf{s}(0) = 1, \mathbf{s}(1) = 2, \mathbf{s}(2) = 3, \ldots;$$

and $\mathbf{0}$ is interpreted as the natural number 0. This model $\mathcal{N}$ is called the **standard model of arithmetic**.

Definition 3.7.1 **Peano Arithmetic**, or **PA**, is the collection consisting of the following six basic axioms:

1. $\forall x\, \neg\mathbf{s}(x) \doteq \mathbf{0}$
2. $\forall x\, \forall y\, [\mathbf{s}(x) \doteq \mathbf{s}(y) \Rightarrow x \doteq y]$
3. $\forall x\, x{+}\mathbf{0} \doteq x$
4. $\forall x\, \forall y\, x{+}\mathbf{s}(y) \doteq \mathbf{s}(x{+}y)$
5. $\forall x\, x{*}\mathbf{0} \doteq \mathbf{0}$
6. $\forall x\, \forall y\, x{*}\mathbf{s}(y) \doteq (x{*}y){+}x$

together with the all the instances of the

First Order Induction Principle

$$\forall y_1 \cdots \forall y_n [\mathbf{B}(\mathbf{0}) \wedge \forall x [\mathbf{B}(x) \Rightarrow \mathbf{B}(\mathbf{s}(x))] \Rightarrow \forall x \mathbf{B}(x)]$$

In this principle **B** is a wff in the language of arithmetic and all free variables of **B** are among $x, y_1, \ldots, y_n$ To improve readability, we wrote $\mathbf{B}(x)$ for **B**, $\mathbf{B}(\mathbf{0})$ for $\mathbf{B}(x//\mathbf{0})$, and $\mathbf{B}(\mathbf{s}(x))$ for $\mathbf{B}(x//\mathbf{s}(x))$.

Peano Arithmetic is of fundamental importance in mathematics, because it captures most of the mathematical facts which are known about the natural numbers.

Axiom 1 says that **0** is not the successor of any element. Axiom 2 says that the successor function **s** is one-one. Axioms 3 and 4 give the inductive definition of $+$ in terms of **0** and **s**. Axioms 5 and 6 give the inductive definition of $*$ in terms of $\mathbf{0}, \mathbf{s}$, and $+$.

The only constant symbol in the vocabulary of Peano Arithmetic is the zero symbol **0**. However, by repeatedly applying the successor function symbol **s** to **0** we obtain a constant term for each natural number. Thus $\mathbf{s}(\mathbf{0})$ stands for 1, $\mathbf{s}(\mathbf{s}(\mathbf{0}))$ stands for 2, and so on. The term

$$\mathbf{n} \doteq \underbrace{\mathbf{s}(\mathbf{s}(\ldots \mathbf{s}}(\mathbf{0})\ldots))$$

with n s's followed by **0** stands for the natural number n. It is called the **numeral of** n and is denoted by **n**. The first few numerals are

$$\mathbf{0} \doteq \mathbf{0}, \mathbf{1} \doteq \mathbf{s}(\mathbf{0}), \mathbf{2} \doteq \mathbf{s}(\mathbf{s}(\mathbf{0})), \ldots$$

Using the six basic axioms alone, one can prove many equations and inequalities involving particular numerals.

We give two examples as illustrations.

Example 3.7.2 Here is a tableau proof of the sentence

$$\neg \mathbf{3} \doteq \mathbf{1}$$

from Axioms 1 and 2 alone. Of course, everyone already knows this inequality. Our point here is that there is a tableau proof of it which uses only the first two axioms of Peano Arithmetic as hypotheses. Only the main steps are shown.

(1)	$\neg\neg \mathbf{s}(\mathbf{s}(\mathbf{s}(\mathbf{0}))) \doteq \mathbf{s}(\mathbf{0})$	$\neg$ to be proved
(2)	$\forall x \neg \mathbf{s}(x) \doteq \mathbf{0}$	Axiom 1
(3)	$\forall x \forall y [\mathbf{s}(x) \doteq \mathbf{s}(y) \Rightarrow x \doteq y]$	Axiom 2
(4)	$\mathbf{s}(\mathbf{s}(\mathbf{s}(\mathbf{0}))) \doteq \mathbf{s}(\mathbf{0})$	By (1)
(5)	$\neg \mathbf{s}(\mathbf{s}(\mathbf{0})) \doteq \mathbf{0}$	By (2)
(6)	$\mathbf{s}(\mathbf{s}(\mathbf{s}(\mathbf{0}))) \doteq \mathbf{s}(\mathbf{0}) \Rightarrow \mathbf{s}(\mathbf{s}(\mathbf{0})) \doteq \mathbf{0}$	By (3)
(7)	$\mathbf{s}(\mathbf{s}(\mathbf{0})) \doteq \mathbf{0}$	By (4) and (6)

By the same method, for any particular natural numbers m and n such that $m > n$, there is a tableau proof of the sentence

$$\neg \mathbf{m} \doteq \mathbf{n}$$

from Axioms 1 and 2 of Peano Arithmetic.

Example 3.7.3 Here is a tableau proof of the sentence

$$\mathbf{1} + \mathbf{2} \doteq \mathbf{3}$$

from Axioms 3 and 4 alone, again showing the main steps.

(1)	$\neg \mathbf{s}(\mathbf{0}) + \mathbf{s}(\mathbf{s}(\mathbf{0})) \doteq \mathbf{s}(\mathbf{s}(\mathbf{s}(\mathbf{0})))$	$\neg$ to be proved
(2)	$\forall x\, x + \mathbf{0} \doteq x$	Axiom 3
(3)	$\forall x \forall y\, x + \mathbf{s}(y) \doteq \mathbf{s}(x + y)$	Axiom 4
(4)	$\mathbf{s}(\mathbf{0}) + \mathbf{0} \doteq \mathbf{s}(\mathbf{0})$	By (2)
(5)	$\mathbf{s}(\mathbf{0}) + \mathbf{s}(\mathbf{0}) \doteq \mathbf{s}(\mathbf{s}(\mathbf{0}) + \mathbf{0})$	By (3)
(6)	$\mathbf{s}(\mathbf{0}) + \mathbf{s}(\mathbf{0}) \doteq \mathbf{s}(\mathbf{s}(\mathbf{0}))$	By (4) and (5)
(7)	$\mathbf{s}(\mathbf{0}) + \mathbf{s}(\mathbf{s}(\mathbf{0})) \doteq \mathbf{s}(\mathbf{s}(\mathbf{0}) + \mathbf{s}(\mathbf{0}))$	By (3)
(8)	$\mathbf{s}(\mathbf{0}) + \mathbf{s}(\mathbf{s}(\mathbf{0})) \doteq \mathbf{s}(\mathbf{s}(\mathbf{s}(\mathbf{0})))$	By (6) and (7)

Again, by the same method, for any three particular natural numbers m, n and p, if $m + n = p$ then the sentence

$$\mathbf{m} + \mathbf{n} \doteq \mathbf{p}$$

has a tableau proof from Axioms 3 and 4 of Peano Arithmetic.

In spite of these examples, one cannot go very far with only the six basic axioms of Peano Arithmetic. The Induction Principle is needed early and often in the study of the natural numbers.

In a formal tableau proof of a sentence from Peano Arithmetic, the cases of the Induction Principle which are needed for the proof are included in the hypothesis list. Many simple and familiar properties of the natural numbers cannot be proved without induction; that is, there is no tableau proof from the six basic axioms alone, but there is a tableau proof from the full set of axioms of Peano Arithmetic including the Induction Principle.

We now give several examples of such sentences. In each example, we first sketch a tableau proof of the sentence from Peano Arithmetic. We then show that the sentence cannot be proved from the six basic axioms alone by describing a model of the six basic axioms in which the sentence is false. It follows from the Soundness Theorem that a sentence which is false in some model of the six basic axioms cannot be provable from them. Thus at least one induction axiom is needed in any tableau proof of the sentence.

Example 3.7.4 The sentence

$$\mathbf{A}_1 : \ \forall x \neg x \doteq \mathbf{s}(x)$$

is provable from Peano Arithmetic but is not provable from the six basic axioms alone.

Proof: To prove this sentence from **PA**, we let **B** be the wff $\neg x \doteq \mathbf{s}(x)$, and prove $\mathbf{A}_1$ from the hypotheses

1. $\forall x \neg \mathbf{0} \doteq \mathbf{s}(x)$
2. $\forall x \forall y [\mathbf{s}(x) \doteq \mathbf{s}(y) \Rightarrow x \doteq y]$

3. $\mathbf{B}(\mathbf{0}) \wedge \forall x [\mathbf{B}(x) \Rightarrow \mathbf{B}(\mathbf{s}(x))] \Rightarrow \forall x \mathbf{B}(x)$

Note that $\forall x \mathbf{B}(x)$ is the same as $\mathbf{A}_1$, the formula to be proved. By hypothesis 1, $\neg \mathbf{0} \doteq \mathbf{s}(\mathbf{0})$, so $\mathbf{B}(\mathbf{0})$ holds. Let a be arbitrary and temporarily assume $\mathbf{B}(a)$, that is, $\neg a \doteq \mathbf{s}(a)$. By hypothesis 2,

$$\mathbf{s}(a) \doteq \mathbf{s}(\mathbf{s}(a)) \Rightarrow a \doteq \mathbf{s}(a).$$

By propositional logic, $\neg \mathbf{s}(a) \doteq \mathbf{s}(\mathbf{s}(a))$, that is, $\mathbf{B}(\mathbf{s}(a))$.

By the Deduction Rule, $\mathbf{B}(a) \Rightarrow \mathbf{B}(\mathbf{s}(a))$, and thus by the Generalization Rule,

$$\forall x [\mathbf{B}(x) \Rightarrow \mathbf{B}(\mathbf{s}(x))].$$

By hypothesis 3, it follows that $\forall x \mathbf{B}(x)$, which is the formula to be proved. **End of Proof.**

A formal tableau proof of the sentence $\mathbf{A}_1$ can be carried out in 12 nodes, and is included in the diskette as PEANO.TBU.

To see that sentence $\mathbf{A}_1$ is not provable from Axioms 1—6 alone, we shall describe a model $\mathcal{M}$ of Axioms 1—6 in which the sentence $\mathbf{A}_1$ is false. The universe set of the model is the set $M = \mathbf{N} \cup \{\infty\}$ formed by adding to the set $\mathbf{N}$ of natural numbers one extra element called ∞. Among elements of $\mathbf{N}$, the function symbols $+, *, \mathbf{s}, \mathbf{0}$ have their usual meaning. To complete the definition of the model, we stipulate that

$$\mathbf{s}_{\mathcal{M}}(\infty) = \infty,$$

$$x +_{\mathcal{M}} \infty = \infty +_{\mathcal{M}} x = \infty,$$

$$0 *_{\mathcal{M}} \infty = \infty *_{\mathcal{M}} 0 = 0,$$

$$x \neq 0 \Rightarrow x *_{\mathcal{M}} \infty = \infty *_{\mathcal{M}} x = \infty.$$

It can be checked that each of the six basic axioms is true in this model. However, we see that the sentence $\mathbf{A}_1$ is false in the model $\mathcal{M}$ by taking $x = \infty$.

Example 3.7.5 The sentence

$$\mathbf{A}_2 : \ \forall x \, \mathbf{0} * x \doteq \mathbf{0}$$

is provable from Peano Arithmetic but is not provable from Axioms 1—6 alone.

Proof: Here is a direct proof of $\mathbf{A}_2$ from **PA** in paragraph form. The following axioms of **PA** are needed in the proof:

1. $\forall x \;\; x + \mathbf{0} \doteq x$
2. $\forall x \;\; x * \mathbf{0} \doteq \mathbf{0}$
3. $\forall x \forall y \;\; x * \mathbf{s}(y) \doteq x * y + x$
4. $\mathbf{0} * \mathbf{0} \doteq \mathbf{0} \wedge \forall x \;\; [\mathbf{0} * x \doteq \mathbf{0} \Rightarrow \mathbf{0} * \mathbf{s}(x) \doteq \mathbf{0}] \Rightarrow \forall x \;\; \mathbf{0} * x \doteq \mathbf{0}.$

By hypothesis 1, we have $\mathbf{0} * \mathbf{0} \doteq \mathbf{0}$. We next prove that

$$\forall x \, [\mathbf{0} * x \doteq \mathbf{0} \Rightarrow \mathbf{0} * \mathbf{s}(x) \doteq \mathbf{0}].$$

Let a be arbitrary and assume that $\mathbf{0} * a \doteq \mathbf{0}$. By hypotheses 3 and 1,

$$\mathbf{0} * \mathbf{s}(a) \doteq \mathbf{0} * a + \mathbf{0} \doteq \mathbf{0} * a.$$

Then

$$\mathbf{0} * \mathbf{s}(a) \doteq \mathbf{0} * a \doteq \mathbf{0}.$$

By the Deduction and Generalization Rules,

$$\forall x \, [\mathbf{0} * x \doteq \mathbf{0} \Rightarrow \mathbf{0} * \mathbf{s}(x) \doteq \mathbf{0}].$$

Then by hypothesis 4, $\forall x \, \mathbf{0} * x \doteq \mathbf{0}$ as required. **End of Proof.**

The formal tableau proof of $\mathbf{A}_2$ from **PA** is left as Exercise 22.

To see that $\mathbf{A}_2$ is not provable from the six basic axioms alone, we modify the model $\mathcal{M}$ in the preceding example by stipulating that

$$0 *_{\mathcal{M}} \infty = 17.$$

This modified model is still a model of Axioms 1—6. (In fact, we can give $0 *_{\mathcal{M}} \infty$ any value at all and still have a model of Axioms 1—6.) To see that the sentence $\mathbf{A}_2$ is false in this model, take ∞ for x.

Example 3.7.6 The sentence

$$\mathbf{A}_3 : \; \forall x \, [x \doteq \mathbf{0} \vee \exists y \, x \doteq \mathbf{s}(y)]$$

is provable from Peano Arithmetic but not from Axioms 1—6 alone.

Proof: The proof of $\mathbf{A}_3$ from **PA** uses the single induction axiom

1. $\mathbf{B}(\mathbf{0}) \wedge \forall x \, [\mathbf{B}(x) \Rightarrow \mathbf{B}(\mathbf{s}(x))] \Rightarrow \forall x \, \mathbf{B}(x)$

where $\mathbf{B}(x)$ is the wff $x \doteq \mathbf{0} \vee \exists y \, x \doteq \mathbf{s}(y)$. Note that the formula $\mathbf{A}_3$ to be proved is $\forall x \, \mathbf{B}(x)$. $\mathbf{B}(\mathbf{0})$ is the sentence

$$\mathbf{0} \doteq \mathbf{0} \vee \exists y \, \mathbf{0} \doteq \mathbf{s}(y),$$

which follows from the equality rule $\mathbf{0} \doteq \mathbf{0}$ by propositional logic. Let a be arbitrary and assume $\mathbf{B}(a)$. The formula $\mathbf{B}(\mathbf{s}(a))$ is

$$\mathbf{s}(a) \doteq \mathbf{0} \vee \exists y \, \mathbf{s}(a) \doteq \mathbf{s}(y),$$

which is easily proved with no hypotheses.

By the Deduction and Generalization Rules,

$$\forall x \, [\mathbf{B}(x) \Rightarrow \mathbf{B}(\mathbf{s}(x))].$$

Then by the hypothesis 1., $\forall x \, \mathbf{B}(x)$, which is the formula to be proved. **End of Proof.**

The formal tableau proof of $\mathbf{A}_3$ from **PA** is left as an exercise for the student.

Example 3.7.7 The sentence

$$\mathbf{A}_4 : \; \forall x \forall y \, x + y \doteq y + x$$

is provable from Peano Arithmetic but not from the six basic axioms alone.

In the computer problem PLUS.TBU, you are asked to give a tableau proof of $\mathbf{A}_4$ from **PA**. To see that neither of the sentences $\mathbf{A}_3$ nor $\mathbf{A}_4$ is provable from Axioms 1—6 alone, we describe a model $\mathcal{M}$ of Axioms 1—6 in which each of the sentences $\mathbf{A}_3$ and $\mathbf{A}_4$ is false. The universe set of $\mathcal{M}$ is the set

$$\{a_0, a_1, a_2, \ldots\} \cup \{b_0, b_1, b_2, \ldots\} \cup \{c\}$$

made up of two "copies" of $\mathbf{N}$ and one additional element $\{c\}$. The function symbols in $\mathcal{M}$ are defined as follows:

$$\mathbf{0}_{\mathcal{M}} = a_0,$$

$$s_{\mathcal{M}}(a_n) = a_{n+1}, \quad s_{\mathcal{M}}(b_n) = b_{n+1}, \quad s_{\mathcal{M}}(c) = c,$$

$$a_m +_{\mathcal{M}} a_n = a_{m+n}, \qquad a_m +_{\mathcal{M}} b_n = b_n, \qquad b_n +_{\mathcal{M}} a_m = b_{n+m},$$

and in all other cases the sum is c.

$$a_m *_{\mathcal{M}} a_n = a_{m*n},$$

$$a_0 *_{\mathcal{M}} x = x *_{\mathcal{M}} a_0 = a_0 \text{ and } a_1 *_{\mathcal{M}} x = x *_{\mathcal{M}} a_1 = x \text{ for all } x$$

and in all other cases the product is c. The student can now check that all the basic axioms of **PA** are true in the model $\mathcal{M}$. To see that the sentence $\mathbf{A}_3$ is false in $\mathcal{M}$, take b_0 for x. To see that the sentence $\mathbf{A}_4$ is false in $\mathcal{M}$, note that

$$a_1 +_{\mathcal{M}} b_0 = b_0, \quad b_0 +_{\mathcal{M}} a_1 = b_1.$$

Using the definition given above for $+_{\mathcal{M}}$, one can see that the relation $\leq_{\mathcal{M}}$ orders the elements of M so that

$$a_0 \leq a_1 \leq \ldots \leq b_0 \leq b_1 \leq \ldots \leq c.$$

We now introduce the new symbol $\leq$ as an abbreviation as follows: For any terms σ, τ of arithmetic, we write $\sigma \leq \tau$ for the sentence [2]

$$\exists z\, \sigma + z \doteq \tau$$

where z does not occur in σ or τ.

With this symbol, the four axioms for linear order can be proved from **PA**.

[2]Note that technically we have not specified a particular sentence since any choice of z not in σ or τ satisfies the condition of the definition. However, as the reader may easily verify, for any variables x, y not in σ or τ, there is a tableau proof of the sentence $[\exists \mathbf{x}\, \sigma + \mathbf{x} \doteq \tau] \Leftrightarrow [\exists \mathbf{y}\, \sigma + \mathbf{y} \doteq \tau]$; thus, any choice of z will do.

The Reflexive Law $\forall x\, x \leq x$ is an abbreviation for the sentence $\forall x \exists z\, x + z = x$, which follows very easily from Axiom 3 of **PA**. The proofs of the other linear order axioms from **PA** are broken into small steps which are included in the Exercises at the end of this chapter.

In the remainder of this section we shall briefly discuss two other forms of arithmetic, one which is much weaker than **PA** and another which is much stronger than **PA**.

Weak Arithmetic, or **WA**, is a particular list of nine axioms which are consequences of **PA**.

Definition 3.7.8 The axioms for Weak Arithmetic consist of the six basic axioms for Peano Arithmetic together with the following three additional axioms;

7. $\forall x\, [x \leq \mathbf{0} \Rightarrow x \doteq \mathbf{0}]$

8. $\forall x \forall y\, [x \leq \mathbf{s}(y) \Rightarrow [x \leq y \vee x \doteq \mathbf{s}(y)]]$

9. $\forall x \forall y\, [x \leq y \vee y \leq x]$.

The three additional axioms 7–9 for Weak Arithmetic use the abbreviation $\leq$ but officially are sentences of the language of arithmetic. Each of these axioms can be proved from **PA**; the proofs are left to the student in the Exercises.

Axiom 7 says that no element is less than $\mathbf{0}$. Axiom 8 says that there are no elements between x and $\mathbf{s}(x)$. Axiom 9 is the Comparability Law for linear order.

Each of the sentences in Examples 3.7.4, 3.7.5, 3.7.6, and 3.7.7 is an example of a sentence which can be proved from **PA** but cannot be proved from **WA**. To see this, recall that in each example we proved the sentence from **PA** and gave a model of the six basic axioms of **PA** in which the sentence is false. In each case, the remaining three axioms of **WA** also hold in the same model.

Weak Arithmetic is a useful technical tool in the proof of the Gödel Incompleteness Theorems and the study of computable functions. It will be developed further in Chapter 5 on the way to the proof of the Gödel Incompleteness Theorem. The above examples show that many familiar facts about the natural numbers cannot be proved from

Weak Arithmetic. In spite of this, Weak Arithmetic has two important advantages. First, it has only finitely many axioms. Second, as we shall see in Chapter 5, the concepts of a wff and a tableau proof in full predicate logic can be developed within Weak Arithmetic as well as within Peano Arithmetic.

We now turn to another induction principle which is more powerful than the First Order Induction Principle of **PA**. It cannot be included in the axiom list of Peano Arithmetic because it is not a wff of first order predicate logic.

Second Order Induction Principle

for every subset $A \subset \mathbf{N}$

$$0 \in A \wedge \forall n\,[n \in A \Rightarrow (n+1) \in A] \Rightarrow \forall n\, n \in A.$$

In this principle the quantifier $\forall n$ means $\forall n \in \mathbf{N}$. The system of axioms consisting of Weak Arithmetic and the Second Order Induction Principle is sometimes called **Second Order Arithmetic**. It is more powerful than Peano Arithmetic but is not a set of sentences of first order logic.

Unlike the First Order Induction Principle, the second order version is a single axiom. However, this axiom quantifies over subsets, rather than elements, of $\mathbf{N}$, and cannot *directly* be formalized in the first order language of arithmetic.

The advantage of this second principle is that, combined with Weak Arithmetic, it captures the standard model $\mathcal{N}$ of arithmetic: If $\mathcal{M}$ is any model of **WA** having universe M, and, on replacing $\mathbf{N}$ by M, the Second Order Induction Principle is true, then $\mathcal{M}$ is **isomorphic to** $\mathcal{N}$; that is, the elements of M can be listed,

$$M = \{m_0, m_1, \ldots\}$$

so that if $+_{\mathcal{M}}$, $*_{\mathcal{M}}$, $\mathbf{s}_{\mathcal{M}}$, $\mathbf{0}_{\mathcal{M}}$ are the interpretations of the function and relation symbols of arithmetic, then we have, for all $k, \ell \in \mathbf{N}$,

$$\begin{array}{ll} \mathbf{0}_{\mathcal{M}} = m_0 & m_k +_{\mathcal{M}} m_\ell = m_{k+\ell} \\ \mathbf{s}_{\mathcal{M}}(m_k) = m_{k+1} & m_k *_{\mathcal{M}} m_\ell = m_{k\ell}. \end{array}$$

(See Theorem 3.7.9 below.)

The disadvantage of the Second Order Induction Principle is that to formalize it one must introduce a second order logic which has variables and quantifiers for predicates as well as for individuals. This logic will need additional rules of proof to take care of the quantifiers over the predicates. There will be just one induction axiom but at the price of a new list of rules of proof. A logic with quantifiers over predicates is called **second order logic**. Second order logic does not have a completeness theorem, and for this reason it has been less important than first order logic in the foundations of mathematics.

The First Order Induction Principle is a reasonable attempt to formalize the Second Order Induction Principle in our language. The idea is to "spread out" the Second Order Induction Axiom over infinitely many distinct sentences to eliminate quantification over subsets. A first attempt at spreading out this axiom would be to have, for every subset A of $\mathbf{N}$, an axiom

$$I_A : [\mathbf{0} \in A \wedge \forall x\,[x \in A \Rightarrow \mathbf{s}(x) \in A]] \Rightarrow \forall x\, x \in A.$$

If A were represented by a unary relation symbol p^A in our vocabulary, we could then write out I_A as the formal sentence

$$\mathbf{C}_A : [p^A(\mathbf{0}) \wedge \forall x\,[p^A(x) \Rightarrow p^A(\mathbf{s}(x))]] \Rightarrow \forall x\, p^A(x).$$

Now, although we have no such relation symbols in our vocabulary, we can represent many subsets A of $\mathbf{N}$ with *wffs* rather than with relation symbols. For instance, the set E of even numbers is represented by the wff $\mathbf{B}$ having only the variable x free:

$$\exists y\ \ y{+}y = x.$$

We obtain:

$$E = \{n \in \mathbf{N}\ :\ \mathcal{N} \models \mathbf{B}(x//n)\}.$$

In fact, every wff of the language determines a subset of $\mathbf{N}$ in exactly the same way. Moreover, if we replace the collection of $\mathbf{C}_A$'s with the collection of first order wffs in our formulation of the second-order axiom we obtain the First Order Induction Principle. Unfortunately, however, since there are only countably many wffs in the language (see Exercise 8) and uncountably many subsets of $\mathbf{N}$ (see Appendix A.6), "most" subsets of $\mathbf{N}$ are not accounted for by the wffs used in the first order axiom. Thus we should not expect every model of Peano Arithmetic to be isomorphic to $\mathcal{N}$. In fact, models which are not isomorphic to $\mathcal{N}$ (called **nonstandard models of arithmetic**) can be constructed using the Compactness Theorem; see Theorem 3.8.3 in the next section. By contrast

Theorem 3.7.9 (Uniqueness Theorem) *Suppose that $\mathcal{M}$ is a model for weak arithmetic and satisfies the Second Order Induction Axiom in the sense that if $A \subset M$ satisfies*

$$\mathbf{0}_{\mathcal{M}} \in A \text{ and } \forall n \in M\,[n \in A \Rightarrow (n+1) \in A]$$

then $A = M$. (Here M is the universe of the model $\mathcal{M}$.) Then $\mathcal{M}$ is isomorphic to the standard model $\mathcal{N}$ of **PA**. *In particular for any sentence* $\mathbf{A}$ *we have $\mathcal{M} \models \mathbf{A}$ if and only if $\mathcal{N} \models \mathbf{A}$.*

Proof: The assertion that $\mathcal{N}$ and $\mathcal{M}$ are isomorphic means that there is a one-one onto function

$$\phi : \mathbf{N} \to M$$

such that

$$\phi(0) = \mathbf{0}^{\mathcal{M}}, \qquad \phi(n+1) = \mathbf{s}^{\mathcal{M}}(\phi(n)), \tag{1}$$

and

$$\phi(m+n) = \phi(m) +^{\mathcal{M}} \phi(n), \qquad \phi(mn) = \phi(m) *^{\mathcal{M}} \phi(n). \tag{2}$$

The equations (1) determine ϕ uniquely by induction (on $\mathbf{N}$). Using induction again and the fact that $\mathcal{M} \models \mathbf{WA}$ we see that ϕ is one-one and that equations (2) hold. (See Exercise 25.) Finally apply the Second Order Induction Principle (for $\mathcal{M}$) to the set

$$A = \{\phi(n) : n \in \mathbf{N}\}.$$

We see that $A = M$ so that ϕ is onto. **End of Proof.**

3.8 Some Applications of Compactness

The Compactness Theorem is one of the most useful theorems in mathematical logic. In this section we shall give three applications which illustrate its usefulness.

Theorem 3.8.1 *Let* $\mathbf{H}$ *be a finite or countable set of sentences. Suppose that for each natural number n,* $\mathbf{H}$ *has a model whose universe set has more than n elements. Then* $\mathbf{H}$ *has a model whose universe set is infinite.*

Proof: For each n, let $\mathbf{E}_n$ be the sentence

$$\forall x_1 \forall x_2 \cdots \forall x_n \exists y [x_1 \neq y \wedge x_2 \neq y \wedge \cdots \wedge x_n \neq y].$$

The sentence $\mathbf{E}_n$ is true in a model $\mathcal{M}$ if and only if the universe set of $\mathcal{M}$ has more than n elements. For each n, the set

$$\mathbf{H} \cup \{\mathbf{E}_1, \mathbf{E}_2, \ldots, \mathbf{E}_n\}$$

has a model, namely any model of $\mathbf{H}$ whose universe set has more than n elements. It follows that each finite subset of the countable set of sentences

$$\mathbf{H} \cup \{\mathbf{E}_1, \mathbf{E}_2, \ldots\}$$

has a model. By the Compactness Theorem, this set of sentences has a model $\mathcal{M}$. Then $\mathcal{M}$ is a model of $\mathbf{H}$ whose universe is infinite, as required. **End of Proof.**

The next application involves groups. In the language of group theory, let x^n be the term for x multiplied by itself n times. That is, x^0 is $\mathbf{e}$, x^1 is x, and x^{n+1} is $(x^n) * x$. In a group $\mathcal{G}$, an element g has **order** n if n is the least natural number such that $g^n = \mathbf{e}$. An element g has **infinite order** if $g^n \neq \mathbf{e}$ for each natural number n.

Theorem 3.8.2 *Let* $\mathbf{H}$ *be a finite or countable set of sentences which contains all the group axioms. Suppose that for each n,* $\mathbf{H}$ *has a model $\mathcal{G}$ which has no elements of order $\leq n$ except the element* $\mathbf{e}$ *of order 0. Then* $\mathbf{H}$ *has a model in which all elements except* $\mathbf{e}$ *have infinite order.*

Proof: For each n let $\mathbf{D}_n$ be the sentence $\forall x[x^n \doteq \mathbf{e} \Rightarrow x \doteq \mathbf{e}]$. Then for each n, $\mathbf{H}$ has a model in which each of the sentences $\mathbf{D}_k, k \leq n$ is true. Therefore each finite subset of the countable set

$$\mathbf{H} \cup \{\mathbf{D}_1, \mathbf{D}_2, \ldots, \}$$

has a model. By the Compactness Theorem, this whole set has a model $\mathcal{M}$. Then $\mathcal{M}$ is a model of $\mathbf{H}$ in which all elements except $\mathbf{e}$ have infinite order.

End of Proof.

Our third application concerns models of arithmetic. By **complete arithmetic** we mean the set of all sentences in the vocabulary of **PA** which are true in the standard model $\mathcal{N}$ of arithmetic . Thus all the axioms of **PA** belong to complete arithmetic. We shall see from the Gödel Incompleteness Theorem in Chapter 5 that there are additional sentences in complete arithmetic which are not tableau provable from **PA** The following application of the Compactness Theorem shows that complete arithmetic, and hence **PA**, has nonstandard models.

Theorem 3.8.3 *There is a model $\mathcal{M}$ of complete arithmetic whose universe set M contains an element ω such that all the sentences*

$$\mathbf{0} \leq \omega, \mathbf{1} \leq \omega, \mathbf{2} \leq \omega, \ldots$$

are true in $\mathcal{M}$. (Such an element ω is called **infinite**, *and models of* **PA** *which have infinite elements are called* **nonstandard models** *of arithmetic.)*

Proof: Add a new constant symbol ω to the vocabulary of **PA**. In this expanded vocabulary, let $\mathbf{H}$ be the union of complete arithmetic and the set of sentences

$$\mathbf{0} \leq \omega, \mathbf{1} \leq \omega, \mathbf{2} \leq \omega, \ldots.$$

Every finite subset $\mathbf{H}_0$ of $\mathbf{H}$ has a model, namely the standard model $\mathcal{N}$ of arithmetic with the extra constant symbol ω interpreted by an element $m \in \mathbf{N}$ which is greater than any n such that the sentence $\mathbf{n} \leq \omega$ belongs to $\mathbf{H}_0$. By the Compactness Theorem, $\mathbf{H}$ has a model $\mathcal{M}$.

End of Proof.

3.9 Tableau Problems (TAB4)

This assignment uses the TABLEAU or TABWIN program. You will construct tableau proofs in full predicate logic. The problems are located in directory TAB4 on the distribution diskette, and the SETUP-DOS or SETUPWIN program will put them in a subdirectory called TAB4 on your hard disk. There are seven problems in this directory, called

GROUP1.TBU, GROUP2.TBU, CALC1.TBU, CALC2.TBU, CALC3.TBU, ZPLUS.TBU, PLUS.TBU.

You should load in each problem with the TABLEAU or TABWIN program, then make a proof sketch on paper, and finally use your proof sketch as a guide to make a formal tableau proof with the TABLEAU or TABWIN program. In many cases your sketch will contain a string of equations. As usual, you should save your answer on your diskette or hard drive, with the name of the problem preceded by an A.

These problems use the full predicate logic with function symbols and equality substitutions. Here are some comments on the problems. You should try the problems with shorter solutions (fewer nodes) first.

GROUP1 (16 nodes).

Hypotheses:

$$\forall x\ \forall y\ \forall z\ x * (y * z) = (x * y) * z,$$

$$\forall x\ \exists y\ x * y = e,$$

$$\forall x\ x * e = x,$$

$$\forall x\ e * x = x.$$

To prove:

$$\forall x\ \exists y\ y * x = e$$

The hypotheses are axioms from group theory with a binary infix operation $*$ and a constant symbol e for the identity element. The first hypothesis is the associative law, the second hypothesis is that every element has a right inverse, and the other two

hypotheses state that e is a two-sided identity element (actually, the fourth hypotheses can be proved from the other three). The sentence to be proved is that every element has a left inverse.

GROUP2 (21 nodes).

Hypotheses:

$$\forall x\ \forall y\ \forall z\ x * (y * z) = (x * y) * z,$$
$$\forall x\ \exists y\ x * y = e,$$
$$\forall x\ x * e = x.$$

To prove:

$$\forall x\ \forall y\ \forall z\ [x * z = y * z \Rightarrow x = y]$$

The hypotheses are the axioms for groups. The sentence to be proved is the cancellation law.

CALC1 (6 nodes).

Hypotheses:

$$\forall y\ \exists x\ f(x) = y,$$
$$\forall x\ g(f(x)) = x.$$

To prove:

$$\forall y\ f(g(y)) = y$$

The hypotheses state that the function f is onto and that g is an inverse function of f. The sentence to be proved is that f is an inverse function of g.

CALC2 (30 nodes).

Hypotheses:

$$\forall x\ \forall y\ [x < y \Rightarrow f(x) < f(y)],$$
$$\forall x\ f(x) < c,$$
$$\forall y\ [\forall x\ f(x) < y \Rightarrow c < y \vee c = y],$$
$$\neg\ \exists x\ \exists y\ [x < y \wedge y < x],$$
$$\forall x\ \forall y\ [x < y \Rightarrow \forall z\ [x < z \vee z < y]].$$

To prove:

$$\forall y\ [y < c \Rightarrow \exists x\ \forall z\ [x < z \Rightarrow y < f(z)]]$$

This is the theorem from calculus which states that a bounded increasing real function $f(x)$ approaches a limit as x approaches infinity. The vocabulary has a constant c, a unary function f, and a binary infix predicate $<$. The first hypothesis states that the function f is increasing, the second and third hypotheses state that c is the least upper bound of the range of f, and the last two hypotheses are needed facts about the order relation. The sentence to be proved states that c is the limit of $f(x)$ as x approaches infinity.

CALC3 (64 nodes).

Hypotheses:

$$\forall x\ \forall y\ \forall z\ [x \leq y \wedge y \leq z \Rightarrow x \leq z],$$
$$\forall x\ \exists y\ \neg\ y \leq x,$$
$$\forall x\ [f(x) \leq 0 \Rightarrow x \leq c],$$
$$\forall y\ [\forall x\ [f(x) \leq 0 \Rightarrow x \leq y] \Rightarrow c \leq y],$$
$$\forall x\ \forall y\ \forall z\ [p(x,y,z) \Leftrightarrow \neg y \leq x \wedge \neg z \leq y],$$
$$\forall x \forall u \forall v [p(u, f(x), v) \Rightarrow \exists s \exists t [p(s,x,t) \wedge \forall y [p(s,y,t) \Rightarrow p(u, f(y), v)]]].$$

To prove:

$$f(c) \leq 0$$

This is the main part of the the Intermediate Value Theorem from calculus. The vocabulary has constants c and 0, a unary function f, a binary infix predicate $\leq$, and a ternary predicate p. The first two hypotheses are facts about the order relation. The next two hypotheses state that c is the least upper bound of the set of all x such that $f(x) \leq 0$. The fifth hypothesis defines the relation $p(x,y,z)$ to mean that y belongs to the open interval (x,z). The long sixth hypothesis uses the relation p to state that the function

$f(x)$ is continuous for all x. The sentence to be proved is that $f(c) \leq 0$.

(A similar proof will show that $0 \leq f(c)$. This leads to the theorem that if f is continuous and $f(a) < 0 < f(b)$ then there is a point c between a and b with $f(c) = 0$.)

The problems ZPLUS and PLUS are examples of proofs using the induction principle for the natural numbers. The vocabulary has a constant 0 for zero, a unary function s for successor, and a binary function $+$ (written in infix notation $x+y$) for the sum. The hypotheses in each problem give the rules for computing the sum. The other hypotheses are cases of the induction principle for natural numbers.

ZPLUS (11 nodes).

Hypotheses:

$$\forall x\ x + 0 = x,$$

$$\forall x\ \forall y\ x + s(y) = s(x + y),$$

$$0+0 = 0 \wedge \forall x\ [0+x = x \Rightarrow 0+s(x) = s(x)] \Rightarrow \forall x\ 0+x = x.$$

To prove:

$$\forall x\ 0 + x = x$$

The third hypothesis is the induction principle for the wff $0+x = x$ in the variable x. The sentence to be proved is that for all x, $0 + x = x$.

PLUS (38 nodes).

Hypotheses:

$$\forall x\ x + 0 = x,$$

$$\forall x\ \forall y\ \ s + s(y) = s(x + y),$$

$$\forall y 0+y = y+0 \wedge [\forall y\ x+y = y+x \Rightarrow \forall y\ s(x)+y = y+s(x)]$$

$$\Rightarrow \forall x \forall y\ x + y = y + x,$$

$$\forall x[x + 0 = 0 + x \wedge \forall y[x + y = y + x \Rightarrow x + s(y) = s(y) + x]$$

$$\Rightarrow \forall y\ x + y = y + x],$$

$$\forall x\ 0 + x = x.$$

To prove:

$$\forall x\ \forall y\ x + y = y + x$$

The third and fourth hypotheses are the induction principle for $\forall y\ x + y = y + x$ in the variable x, and the induction principle for $x + y = y + x$ in the variable y. The last hypothesis is the sentence proved in the preceding problem. The sentence to be proved is the commutative law for the sum.

If you get stuck, you may look at the hints below.

HINT FOR GROUP1: If $a * b = e$ and $b * c = e$ then

$$a = a * e = a * (b * c) = (a * b) * c = e * c = c,$$

so that $b * a = e$.

HINT FOR GROUP2: If $a * c = b * c$ and $c * d = e$ then

$$a = a*e = a*(c*d) = (a*c)*d = (b*c)*d = b*(c*d) = b*e = b.$$

HINT FOR PLUS: If $\forall y\ [a + y = y + a]$ and $s(a) + b = b + s(a)$, then

$$\begin{aligned} s(a) + s(b) &= s(s(a) + b) = s(b + s(a)) = s(s(b + a)) = s(s(a + b)) = \\ &= s(a + s(b)) = s(s(b) + a) = s(b) + s(a). \end{aligned}$$

3.10 Exercises

1. Let **B** be the wff

$$y \doteq \mathbf{s}(x) \wedge \exists y\ x + y \doteq z.$$

(a) Write down the wff $\mathbf{B}(x//\mathbf{0})$.

(b) Is the term $\mathbf{s}(x)$ free for x in **B**? If it is, write down the wff $\mathbf{B}(x//\mathbf{s}(x))$.

(c) Is the term $x * y$ free for x in **B**? If it is, write down the wff $\mathbf{B}(x//x * y)$

(d) Is the term $x * y$ free for y in **B**? If it is, write down the wff $\mathbf{B}(y//x * y)$.

(e) Write down the sentence $\mathbf{B}(v)$ where v is the valuation

$$v = ((x, 2), (y, 4), (z, 6)).$$

2. Prove that for each term τ and each initial segment **U** of the string τ such that the next symbol after **U** is a function symbol f, there is a unique term σ within τ which starts with f, that is, there is a unique term σ such that $\tau = \mathbf{U}\sigma\mathbf{V}$ for some **V**.
(Hint: Similar to the proof of the Unique Readability Theorem for wffs).

3. Suppose τ is a term and σ is a term within τ, that is, $\tau = \mathbf{U}\sigma\mathbf{V}$ for some strings **U** and **V**. Prove that for every other term ρ, the string $\mathbf{U}\rho\mathbf{V}$ obtained by replacing σ by ρ in τ is also a term.
(Hint: Use the preceding Exercise. Hold σ and ρ fixed and argue by induction on the length of τ.)

4. Let **A** be a wff in full predicate logic and let **B** be a wff within **A**, that is, $\mathbf{A} = \mathbf{UBV}$ for some **U** and **V**. Prove that for every wff **C**, the string **UCV** obtained by replacing **B** by **C** in **A** is also a wff.

5. Give a tableau proof of the sentence

$$\forall x \forall y \exists z\ z \doteq f(x, y)$$

6. Give a tableau proof of the sentence

$$\forall y\ [R(y) \Leftrightarrow \exists x\ [R(x) \wedge x \doteq y]].$$

7. In the full predicate logic with a vocabulary consisting of the two binary predicate symbols $\doteq, p$, give tableau proofs of each of the Equality Axioms from Section 2.10. (You may skip the transitive law (3), which is already proved in the text as an example).

8. Suppose there are only countably many function symbols in the vocabulary of a full predicate logic and that M is a countable set. Prove that $TERM(\mathcal{F}, M)$ is a countable set. (Hint: First show that the set of all finite sequences from a countable set is countable. Then show that the set S of all symbols except for the predicate symbols is countable. Finally, show that each term is a finite sequence (or string) of symbols from S). Then show that $WFF(\mathcal{P}, \mathcal{F}, \mathcal{M})$ is countable.

9. Let $\mathbf{N} = \{0, 1, 2, \ldots\}$ be the set of natural numbers and $\phi_0, \phi_1, \phi_2, \ldots$ be a list elements of $FUN_1(\mathbf{N})$, i.e. each $\phi_n : \mathbf{N} \to \mathbf{N}$. Define a function $f : \mathbf{N} \to \mathbf{N}$ such that $f \neq \phi_n$ for all $n = 0, 1, 2, \ldots$. Conclude that the set $FUN_1(\mathbf{N})$ is not countable. Hint: See Exercise 48 on page 139.

10. This exercise gives an example of a premodel which is not a model and shows how the premodel may be transformed into a model. We use the logic of group theory whose vocabulary $\{*, \mathbf{e}\}$ consists of one binary function symbol $*$ and one constant symbol **e**. We build a premodel $\mathcal{G}_m$ for each natural number m with the following specifications:

$$\text{the universe of } \mathcal{G}_m = \mathbf{N}$$

$$*_{\mathcal{G}_m} = +$$

$$\mathbf{e}_{\mathcal{G}_m} = 0$$

$$\doteq_{\mathcal{G}_m} = \equiv_m$$

where $\equiv_m$, called **equality modulo** m, is defined by:

$$x \equiv_m y \iff x - y \text{ is divisible by } m.$$

(For instance, any two even numbers are equal modulo 2, and the following numbers are equal (in pairs) modulo 3 : $2, 5, 8, 11, \ldots$.)

(a) Show that $\mathcal{G}_m$ is a premodel satisfying the axioms of group theory, but that $\mathcal{G}_m$ is *not* a group.

(b) Notice that the elements of $\mathcal{G}_m$ can be organized in an array:

$$\begin{array}{cccc} 0, & 1, & \ldots, & m-1 \\ m, & m+1, & \ldots, & 2m-1 \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \end{array}$$

so that the elements in any column are equal modulo m (but elements from different columns are *not* equal modulo m).

Let $\mathbf{Z}_m$ consist of the elements in the top row of the matrix, i.e.,

$$\mathcal{Z}_m = \{0, 1, \ldots, m-1\}.$$

We interpret $*$ for $\mathbf{Z}_m$ by the operation $+_m$ defined by

$$i +_m j = \begin{cases} \text{the remainder obtained on dividing} \\ i + j \text{ by } m \end{cases}$$

where "$i+j$" signifies ordinary addition of natural numbers. Thus, for example,

$$2 +_5 3 = 0$$

$$3 +_6 4 = 1$$

Show that $\mathcal{Z}_m = (\mathbf{Z}_m, +_m, 0)$ forms an abelian group.

(c) Show that for all m, $\mathcal{G}_m$ and $\mathcal{Z}_m$ satisfy the same first order sentences, i.e., for all sentences $\mathbf{A}$,

$$\mathcal{G}_m \models \mathbf{A} \iff \mathcal{Z}_m \models \mathbf{A}$$

(Hint: For each $n \in \mathbf{N}$, let $\bar{n}$ denote the remainder obtained on dividing n by m. Show that for each wff $\mathbf{A}$ with free variables $x_1, \ldots, x_r$, and for all natural numbers $n_1, \ldots, n_r$,

$$\mathcal{G}_m \models \mathbf{A}(n_1, \ldots, n_r) \iff \mathcal{Z}_m \models \mathbf{A}(\bar{n}_1, \ldots, \bar{n}_r).$$

Do this by induction on the wff $\mathbf{A}$.

(d) Treat $\mathcal{Z}_m$ as a model of the language of arithmetic by interpreting the multiplication symbol $*$ as $*_m$, defined by

$$i *_m j = \begin{cases} \text{the remainder obtained on dividing} \\ i * j \text{ by } m \end{cases}$$

Does $\mathcal{Z}_m$ satisfy **WA**? Which axioms are satisfied and which fail? Does your answer depend on the choice of m? What is the "right" way to define multiplication on $\mathcal{G}_m$ so that it too becomes a model of the language of arithmetic satisfying the same sentences as $\mathcal{Z}_m$?

11. Recall from page 161 that $(S(X), \circ, I_X)$ denotes the permutation group of the set X. Show that for any set X with more than two elements, this group is not abelian. (Why must $S(X)$ be abelian if X has at most two elements?)

12. Prove that every model of the sentence

$$[\forall x \neg \mathbf{s}(x) \doteq \mathbf{0} \wedge \forall x \forall y [\mathbf{s}(x) \doteq \mathbf{s}(y) \Rightarrow x \doteq y]]$$

of full predicate logic has an infinite universe.

13. Give a tableau proof from **PA** of the wff

$$\forall x [x \doteq \mathbf{0} \vee \exists y\, x \doteq \mathbf{s}(y)]$$

(Hint: An informal proof was given in Example 3.7.6. The tableau proof involves the use of the induction axiom where $\mathbf{B}(x)$ is the wff $x \doteq \mathbf{0} \vee \exists y\, x \doteq \mathrm{s}(y)$.)

14. Show that Axiom 7 of **WA**,

$$\forall x\, [x \leq \mathbf{0} \Rightarrow x \doteq \mathbf{0}],$$

is provable from **PA**, by proving it from the six basic axioms and the sentence

$$\forall x\, [x \doteq \mathbf{0} \vee \exists y\, x \doteq \mathrm{s}(y)]$$

from the preceding Exercise.

15. Show that Axiom 8 of **WA**,

$$\forall x \forall y\, [x \leq \mathrm{s}(y) \Rightarrow [x \leq y \vee x \doteq \mathrm{s}(y)]],$$

is provable from **PA** by proving it from the six basic axioms and the sentence from Example 3.7.6.

16. Prove that Axiom 9 of **WA**, the Comparability Law

$$\forall x \forall y [x \leq y \vee y \leq x],$$

is provable from **PA**. In addition to the axioms of **PA**, you may use the Commutative Law for Addition

$$\forall x \forall y\, x + y \doteq y + x$$

from the computer problem PLUS.TBU and Axiom 8 which is proved from **PA** in the preceding exercise.

17. Show that the sentence

$$\forall x \forall y \forall z [x + y \doteq x + z \Rightarrow y \doteq z]$$

is provable from **PA**. (Hint: Prove it from the axioms of **PA** and the Commutative Law for Addition from computer problem PLUS.TBU).

18. Show that there is a tableau proof from **PA** of the associative law of addition,

$$\forall x \forall y \forall z\, (x + y) + z = x + (y + z).$$

(Hint: First come up with an informal proof from the Peano axioms using induction. Then translate it into a tableau proof.)

19. Show that the Transitive Law

$$\forall x \forall y \forall z\, [x \leq y \wedge y \leq z \Rightarrow x \leq z]$$

is provable from **PA**. (Hint: Prove it from the Associative Law of Addition.)

20. Show that the Antisymmetric Law

$$\forall x \forall y\, [x \leq y \wedge y \leq x \Rightarrow x \doteq y]$$

is provable from **PA**. (Hint: Prove it from the basic axioms of **PA**, the Associative Law of Addition, and the sentence from Exercise 17).

21. Give a tableau proof of the sentence

$$\forall x\, [\mathbf{0} * x \doteq \mathbf{0} \Rightarrow \mathbf{0} * \mathrm{s}(x) \doteq \mathbf{0}]$$

from the set of hypotheses

$$\forall x \;\; x + \mathbf{0} \doteq x$$

$$\forall x \forall y \;\; x * \mathrm{s}(y) \doteq x * y + x$$

22. Give a tableau proof of the sentence

$$\forall x \;\; \mathbf{0} * x \doteq \mathbf{0}$$

from Peano arithmetic. Here is a start (showing only the axioms of Peano arithmetic which are needed for your proof).

$$\neg \forall x \;\; \mathbf{0} * x \doteq \mathbf{0}$$

$$\forall x \;\; x + \mathbf{0} \doteq x$$

$$\forall x \;\; x * \mathbf{0} \doteq \mathbf{0}$$

$$\forall x \forall y \;\; x * \mathbf{s}(y) \doteq x * y + x$$

$$\mathbf{0} * \mathbf{0} \doteq \mathbf{0} \wedge \forall x \;\; [\mathbf{0} * x \doteq \mathbf{0} \Rightarrow \mathbf{0} * \mathbf{s}(x) \doteq \mathbf{0}] \Rightarrow \forall x \;\; \mathbf{0} * x \doteq \mathbf{0}$$

23. Show that the Distributive Law

$$\forall x \forall y \forall z \; (x + y) * z = x * y + x * z$$

is tableau provable from **PA**. You may use extra rules of proof such as the Generalization and Deduction Rules, as well as the commutative and associative laws for addition, which were proved from **PA** in earlier exercises.

24. Give a tableau proof of the "strong induction principle"

$$\forall x \, [\forall y \, [y < x \Rightarrow P(y)] \Rightarrow P(x)] \Rightarrow \forall x \, P(x)$$

from the three hypotheses

$$\forall x \neg x < \mathbf{0},$$

$$\forall x \, \forall y \, [x < \mathbf{s}(y) \wedge \neg x < y \Rightarrow x \doteq y],$$

$$\mathbf{B}(\mathbf{0}) \wedge \forall x \, [\mathbf{B}(x) \Rightarrow \mathbf{B}(\mathbf{s}(x))] \Rightarrow \forall x \, \mathbf{B}(x),$$

where $\mathbf{B}(x)$ is the particular wff $\forall y \, [y < x \Rightarrow P(y)]$.

25. Supply the missing details in the proof of Theorem 3.7.9.

26. The following argument purports to prove that any two models for Peano arithmetic are isomorphic.

> Let $\mathcal{M}$ be a model for Peano arithmetic with universe set M and define a map
>
> $$f : \mathbf{N} \to M$$

> by induction:
>
> $$f(0) \doteq \mathbf{0}^{\mathcal{M}}, \;\; f(n+1) \doteq \mathbf{s}^{\mathcal{M}}(f(n)).$$
>
> Since
>
> $$\mathcal{M} \models \forall x \forall y [\mathbf{s}(x) \doteq \mathbf{s}(y) \Rightarrow x \doteq y]$$
>
> it follows by induction that f is one-one. Since $\mathbf{0}^{\mathcal{M}} \in f(\mathbf{N})$ and $\mathbf{s}(u) \in f(\mathbf{N})$ whenever $u \in f(\mathbf{N})$ and since $\mathcal{M}$ models the Induction Principle, it follows that $M = f(\mathbf{N})$, that is, that f is onto. Finally the formulas
>
> $$\begin{aligned} f(0) &= \mathbf{0}^{\mathcal{M}} \\ f(\mathbf{s}(x)) &= \mathbf{s}^{\mathcal{M}}(f(x)) \\ f(x+y) &= f(x) +^{\mathcal{M}} f(y) \\ f(x*y) &= f(x) *^{\mathcal{M}} f(y) \end{aligned}$$
>
> hold, the first two by definition and the last two by induction.

The argument is wrong: Theorem 3.8.3 provides a counter-example. Where is the error?

The following four problems use the Compactness Theorem.

27. Prove that for every set P of prime numbers, there is a model $\mathcal{M}$ of complete arithmetic and an element $a \in M$ such that

$$\mathcal{M} \models \exists x \mathbf{p} * x \doteq a$$

for each prime $p \in P$, and

$$\mathcal{M} \models \neg \exists x \mathbf{p} * x \doteq a$$

for each prime $p \notin P$.

28. Let $\mathbf{H}$ be a finite or countable set of sentences in the language of group theory. Suppose that for each natural number n, $\mathbf{H}$ has a model

which has at least one element of order $\geq n$. Then **H** has a model which has at least one element of infinite order.

29. Let **H** be a set of sentences which contains the axioms for linear order. Suppose **H** has an infinite model. Prove that **H** has a model $\mathcal{M}$ in which there is a countable strictly increasing sequence of elements, that is, there are elements $a_1, a_2, a_3, \ldots \in M$ such that

$$\mathcal{M} \models a_1 < a_2, \mathcal{M} \models a_2 < a_3, \ldots.$$

30. Let **H** be the set of all sentences in the vocabulary $\{\mathbf{0}, 1, \leq, +, *\}$ which are true of the real numbers. Prove that **H** has a model $\mathcal{M}$ with an element ϵ such that $\mathcal{M} \models \mathbf{0} < \epsilon$ but for each natural number n, $\mathcal{M} \models n * \epsilon < 1$. (n is the term formed by adding 1 to itself n times).

Chapter 4

Computable Functions

4.1 Introduction

Consider the following two statements:

1. For any two positive integers m and n, there is a largest integer g which is a factor of m and n.

2. For any two positive integers $m > n$, if m is divided by n obtaining a remainder r, and n is divided by r obtaining a remainder s, and r is divided by s obtaining a remainder t, and so forth, stopping the first time the remainder is zero, then the last nonzero remainder that arises in this process is the largest factor of m and n.

The first of these statements merely asserts the *existence* of the greatest common divisor (*gcd*) of any two positive integers; the second actually gives a procedure to *construct* g. Moreover, this procedure is *mechanical* in the sense that a computer can be programmed to carry out these instructions.

The procedure given in the second statement is known as the **Euclidean algorithm**. An **algorithm** is a finite set of instructions which, when applied to an appropriate input, dictates a unique sequence of simple operations to be applied to the input. For some inputs, the sequence of operations will come to a halt and an **output** will be given; for others, the sequence of operations on the given input may never terminate and there will be no output. The Euclidean algorithm accepts as input any pair of positive integers $m > n$ and in every case produces

an output (namely, $gcd(m,n)$).

As another example, consider the following algorithm **R**: **R** accepts natural numbers as inputs; with input n **R** checks to determine whether $n = 0$ and if so, outputs 0; if not, **R** adds 1 to n and repeats the procedure. Clearly, **R** outputs 0 with input 0 and, with input $n > 0$, continues adding 1 to n forever and gives no output at all.

The two algorithms described above define functions in a natural way. The Euclidean algorithm defines a total function $(m,n) \mapsto \gcd(m,n)$ from pairs of positive integers to positive integers. The second algorithm defines the following "partial" function f:

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ \textit{undefined} & \text{if } n > 0 \end{cases}$$

Which functions are computed by algorithms? In this chapter we provide an answer to that question by devising a simple computing machine, called an **unlimited register machine**, which will execute programs especially designed to run on this machine; these programs will be called **RM programs**. Given a subset S of $\mathbf{N}^n$, a function $f : S \to \mathbf{N}$ will be called **RM computable** if there is an RM program which outputs $f(a_1,\ldots,a_n)$ when it runs with input $(a_1,\ldots,a_n) \in S$, and gives no output for any input lying outside S.

We will find that virtually all functions $S \to \mathbf{N}$ that come up in mathematical practice are RM computable. In addition we will show how finite sequences — and as a result RM programs themselves — can be coded as single natural numbers. We will then be able to construct a universal RM program UNIV which will be able to execute every RM program on any input: If **P** is an RM program which is coded by the number e, the program UNIV will accept as input all pairs of numbers (e,n) and will output the number which **P** outputs on input n (or, if **P** gives no output with input n, then UNIV gives no output with input (e,n)). The universal RM program will allow us to find examples of functions which are not RM computable and will provide examples of "unsolvable" problems.

There are three sets of problems at the end of this chapter. The first two problem sets use the GNUMBER program and are done on a computer. The third problem set contains ordinary pencil and paper problems.

4.2 Numerical Functions and Relations

A **numerical function** is a function f defined on a set of n-tuples of natural numbers:

$$Dom(f) \subset \mathbf{N}^n$$

and taking natural numbers as values:

$$Ran(f) \subset \mathbf{N}.$$

The positive integer n is called the **arity** of the numerical function; it is the number of inputs $x_1, x_2, \ldots, x_n$ required to produce an output $f(x_1, x_2, \ldots, x_n)$. A numerical function with arity n is also called an **n-ary numerical function**, or simply an **n-ary function on N**. (This usage arose from more traditional terminology where *unary* meant 1-ary, *binary* meant 2-ary, *ternary* meant 3-ary, etc.)

If $Dom(f) = \mathbf{N}^n$, f is called a **total function**; if $Dom(f)$ is a subset of $\mathbf{N}^n$, f is called a **partial function**. By "function" we will mean "*total* function" although we will occasionally refer to a function redundantly as a total function if we want to emphasize that its domain is all of $\mathbf{N}^n$. Since $\mathbf{N}^n$ is a subset of itself, every total function is also a partial function, that is, the set of total functions is a subset of the set of partial functions.

An **n-ary numerical relation** is any subset of $\mathbf{N}^n$; note that the graph of an n-ary function (partial or total) is an $n+1$-ary relation. A relation R is determined by its **characteristic function** which is the numerical function c_R defined by

$$c_R(x_1, x_2, \ldots, x_n) = \begin{cases} 1 & \text{if } (x_1, x_2, \ldots, x_n) \in R \\ 0 & \text{if } (x_1, x_2, \ldots, x_n) \notin R \end{cases}$$

for $(x_1, x_2, \ldots, x_n) \in \mathbf{N}^n$. An important difference between numerical functions and numerical relations is that by convention, *relations are always assumed to be totally defined.* Therefore the characteristic function c_R of a numerical relation is always total.

In Chapter 3 we saw examples of total and partial numerical functions: addition and multiplication are total binary functions; in this chapter we call these functions *Add* and *Mult* , respectively. Also, subtraction and division are partial binary functions ; recall that we denote

subtraction by *Subt* and division by *Divide* . Another useful pair of partial functions, which we denote by *Div* and *Remain* , is given by the division algorithm as follows:

$$q = Div(x,y), \quad r = Remain(x,y)$$

if and only if

$$x = qy + r, \quad 0 \leq r < y$$

with domain $\{(x,y) \in \mathbf{N}^2 : y > 0\}$.

We will be interested in extending partial functions to make them total. We give some examples below; we define

- Cut-off subtraction by

$$x \dot{-} y = \begin{cases} x - y & \text{if } y \leq x \\ 0 & \text{if } x < y \end{cases}$$

for $(x,y) \in \mathbf{N}^2$.

- The quotient function by

$$qt(x,y) = \begin{cases} Div(x,y) & \text{if } y > 0 \\ 0 & \text{if } y = 0 \end{cases}$$

for $(x,y) \in \mathbf{N}^2$.

- The remainder function by

$$rm(x,y) = \begin{cases} Remain(x,y) & \text{if } y > 0 \\ 0 & \text{if } y = 0 \end{cases}$$

for $(x,y) \in \mathbf{N}^2$.

WARNING: In the theory of computable functions the domain of a partial function plays an important role. Typically an n-tuple $(x_1, x_2, \ldots, x_n)$ is *not* in the domain of some computable function f because the program which computes $f(x_1, x_2, \ldots, x_n)$ does not terminate normally when the input is $(x_1, x_2, \ldots, x_n)$: it goes into an infinite loop.

The total functions $\dot{-}$, rm, and qt in the above examples turn out to be RM computable. However, it can happen (as we shall see later)

that the total function F defined from the partial function f by the prescription

$$F(x_1, x_2, \ldots, x_n) = \begin{cases} f(x_1, x_2, \ldots, x_n) & \text{if } (x_1, x_2, \ldots, x_n) \in Dom(f) \\ 0 & \text{otherwise.} \end{cases}$$

will *not* be computable, even though f *is* computable.

4.3 The Unlimited Register Machine

In this section we shall describe an abstract computer called the **unlimited Register Machine** or simply **register machine** (RM). It differs from real computers in three ways.

- First, the instruction set of an RM is much smaller than that of a real computer. This makes the RM much easier to study than a real computer (although it also makes the RM less efficient than a real computer), but does not in principle restrict the computing power of the RM; we shall see that the RM can compute anything a more complicated computer can.
- Second, the RM has an infinite memory: it has infinitely many data registers, and infinitely many instruction registers which hold the program instructions. Moreover each register can hold an arbitrarily large number. This idealization makes the RM easy to study and is not as far removed from reality as one might think: any particular calculation on an RM will use only a finite amount of memory, so any particular calculation which can be done by an RM can in principle be performed by a real computer with a large enough finite memory.
- Third, program memory is disjoint from data memory.

The register machine has two countable lists of registers, the **instruction registers** $\mathbf{I}_0, \mathbf{I}_1, \mathbf{I}_2, \ldots$ and the **data registers** $R_1, R_2, R_3, \ldots$. In addition, there is one more register R_0, called the **program counter**. Each instruction register $\mathbf{I}_n$ holds an instruction I_n which is loaded prior

to the execution of a program and does not change. However, all but finitely many of the instruction registers hold the halt instruction H. At any given time in the execution of a program, the program counter and all the data registers hold natural numbers, with all but finitely many of the data registers holding 0. The contents of these registers may change during execution of a program. The program counter R_0 contains the index of the next instruction to be executed, and is initially set to 0 so that the program starts with the instruction I_0.

The RM recognizes the following five kinds of instructions:

(H) Halt Instruction: There is a single halt instruction H which causes the RM to stop execution.

(Z) Zero Instructions: For each $n = 1, 2, \ldots$ there is a zero instruction (Z, n) which causes the RM to set the contents of register R_n to 0, and to increment by 1 the contents of the program counter R_0, leaving the other registers unaltered.

(S) Successor Instructions: For each $n = 1, 2, \ldots$ there is a successor instruction (S, n) which causes the RM to increment by 1 the contents of the register R_n, and to increment by 1 the contents of the program counter R_0, leaving the other registers unaltered.

(T) Transfer Instructions: For each $m = 1, 2, \ldots$ and $n = 1, 2, \ldots$ there is a transfer instruction (T, m, n) which causes the RM to replace the contents of the register R_n by the contents of the register R_m (i.e. transfer R_m to R_n), and to increment by 1 the contents of the program counter R_0, leaving the other registers (including R_m) unaltered.

(J) Jump Instructions: For each $m = 1, 2, \ldots$, each $n = 1, 2, \ldots$, and each $q = 0, 1, 2, \ldots$ there is a jump instruction (J, m, n, q) which causes the RM to put the number q into the program counter R_0 (resulting in a jump to the q-th instruction) if the contents of the registers R_m and R_n are equal, and to increment by 1 the contents of the program counter R_0 otherwise. A jump instruction does not alter any data registers $R_n, n \geq 1$.

H	(do nothing)
(Z, n)	$r_n := 0$, $r_0 := r_0 + 1$
(S, n)	$r_n := r_n + 1$, $r_0 := r_0 + 1$
(T, m, n)	$r_n := r_m$, $r_0 := r_0 + 1$
(J, m, n, q)	if $r_m = r_n$ then $r_0 := q$ else $r_0 := r_0 + 1$

Table 4.1: The RM machine

An **RM-program** is a finite sequence

$$\mathbf{P} = (I_0, I_1, I_2, \ldots, I_p)$$

of such instructions, with the understanding that all the later instructions $I_{p+1}, I_{p+2}, \ldots$ are halt instructions H.

If a program **P** is loaded into the RM's program memory, the data registers $R_1, R_2, \ldots$ are given initial values, and the RM is given the command to start computing, the RM first puts a 0 in the program counter R_0. It then keeps repeating the following procedure: Look up the number r_0 currently in the program counter R_0, and execute the corresponding instruction I_{r_0}, modifying the appropriate data registers and program counter as required. It continues this process until it encounters a halt instruction, at which point the RM stops.

It is possible (even likely) that a program will not stop at all (for example, the program consisting of the single instruction $I_0 = (J, 5, 5, 0)$).

The RM instructions are summarized in Table 4.1. In this table the column on the left gives the instruction and the column on the right gives the result of executing the instruction in conventional programming notation. Here the lower case letter r_n indicates the contents of register R_n and r_0 indicates the value of the program counter.

Note that program memory is indexed starting at 0, i.e. the instructions are numbered $I_0, I_1, I_2, \ldots$ whereas the data memory is indexed starting at 1, i.e. the data registers are numbered $R_1, R_2, \ldots$, reserving R_0 for the program counter.

Each particular RM program **P** uses only finitely many data registers. If ℓ is the largest data register index mentioned in the program instructions, then the program will never use the data registers R_m for $m > \ell$, no matter what the initial register contents were. That is, if

$m > \ell$ then the contents of R_m will never change and will never affect the contents of another register during program execution.

For a given program **P**, the **state** of the register machine is a sequence of natural numbers $(r_0, r_1, r_2, \ldots, r_\ell)$ where r_0 is the program counter contents, ℓ is the highest data register index which appears in the program instructions, and $r_1, \ldots, r_\ell$ are the contents of the data registers $R_1, \ldots, R_\ell$. Since the program begins execution with instruction I_0, the initial state is a sequence $(0, r_1, \ldots, r_\ell)$ with zeroth term 0. The state at time $t+1$ is completely determined by the state at time t. It is sometimes useful to think of the program as a rule for changing from one state to another. Thus the program **P** gives rise to a function

$$NXSTATE_{\mathbf{P}} : \mathbf{N}^{\ell+1} \to \mathbf{N}^{\ell+1}$$

where $(r_0, r_1, \ldots, r_\ell) \in \mathbf{N}^{\ell+1}$ is the state before instruction I_{r_0} is executed, and $NXSTATE_{\mathbf{P}}(r_0, r_1, \ldots, r_\ell)$ is the state after execution. This function is sometimes called the **nextstate function.**

We do not count the infinite sequence of halt instructions at the end as part of the program, so that a program will be a finite rather than an infinite sequence of instructions. We say that two RM programs

$$P = (I_0, \ldots, I_p),\ Q = (J_0, \ldots, J_q)$$

are **equivalent** if they are the same except for a different finite number of halt instructions at the end; that is, if $p \leq q$,

$$I_0 = J_0, \ldots, I_p = J_p, \text{ and } J_{p+1} = H, \ldots, J_q = H.$$

Two equivalent RM programs will have exactly the same computations and will be displayed alike by GNUMBER. Given an RM program **P**, the smallest RM program which is equivalent to **P** is the program consisting of all instructions of **P** up through the last nonhalt instruction. We regard the empty sequence as an RM program equivalent to an RM program which has only Halt instructions.

4.4 RM computability

In this section we study functions which are computed by register machine programs.

Register machines have no special provision for input or output. Instead we consider the input to the RM to be the sequence of values in the registers $R_1, R_2, \ldots$ when the RM starts, and the output from the RM to be the value in the register R_1 if and when the RM halts. (Sometimes we allow two outputs, say R_1 and R_2, although this should really be regarded as computing two different functions at the same time, or as computing one function with values in $\mathbf{N}^2$.)

We now give a formal definition of RM computable functions and relations. In the sequel, we shall say that an RM program **P** halts on some input when we actually mean to say that when running the program **P** with the given input, the RM eventually halts.

An RM program **P** computes an n-ary partial numerical function $\Phi_{\mathbf{P}}^{(n)}$ as follows:

- The domain $Dom(\Phi_{\mathbf{P}}^{(n)})$ of $\Phi_{\mathbf{P}}^{(n)}$ is the set of all n-tuples
$$(a_1, a_2, \ldots, a_n) \in \mathbf{N}^n$$
such that the program **P** eventually halts if it is started with register R_j set to a_j for $j = 1, 2, \ldots, n$ and all other registers set to 0.

- For any n-tuple $(a_1, a_2, \ldots, a_n) \in Dom(\Phi_{\mathbf{P}}^{(n)})$ the value
$$\Phi_{\mathbf{P}}^{(n)}(a_1, a_2, \ldots, a_n)$$
is the number in register R_1 when the program **P** halts (after it has been started as above).

Recall that the **characteristic function** of an n-ary relation R on $\mathbf{N}$ is the total function C_R from $\mathbf{N}^n$ into the set $\{0, 1\}$ defined by
$C_R(x_1, \ldots, x_n) = 1$ if $R(x_1, \ldots, x_n)$ is true,
$C_R(x_1, \ldots, x_n) = 0$ if $R(x_1, \ldots, x_n)$ is false.

Definition 4.4.1 An n-ary numerical function f is called **RM computable** if there is an RM program **P** which computes f; that is, if there is a program **P** with

$$f = \Phi_{\mathbf{P}}^{(n)}.$$

An n-ary relation R is called RM computable if its characteristic function C_R is RM computable.

4.5 Examples of RM-Computable Functions

In this section we give some simple examples of RM-computable functions.

Example 4.5.1 The addition function *Add* is defined by

$$Add(x,y) = x + y$$

for $(x,y) \in \mathbf{N}^2 = Dom(Add)$. It is RM-computable.

Example 4.5.2 The multiplication function *Mult* is defined by

$$Mult(x,y) = x * y$$

for $(x,y) \in \mathbf{N}^2 = Dom(Mult)$. It is RM-computable.

Example 4.5.3 The predecessor function *Pred* is defined by

$$Pred(x) = \begin{cases} x-1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \end{cases}$$

It is RM-computable.

Example 4.5.4 The cut-off subtraction function *DotMinus* is defined by

$$DotMinus(x,y) = \begin{cases} x-y & \text{if } y \leq x \\ 0 & \text{if } x < y \end{cases}$$

for $(x,y) \in \mathbf{N}^2 = Dom(DotMinus)$. It is RM-computable. We often write $x \dot{-} y$ for $DotMinus(x,y)$:

$$x \dot{-} y = DotMinus(x,y).$$

Example 4.5.5 The functions *Div* and *Remain* are defined by

$$Div(x,y) = q, \quad Remain(x,y) = r,$$

if

$$x = qy + r, \text{ where } 0 \leq r < y$$

for $(x,y) \in \mathbf{N}^2$ and $y \neq 0$. They are undefined when $y = 0$. Both functions are RM-computable.

We now proceed to the RM programs to compute the functions in the above list. Each program is presented in three forms: in "pseudocode," in "assembly code," and in "machine code."

Pseudocode is useful for writing a first version of a program. It lists the main steps of a program in English, and may contain loops and if-then tests. Certain conventions will be followed. The letters `a,b,c,` ... will correspond to the contents of the first registers $R_1, R_2, R_3, \ldots$. Other "variables" correspond to other register contents and have names which suggest how they are used in the program. A pseudocode listing will begin with the program name and the intended input and output of the program. Sometimes the name of an earlier RM program with indicated inputs will appear as a line within a new program. The start of a loop will be indicated by a line such as

```
do until s = t,
```

and the end of the loop will be indicated by a single line

```
loop.
```

The program will repeat the intervening sequence of steps (the loop) until `s = t` becomes true, and will then go on to the line after the loop. If `s = t` is already true the first time the loop is encountered, the loop is never executed. If `s = t` never becomes true, the loop will be repeated forever and the computation will never halt.

The **assembly code** for a program matches the final RM program line by line, but uses descriptive names instead of numbers for the register contents and the targets of jump instructions. The assembly code has two kinds of jump instructions, the ordinary jump with three arguments as in an RM program, and an *unconditional jump* with just one argument, which always causes the program to jump to the target line. The final RM program in **machine code** is listed next to the assembly code.

The assembly code is translated into machine code (the final RM program) in a routine manner. For example, let us go through this translation for the ADD program. The labels "LOOP" and "DONE" stand for two of the instruction numbers:

$$\text{LOOP} = 1, \text{DONE} = 5.$$

(The only instructions which need labels are those which appear somewhere in the program as jump targets). The labels "a," "b," and "count" in the assembly code stand for register numbers. Choose a register number for each of these labels:

$$a = 1, b = 2, c = 3.$$

To form the machine code RM program, first write down the instruction numbers 0 through 5, then copy the instruction letters from the assembly code, then insert a pair 1 1 after each unconditional jump instruction to make it into an ordinary RM jump instruction, and finally replace each label by the corresponding jump target or register number. (A different choice of register numbers would give another RM program which does the same thing but in different registers.)

ADD

```
program ADD(a,b)
  input: a = x, b = y
  output:  a = x + y

    let count = 0
    do until count = b
      let a = a + 1
      let count = count + 1
    loop
end of program ADD
```

```
        Z   count              0:  Z 3
LOOP    J   count,b, DONE      1:  J 3 2 5
        S   a                  2:  S 1
        S   count              3:  S 3
        J   LOOP               4:  J 1 1 1
DONE    H                      5:  H
```

Figure 4.1: Pseudocode, assembly code, machine code for ADD

MULT

```
program MULT(a,b)
  input: a = x,  b = y
  output:  a = x * y

     let accum = 0
     let i = 0
       do  until  i = b
         let i = i + 1
         ADD(accum,a)
       loop
       let a=accum
end of program MULT
```

```
        Z   accum          0: Z 3
        Z   i              1: Z 4
LOOP    J   b,i, DONE      2: J 2 4 10
        S   i              3: S 4
        Z   count          4: Z 5
ALOOP   J   count, a, ADONE 5: J 5 1 9
        S   accum          6: S 3
        S   count          7: S 5
        J   ALOOP          8: J 1 1 5
ADONE   J   LOOP           9: J 1 1 2
DONE    T   accum,a       10: T 3 1
        H                 11: H
```

Figure 4.2: Pseudocode, assembly code, machine code for MULT

PRED

```
program PRED(a)
  input:  a = x
  output: a = x - 1 if x > 0,
          a = 0     if x = 0.

     if a = 0 then halt
     let prev = 0
     let next = 1
     do  until  a = next
       let next = next + 1
       let prev = prev + 1
     loop
     let a = prev
end of program PRED
```

```
        Z   prev           0: Z 2
        J   a,prev,DONE    1: J 1 2 8
        Z   next           2: Z 3
        S   next           3: S 3
LOOP    J   a,next,DONE    4: J 1 3 8
        S   next           5: S 3
        S   prev           6: S 2
        J   LOOP           7: J 1 1 4
DONE    T   prev,a         8: T 2 1
        H                  9: H
```

Figure 4.3: Pseudocode, assembly code, machine code for PRED

DOTMINUS

```
program DOTMINUS(a,b)
  input:   a = x, b = y
  output:  a = x-y  if x>y
           a = 0    otherwise

     let count = 0
     do until count = b
        PRED(a)
        let count = count + 1
     loop
end of program DOTMINUS
```

```
          Z   count              0: Z 3
  LOOP    J   count,b, DONE      1: J 3 2 13
          Z   prev               2: Z 5
          J   a,prev,PDONE       3: J 1 5 10
          Z   next               4: Z 4
          S   next               5: S 4
  PLOOP   J   a,next, PDONE      6: J 1 4 10
          S   next               7: S 4
          S   prev               7: S 5
          J   PLOOP              8: J 1 1 6
  PDONE   T   prev, a            9: T 5 1
          S   count             10: S 3
          J   LOOP              11: J 1 1 1
  DONE    H                     12: H
```

Figure 4.4: Pseudocode, assembly code, machine code for DOTMINUS

DIVREM

```
program DIVREM(a,b)
  input:   a = x, b = y
  output:  a = q, b = r  where
           x = qy + r and 0 <= r < y.
           (undefined if y=0)

     if b = 0 then hang
     let (count, q, r) = (0,0,0)
     do until count = a
        if r = b then let (q, r) = (q+1, 0)
        else          let r = r+1
        let count = count+1
     loop
     let (a,b) = (q,r)
end of program DIVREM
```

```
          Z   count              0: Z  5
  HANG    J   b,count, HANG      1: J  2  5  1
          Z   q                  2: Z  3
          Z   r                  3: Z  4
  TEST    J   r,b, INCQ          4: J  4  2  9
          J   count,a, DONE      5: J  5  1  12
          S   r                  6: S  4
          S   count              7: S  5
          J   TEST               8: J  1  1  4
  INCQ    S   q                  9: S  3
          Z   r                 10: Z  4
          J   TEST              11: J  1  1  4
  DONE    T   q, a              12: T  3  1
          T   r, b              13: T  4  2
```

Figure 4.5: Pseudocode, assembly code, machine code for DIVREM

4.6 Gödel Numbers, Extract, and Put

In this section we introduce a way of representing finite sequences of natural numbers by single natural numbers. This scheme is called a Gödel numbering scheme, and will be used in the construction of a universal RM program. The GNUMBER program has a built-in Gödel numbering scheme which uses the even decimal positions (starting from 0 on the left) as markers to show where a new term begins, and uses the odd decimal positions for the digits of the terms in the sequence to be coded. A 2 marker means that a new term is beginning, and a 1 marker means that the old term is continuing. We take 0 to be the Gödel number of the empty sequence. For example, the Gödel number (or **G.N.**) of the sequence

$$(54, 6, 217)$$

is (with the original digits in large type)

$$2\,5\,1\,4\,2\,6\,2\,2\,1\,1\,1\,7.$$

This is the Gödel number in **standard form**, or the **standard** Gödel number. In order to make every number a Gödel number of some sequence, we adopt the convention that any single digit number, 0 through 9, is taken to be a Gödel number of the empty sequence. For numbers with two or more digits, we treat every digit in an even position (starting from 0 on the left) as a marker. The initial digit can be any digit except 0 and is the first marker. Any marker > 2 has the same effect as a 2 and starts a term of the sequence. Any 0 marker has the same effect as a 1 and continues a term. An extra marker at the end is ignored. After computing the sequence, any initial zeros which may appear in a term are ignored. For example, the natural number

$$1\,5\,1\,6\,3\,0\,1\,0\,1\,9\,0\,7\,4.$$

is a Gödel number of the sequence

$$(56, 97).$$

Let $\mathbf{N}^*$ denote the set of all finite sequences of natural numbers:

$$\mathbf{N}^* = \bigcup_{n=0}^{\infty} \mathbf{N}^n$$

where $\mathbf{N}^n$ is the set of sequences of length n (and $\mathbf{N}^0$ is the singleton whose only element is the empty sequence). Define two functions

$$\# : \mathbf{N}^* \to \mathbf{N}, \qquad \mathit{seq} : \mathbf{N} \to \mathbf{N}^*$$

where $\#(\sigma)$ is the standard Gödel number for the sequence σ and $\mathit{seq}(n)$ is the sequence σ having n as a Gödel number. The function $\#$ is one-one (two sequences having the same standard Gödel number are equal) and the function *seq* is onto (every number is a Gödel number of some sequence). Moreover, *seq* is a left inverse to $\#$:

$$\mathit{seq}(\#(\sigma)) = \sigma$$

for every finite sequence $\sigma \in \mathbf{N}^*$. Thus each finite sequence of natural numbers has several Gödel numbers but a unique Gödel number in standard form, and each natural number is a Gödel number of a unique finite sequence of natural numbers.

Gödel numbers of RM programs are especially important, because they are central to our goal of using RM programs to study RM programs. The first step in assigning Gödel numbers to programs is to introduce a numerical code (called an **opcode**) for each of the five RM instructions letters. We use the natural numbers 1, 2, 3, 4, 5 as codes for the RM instruction letters H, Z, S, T, and J, respectively. When we replace the RM instruction letters by their codes, each RM instruction becomes a sequence of from 1 to 4 natural numbers. By the Gödel number $\#(I)$ of an RM instruction I we mean the Gödel number of that sequence. For example, the Gödel number $\#(T, 5, 43)$ of the RM instruction $(T, 5, 43)$ is given by

$$\#(T, 5, 43) = \#(4, 5, 43) = 2\,4\,2\,5\,2\,4\,1\,3.$$

Finally, each RM program is a finite sequence

$$\mathbf{P} = (I_0, I_1, \ldots, I_p)$$

of instructions, and the Gödel number $\#(\mathbf{P})$ of the program $\mathbf{P}$ is defined to be the Gödel number

$$\#(\mathbf{P}) = \#(\#(I_0), \#(I_1), \ldots, \#(I_p))$$

of the sequence of the Gödel numbers of the instructions of the program. For example, the Gödel number of the program

$$\begin{array}{ll} T & 5\,43 \\ S & 6 \\ Z & 1 \end{array}$$

is the Gödel number of the sequence

$$(2\,4\,2\,5\,2\,4\,1\,3, 2\,3\,2\,6, 2\,2\,2\,1),$$

which is

$$2\,2\,1\,4\,1\,2\,1\,5\,1\,2\,1\,4\,1\,1\,1\,3\,2\,2\,1\,3\,1\,2\,1\,6\,2\,2\,1\,2\,1\,2\,1\,1.$$

We shall now introduce two new total functions,

$$\mathit{Extract}(x, y), \quad \mathit{Put}(x, y, z)$$

and show that they are RM computable.

$\mathit{Extract}(x, y)$ is the y-th term of the sequence with Gödel number x, with $\mathit{Extract}(x, y) = 0$ if this sequence has fewer than the y terms needed. (The Gödel number x need not be in standard form).

$\mathit{Put}(x, y, z)$ is equal to the standard Gödel number of the sequence which is formed by putting x into the y-th term of the sequence with Gödel number z, first adding as many 0 terms as necessary if the sequence with Gödel number z has fewer than y terms. These functions are useful in manipulating Gödel numbers, but have rather long and slow RM programs.

The following functions:

$$\mathit{Length}(x), \ \mathit{Digit}(x, i), \ \mathit{Terms}(x), \ \mathit{Start}(x, y), \ \mathit{PutEnd}(x, y)$$

defined below are RM computable. Using pseudocode, we shall describe RM programs LENGTH, DIGIT, TERMS0, START, and PUTEND which compute them. These programs will be used only to

show that the two functions $\mathit{Extract}(x, y)$ and $\mathit{Put}(x, y, z)$ are RM computable. (The RM program which computes $\mathit{Terms}(x)$ will be denoted by TERMS0 to distinguish it from the shorter Advanced RM program TERMS which is on the distribution diskette.)

At this point the reader should be convinced that given pseudocode for a new function in terms of old functions, and given RM programs for the old functions, one can routinely construct an RM program for the new function. For convenience, we include with each of the functions below a short algorithm for computing it; each such algorithm briefly describes the behavior of its corresponding pseudocode program.

(1) $\mathit{Length}(x) =$ *number of decimal digits in* x.

For example,

$$\begin{array}{lllllllll} \mathit{Length}(0) & = & \mathit{Length}(1) & = & \cdots & = & \mathit{Length}(9) & = & 1, \\ \mathit{Length}(10) & = & \mathit{Length}(11) & = & \cdots & = & \mathit{Length}(99) & = & 2, \\ \mathit{Length}(100) & = & \mathit{Length}(101) & = & \cdots & = & \mathit{Length}(999) & = & 3, \end{array}$$

and so on.

Short Algorithm: Successively divide x by 10, using *Div*, until 0 is reached. Output the number of divisions required.

(2) $\mathit{Digit}(x, i) =$ *the* i*-th decimal digit of* x *if* $i < \mathit{Length}(x)$, $\mathit{Digit}(x, i) =$ 0 *otherwise.*

We start counting with $i = 0$ on the left. For example,

$$\begin{array}{lll} \mathit{Digit}(907, 0) & = & 9, \\ \mathit{Digit}(907, 1) & = & 0, \\ \mathit{Digit}(907, 2) & = & 7, \\ \mathit{Digit}(907, n) & = & 0 \text{ for all } n \geq 3. \end{array}$$

Short Algorithm: If $i \geq \mathit{Length}(x)$, output 0. Otherwise, successively divide x by 10 using *Div* so that the i-th digit d is moved to the one's place (*Div* is applied $\mathit{Length}(x) - i$ times). Apply $\mathit{Remain}(\cdot, 10)$ to the result to output d.

(3) *Terms*(x) = *number of terms in the sequence with G.N.* x*, with the empty sequence having 0 terms.*

For example,

$$\mathit{Terms}(2\ 5\ 1\ 4\ 2\ 6\ 2\ 2\ 1\ 1\ 1\ 7)=3,$$

$$\mathit{Terms}(1\ 5\ 0\ 4\ 3\ 6\ 4\ 2\ 1\ 1\ 0\ 7\ 9)=3.$$

Short Algorithm: If x has an odd number of digits, use *Div* to drop the last digit. Use a counter to keep track of how many terms are in the sequence. If $x > 0$, initialize the counter at 1 because the zeroth (leftmost) digit is a marker which starts a term of the sequence, regardless of its value. Search the even-placed digits of x excluding the zeroth digit, and increment the counter whenever a marker > 1 is found; output the number in the counter after all even-placed digits have been tested.

(4) *Start*(x, y) = *the position of marker for the start of the* y*-th term in the sequence with G.N* x *if* $y <$ *Terms*(x)*, undefined otherwise.*

Count terms from 0 on the left. For example,

$$\mathit{Start}(\underline{2}\ 5\ 1\ 4\ 2\ 6\ 2\ 2\ 1\ 1\ 1\ 7,0)=0,$$

$$\mathit{Start}(2\ 5\ 1\ 4\ \underline{2}\ 6\ 2\ 2\ 1\ 1\ 1\ 7,1)=4,$$

$$\mathit{Start}(2\ 5\ 1\ 4\ 2\ 6\ \underline{2}\ 2\ 1\ 1\ 1\ 7,2)=6.$$

Short Algorithm: If $y \geq$ *Terms*(x), do not output anything; otherwise check the even-placed digits for markers > 1; use a counter to keep track of how many such markers are found, and another counter to record the position of each. When the yth such marker is found, output its position.

(5) *PutEnd*(x,y) = *the standard G.N. of the sequence formed by adding* y *as one more term to the end of the sequence with G.N* x *if* x *is a G.N. in standard form. Don't care otherwise.*

For example,

$$\mathit{PutEnd}(251426221117, 98) = 2514262211172\underline{918}$$

Short Algorithm: If x is not a standard Gödel number, the output can be anything. Otherwise, adjoin a 2 to the end of x (i.e., let $x' = x * 10 + 2$) and then use a loop to successively adjoin to the end of this new value of x the zeroth digit of y, then a 1, then the first digit of y, then a 1, etc., until the last digit of y has been adjoined.

(6) *Extract*(x, y) = *the* y*-th term of the sequence with G.N.* x *if* $y <$ *Terms*(x)*.* *Extract*$(x, y) = 0$ *otherwise.*

Following the precedent set in defining the function *Digit*, we make *Extract* a total function by giving it the value 0 when $y \geq$ *Terms*(x). For example,

$$\begin{aligned}
\mathit{Extract}(\underline{251}426221117, 0) &= 54\\
\mathit{Extract}(251\underline{42}6221117, 1) &= 6\\
\mathit{Extract}(251426\underline{221117}, 2) &= 217\\
\mathit{Extract}(251426221117, 3) &= 0
\end{aligned}$$

Short Algorithm: Record the digit d in the *Start*$(x, y)+1$ position, and use a loop to successively adjoin to the end of d the digits in positions *Start*$(x, y) + 3$, *Start*$(x, y) + 5$,...and so forth. Stop the process when the next even-numbered position is occupied by a marker > 1; output the number that has been obtained from this loop.

(7) *Put*(x, y, z) = *the standard G.N. of the sequence formed by putting* x *into the* y*-th term of the sequence with G.N.* z*, first adding as many 0 terms as necessary if* z *has fewer than* y *terms.*

For example,

$$Put(99,2,251426221117) = 2514262\underline{919}$$
$$Put(99,5,251426221117) = 25142622111720202\underline{919}$$

Note that $Put(x,y,z)$ is always a Gödel number in standard form; thus, it not only replaces the yth term of the sequence coded by z, but also changes the markers for the other terms to 1's and 2's as appropriate.

Short Algorithm: To change the markers which occur before the yth term of z to 1's or 2's, use a loop which successively applies *Extract* and *PutEnd* to the zeroth, first, second,...terms of (the sequence coded by) z (remembering to adjoin 0-terms if $Terms(z) < y$), thereby obtaining a code u for a sequence of y terms. Now use *PutEnd* to adjoin to u a yth term x. Finally, if $Terms(z) > y+1$, repeat the process above of applying *Extract* and *PutEnd* to change all markers after the yth to 1's and 2's, as appropriate.

Here are pseudocode descriptions of programs computing each of these functions. The RM programs for *Length* and *Digit* are assigned as exercises for the student. These functions can be tested with the GNUMBER program and do not take too much time when applied to numbers with fewer than six digits. The other functions are more difficult optional exercises. It is still possible to write RM programs for them with the GNUMBER editor, but the *Extract* and *Put* functions are too slow to be tested out.


```
program LENGTH(a)
 input:  a = x
 output: a = number of decimal digits in x

   let len = 1
   let num = a
   let num = Div(num,10)
   do until num = 0
     let num = Div(num,10)
     let len = len+1
   loop
   let a = len
end of program LENGTH
```

```
program DIGIT(a,b)
 input:  a = number, b = position
 output: a = Digit(number,position)

   let place = Length(a)
   DOTMINUS(place,b)
   if place = 0 then let a = 0
   PRED(place)
   let num = a
   let times = 0
   do until times = place
      let num = Div(num,10)
      let times = times + 1
   loop
   let a = Remain(num,10)
end of program DIGIT
```

```
program TERMS0(a)
 input:  a = x
 output: a = number of terms in the sequence
             with Godel number x.

   let count = 0
   let pos = 0
   do until pos + 2 > Length(a)
     let d = Digit(a,pos)
     if (count = 0 or d >1) then let count = count + 1
     let pos = pos + 2
   loop
   let terms = count
end of program TERMS0
```

```
program START(a,b)
 input:  a = x, b = i
 output: a = the position of the start marker of the term of x
             with index i.  Undefined if i >= Terms(x).

   let pos = 0
   let count = 0
   do until count >= b
      let pos = pos + 2
      let d = Digit(a,pos)
      if  d >1 then let count = count + 1
   loop
   let a = pos
end of program START
```



```
program EXTRACT(a,b)
 input: a = source, b = i
 output: c = the i-th term of the sequence
             with Godel number source if
             i < Terms(source), 0 otherwise.

   if b >= Terms(a) then
     let c = 0, halt
   let position = Start(a,b)
   let term = 0
   let marker = 0
   do until marker > 1
     let position = position + 1
     let d = Digit(a,position)
     let term = 10 * term + d
     let position = position + 1
     let marker = Digit(a,position)
   loop
   let c = term
end of program EXTRACT
```

```
program PUTEND(a,b)
 input:  a = a standard G.N. x, b = y
 output: a = the standard Godel number
             of the sequence formed by
             putting the number y onto the end of
             the sequence with Godel number x.

   let ab = a*10 + 2
   let place = 0
   let len = Length(b)
   do until place = len
      let d = Digit(b,place)
      let ab = ab*10 + d
      let place = place+1
      if place < len then let ab = ab*10 +1
   loop
   let a = ab
end of program PUTEND
```



```
program PUT(a,b,c)
 input:  a = source, b = i, c = target
 output: c = the standard Godel number
             of the sequence formed by
             putting the number source into the i-th term
             of the sequence with Godel number target.

   let inarray = c
   let outarray = 0
   let index = 0
   do until index = b
     let term = Extract(inarray,index)
     let outarray = Putend(outarray,term)
     let index = index + 1
   loop
   let outarray = Putend(outarray,a)
   let index = index + 1
   do until index >= Terms(inarray)
     let term = Extract(inarray,index)
     let outarray = Putend(outarray,term)
     let index = index + 1
   loop
   let c = outarray
end of program PUT
```

4.7 The Advanced RM

The advanced RM machine, or ARM, is formed by adding to the ordinary RM machine the two new instructions E for *Extract* and P for *Put*.

(E) Extract Instructions: For each $m = 1, 2, \ldots$, each $i = 1, 2, \ldots$, and each $n = 1, 2, \ldots$, there is an Extract instruction (E, m, i, n) which causes the ARM to replace the contents of register R_n by $Extract(r_m, r_i)$ leaving the other registers unchanged. Here r_i and r_m are the contents of registers R_i and R_m respectively, before the instruction is executed.

(P) Put Instructions: For each $m = 1, 2, \ldots$, each $i = 1, 2, \ldots$, and each $n = 1, 2, \ldots$, there is a Put instruction (P, m, i, n) which causes the ARM to replace the contents of register R_n by $Put(r_m, r_i, r_n)$ leaving the other registers unchanged. Here r_m, r_i, and r_n are the contents of registers R_m, R_i and R_n respectively (before the instruction is executed).

Since the functions *Extract* and *Put* are RM computable, any function which is computable by an advanced RM program is already computable by an RM program in the original sense, using only the instructions H, Z, S, T, and J. We make this precise in Theorem 4.7.1 below.

An **ARM program** is a sequence

$$\mathbf{P} = (I_0, I_1, I_2, \ldots, I_p)$$

of ARM instructions. As for the RM each program and each n determine a partial function $\Phi_{\mathbf{P}}^{(n)}$ defined on a subset $Dom(\Phi_{\mathbf{P}}^{(n)})$ of $\mathbf{N}^n$. Just as in Section 4.3, an ARM program $\mathbf{P}$ determines a **nextstate function**

$$NXSTATE_{\mathbf{P}} : \mathbf{N}^{\ell+1} \to \mathbf{N}^{\ell+1}$$

where ℓ is the highest number of a register mentioned in the program $\mathbf{P}$, $(r_0, r_1, r_2, \ldots, r_\ell) \in \mathbf{N}^{\ell+1}$ is the state before instruction I_{r_0} is executed, and $NXSTATE_{\mathbf{P}}(r_0, r_1, r_2, \ldots, r_\ell)$ is the state after execution.

H	(do nothing)
(Z, n)	$r_n := 0$, $r_0 := r_0 + 1$
(S, n)	$r_n := r_n + 1$, $r_0 := r_0 + 1$
(T, m, n)	$r_n := r_m$, $r_0 := r_0 + 1$
(J, m, n, q)	if $r_m = r_n$ then $r_0 := q$ else $r_0 := r_0 + 1$
(E, m, i, n)	$r_n := r_m[r_i]$, $r_0 := r_0 + 1$
(P, m, i, n)	$r_n[r_i] := r_m$, $r_0 := r_0 + 1$

Figure 4.6: The ARM machine

The ARM-instructions are summarized in Figure 4.6. In this figure, $a[x]$ denotes the x-th element of the sequence with Gödel number a. The column on the left gives the instruction and the column on the right gives the result of executing the instruction in conventional programming notation. Here the lower case letter r_n indicates the contents of register R_n and r_0 indicates the value of the program counter.

Theorem 4.7.1 *A function is ARM computable if and only if it is RM computable.*

Proof: Clearly an RM computable function is ARM computable since every RM program is an ARM program. The converse is true because we may always transform an ARM program to an RM program which behaves in the same way. We simply replace every *Extract* instruction (E, m, i, n) by an RM program which computes the *Extract* function $Extract(r_m, r_i)$ (with inputs r_m, r_i the contents of R_m, R_i) and puts the result in R_n, and every *Put* instruction (P, m, i, n) by an RM program which compute the *Put* function $Put(r_m, r_i, r_n)$ (with inputs r_m, r_i, r_n the contents of R_m, R_i, R_n) and put the result in R_n. We must take care that these inserted programs do not change any registers (other than R_n) used by the original ARM program.

The advanced GNUMBER program replaces the *Extract* and *Put* functions by extra instructions E and P. In principle, any "advanced" RM program with the E and P instructions can be replaced by an ordinary RM program which computes the same function. However, RM programs which involve computations of Gödel numbers are often so long and slow without the extra E and P instructions that nobody

will live long enough to see the output. The extra instructions are a pragmatic compromise which will allow us to experiment with some important programs involving Gödel numbers.

The Gödel number of an advanced RM program is defined in the same way as for an ordinary RM program, with the two new instruction letters E and P having the opcodes 6 and 7. In the sections which follow we shall use the advanced RM machine to build programs which manipulate Gödel numbers of programs. As an aid in the testing of RM programs which manipulate Gödel numbers of RM programs, GNUMBER has a command which places the standard Gödel number of the current RM program in a given register, and a command which replaces the current RM program with the RM program whose Gödel number (not necessarily in standard form) is in a given register.

4.8 Closure Theorems

One of the easiest ways to show that a complicated function is RM computable is to show that it can be built up using operations which produce RM computable functions from other RM computable functions. In this section several common operations are discussed: composition, primitive recursion, course of values recursion, parametrization, and unbounded minimalization. We shall prove several theorems showing that if the original function is RM computable then the new function is also RM computable. Such theorems are called closure theorems, because they say that the set of all RM computable functions is closed under the operation used to form a new function.

Throughout this section, all partial functions mentioned will be understood to be numerical functions. Remember that if we say that f is a partial function, we do not exclude the possibility that f might be total. Every total function is a partial function, but there are many partial functions which are not total. To simplify the exposition, we shall state the closure theorems for partial functions of one variable, with the understanding that results for n variables can be proved in a similar way. Since the RM computable functions are partial functions, we define composition, primitive recursion, and other operations on partial rather than total functions.

Composition: Let $g_1, \ldots, g_m$ be k-ary functions and let h be an m-ary function. The **composition** $h(g_1, \ldots, g_k)$ is the new k-ary function f defined by

$$f(a_1, \ldots, a_k) = h(g_1(a_1, \ldots, a_k), \ldots, g_m(a_1, \ldots, a_k)),$$

where $f(a_1, \ldots, a_k)$ is undefined if any part of the right side of the equation is undefined. In the case of one variable, if g and h are unary partial functions, then their composition $g \circ h$ is the unary partial function f such that

$$f(x) = g(h(x))$$

whenever both $h(x)$ and $g(h(x))$ are defined, and $f(x)$ is undefined otherwise. If g_1 and g_2 are unary partial functions and h is a binary partial function, the composition $h(g_1, g_2)$ is the unary partial function f such that

$$f(x) = h(g_1(x), g_2(x))$$

whenever $g_1(x)$, $g_2(x)$ and $h(g_1(x), g_2(x))$ are all defined, and $f(x)$ is undefined otherwise.

Primitive Recursion: Let h be a binary partial function. The partial function obtained from h by primitive recursion is the unary partial function f such that

$$f(0) = 1$$

and for all x,

$$f(x+1) = h(f(x), x)$$

if $f(x)$ and $h(f(x), x)$ are both defined, and $f(x+1)$ is undefined otherwise.

Note that in this definition, if $f(x)$ is undefined then all later values $f(y)$, $y > x$ will be undefined. Thus f will either be total, i.e. defined for all x, or the domain of f will be a finite initial segment $\{0, 1, \ldots, n\}$ of the natural numbers.

Course of Values Recursion: Let h be a binary partial function. The partial function obtained from h by course of values recursion is the unary partial function g such that

$$g(0) = 1$$

and for all x,

$$g(x+1) = h(\#(g(0), g(1), \ldots, g(x)), x)$$

if $g(0), \ldots, g(x)$ and $h(\#(g(0), g(1), \ldots, g(x)), x)$ are all defined, and $g(x+1)$ is undefined otherwise. In this definition, $\#(g(0), g(1), \ldots, g(x))$ stands for the standard Gödel number of the sequence $(g(0), \ldots, g(x))$ in the notation of Section 4.6.

Again, if $g(x)$ is undefined then all later values $g(y)$, $y > x$ will be undefined. Thus g will either be total or $Dom(g)$ will be a finite initial segment $\{0, 1, \ldots, n\}$ of the natural numbers.

Parametrization: Let f be a binary partial function. The parametrization of f is the sequence of unary partial functions f_n, $n = 0, 1, \ldots$ defined by

$$f_n(x) = f(x, n).$$

Unbounded Minimalization: This is a way of getting a unary partial function from a binary relation. Let R be a binary relation. The partial function obtained from R by unbounded minimalization is the unary partial function

$$f(x) = \mu y\, R(x, y)$$

where $f(x)$ is the least y such that $R(x, y)$ if $\exists y\, R(x, y)$, and $f(x)$ is undefined otherwise.

The symbol μy is read "the least y such that." It is called the unbounded minimalization operator.

Definition 4.8.1 We say that a set $\mathcal{F}$ of partial functions is **closed under composition** if any partial function obtained from partial functions in $\mathcal{F}$ by composition belongs to the set $\mathcal{F}$. Closure under **primitive recursion**, **course of values recursion**, and **parametrization** are defined in a similar way. A set $\mathcal{F}$ of partial functions is **closed under unbounded minimalization** if for any relation R whose characteristic function belongs to $\mathcal{F}$, the partial function obtained from R by unbounded minimalization belongs to $\mathcal{F}$.

In this section we shall prove the

Theorem 4.8.2 (Closure Theorem) *The set of RM computable functions is closed under composition, primitive recursion, course of values recursion, parametrization, and unbounded minimalization.*

Before starting on the proof of the Closure Theorem, we need to develop an efficient way of combining two RM or ARM programs.

If **P** is an ARM program, the **length of P**—denoted $n(\mathbf{P})$—is the number of instructions in **P**, not counting halt instructions at the end. The **empty program** is the program consisting entirely of halt instructions, and has length zero. Since the instructions of an ARM program are numbered beginning with 0, if **P** is not the empty program then the $(n(\mathbf{P}) - 1)$th instruction is the last nonhalt instruction in **P**.

The total function f, where $f(x)$ is the length of the ARM program **P** with Gödel number x, is RM computable. To make an ARM program which computes f, start with the program TERMS [1], which gives the number of instructions in **P**, and then add a loop which will subtract 1 from the output for each halt instruction at the end of **P**.

Given two ARM programs **P** and **Q**, their **join PQ** is the new ARM program consisting of the program **P** followed by the program **Q**, with **Q** starting immediately after the last nonhalt instruction of **P**, and with each instruction number and each jump target in **Q** increased by the length of **P**.

There is an ARM program called JOIN[2] which is a useful building block for other programs, and computes the Gödel number of the join of two ARM programs **P** and **Q** from the Gödel numbers and lengths of **P** and **Q**. If the Gödel numbers of **P** and **Q** are placed in registers R_1 and R_2, the program lengths $n(\mathbf{P})$ and $n(\mathbf{Q})$ are placed in registers R_3 and R_4, and the numbers 0 – 5 are placed in registers R_{20} – R_{25} the program JOIN will eventually halt with the Gödel number of the join **PQ** in register R_1.

Lemma 4.8.3 *Let $c(x, y)$ be the total function defined as follows. If x and y are Gödel numbers of ARM programs* **P** *and* **Q**, *then $c(x, y)$ is*

[1] Included on the problem diskette for the advanced RM machine
[2] Included on the problem diskette for the advanced RM machine

the Gödel number of the join **PQ**. *Otherwise* $c(x,y) = 0$. *The function* c *is RM computable.*

Proof: An ARM program to compute c can be pieced together using the TERMS and JOIN programs given in the problem diskette. **End of Proof.**

If **P** and **Q** are sufficiently well designed, the join **PQ** will compute the composition $g \circ f$ of the unary partial function g computed by **Q** and the unary partial function f computed by **P**.

For example, let **P** be the program

```
0:   S  1
1:   S  1
```

and let **Q** be the program

```
0:   T  1  2
1:   Z  3
2:   J  2  3  6
3:   S  1
4:   S  3
5:   J  1  1  1
```

Here, **P** computes the function $f(x) = x + 2$ and **Q** computes the function $g(x) = 2x$. Then the join **PQ** is the following program, which computes the function $g(f(x)) = 2x + 4$:

```
0:   S  1
1:   S  1
2:   T  1  2
3:   Z  3
4:   J  2  3  8
5:   S  1
6:   S  3
7:   J  1  1  3
```

In this example, the join **PQ** has the effect of first executing the program **P**, ending up at the initial instruction of **Q** with the output of **P** in register R_1, and then executing **Q**.

We shall now introduce conditions under which the join of two programs will compute the composition of two partial functions, as in the example. We first define the regular programs, which behave well as the first part of a join, and then define the neatly computing programs, which are regular and also behave well as the second part of a join.

Definition 4.8.4 We will call an ARM program **P regular** if **P** has no halt instructions before the last nonhalt instruction, and no target of a jump instruction in **P** is greater than the program length $n(\mathbf{P})$.

The three programs listed above are regular.

Let us consider a joined program **PQ** whose first part **P** is regular. Suppose **P** and **Q** are ARM programs and **P** is regular. Then the join program **PQ** will stay within the first $n(\mathbf{P})$ instructions and therefore do exactly the same thing as **P** does until **P** halts. If **P** never halts with input $x_1, x_2, \ldots$, then **PQ** never halts with input $x_1, x_2, \ldots$. If **P** with input $x_1, x_2, \ldots$ halts at step t, then **PQ** with input $x_1, x_2, \ldots$ will have the same state as **P** at time t, with the program counter at $n(\mathbf{P})$ where the **Q** part of the join program begins.

There is one more problem to be dealt with. The program **P** might place nonzero data in registers $R_2, R_3, \ldots$ while computing its output in R_1. In order to be sure that **PQ** computes the composition, we must know that the output of **Q** in R_1 depends only on the initial contents of R_1 and is not affected by the initial contents of the other data registers.

Definition 4.8.5 An RM program **P** is said to **neatly compute** an n-ary function f if **P** is regular and computes f in the following sense: if the registers of the RM are initialized so that the registers R_1 through R_n hold the numbers a_1 through a_n, and the program **P** is loaded into the machine and executed (starting with instruction I_0), then, *no matter what the other registers contain initially,*

- if $(a_1, a_2, \ldots, a_n) \in Dom(f)$, then the program eventually halts with register R_1 holding the value $f(a_1, a_2, \ldots, a_n)$ of the function; and
- if $(a_1, a_2, \ldots, a_n) \notin Dom(f)$, then the program never halts, i.e. computes forever.

In other words, **P** neatly computes the n-ary function f if and only if **P** is regular and for any $m \geq n$ and any numbers $a_1, a_2, \ldots, a_m$ we have both the condition

$$(a_1, a_2, \ldots, a_n) \in Dom(f) \Leftrightarrow (a_1, a_2, \ldots, a_m) \in Dom(\Phi_{\mathbf{P}}^{(m)})$$

and the condition that

$$f(a_1, a_2, \ldots, a_n) = \Phi_{\mathbf{P}}^{(m)}(a_1, a_2, \ldots, a_m)$$

for $(a_1, a_2, \ldots, a_m) \in Dom(\Phi_{\mathbf{P}}^{(m)})$.

Again, the functions in our example above are neatly computed by their ARM programs. We now show that joins of neatly computing programs neatly compute compositions of partial functions.

Lemma 4.8.6 *Suppose that* **P** *and* **Q** *neatly compute the unary partial functions f and g. Then the join* **PQ** *neatly computes the composition $g \circ f$.*

Proof: Let h be the unary partial function computed by **PQ**. Start with x in R_1. Since **P** is regular, **P** and **PQ** will do exactly the same thing until **P** halts. Thus if $f(x)$ is undefined, then both **P** and **PQ** will go on forever, so $h(x)$ is undefined.

Suppose that $f(x)$ is defined. Then **P** will halt at some time t with $f(x)$ in R_1, so **PQ** at time t will have $f(x)$ in R_1 and the program counter at $n(\mathbf{P})$ where **Q** begins. Since **P** neatly computes f, this happens no matter what the initial contents of the other registers $R_n, n > 1$ were.

The program **PQ** will now do the same thing as **Q** would do starting from the data register contents left by **P** at time t. Since **Q** neatly computes g, the output of **PQ** in R_1 depends only on the contents of R_1 at time t, not on the other data registers. If $g(f(x))$ is undefined, the program **PQ** will never halt, so $h(x)$ is undefined. If $g(f(x))$ is defined, the program **PQ** will eventually halt with $g(f(x))$ in R_1, so $h(x) = g(f(x))$. Thus $\mathcal{P}\mathbf{Q}$ neatly computes h, and $h = g \circ f$. **End of Proof.**

The following proposition shows that there are enough neatly computing programs to capture all RM computable functions.

Proposition 4.8.7 *If an RM program* **P** *computes an n-ary partial function f, then there is a program* **Q** *which neatly computes the same partial function f.*

Proof: To make the computation neat, first add additional steps at the beginning of the program **P** which put zero in all the registers which are used in the program except for the first n registers. Let m be the length of this adjusted program. Renumber the targets of the jump instructions by increasing each by m.

To make the program regular, decrease to m any targets of jump instructions in this new program which exceed m. Replace all halt instructions by the instruction $(J, 1, 1, m)$. The new program **Q** computes f neatly, and is clearly regular as well. **End of Proof.**

We now prove the composition part of the Closure Theorem. We shall prove even more, that the Gödel number of a program for the composition is given by an RM computable function.

Theorem 4.8.8 (Closure Under Composition) *If g and h are RM computable functions of one variable, then the composition $g \circ h$ is RM computable. Moreover, there is an RM computable total function c of two variables such that whenever x and y are the Gödel numbers of ARM programs which neatly compute g and h respectively, then $c(x, y)$ is the Gödel number of an ARM program which neatly computes $g \circ h$.*

Proof: The first part of the theorem follows from Lemma 4.8.6, which shows that if **P** neatly computes g and **Q** neatly computes h, then the join **PQ** neatly computes the composition $g \circ h$. It now follows that the function c given in Lemma 4.8.3 does the job [3].

Theorem 4.8.9 (Closure Under Primitive Recursion) *If h is an RM computable function of two variables, then the partial function f of one variable given by the rule*

$$f(0) = 1, \quad f(n+1) = h(f(n), n)$$

[3]The function c will be computed by the ARM program COMPOSE assigned in computer problem set GN6.

is RM computable. Moreover, there is an RM computable total function r of one variable such that for all x, if x is the Gödel number of an RM program which neatly computes h then $r(x)$ is the Gödel number of an RM program which neatly computes the new partial function f given by the above rule.

Proof: We sketch the proof of the first part of the theorem. Let **P** neatly compute h. Take m large enough so that **P** does not use any register beyond R_m, that is, no register number larger than m appears in the instructions of **P**. We describe a new program **Q** which neatly computes f. First, **Q** saves the original input a in R_{m+1}, and puts a zero in R_{m+2}. The number in R_{m+2}, which we shall call x, will be used as a counter which works its way from 0 to a. **Q** then puts a 1 in R_1.

Now **Q** checks whether $a = x$. If so, **Q** halts. Otherwise, **Q** puts x into R_2, runs the program **P**, and increases x by 1. It then repeats the process given in the current paragraph.

The program **Q** can be built by joining a few instructions before and after **P**

To prove the second part of the theorem, an ARM program must be produced which computes the total function r[4]. **End of Proof.**

Theorem 4.8.10 (Closure Under Course of Values Recursion) *If h is an RM computable function of two variables, then the partial function g of one variable given by the rule*

$$g(0) = 1, \quad g(n+1) = h(\#((g(0), \ldots, g(n)), n)$$

is RM computable. Moreover, there is an RM computable total function r of one variable such that for all x, if x is the Gödel number of an RM program which neatly computes h then $r(x)$ is the Gödel number of an RM program which neatly computes the new partial function g given by the above rule.

Again, this can be proved by producing an appropriate RM program. [5]

[4]This is assigned as the problem RECUR in computer problem set GN6.

[5]This is assigned as the problem CVREC in problem set GN6.

Theorem 4.8.11 (Parametrization) *If f is an RM computable function of two variables, then for each natural number n, the one variable partial function $f_n(x) = f(x, n)$, obtained by holding the second argument fixed at n, is RM computable. Moreover, there is an RM computable total function p of two variables such that for all x and y, if x is the Gödel number of an RM program which neatly computes f then $p(x, y)$ is the Gödel number of an RM program which neatly computes f_y.*

The first part is proved as follows. Let **P** be an RM program which neatly computes f. For each n, let $\mathbf{Q}_n$ be the program which has one instruction $(Z, 2)$ followed by n copies of the instruction $(S, 2)$. Then the join $\mathbf{Q}_n\mathbf{P}$ computes the partial function f_n, because it puts n in register 2 and then executes **P**. The second part is proved by producing the RM program PARAM which does the job.[6] If PARAM is executed with the Gödel number of an RM program **P** as the first input and n as the second input, it will halt with the Gödel number of the program $\mathbf{Q}_n\mathbf{P}$ as output.

Theorem 4.8.12 (Closure Under Unbounded Minimalization) *Let R be an RM-computable binary relation. Then the partial unary function f defined by*

$$f(x) = \mu y\, R(x, y)$$

is RM-computable. Moreover, there is an RM computable total function r of one variable such that for all x, if x is the Gödel number of an RM program which neatly computes the characteristic function of R then $r(x)$ is the Gödel number of an RM program which neatly computes the new partial function f given by the above rule.

To prove the theorem, an RM program must be produced which computes the total function r.[7]

[6]Included on the problem diskette as an example.

[7]This is assigned as the problem UBMIN in computer problem set GN6.

4.9 Universal RM Programs

An RM program **U** is **universal for one input** if for all RM programs **P** with one input there is a number e such that for all x the output of **U** computing on input (e, x) is the same as the output of **P** computing on input x. (The program **U** never halts on (e, x) just in case **P** never halts on x.) We sometimes call **P** the **simulated program with index** e. More generally, an RM program **U** is **universal for** n **inputs** if for all RM programs **P** with n inputs there is a number e such that for all $x_1, \ldots, x_n$ the output of **U** computing on inputs $(e, x_1, \ldots, x_n)$ is the same as the output of **P** computing on inputs $(x_1, \ldots, x_n)$.

Theorem 4.9.1 (Universal Machine) *For every n there is an RM program which is universal for n inputs.*

We shall prove this theorem in case $n = 2$ by producing a universal RM program UNIV for two inputs. We leave the problem of modifying the program to produce a universal program on n inputs as an exercise for the reader (Exercise 11). The following remark takes care of the case of one input.

Remark 4.9.2 If UNIV is a universal RM program for two inputs, then the program UNIV1 formed by joining the single instruction $(Z, 3)$ to the beginning of UNIV is a universal RM program for one input.

This is because, by definition, the output of an ARM program computing on n inputs is obtained by starting the program with the given inputs in the first n registers and zero in all other registers.

To make the task of producing a universal program easier, we shall use the advanced RM instructions. (It follows from Theorem 4.7.1 that there is also an ordinary RM program which does the job.) To keep things balanced, the universal program will simulate advanced as well as ordinary RM programs. The ARM program UNIV listed below is the same as the one supplied on the diskette.

UNIV will use several Gödel numbers of sequences of numbers. We identify the instructions H, Z, S, T, J, E, and P with the natural numbers 1,2,3,4,5,6,7. An ARM instruction is then a sequence of at most 4 natural numbers, and an ARM program **P** is a finite sequence of

instructions. The state of an ARM program **P** during a computation is another finite sequence of natural numbers, giving the contents of each register used in the program.

UNIV accepts as input a triple e, x, y in registers R_1, R_2 and R_3. The number e is interpreted as the Gödel number of an ARM program **P**. (If the sequence display is used in the GNUMBER program, e will appear as a finite sequence of Gödel numbers for the instructions of **P**.) The output of UNIV will be the same as the output of the program **P** with input x, y. UNIV works by simulating an ARM machine running the program **P**. The contents of the registers of the simulated machine are coded as a Gödel number for a single finite sequence of natural numbers, which is held in register R_4 (the fourth register of the universal machine). The zeroth term of the sequence coded in R_4 is the program counter of the simulated machine. For $n \geq 1$, the n-th term of the sequence coded in R_4 is the n-th register of the simulated machine. UNIV begins by initializing constants and clearing register R_4 to zero. It then places x and y into the simulated registers one and two. It does this by using the Put command to make the first term of the sequence coded in R_4 equal to x and the second term equal to y. At this point the simulated program counter, which is the zeroth term of the sequence coded in R_4, contains a zero. UNIV next analyzes the zeroth simulated instruction, whose Gödel number is the zeroth term of the sequence coded by the input e in register R_1, and performs the indicated operation on the contents of the simulated program counter and registers coded in R_4. It then repeats the process, extracting the simulated program counter from the zeroth term in R_4, and the simulated instruction from R_1. In this way, UNIV does the same thing to the simulated registers in R_4 that the program **P** would do to its registers.

For example, suppose that the simulated program counter, which is the zeroth term in R_4, is 5, and the fifth simulated instruction is $Z3$. We identify $Z3$ with the sequence $(2, 3)$, whose Gödel number 2223 would be the fifth term in R_1. UNIV will use the Put command to place a zero in the third simulated register, which is the third term in R_4.

Here is a list of the registers used in the program UNIV. For each register, we give a name for the contents to use in comments, and a

verbal description.

R_1 : a. The input e, a Gödel number of the simulated program **P**. (In the sequence display, e is shown as a sequence of Gödel numbers of instructions of **P**. Each instruction is itself a sequence of from one to four numbers.) The output of the program also goes here.

R_2 : b. The input x.

R_3 : c. The input y.

R_4 : reg. A Gödel number for the state of the simulated machine, i.e. the finite sequence consisting of the contents of the simulated program counter and the simulated registers.

R_5 : pc. The simulated program counter, which is the zeroth term coded in reg.

R_6 : quad. The Gödel number of the pc-th simulated instruction, which is a sequence of from one to four numbers. quad is the pc-th term of the simulated program e in R_1.

R_7 : op. The zeroth term of the simulated instruction quad. This term is an opcode for one of the commands H,Z,S,T,J,E,P.

R_8 : s1. The first term of the simulated instruction quad (or zero if the instruction is of length 1).

R_9 : s2. the second term of the simulated instruction quad (or zero if the instruction is of length < 3).

R_{10} : s3. The third term of the simulated instruction quad (or zero if the instruction is of length < 4).

R_{11} : v1. The contents of simulated register number s1, i.e. the s1-th term of reg.

R_{12} : v2. The contents of simulated register number s2, i.e. the s2-th term of reg.

R_{13} : v3. The contents of simulated register number s3, i.e. the s3-th term of reg. (Note that in case the simulated instruction number pc is a jump instruction, then s3 is another instruction number and not a register number).

R_{14} : Unused.

R_{15} : time. The time for the simulated program. (This is not needed, but is helpful when experimenting with the program).

R_{20} : zero. The constant 0.

R_{21} : one. The constant 1.

R_{22} : two. The constant 2.

R_{23} : three. The constant 3.

R_{24} : four. The constant 4.

R_{25} : five. The constant 5.

R_{26} : six. The constant 6.

R_{27} : seven. The constant 7.

We first give a pseudocode description of UNIV, using the "variable" names in the preceding list for the contents of the registers used by UNIV.

```
program UNIV(a,b,c)
 input:  a = e, b = x, c = y
 output: a = P(x)
  let zero = 0, one = 1, ... , seven = 7
  let time = 0
  let reg = 0
  let reg[one] = b, reg[two] = c
  let pc = 0
  let op = 0
  do until op = H
    let quad = inst[pc]
    let op = quad[zero]
    let s1 = quad[one],   v1 = reg[s1]
    let s2 = quad[two],   v2 = reg[s2]
    let s3 = quad[three], v3 = reg[s3]
    if op = Z then
        let reg[s1] = zero,  pc = pc+1
    else if op = S then
        let v1 = v1 + 1,  reg[s1] = v1,  pc = pc+1
    else if op = T then
         let reg[s2] =  v1,  pc = pc+1
    else if op = J then
        if v1 = v2  then let pc = s3  else  let pc = pc+1
    else if op = E then
        let v3 = v1[v2],  reg[s3] =v3,  pc = pc+1
    else if op = P then
        let v3 = v1[v2],  reg[s3] = v3,  pc = pc+1
    else let op = H
    let reg[zero] = pc
    let time = time + 1
  loop
  let a = reg[one]
end of program UNIV
```


The listings in figures 4.7 and 4.8 give "assembly code" for the universal program. Adjacent to the assembly code is the actual "machine language" which the assembly code describes. The program could be shortened by several steps, but instead is designed to match the pseudocode listing. Here's an outline:

Initialization Instructions 00-14 initialize the constants `zero` through `seven`. Instructions 15-20 initialize the simulated time counter `time`, program counter `pc`, and register sequence `reg`.

Main Loop Instructions 21-29 initialize the main loop by extracting the opcode `op` of the instruction to be executed, the registers `s1`, `s2`, `s3` used in this instruction, and the values `v1`, `v2`, `v3` held in this registers. Instructions 30-37 jump to the appropriate interpreter subroutine. Instructions 57-58 increment the time counter and restart the loop.

Action Instructions 38-56 contain the interpreter subroutines.

Output Instructions 59-60 place the output in R_1 and halt.

```
      Z  zero                    0: Z  20
      T  zero, one               1: T  20  21
      S  one                     2: S  21
      T  one, two                3: T  21  22
      S  two                     4: S  22
      T  two, three              5: T  22  23
      S  three                   6: S  23
      T  three, four             7: T  23  24
      S  four                    8: S  24
      T  four, five              9: T  24  25
      S  five                   10: S  25
      T  five, six              11: T  25  26
      S  six                    12: S  26
      T  six, seven             13: T  26  27
      S  seven                  14: S  27
      Z  time                   15: Z  15
      Z  reg                    16: Z   4
      P  b, one, reg            17: P   2  21   4
      P  c, two, reg            18: P   3  22   4
      Z  pc                     19: Z   5
      Z  op                     20: Z   7
LOOP  J  op,one,EXIT            21: J   7  21  59
      E  a, pc, quad            22: E   1   5   6
      E  quad, zero, op         23: E   6  20   7
      E  quad, one, s1          24: E   6  21   8
      E  quad, two, s2          25: E   6  22   9
      E  quad, three, s3        26: E   6  23  10
      E  reg, s1, v1            27: E   4   8  11
      E  reg, s2, v2            28: E   4   9  12
      E  reg, s3, v3            29: E   4  10  13
      J  op, two, ZERO          30: J   7  22  38
      J  op, three, SUCC        31: J   7  23  40
      J  op, four, TRANS        32: J   7  24  43
      J  op, five, JUMP         33: J   7  25  45
      J  op, six, EXTR          34: J   7  26  47
      J  op, seven, PUT         35: J   7  27  50
      T  one, op                36: T  21   7
      J  DONE                   37: J   1   1  56
```

Figure 4.7: The Universal Program (Assembly Code)


```
ZERO  P  zero, s1, reg    38: P  20   8   4
      J  NEXT             39: J   1   1  53
SUCC  S  v1               40: S  11
      P  v1, s1, reg      41: P  11   8   4
      J  NEXT             42: J   1   1  53
TRANS P  v1, s2, reg      43: P  11   9   4
      J  NEXT             44: J   1   1  53
JUMP  J  v1, v2, SETPC    45: J  11  12  55
      J  NEXT             46: J   1   1  53
EXTR  E  v1, v2, v3       47: E  11  12  13
      P  v3, s3, reg      48: P  13  10   4
      J  NEXT             49: J   1   1  53
PUT   P  v1, v2, v3       50: P  11  12  13
      P  v3, s3, reg      51: P  13  10   4
      J  NEXT             52: J   1   1  53
NEXT  S  pc               53: S   5
      J  DONE             54: J   1   1  56
SETPC T  v3, pc           55: T  10   5
DONE  P  pc, zero, reg    56: P   5  20   4
      S  time             57: S  15
      J  LOOP             58: J   1   1  21
EXIT  E  reg, one, a      59: E   4  21   1
      H                   60: H
```

Figure 4.8: Subroutines for the Universal Program

4.10 Church's Thesis

We introduced our RM computer as an attempt to capture the notion of an algorithm. And certainly every partial function which is RM computable is computable by an algorithm (using the program itself as the desired algorithm). But what about the converse? (Is every partial function computable by an algorithm in fact *RM computable*?) Until a generally accepted formal definition of "algorithm" is designed, no formal proof of the converse is possible. However, every known attempt to describe the class of algorithmically computable functions – using computing machines (like our RM computer), formal systems (like Weak or Peano Arithmetic), recursiveness, and others – has resulted in exactly the same class of computable functions. In Chapter 5 we shall make use of two of these alternative characterizations of the class of computable functions, the recursive functions and the functions which are representable in Weak Arithmetic. This confluence of ideas suggests that the class of RM computable functions is both natural and comprehensive. Secondly, no one has ever described an (intuitively) algorithmic function which didn't turn out to be RM computable. These considerations have led mathematicians to accept the following statement:

CHURCH'S THESIS

> *Every partial or total function which can be computed by an algorithm is an RM computable partial or total function.*

Let us emphasize that Church's Thesis is not a *theorem* but rather is a *heuristic principle* for which there is a great deal of evidence. The reason Church's Thesis is only a heuristic principle is that we do not have a mathematically rigorous definition of the word "algorithm." We can agree that many particular examples are algorithms, but statements about the class of all algorithms are hard to make precise. Using Church's Thesis frequently makes the job of verifying that certain partial functions are RM computable much easier, since it allows us to point to a simple algorithm rather than a tedious RM program to establish RM computability. Actually, we already began using a form of Church's Thesis in Section 4.6 when we claimed that the functions

Length , *Digit* , etc. were RM computable after exhibiting only pseudocode programs for them. The task of actually writing RM programs in place of these pseudocode programs has been left to the exercises. The proof in this book that every ARM computable function is RM computable used Church's Thesis to show that these pseudocode programs can be replaced by actual RM programs. When this proof is supplemented by the actual RM programs required by the exercises, we obtain a rigorous proof that every ARM computable function is RM computable.

The theorem that there exists a universal ARM program for one input is a good example of a theorem which can be proved more easily if one uses Church's Thesis. In this chapter we gave an explicit example of a universal ARM program, without relying on Church's thesis. The following proof uses Church's Thesis to show very quickly that there *exists* a universal ARM program without actually producing the program.

Proof that there is a universal ARM program (using Church's Thesis): We show that the partial function *Univ* given by

$$Univ(e, x) = f_e(x),$$

where f is the partial function computed by the ARM program with Gödel number e, is RM computable. By Church's Thesis, all we have to do is describe an algorithm which computes this partial function. Here it is: Write down the ARM program which has Gödel number e. Run that program with input x in register R_1 and 0 in all other registers. If the computation eventually halts, $Univ(e, x) = a$ where a is the number in register R_1 at the halt. Otherwise, $Univ(e, x)$ is undefined. **End of Proof.**

In the next chapter we shall use Church's Thesis to show that the some of the central notions of predicate logic are RM computable.

Whenever we use Church's Thesis in a proof in this book, it is possible to give a completely rigorous proof without Church's Thesis. In cases where these rigorous proofs are long and bereft of new ideas, it is better to accept Church's Thesis and use the extra time elsewhere.

4.11 The Halting Problem

Recall from page 199 that a numerical relation R in n variables is said to be RM **computable**[8] if there is an RM program which produces the output 1 (for yes) if the input satisfies the relation, and produces the output 0 (for no) if not. Such a program is said to **compute** the relation R. A relation R which is not RM computable is said to be **undecidable**; we also say that the **decision problem** for R is undecidable. According to Church's thesis, if a relation is undecidable, then it is impossible to design an algorithm which, given any input, will always produce the answer **yes** if the input belongs to R and the answer **no** if the answer does not belong to R. One of the main purposes of the RM machine is to show that various interesting relations are undecidable. In this section we shall use the universal RM program to give a first example of an interesting undecidable relation. Other examples will be given in 5.10 and Exercise 17.

Theorem 4.11.1 (Halting Problem) *Let UNIV1 be the universal program for RM programs with one input. Let $R(x,y)$ be the set of all pairs x,y of natural numbers such that UNIV1 computing on inputs x,y eventually halts. The relation R is undecidable, i.e. it is not RM computable.*

Proof: The proof is by contradiction. Suppose that R is RM computable. Then there is an RM program **P** which computes the relation R. Let u be the partial function of two variables computed by UNIV1. By joining the program **P** with UNIV1 and doing some easy housekeeping, we can form an RM program **Q** which, when computing on input x, halts with output 0 if $R(x,x)$ is false and halts with output $u(x,x)+1$ if $R(x,x)$ is true. Let n be Gödel number of **Q**. The program **Q** will eventually halt with any input because it computes a total function. The program UNIV1 computing on input n,n will eventually halt with the same output as **Q** computing on input n. But UNIV1 computing on input n,n will have output $u(n,n)$, and by the definition of **Q**, **Q** computing on input n will halt with output $u(n,n)+1$. Thus

[8]The word **decidable** is often used as a synonym for *computable*.

$u(n,n) = u(n,n)+1$, which is a contradiction. Therefore R cannot be RM computable. **End of Proof.**

There is a striking resemblance between the preceding proof and the arguments used in the proofs of each of the following: Russell's result that the common notion of "set" is self-contradictory (i.e., Russell's paradox – Exercise 2.46); Cantor's Theorem that there can be no function from a set X onto the set of all subsets of X (Theorem A.6 in the Appendix); and the result that there is an RM computable function which is not primitive recursive (Exercise 4.31).

The powerful technique common to these arguments is known as Cantor's **diagonal method**. The idea is to prove that a certain binary relation $R(x,y)$ cannot have some property by looking at the *diagonal relation* $R(x,x)$ in two different ways. The diagonal method will be used again in the next chapter to prove Gödel's Incompleteness Theorems.

4.12 Church's Theorem

Church's theorem says that we cannot program a computer to accept as input a wff of predicate logic and produce as output a zero or one according to whether or not the input wff is valid. We will prove this by contradiction; under the (false) assumption that such a program exists, we will show how to construct another program which solves the halting problem. Since the latter program does not exist (by Theorem 4.11.1) neither does the former.

A vocabulary sufficient for describing the behavior of an RM program **P** which uses only the registers

$$R_0, R_1, \ldots, R_\ell$$

(where R_0 is the program counter), contains the equality symbol $\doteq$, a constant symbol **0** for the number zero, a unary function symbol **s** for the successor function, and an $(\ell+1)$-ary predicate symbol **R**. As in Section 3.7 every non-negative integer n has a name **n** called the **numeral** which denotes n. For example,

$$\mathbf{3} \doteq \mathrm{s}(\mathrm{s}(\mathrm{s}(0))).$$

Theorem 4.12.1 *For every RM program* $\mathbf{P}$ *which uses only the registers* $R_0, \ldots, R_\ell$ *there is a wff*

$$\mathbf{A_P}(x_1, x_2, \ldots, x_\ell)$$

with free variables $(x_1, x_2, \ldots, x_\ell)$ *such that for all* ℓ*-tuples* $(a_1, a_2, \ldots, a_\ell) \in \mathbf{N}^\ell$ *the sentence*

$$\mathbf{A_P}(\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_\ell)$$

is valid if and only if the program $\mathbf{P}$ *halts on input* $(a_1, \ldots, a_\ell)$.

Proof: The intended interpretation of the wff $\mathbf{R}(x_0, x_1, x_2, \ldots, x_\ell)$ is that the state of a register machine running the program $\mathbf{P}$ is $(x_0, x_1, \ldots, x_\ell)$; that is, the register R_j holds the value x_j. We shall be more precise about this below. It is important to remember that when the register machine is running, the register R_0 plays a special role: it holds the index of the next instruction to be executed.

Suppose that the program $\mathbf{P}$ is given by

$$\mathbf{P} = (I_0, I_1, I_2, \ldots, I_{p-1}).$$

We may assume without loss of generality that the program $\mathbf{P}$ is regular. Thus $I_p = H$ and if $I_j = (J, m, n, q)$ then $q \leq p$.

To each instruction I_j $(j = 0, 1, \ldots, p)$ of the program $\mathbf{P}$ we associate a wff $\mathbf{I}_j$ as follows.

- If $I_j = (Z, n)$ then $\mathbf{I}_j$ is the wff

$$\mathbf{R}(\mathbf{j}, y_1, \ldots, y_n, \ldots, y_\ell) \Rightarrow \mathbf{R}(\mathbf{s(j)}, y_1, \ldots, \mathbf{0}, \ldots, y_\ell).$$

- If $I_j = (S, n)$ then $\mathbf{I}_j$ is the wff

$$\mathbf{R}(\mathbf{j}, y_1, \ldots, y_n, \ldots, y_\ell) \Rightarrow \mathbf{R}(\mathbf{s(j)}, y_1, \ldots, \mathbf{s}(y_n), \ldots, y_\ell).$$

- If $I_j = (T, m, n)$ then $\mathbf{I}_j$ is the wff

$$\mathbf{R}(\mathbf{j}, \ldots, y_n, \ldots, y_m, \ldots) \Rightarrow \mathbf{R}(\mathbf{s(j)}, \ldots, y_m, \ldots, y_m, \ldots).$$

- If $I_j = (J, m, n, q)$ then $\mathbf{I}_j$ is the wff $\mathbf{I}'_j \wedge \mathbf{I}''_j$ where $\mathbf{I}'_j$ is

$$[y_m \doteq y_n \wedge \mathbf{R}(\mathbf{j}, y_1, \ldots, y_\ell)] \Rightarrow \mathbf{R}(\mathbf{q}, y_1, \ldots, y_\ell).$$

and $\mathbf{I}''_j$ is

$$[\neg y_m \doteq y_n \wedge \mathbf{R}(\mathbf{j}, y_1, \ldots, y_\ell)] \Rightarrow \mathbf{R}(\mathbf{s(j)}, y_1, \ldots, y_\ell).$$

Denote by $\mathbf{C_P}$ the sentence

$$\forall y_1 \ldots \forall y_\ell [\mathbf{I}_0 \wedge \ldots \wedge \mathbf{I}_{p-1}]$$

(recall that I_p is a halt instruction). Denote by $\mathbf{B}$ the sentence

$$\forall x \neg \mathbf{s}(x) \doteq 0 \wedge \forall x \forall y \, [\mathbf{s}(x) \doteq \mathbf{s}(y) \Rightarrow x \doteq y],$$

and by $\mathbf{A_P}(x_1, \ldots, x_\ell)$ the wff

$$[\mathbf{B} \wedge \mathbf{C_P} \wedge \mathbf{R}(0, x_1, \ldots, x_\ell)] \Rightarrow \exists z_1 \ldots \exists z_\ell \mathbf{R}(\mathbf{p}, z_1, \ldots, z_n).$$

We must prove that for each $(a_1, \ldots, a_\ell) \in \mathbf{N}^\ell$ the following are equivalent:

(I) The sentence $\mathbf{A_P}(\mathbf{a}_1, \ldots, \mathbf{a}_\ell)$ is valid.

(II) The RM program $\mathbf{P}$ halts on the input $(a_1, \ldots, a_m)$, that is,

$$(a_1, \ldots, a_\ell) \in Dom(\Phi^{(\ell)}_{\mathbf{P}}).$$

Choose $(a_1, \ldots, a_\ell) \in \mathbf{N}^\ell$. For $k = 0, 1, \ldots, \ell$ and $t = 0, 1, 2, \ldots$ let $r_k(t)$ denote the value in register R_k after t steps when the RM is running program $\mathbf{P}$ from the initial state $(0, a_1, \ldots, a_\ell)$. In terms of the notation introduced on page 198 this means that $r_k(t)$ is defined inductively by

$$(r_0(0), r_1(0), \ldots, r_\ell(0)) = (0, a_1, \ldots, a_\ell)$$

and

$$(r_0(t+1), r_1(t+1), \ldots, r_\ell(t+1)) = NXSTATE_{\mathbf{P}}(r_0(t), r_1(t), \ldots, r_\ell(t)),$$

where $NXSTATE_{\mathbf{P}}$ is the next state function of the program $\mathbf{P}$. For each $t = 0, 1, 2, \ldots$ let $\mathbf{D}_t$ denote the sentence

$$\mathbf{R}(\mathbf{r_0(t)}, \mathbf{r_1(t)}, \ldots, \mathbf{r_\ell(t)})$$

which results from $\mathbf{R}(x_0, x_1, \ldots, x_\ell)$ by replacing each x_k by the numeral $\mathbf{r_k(t)}$ which denotes the number $r_k(t)$. The next state function returns its input unchanged if R_0 points to a halt instruction (or contains a value larger than p) so, if $\mathbf{P}$ halts on the input $(a_1, a_2, \ldots, a_\ell)$ then the list

$$\mathbf{D}_0, \mathbf{D}_1, \mathbf{D}_2, \ldots \tag{1}$$

terminates in the sense that $\mathbf{D}_T = \mathbf{D}_{T+1} = \cdots$ for sufficiently large T.

We are now ready to prove the theorem, that is, to prove that (I) and (II) are equivalent. Assume (I). Define a model $\mathcal{M}$ with universe $\mathbf{N}$, the natural numbers, by taking $\mathbf{s}_{\mathcal{M}}(n) = n + 1$, $\mathbf{0}_{\mathcal{M}} = 0$, and

$$\mathcal{M} \models \mathbf{R}(\mathbf{b_0}, \mathbf{b_1}, \ldots, \mathbf{b_n})$$

iff $\mathbf{R}(\mathbf{b_0}, \mathbf{b_1}, \ldots \mathbf{b_n})$ appears in the list (1). Now

$$\mathcal{M} \models \mathbf{I}_j(\mathbf{b_0}, \mathbf{b_1}, \ldots, \mathbf{b}_\ell)$$

because each of these sentences asserts that if some state occurs during the computation, some other state occurs at the next step. Hence

$$\mathcal{M} \models \mathbf{C_P}.$$

$\mathcal{M} \models \mathbf{B}$ because in the model $\mathcal{M}$ the successor function and zero have their usual interpretations. Moreover, $\mathcal{M} \models \mathbf{R}(\mathbf{0}, \mathbf{a_1}, \ldots, \mathbf{a}_\ell)$ since this sentence is $\mathbf{D}_0$, the first sentence in the list (1). Thus

$$\mathcal{M} \models \mathbf{B} \wedge \mathbf{C_P} \wedge \mathbf{R}(\mathbf{0}, \mathbf{a_1}, \ldots, \mathbf{a}_\ell) \tag{2}.$$

Since we are assuming that $\mathbf{A_P}(\mathbf{a_1}, \ldots, \mathbf{a}_\ell)$ is valid this sentence holds (in particular) in the model $\mathcal{M}$:

$$\mathcal{M} \models \mathbf{A_P}(\mathbf{a_1}, \ldots, \mathbf{a}_\ell) \tag{3}$$

But we have seen that the antecedent of $\mathbf{A_P}$ holds in $\mathcal{M}$ so the consequent must holds as well:

$$\mathcal{M} \models \exists z_1 \ldots \exists z_\ell \mathbf{R}(\mathbf{p}, z_1, \ldots, z_\ell). \tag{4}$$

In other words there are numbers $b_1, b_2, \ldots, b_\ell \in \mathbf{N}$ such

$$\mathcal{M} \models \mathbf{R}(\mathbf{p}, \mathbf{b_1}, \ldots, \mathbf{b}_\ell) \tag{5},$$

so that the sentence

$$\mathbf{R}(\mathbf{p}, \mathbf{b_1}, \ldots, \mathbf{b}_\ell)$$

occurs in the list (1). But this means that $\mathbf{P}$ halts.

Now we prove the converse. Assume II, that is, that the computation halts. We must show that the sentence $\mathbf{A_P}(\mathbf{a_1}, \ldots, \mathbf{a}_\ell)$ is valid. To do this we choose a model $\mathcal{M}$ and prove (3). If (2) is false then (3) follows trivially. Assume (2). Then by induction on t we have that

$$\mathcal{M} \models \mathbf{D}_t$$

for $t = 0, 1, 2, \ldots$. Since the computation halts, the sentence

$$\mathbf{R}(\mathbf{p}, \mathbf{b_1}, \ldots, \mathbf{b}_\ell)$$

appears in the list (1). Therefore this sentence holds in $\mathcal{M}$, that is, (5) holds. It follows that (4) holds, and therefore (3) holds. **End of Proof.**

Church's theorem says that we cannot program a computer to accept as input a wff of predicate logic and produce as output a zero or one according to whether or not the input wff is valid. To make this precise we must assign a Gödel number $\#(\mathbf{A})$ to each wff $\mathbf{A}$ of predicate logic. There are many ways of defining such a Gödel numbering. The scheme we shall use here takes advantage of the Gödel numbering of finite sequences of natural numbers already developed in this chapter.

The first step is to assign a natural number, called a **code**, to each symbol s of the full predicate logic with vocabulary $\{\doteq, \mathbf{0}, \mathbf{s}, \mathbf{R}\}$, where $\mathbf{R}$ is an $(\ell + 1)$-ary predicate symbol. Let

$$v_0, v_1, v_2, \ldots$$

be a list of all the variables of first order logic. We shall assign the even number $2n$ as the code of the n^{th} variable v_n, and assign odd numbers as codes of the other symbols, including brackets and parentheses, as follows:

symbol	$\neg$	$\wedge$	$\vee$	$\Rightarrow$	$\Leftrightarrow$	$\exists$	$\forall$	$\doteq$
code	1	3	5	7	9	11	13	15

symbol	[	]	(	)	**0**	**s**	**R**	,
code	17	19	21	23	25	27	29	31

Next we define the Gödel number $\#(\mathbf{T})$ of a string $\mathbf{T}$ of symbols to be the Gödel number of the sequence of codes of the symbols. For example,

$$\#(\mathbf{0} + v_5 \doteq \mathbf{s}(v_0)) = \#(25, 29, 10, 15, 27, 21, 0, 23).$$

Each term and each wff, being a string of symbols, now has a Gödel number.

Lemma 4.12.2 *Let* $\mathbf{B}$ *be a wff in a full predicate logic with the vocabulary* $\{\doteq, \mathbf{0}, \mathbf{s}, \mathbf{R}\}$*, and let* $\beta_{\mathbf{B}}$ *be the total function from* $\mathbf{N}^2$ *into* $\mathbf{N}$ *defined by*

$$\beta_{\mathbf{B}}(a, b) = \#(\mathbf{B}(\mathbf{a}, \mathbf{b}))$$

where $\mathbf{B}(\mathbf{a}, \mathbf{b})$ *is the wff obtained from* $\mathbf{B}$ *by replacing all free occurrences of the the variable* x_1 *by the numeral* $\mathbf{a}$ *and all free occurrences of the the variable* x_2 *by the numeral* $\mathbf{b}$*. Then* $\beta_{\mathbf{B}}$ *is RM computable.*

Proof: By Church's Thesis, it is enough to describe an algorithm with input (a, b) which computes $\beta_{\mathbf{B}}(a, b)$. We sketch such an algorithm. The first step is to build a parsing sequence for $\mathbf{B}$. This can be done using an exercise from Chapter one, that each wff either starts with a negation symbol or quantifier, or has a main connective which is the unique binary connective preceded by one more left bracket than right bracket. Working through the parsing sequence, underline each bound occurrence of each variable. This can be done by underlining all occurrences of a variable which come from underlined occurrences earlier in the parsing sequence, and also underlining all occurrences of

x in a wff starting with $\forall x$ or $\exists x$. Form the string $\mathbf{b}$ consisting of b **s** symbols followed by one **0** symbol, and do the same for $\mathbf{a}$. Then replace in $\mathbf{B}$ all nonunderlined occurrences of x_1 by $\mathbf{a}$ and all nonunderlined occurrences of x_2 by $\mathbf{b}$. This results in the wff $\mathbf{B}(\mathbf{a}, \mathbf{b})$. Finally, compute the Gödel number of this wff as the output. **End of Proof.**

Recall from page 199 that a subset $V \subset \mathbf{N}$ is called **computable** iff its characteristic function c_V is RM computable (See page 193.) The set V is **undecidable** iff it is not RM computable.

Theorem 4.12.3 (Church's Theorem) *Let* $(\mathcal{P}, \mathcal{F})$ *be a vocabulary containing at least the symbols* $\{\doteq, \mathbf{0}, \mathbf{s}\}$ *and an* ℓ*-ary predicate symbol for each* ℓ*. Then the set*

$$V = \{\#(\mathbf{C}) : \models \mathbf{C}\}$$

of Gödel numbers of valid sentences in the full predicate logic with vocabulary $(\mathcal{P}, \mathcal{F})$ *is undecidable.*

Proof: As in Theorem 4.11.1 let S denote the set of pairs (a, b) such that the universal machine UNIV1 halts on the input (a, b). Theorem 4.11.1 says that S is undecidable (i.e. not computable). Under the assumption that V is computable we shall derive the contradiction that S is RM computable. Denote by $\mathbf{U}(x_1, x_2)$ the wff

$$\mathbf{A}_{UNIV1}(x_1, x_2, \mathbf{0}, \mathbf{0}, \ldots, \mathbf{0})$$

which results from the wff $\mathbf{A}_{UNIV1}$ by substituting $\mathbf{0}$ for the free variables other than x_1 and x_2. By Theorem 4.12.1 we have

$$S = \{(a, b) : \mathbf{U}(\mathbf{a}, \mathbf{b}) \text{ is valid}\}.$$

In other words

$$(a, b) \in S \iff \#(\mathbf{U}(\mathbf{a}, \mathbf{b})) \in V$$

or

$$c_S(a, b) = c_V(\beta_{\mathbf{U}}(a, b)).$$

By the preceding Lemma, the right hand side is an RM computable function of (a, b) under the assumption that c_V is RM computable. But

this says that c_S is RM computable which contradicts Theorem 4.11.1. **End of Proof.**

We can see from the proof of Church's Theorem that we did not really need to assume that the vocabulary has an ℓ-ary predicate symbol for each ℓ. Instead, we only needed a single ℓ-ary predicate symbol **R** where $R_{\ell-1}$ is the last register used by the universal RM program UNIV1. In fact, it can be shown that in any predicate logic with at least one predicate symbol which is binary or larger, the set of Gödel numbers of valid sentences is undecidable.

4.13 Simple Gnumber Problems (GNUM5)

This is the first of two problem sets using the GNUMBER or GNUMWIN program. In this assignment you only need the SIMPLE form of the program, which you start by hitting the S or RETURN key when you see the title screen.

The following sample register machine programs are located in directory GNUM5 on the distribution diskette. The SETUPDOS or SETUPWIN program will put them in a subdirectory called GNUM5 on your hard disk. The RM programs

ADD, MULT, PRED, DOTMINUS, and DIVREM

are explained in the text, and the commented listings on the distribution diskette are reproduced in Appendix B. Your problem assignment is to type in RM programs which compute the following functions. Test your answers out using the GNUMBER program (for DOS) or the GNUMWIN program (for Windows), then file your answers on your diskette and give them the names indicated.

In the formulas, x, y are the numbers in registers R_1, R_2 before running the program, and a, b are the numbers in these registers after running the program.

EQUAL:	$a = 1$ if $x = y, a = 0$ if not $x = y$
SQUARE:	$a = x * x$
ROOT:	$a =$ square root of x if x is a perfect square,

	undefined otherwise.
LESS:	$a = 1$ if $x < y$, $a = 0$ otherwise.
FACTRL:	$a = x!$ $(a = 1 * 2 * ... * x$ if $x > 0, a = 1$ if $x = 0)$
EXP:	$a = x$ raised to the y-th power if $x > 0$,
	$a = 0$ if $x = 0$ and $y > 0$
	undefined if $x = y = 0$.
PRIME:	$a = 1$ if x is prime $(2, 3, 5, 7, 11, \ldots)$, otherwise $a = 0$.
LENGTH:	$a =$ the number of decimal digits in x.
	(For example, 7402 has length 4).
DIGIT:	$a =$ the y-th digit in x, counting from 0 on the left
	if $y <$ the number of decimal digits in x, $a = 0$ otherwise.
	(For example, the 0-th digit of 7402 is 7).

In solving these problems, you may load in the sample programs and use them as building blocks if you wish.

The following functions are optional problems which require longer RM programs built up from LENGTH and DIGIT:

TERMS0:	$a =$ number of terms in the sequence with Gödel number x
	(not necessarily in standard form),
	the empty sequence having zero terms.
START:	$a =$ position of marker for the start of the y-th
	term in the sequence with Gödel number x
	(not necessarily in standard form),
	counting from 0 on the left.
	Undefined if $Terms(x) \leq y$.
PUTEND:	$a =$ the Gödel number of the sequence formed by adding y
	as one more term to the end of the sequence with Gödel
	number x,
	if x is a Gödel number in standard form.
	It doesn't matter what happens if x is not a standard
	Gödel number.
EXTRACT:	$c = Extract(x, y)$.
PUT:	$c = Put(x, y, z)$.

4.14 Advanced Gnumber Problems (GNUM6)

This assignment uses the Advanced form of the GNUMBER or GNUMWIN program. In the GNUMBER program, you start the advanced form by pressing the A key when you see the title screen.

The problems in this assignment deal with Gödel numbers of register machine programs. Each ARM instruction is a sequence consisting of an instruction letter and up to four numbers. The instruction letters are identified with numbers as follows:

$$H = 1, \quad Z = 2, \quad S = 3, \quad T = 4, \quad J = 5, \quad E = 6, \quad P = 7.$$

Each instruction, being a finite sequence of numbers, has a Gödel number. An ARM program **P** is a finite sequence of instructions $p_1, \ldots, p_n$. If instruction number m has Gödel number g_m, then the Gödel number of the whole program **P** is the Gödel number of the sequence $g_1, \ldots, g_n$.

The following sample advanced register machine programs are located in directory GNUM6 on the distribution diskette. The SETUPDOS or SETUPWIN program will put them in a subdirectory called GNUM6 on your hard disk. These ARM programs are named

FIVE, TERMS, JOIN, PARAM, NXSTATE0, NXSTATE, and UNIV.

In the Appendix there are pseudocode listings of these programs, as well as a reproduction of the commented listings which are on the diskette.

The following paragraphs explain the effect of these programs on the input and output registers. In the formulas, x, y, z, t are the numbers in registers R_1, R_2, R_3, R_4 before running the program, and a, b are the numbers in these registers after running the program.

FIVE: Puts the constants 0 through 5 in registers R_{20} through R_{25}. (It is often convenient to place this at the start of a program).

TERMS: If x is the standard Gödel number of a sequence, then a is the number of terms of the sequence. Otherwise a is zero.

JOIN: If x and y are Gödel numbers of ARM programs **P** and **Q** (not necessarily in standard form), z and t the numbers of instructions in **P** and **Q**, and registers R_{20} through R_{25} already contain 0 through 5, then a is the standard Gödel number of the ARM program **P** followed by **Q** with each jump target of **Q** increased by the number of instructions in **P**. Otherwise the output a can be anything. Extra bonus: this program ends with $z + t$ in R_9.

PARAM: If x is the standard Gödel number of an ARM program which neatly computes a function $f(.,.)$ of two variables, then a is the standard Gödel number of an ARM program which neatly computes the function $g(.) = f(y,.)$ of one variable. Otherwise the output a can be anything.

NXSTATE: If x is a Gödel number of an ARM program and y is a Gödel number of a sequence representing the register state, then b will be the standard Gödel number of the next state. (y and b are in R_4) The inputs need not be Gödel numbers in standard form.

NXSTATE0: If x is a Gödel number of an ARM program, registers R_{20} through R_{27} hold the constants 0 through 7, and y is a Gödel number of a sequence representing the register state, then b will be the standard Gödel number of the next state. (y and b are in R_4) (The program NXSTATE consists of a list of instructions which puts 0 through 7 in registers R_{20} through R_{27}, followed by the program NXSTATE0).

UNIV: The universal program in two variables. If x is a Gödel number of an ARM program **P** (not necessarily in standard form), then a is the output of the program **P** with inputs y and z in R_1 and R_2 and zero inputs elsewhere.

Your problem assignment is to type in register machine programs which compute the following functions. Test your answers out using the GODEL and UNGODEL commands in the GNUMBER program, then file your answers on your diskette and give them the names indicated. The approximate number of steps required for the program is shown. When you need small constants, it is recommended that you start your program with FIVE to put 0 through 5 in registers R_{20} through R_{25}.

CONCAT: (7 steps) If x and y are Gödel numbers of sequences of numbers in standard form, and z and t are the numbers of terms in these sequences, then a is the Gödel number of the first sequence followed by the second sequence. (Concatenation of two sequences). Otherwise the output a can be anything.

CONST: (19 steps) a is the Gödel number of an RM program which puts the constant x in R_1.

STAND: (25 steps) Given an input x, if **S** is the sequence which has x as a Gödel number (not necessarily in standard form), then a is the Gödel number of **S** in standard form.

SUCC: (23 steps) If x is the Gödel number of an ARM program in standard form which computes a function $f(.)$, then a is the Gödel number of an ARM program which computes the function $f(.)+1$. Otherwise the output a can be anything.

TOPREG: (28 steps) If x is the Gödel number in standard form of an ARM program **P**, then a is the largest number of a register mentioned in the first y instructions of **P**. Otherwise the output a can be anything.

COMPOSE: (61 steps) If x and y are Gödel numbers in standard form of ARM programs which neatly compute functions $g(.)$ and $h(.)$ of one variable, then a is the Gödel number in standard form of an ARM program which neatly computes the composition function $f(.) = g(h(.))$. Otherwise the output a can be anything.

BEFORE: (63 steps) $a = 1$ if the ARM program with Gödel number x, inputs y and z in R_1 and R_2, and zero inputs elsewhere, halts before t steps, and $a = 0$ otherwise. (Hint: This can be done by slightly modifying the ARM program UNIV. UNIV puts the time in Register 15)

RECUR: (90 steps) If x is the standard Gödel number of an ARM program **P** which neatly neatly computes a function $h(.,.)$, y is the largest register mentioned by **P**, and z is the number of instructions of **P**, then a is the standard Gödel number of an ARM program which neatly computes the function $f(\cdot)$ obtained from h by primitive recursion in the form

$$f(0) = 1, f(u+1) = h(f(u), u).$$

Otherwise the output a can be anything.

NEAT: (93 steps) If x is the Gödel number in standard form of an ARM program **P** which computes a function f in one variable, y is the largest register mentioned in **P**, and z is the number of instructions of **P**, then a is the Gödel number in standard form of an ARM program which neatly computes f. Otherwise the output a can be anything.

CVREC: (139 steps) If x is the Gödel number in standard form of an ARM program **P** which neatly neatly computes a function $h(.,.)$, y is the largest register mentioned by **P**, and z is the number of instructions of **P**, then a is the standard Gödel number of an ARM program which neatly computes the function $f(\cdot)$ obtained from h by course-of-values recursion in the form

$$f(0) = 1, f(u+1) = h(GN((f(0), ..., f(u)), u).$$

Otherwise the output a can be anything.

UBMIN: (165 steps) If x is the Gödel number in standard form of an ARM program **P** which neatly computes the characteristic function $h(.,.)$ of a binary relation $R(.,.)$, then a is the standard Gödel number of an ARM program which neatly computes the function $f(x) = \mu y\, R(x,y)$ obtained from R by unbounded minimalization. Otherwise the output a can be anything.

In solving your problems, you may load in the sample programs and use them as building blocks if you wish. Remember that the LOAD command can load ARM programs from the diskette starting at any point within your current instruction list.

4.15 Exercises

1. (a) Write an RM program which diverges (never halts) for every input.

(b) Write an RM program **P** such that: $P(x, y)$ never halts if $x = y$, $P(x, y)$ halts with output 0 in register one if $x \neq y$.

2. Write an RM program which uses only the instructions Z, S, and J, and has the effect of placing the number in register 3 into register 7, with all other registers left unchanged. (This shows that the T command can always be avoided in RM programs).

3. Show that any finite set, considered as a unary relation, is RM computable.

4. Show that the Fibonacci sequence

$$a_0 = 1, a_1 = 1, a_2 = 2, a_3 = 3, a_4 = 5, a_5 = 8, a_6 = 13, \ldots,$$

obtained by the rules

$$a_0 = 1,\ a_1 = 1,\ a_{n+2} = a_n + a_{n+1},$$

is RM computable.

5. Show that the zero function Z, successor function S, and projection functions I_i^n, defined by

$$Z(x) = 0$$
$$S(x) = x + 1$$
$$I_i^n(x_1, \ldots, x_n) = x_i$$

are RM computable.

6. (a) Write an RM program which computes a one-one onto mapping $\mathbf{N} \times \mathbf{N} \to \mathbf{N}$.

(b) Write an RM program which computes a one-one onto mapping $\mathbf{N} \to \mathbf{N} \times \mathbf{N}$.

7. Suppose instead of using the RM instruction set we use JN, S, Z, T, H, where $(JN, 1, 2, 6)$ would mean jump to instruction 6 if the contents of register 1 is not equal to the contents of register 2. Prove that every computable function is computable in this new sense.

8. Suppose we consider programs that only use the instructions S,Z,T,H; i.e. no jump instructions at all.

(a) Show that every function computable in this sense is total.

(b) Show that not every total computable function is computable in this sense.

9. Write an ARM program which computes the function

$f(x) =$ the standard Gödel number of the sequence $(0, 1, ..., x)$.

10. Give a universal ARM program for three inputs.

11. Prove that for each natural number n, there is a universal ARM program for n inputs.

12. Suppose an RM program P neatly computes a function f of one variable, and another RM program Q neatly computes a function g of one variable. Describe an RM program S such that, given inputs x in R_1 and y in R_2, S will halt with output 0 in R_1 if $f(x)$ and $g(y)$ are both defined, and S will never halt otherwise.

13. Let UNIV be a universal register machine program for two inputs. (That is, if p is the Gödel number of a program P, then UNIV with inputs p, x, y in registers R_1, R_2, R_3, and 0 in all other registers will produce the same output in R_1 as program P with inputs x, y in registers R_1, R_2 and 0 in all other registers). Let u be the Gödel number of UNIV. Show that UNIV with inputs u, u, u in registers R_1, R_2, R_3 and 0 in all other registers will eventually halt with output 0 in R_1.

14. Let us say that an ARM program U *simulates* an ARM program P which has Gödel number p if for all x and b, U with inputs p and x in $R1$ and $R2$ and zero elsewhere halts with output b in $R1$ if and only if P with input x in $R1$ and zero elsewhere halts with output b in $R1$. Suppose that U computes a total function of two variables, and that U simulates every ARM program P such that P has n instructions and computes a total function of one variable. Prove that U has at least $n-1$ instructions. (Hint: use a diagonal argument).

15. Define a *super* ARM to be an ARM with an extra instruction $N\,k\,m$ which acts as follows. Before: R_k holds the Gödel number of a simple RM program P and R_m holds the Gödel number of a state S. After: R_m holds the Gödel number of the new state formed by executing the j-th instruction of P where j is the 0-th term of S, and the program counter of the super ARM is increased by 1. Write a super ARM program U which is universal for RM programs in one input, i.e. which simulates every RM program in the sense of the preceding exercise. (Can be done in 7 instructions).

16. Suppose the numerical relation $R(x,y)$ is decidable. Show that the relation

$$\exists z[z \leq y \wedge R(x,z)]$$

is also decidable.

17. Show that the following relations are undecidable (Hint: In each case, assume the relation is decidable and prove that under that assumption the Halting Problem is decidable, contrary to Theorem 4.11.1):

(a) The set of all pairs (e,x) such that e is the Gödel number of an RM program which never halts with input x.

(b) The set of all pairs (e,x) such that e is the Gödel number of an RM program which outputs 0 with input x.

(c) The set of all numbers e such that e is the Gödel number of an RM program which computes a total function.

18. Give an example of an RM computable partial function whose graph (considered as a binary relation) is not RM computable.

In the following exercises, we introduce a new kind of machine, the **LRM machine**. It is obtained by modifying the definition of the RM machine as follows: The J (jump) instructions are eliminated and in their place are added the L (loop) instructions and the N (next) instructions. In any legal LRM the L and N instructions occur in pairs: for every L instruction there is a corresponding N instruction occurring later in the program. The instructions work as follows:

(Z, S, T, H) The LRM machine has the Z (zero), S (successor), T (transfer), and H (halt) instructions which operate in exactly the same way as in the RM machine.

For every $n = 1, 2, 3 \ldots$ there is a **loop instruction** (L,n) whose effect is to execute the steps between the *loop instruction* and the corresponding *next instruction* r_n times where r_n is the value in the register R_n when the loop instruction is encountered. After these r_n repetitions have been performed, the program jumps to the step immediately following the corresponding *next instruction.* (If $r_n = 0$, the program immediately jumps to the corresponding next instruction.)

For every $q = 0, 1, 2, \ldots$ there is a **next instruction** (N,q). The q-th instruction in the program must be a *loop instruction* (L,n). The (N,q) instruction acts like an unconditional jump: $(J,1,1,q)$. Notice that changing the value of R_n within the loop does not affect the number of times the loop is executed.

A **legal LRM program** is a finite list of LRM instructions which satisfies the following three requirements.

(1) The L and N instructions all occur in pairs as described above.

(2) There are no H instructions before the last nonhalt instruction.

(3) If any loop instruction occurs within a program fragment of the form

$$((L,n), I_{q+1}, \ldots, I_r, (N,q))$$

the corresponding next instruction must also occur in this fragment.

Thus the (L, N) pairs may be nested but if one loop starts within another loop, it must also end within the other loop .

A function $f(x_1, x_2, \ldots, x_n)$ is **LRM computable** if there is a legal LRM program **P** which computes it in the following sense: If the program is run after the registers are initialized so that for $k = 1, 2, \ldots, n$ the register R_k holds the value x_k and all other registers hold the value 0, then when it halts the register R_1 holds the value $f(x_1, x_2, \ldots, x_n)$. We call $f(x_1, x_2, \ldots, x_n)$ the n-ary **function computed by the LRM program P** on inputs $(x_1, x_2, \ldots, x_n)$.

For any legal LRM program **P** there is an RM program **Q** which performs in exactly the same way. It can be constructed as follows: Each part of the LRM program of the form

```
ALOOP   L   x           q: L n
        ⋮
        N   ALOOP       r: N q
```

is replaced by

```
        Z   count           q  : Z c
ALOOP   J   x,count, r+3    q+1: J n c r+3
        S   count           q+2: S c
        ⋮
        J   ALOOP           r+2: J 1 1 q+1
```

Here c is a register used nowhere else in the program (a different one for each (L, N) pair) and each time the replacement is made all the jump instructions (J, m, k, t) occurring after the L instruction must be corrected to $(J, m, k, t + 2)$. The replacement is repeated until no *loop* and *next* instructions remain.

19. The following LRM program computes the addition function:

```
ALOOP   L   y           0: L 2
        S   x           1: S 1
        N   ALOOP       2: N 0
```

Find an equivalent RM program.

20. The following LRM program computes the multiplication function:


```
        Z   z           0: Z 3
MLOOP   L   x           1: L 1
ALOOP   L   y           2: L 2
        S   z           3: S 3
        N   ALOOP       4: N 2
        N   MLOOP       5: N 1
        T   z,x         6: T 3 1
```

Find an equivalent RM program.

21. Write an LRM program which computes the characteristic function of the non-zero integers (that is, $f(x) = 0$ if $x = 0$ and $f(x) = 1$ if $x \neq 0$.)

22. Write an LRM program which computes the characteristic function of the set of odd integers.

23. Write an LRM program which computes cut-off subtraction:

$$x \dot{-} y = \begin{cases} x - y & \text{if } y \leq x \\ 0 & \text{if } x < y. \end{cases}$$

24. Write an LRM program which computes the quotient function:

$$qt(x, y) = \begin{cases} q & \text{if } x = qy + r \ 0 \leq r < y \\ 0 & \text{if } y = 0. \end{cases}$$

25. Write an LRM program which computes the remainder function:

$$rm(x, y) = \begin{cases} r & \text{if } x = qy + r \ 0 \leq r < y \\ 0 & \text{if } y = 0. \end{cases}$$

26. Show that if g, h, and p are LRM computable then so is the function f defined by

$$f(x) = \begin{cases} g(x) & \text{if } p(x) = 0 \\ h(x) & \text{otherwise.} \end{cases}$$

27. Prove that a legal LRM program always halts (on any inputs).

28. In this problem the set of primitive recursive functions is defined, and you are to show that all primitive recursive functions are LRM computable. We mentioned briefly in the text that the primitive recursive functions form the smallest class of numerical functions that contains the zero function Z, the successor function S, and the projection functions I_i^n (defined in Exercise 5) and that is closed under composition and primitive recursion. Here's the precise definition:

The set of **primitive recursive functions** is the smallest set of numerical functions such that

(1) The zero function Z is primitive recursive.

(2) The zero function S is primitive recursive.

(3) The projection functions I_i^n are primitive recursive.

(4) If $h : \mathbf{N}^m \to \mathbf{N}$ and the m functions $g_i : \mathbf{N}^n \to \mathbf{N}$ for $i = 1, 2, \ldots, m$ are all primitive recursive then the function $f : \mathbf{N}^n \to \mathbf{N}$ defined by

$$f(x_1, x_2, \ldots, x_n) = h(g_1(x_1, x_2, \ldots, x_n), \ldots, g_m(x_1, x_2, \ldots, x_n))$$

for $(x_1, x_2, \ldots, x_n) \in \mathbf{N}^n$ is also primitive recursive.

(5) if the functions $g : \mathbf{N}^n \to \mathbf{N}$ and $h : \mathbf{N}^{n+2} \to \mathbf{N}$ are primitive recursive, then the function $f : \mathbf{N}^{n+1} \to \mathbf{N}$ defined by

$$\begin{aligned} f(x_1, x_2, \ldots, x_n, 0) &= g(x_1, x_2, \ldots, x_n) \\ f(x_1, x_2, \ldots, x_n, y+1) &= h(x_1, x_2, \ldots, x_n, y, f(x_1, x_2, \ldots, x_n, y)) \end{aligned}$$

is also primitive recursive.

To prove that every primitive recursive function is LRM computable you must show that

- Z, S, and I_i^n are LRM computable;
- if the functions $h, g_1, \ldots, g_m$ are LRM computable, the the function f obtained from $h, g_1, \ldots, g_m$ by composition as in (4) is also LRM computable;
- if the functions g and h are LRM computable, the the function f obtained from g and h by primitive recursion as in (5) is also LRM computable.

29. Prove that a function is primitive recursive if and only if it is LRM computable. Hint: For $j, n = 1, 2, 3, \ldots$ denote by

$$\Phi_{\mathbf{P}}^{((n,j))}(x_1, x_2, \ldots, x_n)$$

the contents in register R_j when the LRM program $\mathbf{P}$ is run starting with x_k in register R_k for $k = 1, 2, \ldots, n$ and 0 in registers $R_{n+1}, R_{n+2}, \ldots$. (According to the definition a function f is LRM computable iff $f = \Phi_{\mathbf{P}}^{((n,1))}$ for some LRM program $\mathbf{P}$.) Prove that these functions are all primitive recursive by induction on the length of the LRM program $\mathbf{P}$.

30. The ALRM machine is obtained from the LRM machine by adding the E (extract) and P (put) instructions. Prove that a function is ALRM computable if and only if it is LRM computable.

31. All the *total* functions we have discussed so far are primitive recursive; In this exercise we construct a total RM computable function which is not primitive recursive. The basic idea is to describe a "universal" LRM program and show that the function computed by this program is RM computable but not LRM computable. First, modify the notion of Gödel number to apply to LRM programs. Prove that for every n there is a totally defined, RM computable, $(n+1)$-ary function

$$\psi = \psi(e, x_1, x_2, \ldots, x_n)$$

such that whenever e is the Gödel number of a legal LRM program $\mathbf{P}$, the number $\psi(e, x_1, x_2, \ldots, x_n)$ is the value of the function computed by $\mathbf{P}$ on inputs $(x_1, x_2, \ldots, x_n)$. Show ψ is RM computable but not LRM computable. Hint: See Exercise 9 on page 183.

32. Another total RM computable function which is not primitive recursive is the **Ackermann function** $\psi(p, z)$ defined by

$$\psi(p, z) = \psi_p(z)$$

where $\psi_0, \psi_1, \psi_2, \ldots$ is the sequence of primitive recursive functions defined inductively by

$$\psi_0(z) = z + 1$$

and

$$\psi_{p+1}(0) = \psi_p(1), \qquad \psi_{p+1}(z+1) = \psi_p(\psi_{p+1}(z)).$$

For example, $\psi_1(z) = z + 2$, $\psi_2(z) = 2z + 2$, $\psi_3(z) = 2^{z+2} + 3(2^z - 1)$. Show that for every n-ary primitive recursive function f there exists p such that

$$f(x_1, x_2, \ldots, x_n) < \psi_p(x_1 + x_2 + \cdots + x_n)$$

for all $(x_1, x_2, \ldots, x_n) \in \mathbf{N}^n$. Conclude that ψ is not primitive recursive (even though each ψ_p is).[9]

33. Write an RM program to compute the Ackermann function.

34. Definition. The class of **partial recursive functions** is the smallest class of numerical partial functions which contains the zero function, the successor function and the projection functions and which is closed under composition, primitive recursion, and unbounded minimalization.

Show that every partial recursive function is RM computable.

35. Show that every RM computable partial function is partial recursive.

36. Prove that there are only countably many RM programs, hence, only countably many RM computable functions. (Hint: Use the fact that every RM program has a Gödel number.)

[9]This exercise is tough. If you give up, see Epstein and Carnielli, *Computability*, Wadsworth & Brooks/Cole (1989) pages 110-114.

Chapter 5

The Incompleteness Theorems

Gödel's First Incompleteness Theorem says that there are sentences in the language of arithmetic which are true in the standard model $\mathcal{N}$ but are not provable from Peano Arithmetic. In itself, this result is not so surprising. It merely says the set of axioms **PA** for Peano Arithmetic is *incomplete* meaning that it not sufficiently powerful to enable us to give tableau proofs for *all* the true sentences of arithmetic. At this point one can still hope that we can add some additional axioms to **PA** to obtain a system of axioms which truly characterizes the natural numbers.

However, the proof of Gödel's Theorem shows much more:

> *No set* **H** *of axioms for arithmetic can have both the properties that (a) every sentence* **A** *which is true in the standard model* $\mathcal{N}$ *is a logical consequence of* **H** *and (b) there is a computer program which decides whether a given sentence* **B** *is an element of* **H**.

This means that there are intrinsic limitations on the methods mathematicians have used for centuries to arrive at the truth. Gödel's Second Incompleteness Theorem is even more devastating: *No system satisfying (b) is powerful enough to prove its own consistency.* This means that there is no tableau proof from the hypothesis set **H** that the set **H** is not at the root of a contradictory tableau.

The Incompleteness Theorems are closely related to the well known **Liar Paradox.**

Consider the sentence

This sentence is false.

If true it must be false – if false it is true. This version has caused a few philosophers to loose quite a lot of sleep over the centuries. Gödel's insight was to construct a sentence of arithmetic whose meaning is

This sentence cannot be proved.

If it could be proved it would be false. Hence it cannot be proved. But then it is true!

The construction of Gödel's sentence will borrow ideas from Cantor's diagonal method, which was discussed in Section 4.11 following the Halting Problem.

5.1 Coding Tableaus

To construct Gödel's sentence we must devise a way to formulate statements about **PA** within **PA**. This coding process is something like the Gödel numbering used in Chapter 4 to define universal machines.

We shall write $\#(a_1, \ldots, a_n)$ for the Gödel number in standard form of a finite sequence $(a_1, \ldots, a_n)$ of natural numbers as developed in Chapter 4.6, and also write $\#(t)$ for the codes which we shall introduce for other kinds of objects t. We shall use these codes to show that various numerical relations are computable, culminating in the proof relation $\mathrm{PRF}_{\mathbf{H}}$.

All of the proofs in this section proceed by giving an intuitive algorithm which computes the characteristic function of a set or relation on the natural numbers, and then invoking Church's Thesis to show that the relation is computable. A characteristic function is total and has the output 1 if the answer is yes and 0 if the answer is no. Thus for each input, our algorithms will halt in a finite number of steps and have either 1 or 0 as output.

The first step is to assign a code $\#(s)$ to each symbol s of the language of arithmetic. We may do this in the same way as we did in Section 4.10 on Church's Thesis in Chapter 4, except that we must now give codes to the symbols + and ∗. We assign the even numbers as codes of individual variables, and assign odd numbers as codes of the other symbols as follows:

symbol	$\neg$	$\wedge$	$\vee$	$\Rightarrow$	$\Leftrightarrow$	$\exists$	$\forall$	$\doteq$
code	1	3	5	7	9	11	13	15

symbol	[	]	(	)	**0**	**s**	+	∗
code	17	19	21	23	25	27	29	31

We define the code $\#(\mathbf{S})$ of a string $\mathbf{S}$ of symbols to be the Gödel number of the sequence of codes of the symbols. Each term and each wff, being a string of symbols, now has a code.

Next, we assign to each finite sequence of strings the Gödel number of the sequence of codes of terms of the sequence, that is,

$$\#(\mathbf{S}_1, \ldots, \mathbf{S}_n) = \#(\#(\mathbf{S}_1), \ldots, \#(\mathbf{S}_n)).$$

Note that a natural number can be used as a code in three ways: as a code of a symbol, as a code of a string of symbols, and as a code of a finite sequence of strings of symbols. As we continue we will introduce other types of codes. Thus when we write $\#(t)$ for the numerical code of an object t, we must specify whether t is a symbol, a string of symbols, a finite sequence of strings of symbols, or some other type of object.

We now show that the sets of codes of terms and of wffs are computable. This will be done using parsing sequences.

Lemma 5.1.1 *The set of codes of parsing sequences of terms is computable.*

Proof: We shall outline an algorithm which, given a natural number c as input, outputs 1 if c is the code of a parsing sequence for a term, and outputs 0 otherwise. The lemma will then follow by Church's Thesis.

First, form the sequence $(a_0, \ldots, a_n)$ of natural numbers with Gödel number c. Check to see whether the sequence is nonempty and each

a_i is the code of a string of symbols. If not, output 0 and stop. If so, run through $i = 0, \ldots, n$ and check whether a_i is either the code of a single variable or constant symbol, or is the code of a string of symbols obtained from one or two earlier strings in the list by one of the rules of formation for terms. If the answer is yes at each step, c is the code of a parsing sequence of a term, so we output 1 and stop. Otherwise output 0 and stop. The lemma now follows by Church's Thesis. **End of Proof.**

We say that a string **T** is a **substring** of a string **S** if **T** is a consecutive part of **S**, that is, **S** = **UTV** for some (possibly empty) strings **U** and **V**.

Theorem 5.1.2 *The set of codes of terms is computable.*

Proof: We outline an algorithm which, given a natural number c as input, outputs 1 if c is the code of a term, and outputs 0 otherwise.

Form the sequence $(a_0, \ldots, a_n)$ of natural numbers with Gödel number c. If the sequence is empty or some a_i is not the code of a symbol, output 0 and stop. Otherwise, c is the code of a string **S** of symbols. We wish to determine whether **S** has a parsing sequence. If there is a parsing sequence for **S**, then there is one with no repetitions, and each string of the sequence must be a substring of **S**. There are only finitely many sequences of distinct substrings of **S**. List all of these in a systematic way and use the preceding lemma to check whether at least one of them is a parsing sequence whose last term is **S**. Output 1 if yes and 0 if no, then stop. Again, the theorem now follows by Church's Thesis. **End of Proof.**

In the rest of this section, it should always be understood that Church's Thesis is to be invoked at the end of the proof.

Lemma 5.1.3 *The set of codes of atomic wffs is computable.*

Proof: Given input c, form the string of **S** symbols with code c. First check to see whether **S** has exactly one equality symbol. If not, output 0 and stop. If so, then **S** has the form **T** = **U**. If both **T** and **U** are terms, then **S** is an atomic wff, so we output 1 and stop. Otherwise output 0 and stop. **End of Proof.**

Lemma 5.1.4 *The set of codes of parsing sequences for wffs is computable.*

Proof: Similar to the corresponding result for terms, but one must check that each string in the sequence is either an atomic wff or is obtained from two earlier strings in the sequence by a rule of formation for wffs. **End of Proof.**

Theorem 5.1.5 *The set of codes of wffs is computable.*

Proof: Similar to the proof that the set of codes of terms is computable. **End of Proof.**

Since any finite set of natural numbers is computable, and Weak Arithmetic is a finite set of sentences called axioms, the set of codes of axioms of Weak Arithmetic is computable. Although Peano Arithmetic has an infinite set of axioms, we now show that the set of codes of its axioms is also computable.

Theorem 5.1.6 *The set of codes of axioms of Peano Arithmetic is computable*

Proof: Given an input c, we first use the preceding theorem to determine whether c is the code of a wff **A**. If not, output 0 and stop. If c is the code of a wff **A**, we next check whether **A** is one of the nine axioms of Weak Arithmetic. If it is, output 1 and stop. If not, we must check whether **A** is a case of the First Order Induction Scheme. We do this by systematically running through each of the finitely many substrings **B** of **A**, check whether **B** is a wff, and if so, check whether **A** is the string

$$\mathbf{B}(\mathbf{0}) \wedge \forall x\,[\mathbf{B}(x) \Rightarrow \mathbf{B}(\mathrm{s}(x))] \Rightarrow \forall x\,\mathbf{B}(x)]$$

for some variable x. If so, output 1 and stop. If **A** is neither an axiom of Weak Arithmetic or a case of the First Order Induction Scheme, output 0 and stop. **End of Proof.**

By refining the above arguments, one can show that various relations on strings of symbols which are part of the syntax of predicate logic are computable. For example, the set of codes of sentences is computable.

We now introduce codes for tableaus. A tableau has finitely many nodes including a root node, a parent function, a finite set of wffs called the hypothesis set attached to the root node, and a wff attached to each nonroot node.

For simplicity, we may take the nodes of a tableau $\mathbf{T}$ with $n+1$ nodes to be the natural numbers $0, 1, \ldots, n$, with 0 being the root node. The tableau can then be completely described by three finite sequences, the sequence of numbers

$$(\pi(1), \ldots, \pi(n))$$

where $\pi(i)$ is the parent node of the nonroot node i, the sequence

$$(\mathbf{B}_1, \ldots, \mathbf{B}_k)$$

of hypothesis wffs which are attached to the root node, and the sequence

$$(\Phi(1), \ldots, \Phi(n))$$

where $\Phi(i)$ is the wff attached to the nonroot node i.

For each nonroot node $i \in \{1, \ldots, n\}$, the parent node $\pi(i)$ belongs to the set $\{0, 1, \ldots, n\}$ of nodes. Let us assume further that the nodes were listed in such a way that for each $i \in \{1, \ldots, n\}$, $\pi(i) < i$. This can be done for any tableau by listing the nodes in the order in which the tableau was built using the extension rules, because a nonroot node is always added to a tableau after its parent. Note that the requirement that $\pi(i) < i$ for each nonroot node $i > 0$ guarantees that the root node 0 will be reached from any nonroot node i in finitely many steps by repeatedly taking parents.

Since we already have assigned codes to sequences of natural numbers and to sequences of strings, we may now take the code of a tableau $\mathbf{T}$ to be the Gödel number of the triple

$$\#(\mathbf{T}) = \#(a, b, c)$$

where a, b, and c are the codes

$$a = \#((\pi(1), \ldots, \pi(n)),$$

$$b = \#(\mathbf{B}_1, \ldots, \mathbf{B}_k),$$

and

$$c = \#(\Phi(1), \ldots, \Phi(n)).$$

Theorem 5.1.7 *The set of codes of tableaus, and the set of codes of tableau confutations, are computable.*

Proof: Given an input t, we first need an algorithm to check whether t is the code of a tableau. First, check whether t is the Gödel number of a triple (a, b, c) of natural numbers. If not, output 0 and stop. If so, check whether a is the Gödel number of a sequence of some length n such that each term of the sequence is a natural number less than n, that is, a is the Gödel number of a parent function π. If not, output 0 and stop. If so, then check whether $\pi(i) < i$ for each $i \in \{1, \ldots, n\}$. If the answer is no, output 0 and stop. Otherwise, check whether b and c are codes of sequences of codes of wffs. If not, output 0 and stop. If b and c are sequences of codes of wffs, check whether each nonroot node is obtained from an ancestor node using a tableau extension rule. This gives an algorithm for checking whether t is the code of a tableau. Output 1 if yes and 0 if no, then stop. This shows (by Church's Thesis as usual) that the set of codes of tableaus is computable.

To show that the set of codes of tableau refutations is computable, we first determine by the above algorithm whether an input t is the code of a tableau. If not, output 0 and stop. If t is the code of a tableau $\mathbf{T}$, we can then check whether $\mathbf{T}$ is a tableau confutation by systematically checking each branch of $\mathbf{T}$ to see whether it contains a contradictory pair. Output 1 if every branch contains a contradictory pair, and output 0 if not, then stop. **End of Proof.**

Definition 5.1.8 Let $\mathbf{H}$ be a set of wffs in the language of arithmetic. The **proof relation for H** is the binary numerical relation $\mathrm{PRF}_{\mathbf{H}}$ consisting of all pairs (x, y) such that x is the code of a wff and y is the code of a tableau proof of the wff coded by x from $\mathbf{H}$.

Theorem 5.1.9 *Let* $\mathbf{H}$ *be a set of wffs such that the set of codes of elements of* $\mathbf{H}$ *is computable. Then the proof relation* $PRF_{\mathbf{H}}$ *for* $\mathbf{H}$ *is computable.*

Proof: First check whether the input x is the code of a wff. If not, output 0 and stop. If x is the code of a wff $\mathbf{A}$, then check whether y is the code of a tableau refutation, say $\mathbf{T}$. If not, output 0 and stop.

Otherwise, check to see whether each hypothesis attached to the root of $\mathbf{T}$ is either an element of $\mathbf{H}$ or the negation of $\mathbf{A}$. In this step we use the assumption that the set of codes of elements of $\mathbf{H}$ is computable, so that we have a procedure for checking whether a number is the code of an element of $\mathbf{H}$. If we get a yes answer for each hypothesis, then (x, y) belongs to the relation $\mathrm{PRF}_{\mathbf{H}}$, and we output 1 and stop. Otherwise (x, y) does not belong to $\mathrm{PRF}_{\mathbf{H}}$, so we output 0 and stop. **End of Proof.**

5.2 Definability and Representability

In this section we introduce two ways in which a formula of arithmetic can express a numerical relation– definability in $\mathcal{N}$ and representability.

$\mathcal{N}$ is the standard model of arithmetic, whose universe is the set of natural numbers $\mathbf{N}$ and which has the usual interpretations of the symbols $\mathbf{0}, \mathbf{s}, +, *$. Recall from Chapter 3 that for each natural number m, the corresponding numeral $\mathbf{m}$ is the constant term consisting of m successor symbols $\mathbf{s}$ followed by the zero symbol $\mathbf{0}$. In $\mathcal{N}$, each numeral $\mathbf{m}$ will be interpreted by the element m of $\mathbf{N}$.

We shall often substitute numerals for free variables in a wff $\mathbf{A}$. In most cases it will be clear from the context which numeral goes with which free variable, and in such cases we shall write the sentence resulting from the substitution in the short form

$$\mathbf{A}(\mathbf{a_1}, \ldots, \mathbf{a_n})$$

instead of the long form

$$\mathbf{A}(x_1//\mathbf{a_1}, \ldots, x_n//\mathbf{a_n}).$$

Remember from Chapter 2 that for each model $\mathcal{M}$ with universe set M and each formula $\mathbf{A}$ with n free variables, the set of all n-tuples of elements of M which satisfy $\mathbf{A}$ in $\mathcal{M}$ is called the *graph* of $\mathbf{A}$ in $\mathcal{M}$. We now apply this concept to the standard model $\mathcal{N}$ of arithmetic.

Definition 5.2.1 Let R be an n-ary relation on $\mathbf{N}$ and let $\mathbf{B}$ be a wff in the vocabulary of arithmetic with the free variables $x_1, \ldots, x_n$.

We say that R is the **graph of B in $\mathcal{N}$**, or that R is **defined by B in $\mathcal{N}$**, if for all $a_1, \ldots, a_n \in \mathbf{N}$,

$$(a_1, \ldots, a_n) \in R \Longleftrightarrow \mathcal{N} \models \mathbf{B}(\mathbf{a_1}, \ldots, \mathbf{a_n}).$$

We say that R is **definable in $\mathcal{N}$** if it is defined by some wff $\mathbf{B}$ in $\mathcal{N}$.

Similarly, an n-ary function $f : \mathbf{N}^n \to \mathbf{N}$ is said to be defined by a wff $\mathbf{C}$ with free variables $x_1, \ldots, x_n, y$ if the $(n+1)$-ary relation $f(a_1, \ldots, a_n) = b$ is defined by $\mathbf{C}$ in $\mathcal{N}$.

Not all relations on $\mathbf{N}$ are definable in $\mathcal{N}$; in fact there are uncountably many relations on $\mathbf{N}$ but only countably many definable relations on $\mathbf{N}$. For example, Tarski's Theorem, Theorem 5.5.8, shows that the set of all codes of sentences which are true in $\mathcal{N}$ is not definable in $\mathcal{N}$.

Since the symbols $\mathbf{0}, \mathbf{s}, +, *$ of arithmetic are interpreted in the natural way in $\mathcal{N}$, the zero function, successor function, addition function, and multiplication function are defined in $\mathcal{N}$ by the atomic formulas $\mathbf{0} \doteq y, \mathbf{s}(x) \doteq y, x_1 + x_2 \doteq y, x_1 * x_2 \doteq y$. Similarly, the equality relation is defined by the atomic formula $x \doteq y$.

Other examples are easy to work out. The order relation $\leq$ is defined in $\mathcal{N}$ by the formula $\exists z\, x + z \doteq y$, the square function is defined in $\mathcal{N}$ by the formula $x * x \doteq y$, and the set of even numbers is defined in $\mathcal{N}$ by the formula $\exists z\, x \doteq z + z$.

We shall need another, much stronger, way in which a formula in the language of arithmetic can express a numerical relation– representability. Let us first recall the nine axioms of Weak Arithmetic.

Axioms of Weak Arithmetic

1. $\forall x \neg \mathbf{s}(x) \doteq \mathbf{0}$
2. $\forall x \forall y\, [\mathbf{s}(x) \doteq \mathbf{s}(y) \Rightarrow x \doteq y]$
3. $\forall x\, x{+}\mathbf{0} \doteq x$
4. $\forall x \forall y\, x{+}\mathbf{s}(y) \doteq \mathbf{s}(x{+}y)$
5. $\forall x\, x{*}\mathbf{0} \doteq \mathbf{0}$
6. $\forall x \forall y\, x{*}\mathbf{s}(y) \doteq (x{*}y){+}x$

7. $\forall x\,[x \leq \mathbf{0} \Rightarrow x \doteq \mathbf{0}]$

8. $\forall x \forall y\,[x \leq \mathbf{s}(y) \Rightarrow [x \leq y \vee x \doteq \mathbf{s}(y)]]$

9. $\forall x \forall y\,[x \leq y \vee y \leq x]$.

Definition 5.2.2 Let $\mathbf{B}$ be a wff with free variables $x_1, \ldots, x_n$ in the language of arithmetic and let R be an n-ary relation on N. We say that $\mathbf{B}$ **represents** R if for all $a_1, \ldots, a_n$ in $\mathbf{N}$,

1. If $(a_1, \ldots, a_n) \in R$, then $\mathbf{WA} \vdash \mathbf{B}(\mathbf{a_1}, \ldots, \mathbf{a_n})$,

2. If $(a_1, \ldots, a_n) \notin R$, then $\mathbf{WA} \vdash \neg\mathbf{B}(\mathbf{a_1}, \ldots, \mathbf{a_n})$.

We say that a wff $\mathbf{C}$ with free variables $x_1, \ldots, x_n, y$ represents the numerical function f of n variables if $\mathbf{C}$ represents the relation

$$f(x_1, \ldots, x_n) = y.$$

Finally, a relation or function is **representable** if some wff represents it.

We shall call clause (1) in the above definition the "first half" and clause (2) the "second half" of representability.

The value of knowing that a particular n-ary relation R is representable is that true statements of the form $(a_1, \ldots, a_n) \in R$ or $(a_1, \ldots, a_n) \notin R$ can be translated into provable first-order sentences from Weak Arithmetic in which references to R are replaced by the representing wff and natural numbers m are replaced by *numerals* $\mathbf{m}$. Important information about a theory (like **PA**) can often be uncovered by showing that the theory is able to "mirror" –via representability– some well-understood portion of mathematics. The next proposition shows that every relation which is representable is definable in $\mathcal{N}$.

Proposition 5.2.3 *If a wff* $\mathbf{B}$ *represents a relation* R*, then* $\mathbf{B}$ *defines* R *in* $\mathcal{N}$.

Proof: Since each axiom of **WA** is true in $\mathcal{N}$, every wff which is provable from **WA** is true in $\mathcal{N}$. Suppose $\mathbf{B}$ represents R, and let $a_1, \ldots, a_n$ be natural numbers. If $(a_1, \ldots, a_n) \in R$ then $\mathbf{WA} \vdash \mathbf{B}(\mathbf{a_1}, \ldots, \mathbf{a_n})$ and hence $\mathcal{N} \models \mathbf{B}(\mathbf{a_1}, \ldots, \mathbf{a_n})$. On the other hand, if $(a_1, \ldots, a_n) \notin R$, then $\mathbf{WA} \vdash \neg\mathbf{B}(\mathbf{a_1}, \ldots, \mathbf{a_n})$, so $\mathcal{N} \models \neg\mathbf{B}(\mathbf{a_1}, \ldots, \mathbf{a_n})$, and hence it is not the case that $\mathcal{N} \models \mathbf{B}(\mathbf{a_1}, \ldots, \mathbf{a_n})$. This shows that $\mathbf{B}$ defines R in $\mathcal{N}$. **End of Proof.**

We shall see later that there are relations which are definable in $\mathcal{N}$ but not representable. An example of such a relation is the set of all Gödel numbers of formulas which are provable from **WA**. This is not easy to see, and is one of the consequences of the incompleteness theorems.

Functions which certainly ought to be representable are those which correspond to function symbols in the language of arithmetic, namely, zero, addition, multiplication, and the successor function. We would also expect that the relations $=$ and $\leq$ are representable. We have already seen that each of these is definable in $\mathcal{N}$. Additional work is needed to show that they are representable. To prove that a relation or function is represented by a wff, one must show that each of an infinite list of other wffs is provable in Weak Arithmetic. To give some idea of what is involved, we now show that the relations $=$ and $\leq$ and the addition function are representable.

Proposition 5.2.4 *The equality relation is represented by the wff*

$$x \doteq y.$$

Proof: For the first half of the definition of representability, suppose that $a = b$. Then $\mathbf{a}$ and $\mathbf{b}$ are the same term, so $\mathbf{a} \doteq \mathbf{b}$ is provable from the empty hypothesis set and hence is provable from **WA**.

For the second half we must to show that whenever $a < b$, $\mathbf{WA} \vdash \neg\mathbf{a} \doteq \mathbf{b}$. This was done for the particular case $a = 1, b = 3$ in Chapter 3. The same method can be used in general, but requires an induction on natural numbers. We show by induction on n that

(1) $\qquad n < m$ implies $\mathbf{WA} \vdash \neg\,\mathbf{n} \doteq \mathbf{m}$.

Basis Step: Let $n = 0$ and write $m = k + 1$. Then $\mathbf{s(k)} \doteq \mathbf{m}$ and so by Axiom 1, we have

$$\mathbf{WA} \vdash \neg\, \mathbf{0} \doteq \mathbf{s(k)}$$

as required.

Successor Step: Assuming (1) we show

(2) $$n + 1 < m \text{ implies } \mathbf{WA} \vdash \neg\, \mathbf{s(n)} \doteq \mathbf{m}.$$

Write $m = k + 1$. Using the fact that $n < k$ and using (1) (with m replaced by k) as a hypothesis, we have the following tableau proof of (2). Rather than writing out all nine axioms of **WA** as hypotheses for our tableau, only those that are needed in the proof are shown.

(3)	$\neg\, \mathbf{n} \doteq \mathbf{k}$	Inductive hypothesis
(4)	$\forall x\, \forall y\, [\mathbf{s}(x) \doteq \mathbf{s}(y) \Rightarrow x \doteq y]$	Axiom 2
(5)	$\neg\, \neg\, \mathbf{s(n)} \doteq \mathbf{s(k)}$	$\neg$ to be proved
(6)	$\mathbf{s(n)} \doteq \mathbf{s(k)}$	by (5)
(7)	$\mathbf{s(n)} \doteq \mathbf{s(k)} \Rightarrow \mathbf{n} \doteq \mathbf{k}$	by (4) twice
(8)	$\mathbf{n} \doteq \mathbf{k}$	by (6) and (7)

Thus equality is represented by the wff $x \doteq y$. **End of Proof.**

We make a few observations about the proof. First of all, what lets us use induction when the sentences of the First Order Induction Principle are not among the axioms **WA**? What we have done is to use ordinary induction on the natural numbers *outside of our formal system* to obtain an infinite sequence of proofs from **WA**; for each $n < m$ we obtained a proof from **WA** of the sentence $\neg\mathbf{n} \doteq \mathbf{m}$. This was possible

because the superscripts m and n are ordinary natural numbers –not formal expressions in **WA**– and so ordinary induction applies.

By contrast, the proof of $\forall x\, \neg x \doteq \mathbf{s}(x)$ from Peano Arithmetic in Example 3.7.4 used the formal induction axiom

$$\mathbf{B(0)} \wedge \forall x\, [\mathbf{B}(x) \Rightarrow \mathbf{B}(\mathbf{s}(x))] \Rightarrow \forall x\, \mathbf{B}(x)$$

of **PA**, where $\mathbf{B}(x)$ is $\neg x \doteq \mathbf{s}(x)$. Ordinary induction did not apply in that case because the x in $\neg x \doteq \mathbf{s}(x)$ is a variable in predicate logic, not an ordinary natural number.

Secondly, notice that we used the induction hypothesis (3) in the hypothesis set of our tableau. This is a technique that is very useful in working out tableau proofs and is an example of the Learning Rule introduced in Section 2.13: Given sentences **A** and **B**, if $\mathbf{H} \vdash \mathbf{A}$, then by the Learning Rule we can use **A** on any branch of a tableau with hypothesis set $\mathbf{H} \cup \{\neg\mathbf{B}\}$ in building a tableau proof of **B** from **H**. In particular, **A** can be assumed to be in the hypothesis set of such a tableau.

The extra rules of tableau proofs introduced in Section 2.13 for pure predicate logic also hold for full predicate logic. A useful application of the Learning Rule in full predicate logic is that if $\mathbf{H} \vdash \mathbf{A}$ and $\mathbf{H} \vdash \sigma \doteq \tau$, then $\mathbf{H} \vdash \mathbf{A}(\sigma//\tau)$, where σ and τ are terms and τ is free for σ in **A**.

Proposition 5.2.5 *The addition function is represented by the wff*

$$x + y \doteq z.$$

Proof: For the first half, we must show that for all $m, n, p \in \mathbf{N}$ such that $m + n = p$,

$$\mathbf{WA} \vdash \mathbf{m{+}n} \doteq \mathbf{p}.$$

Note that $\mathbf{m} + \mathbf{n}$ is a different term than $\mathbf{p}$. For example, $\mathbf{2} + \mathbf{3}$ is the term $\mathbf{s(s(0))} + \mathbf{s(s(s(0)))}$, while $\mathbf{5}$ is the term $\mathbf{s(s(s(s(s(0)))))}$.

A tableau proof is shown in Figure 5.1. It proceeds holding m fixed and using an induction on n to show that for each n, there is a tableau proof of $\mathbf{m} + \mathbf{n} \doteq \mathbf{p}$ from **WA** where $p = m + n$.

Basis Step ($n = 0$) We show $\mathbf{WA} \vdash \mathbf{m{+}0 \doteq m}$.

(1)	$\forall x\, x{+}\mathbf{0} \doteq x$	Axiom 3
(2)	$\neg\, \mathbf{m + 0 \doteq m}$	$\neg$ to be proved
	|	
(3)	$\mathbf{m{+}0 \doteq m}$	by (1)

Induction Step Let $p = m + n$. We assume $\mathbf{WA} \vdash \mathbf{m{+}n \doteq p}$ and prove $\mathbf{WA} \vdash \mathbf{m{+}s(n) \doteq s(p)}$.

(4)	$\mathbf{m{+}n \doteq p}$	Induction hypothesis
(5)	$\forall x \forall y\, x{+}\mathbf{s}(y) \doteq \mathbf{s}(x{+}y)$	Axiom 4
(6)	$\neg\mathbf{m{+}s(n) \doteq s(p)}$	$\neg$ to be proved
	|	
(7)	$\mathbf{m{+}s(n) \doteq s(m{+}n)}$	by (5) (twice)
	|	
(8)	$\mathbf{m{+}s(n) \doteq s(p)}$	by (4), (7) and an $=$ rule

Figure 5.1: A tableau proof of $\mathbf{m{+}n \doteq p}$

For the second half of representability, we must show that

$$m + n \neq r \text{ implies } \mathbf{WA} \vdash \neg\mathbf{m{+}n \doteq r}.$$

Let $p = m + n$ and assume that $p \neq r$. By the representability of $=$ we have

$$\mathbf{WA} \vdash \neg\mathbf{p \doteq r}.$$

Substituting $\mathbf{m + n}$ for $\mathbf{p}$ (using the first half of representability and the Learning Rule), we have

$$\mathbf{WA} \vdash \neg\mathbf{m + n \doteq r}$$

as required. **End of Proof.**

The following lemma is often useful in proving that things are representable. We shall use it in showing that the order relation on $\mathbf{N}$ is representable.

Lemma 5.2.6 *For any natural number n,*

$$\mathbf{WA} \vdash \forall x\, [x{\leq}\mathbf{n} \Leftrightarrow x \doteq \mathbf{0} \vee x \doteq \mathbf{1} \vee \cdots \vee x \doteq \mathbf{n}].$$

Proof: We proceed by induction on n.

Basis step: We must prove from **WA** that

(1) $$\forall x\, [x \leq \mathbf{0} \Leftrightarrow x \doteq \mathbf{0}].$$

Consider any x. If $x \doteq \mathbf{0}$ then by Axiom 1, $x + \mathbf{0} \doteq \mathbf{0}$, so $\exists y\, x + y \doteq \mathbf{0}$ and $x \leq \mathbf{0}$. If $x \leq \mathbf{0}$, then $x \doteq \mathbf{0}$ by Axiom 3. Therefore (1) is provable from **WA**.

Successor step: Assume that

(2) $$\forall x\, [x \leq \mathbf{n} \Leftrightarrow x \doteq \mathbf{0} \vee \cdots \vee x \doteq \mathbf{n}]$$

is provable from **WA**. We must show that

(3) $$\forall x\, [x \leq \mathbf{s(n)} \Leftrightarrow x \doteq \mathbf{0} \vee \cdots \vee x \doteq \mathbf{n} \vee x \doteq \mathbf{s(n)}]$$

is provable from **WA**. By the Learning Rule, it is enough to prove (3) from the hypothesis set **WA** plus the extra hypothesis (2).

Consider any x. Assume that $x \leq \mathbf{s(n)}$. By Axiom 8,

$$x \leq \mathbf{n} \vee x \doteq \mathbf{s(n)}.$$

Then by (2),

$$x \doteq \mathbf{0} \vee \cdots \vee x \doteq \mathbf{n} \vee x \doteq \mathbf{s(n)}.$$

For the other direction, assume that

$$x \doteq \mathbf{0} \vee \cdots \vee x \doteq \mathbf{n} \vee x \doteq \mathbf{s(n)}.$$

By (2) again,

$$x \leq \mathbf{n} \vee x \doteq \mathbf{s(n)}.$$

Expanding the abbreviation for $x \leq y$, we have

$$\exists z\, x + z \doteq \mathbf{n} \vee x \doteq \mathbf{s(n)}.$$

If $x + z \doteq \mathbf{n}$, then by Axiom 4, $x + \mathbf{s}(z) \doteq \mathbf{s(n)}$ and hence $x \leq \mathbf{s(n)}$. If $x \doteq \mathbf{s(n)}$ then by Axiom 3, $x + \mathbf{0} \doteq \mathbf{s(n)}$, so again $x \leq \mathbf{s(n)}$.

We have shown that

$$x \leq \mathbf{s(n)} \Leftrightarrow x \doteq \mathbf{0} \vee \cdots \vee x \doteq \mathbf{n} \vee x \doteq \mathbf{s(n)}.$$

The required wff (3) follows by the Generalization Rule. **End of Proof.**

Proposition 5.2.7 *The order relation* $\{(x, y) : x \leq y\}$ *on* $\mathbf{N}$ *is represented by the wff* $\exists z\, x + z \doteq y$.

Proof: We begin with the first half. If $m \leq n$, let k be such that $m + k = n$. By Proposition 5.2.5 we have

$$\mathbf{WA} \vdash \mathbf{m{+}k} \doteq \mathbf{n}.$$

It follows that

$$\mathbf{WA} \vdash \exists z\, \mathbf{m}{+}z \doteq \mathbf{n}.$$

To establish the second half, assume $m \not\leq n$. Then $n < m$. Since the equality relation is represented by the wff $x \doteq y$, for all $j \leq n$ we have

$$\mathbf{WA} \vdash \neg \mathbf{j} \doteq \mathbf{m}.$$

Using Lemma 5.2.6 along with each of the above statements in our hypothesis set, we have the following tableau:

(1)	$\forall x\,[x \leq \mathbf{n} \Rightarrow x \doteq \mathbf{0} \vee x \doteq \mathbf{1} \vee \cdots \vee x \doteq \mathbf{n}]$	
(2)	$\neg \mathbf{0} \doteq \mathbf{m}$	
(3)	$\neg \mathbf{1} \doteq \mathbf{m}$	
(⋮)	⋮	
$(n+2)$	$\neg \mathbf{n} \doteq \mathbf{m}$	
$(n+3)$	$\neg\neg \mathbf{m} \leq \mathbf{n}$	¬ to be proved
$(n+4)$	$\mathbf{m} \leq \mathbf{n}$	by $(n+11)$
$(n+5)$	$\mathbf{m} \leq \mathbf{n} \Rightarrow \mathbf{m} \doteq \mathbf{0} \vee \cdots \vee \mathbf{m} \doteq \mathbf{n}$	by (10)
$(n+6)$	$\mathbf{m} \doteq \mathbf{0} \vee \cdots \vee \mathbf{m} \doteq \mathbf{n}$	by $(n+5)$
	$\neg \mathbf{m} \leq \mathbf{n}$	by $(n+5)$
$(n+7)$	$\mathbf{m} \doteq \mathbf{0}$ $\quad$ $\mathbf{m} \doteq \mathbf{1}$ $\quad$ $\cdots$ $\quad$ $\mathbf{m} \doteq \mathbf{n}$	by $(n+6)$

For readability, in step $(n+7)$ we applied the $\boxed{\vee}$ rule n times simultaneously. **End of Proof.**

Here is an example which shows in a simple case what can (and

cannot) be done with a relation that is representable.

In Exercise 6 the reader is asked to verify that the set E of even numbers is represented by the wff

$$\mathbf{E}(x) = \exists z\, x \doteq z + z.$$

Now, consider the following simple property of the even numbers:

(*) The sum of any odd number and any even number is an odd number.

This fact can be expressed formally by the wff

$$\mathbf{B}(x, y) = \neg\mathbf{E}(x) \wedge \mathbf{E}(y) \Rightarrow \neg\mathbf{E}(x + y).$$

Because $\mathbf{E}(x)$ represents E, it is easy to show (see Exercise 6) that for all $m, n \in \mathbf{N}$

$$\mathbf{WA} \vdash \mathbf{B(m, n)}.$$

Thus, this simple property of the even numbers is reflected in the formal setting of **WA**–*with respect to the numerals* $\mathbf{0, 1, 2, \cdots}$. It should be emphasized that the notion of representability we are using here is *not* strong enough to guarantee that such properties can always be translated and proved in **WA** without such a restriction on the substitution values. In the present example, (*) could be translated into the following *sentence*:

$$\mathbf{C} = \forall x \forall y\, [\neg\mathbf{E}(x) \wedge \mathbf{E}(y) \Rightarrow \neg\mathbf{E}(x + y)].$$

This sentence makes a much stronger assertion than $\mathbf{B(m, n)}$ for all m and n: it states that the property (*) holds for all possible interpretations of variables in a model of **WA**, not merely the standard ones. As a matter of fact, it can be shown (see Exercise 6) that $\mathbf{WA} \nvdash \mathbf{C}$; a counter-model is given in Example 3.7.4 in Chapter 3.

5.3 The Equivalence Theorem

In the preceding section, we developed a very short list of representable functions and relations. The following theorem shows that the set of representable relations is richer than one might think from our examples, and in fact is the same as the set of all computable relations.

Theorem 5.3.1 (Equivalence Theorem) .

A numerical relation is representable if and only if it is computable. Similarly, a total numerical function is representable if and only if it is computable.

We now prove one half of the Equivalence Theorem using Church's Thesis. The other half of the theorem will be proved in the next section.

Proof, first half: Using Church's Thesis, we prove that every representable relation is computable.

Let the n-ary relation R be represented by the wff $\mathbf{A}$. We describe an algorithm for computing the characteristic function of R. Consider an input $(a_1, \ldots, a_n)$. Let $\mathbf{B}$ be the wff $\mathbf{A(a_1, \ldots, a_n)}$. Repeat the following process for each $m = 0, 1, 2, \ldots$: Using the computability of the proof relation $\mathrm{PRF}_{\mathbf{WA}}$, determine whether or not m is the code of a tableau proof of $\mathbf{B}$ from **WA**, and if so, then output 1 and stop. Otherwise, determine whether or not m is the code of a tableau proof of $\neg\mathbf{B}$ from **WA**, and if so, output 0 and stop. Since $\mathbf{A}$ represents R, for each input $(a_1, \ldots, a_n)$ there will be either a tableau proof of $\mathbf{B}$ or of $\neg\mathbf{B}$, so the algorithm will eventually stop and produce an output.

This computes the characteristic function of R as required. By Church's Thesis, R is computable.

Now suppose the total function f in n variables is representable, and let R be the relation $f(x_1, \ldots, x_n) = y$. By definition, the relation R is representable, and by the first paragraph its characteristic function is neatly computable by some RM program $\mathbf{P}$. We shall make a new RM program $\mathbf{Q}$ which, for an input $(a_1, \ldots, a_n)$, computes in turn the characteristic function of $(a_1, \ldots, a_n, b) \in R$ for $b = 0, b = 1, \ldots$, continuing until an answer of 1 is found, and then outputs the current value of b. To do this, let R_k be a register beyond the last register which is used by $\mathbf{P}$ and let p be the length of $\mathbf{P}$. $\mathbf{Q}$ is the program

0	Z	k		
1	Z	k+1		
2	S	k+1		
3	**P**			
$p+3$	J	1	k+1	$p+6$
$p+4$	S	k		
$p+5$	J	1	1	3
$p+6$	T	k	1	
$p+7$	H			

For each input, the program **Q** will eventually halt because the function f is total, and **Q** will compute the original function f. **End of Proof.**

Note that the Equivalence Theorem as stated only applies to *total* functions. What happens in the case of *partial* functions? It turns out that every representable partial function is computable, but there are computable partial functions which are not representable. One explanation for this difference is that the class of computable functions is closed under unbounded minimalization (recall Theorem 4.8.2) while the class of representable functions is not. Here is an example of a partial function which is defined using unbounded minimalization from a computable relation (and hence is itself computable) but which is *not* representable.

Example. Define the ternary relation R by

$$(e,a,b) \in R \iff \left\{ \begin{array}{l} e \text{ is the Gödel number of an} \\ \text{RM program which halts with input } a \\ \text{after executing } b \text{ instructions.} \end{array} \right.$$

Define the partial function f by

$$f(e,a) = \mu\, b\, (e,a,b) \in R.$$

(As usual, we understand by this definition that f has the same domain as the function on the right and agrees with it on this domain.) We can compute f with the following RM program **Q**: With input e, a, **Q** executes UNIV1 and halts if and only if UNIV1 halts. If UNIV1 halts, then **Q** outputs the number of steps needed by the program $\mathbf{P}_e$ coded by e to halt on input a. (See Advanced GNUMBER problem BEFORE in Section 4.14.)

To see that f is not representable, first note that by Exercise 5, if a partial function is representable, its domain is representable (as a unary relation). By Theorem 4.11 on the undecidability of the Halting Problem, the domain of this particular function f is not computable. By the Equivalence Theorem, all representable relations are computable. Then the domain of f, and hence f itself, is not representable.

In order to represent all computable partial functions, we shall need another notion, called weak representability.

Definition 5.3.2 An n-ary relation R on **N** is **weakly represented** by a wff **B** with free variables $x_1, \ldots, x_n$ if for all $a_1, \ldots, a_n$ in **N**,

$$(a_1, \ldots, a_n) \in R \iff \mathbf{WA} \vdash \mathbf{B}(\mathbf{a_1}, \ldots \mathbf{a_n}).$$

A function $f(x_1, \ldots, x_n)$ is weakly represented by a wff **C** with free variables $x_1, \ldots, x_n, y$ if the $n+1$-ary relation $f(x_1, \ldots, x_n) = y$ is weakly represented by **C**.

Every representable relation or function is weakly representable, but a relation can be weakly representable and not representable. The incompleteness theorems will show, as an example, that the set of all codes of sentences which are provable from **WA** is weakly representable but not representable.

The difference between representability and weak representability is that in the case $(a_1, \ldots, a_n) \notin R$, weak representability merely requires that $\mathbf{B}(\mathbf{a_1}, \ldots, \mathbf{a_n})$ is not provable from **WA**, while representability requires that the wff $\neg\mathbf{B}(\mathbf{a_1}, \ldots, \mathbf{a_n})$ is provable from **WA**.

Here is an Equivalence Theorem for partial functions.

Theorem 5.3.3 *A partial numerical function is weakly representable if and only if it is computable.*

As we did for the Equivalence Theorem, we shall now prove one half of the above theorem using Church's Thesis, leaving the proof of the other half for the next section.

Proof, first half: We prove that every weakly representable function is computable. Suppose that $f(x_1, \ldots, x_n)$ is weakly represented by a

wff **B**. Then f can be computed by the following algorithm. We are given an input $(a_1, \ldots, a_n)$. For $m = 0, 1, 2, \ldots$, systematically list all tableaus with at most m nodes, only wffs of length at most m, and at most the variables $v_0, \ldots, v_m$, whose hypotheses are **WA** together with a wff of the form

$$\neg \mathbf{B}(\mathbf{a_1}, \ldots, \mathbf{a_n}, \mathrm{b}).$$

Continue until a tableau proof is found, going on forever if a tableau proof is never found. If a tableau proof is found, stop with output b where the extra hypothesis is

$$\neg \mathbf{B}(\mathbf{a_1}, \ldots, \mathbf{a_n}, \mathrm{b}).$$

End of Proof.

5.4 Computable Implies Representable

In this section we prove the second half of the Equivalence Theorem, that every total computable function is representable in Weak Arithmetic. The proof will make use of the notion of a wff being **definable in $\mathcal{N}$**, which was introduced in Definition 5.2.1. The main steps will be as follows.

- Introduce the notion of a Σ_1 wff, which is a wff with one existential quantifier followed by bounded quantifiers.
- Prove that for each RM program **P**, the state relation for **P**, which relates the original input, the time, and the register contents at that time, is definable in $\mathcal{N}$ by a Σ_1 wff.
- Using the state relation, show that every computable function is definable in $\mathcal{N}$ by a Σ_1 wff.
- Show that every total function which is definable in $\mathcal{N}$ by a Σ_1 wff is representable.
- Conclude that every computable total function is representable.

Along the way, we shall also show that a relation is weakly representable if and only if it is definable in $\mathcal{N}$ by a Σ_1 wff. This shows that every computable (partial) function is weakly representable, and completes the proof of Theorem 5.3.3.

Definition 5.4.1 We introduce two abbreviations in the language of arithmetic. Let **A** be a wff and let x, y be distinct variables.

The **bounded existential quantifier**:

$$(\exists x \leq y)\mathbf{A} \text{ means } \exists x\,[x \leq y \wedge \mathbf{A}],$$

The **bounded universal quantifier**:

$$(\forall x \leq y)\mathbf{A} \text{ means } \forall x\,[x \leq y \Rightarrow \mathbf{A}].$$

The bounded quantifiers are defined so as to match the usual meaning that one would expect them to have. $(\exists x \leq y)\mathbf{A}$ means that "There exists an x which is $\leq y$ such that **A** holds". $(\forall x \leq y)\mathbf{A}$ means that "For all x such that $x \leq y$, **A** holds".

Definition 5.4.2 A wff **A** is **bounded** if it can be built up in finitely many steps using the following rules of formation:

(1) Every atomic wff is bounded.

(2) If **A** is bounded, so is $\neg\mathbf{A}$.

(3) If **A** and **B** are bounded, so are $\mathbf{A} \circ \mathbf{B}$ where $\circ \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$.

(4) If **A** is bounded, so are $(\exists x \leq y)\mathbf{A}$ and $(\forall x \leq y)\mathbf{A}$.

Thus a bounded wff is a wff all of whose quantifiers may be written as bounded quantifiers.

Several familiar numerical functions and relations are definable in $\mathcal{N}$ by bounded wffs.

The equality relation, constant functions, successor function, addition function, and multiplication function are defined in $\mathcal{N}$ by wffs which have no quantifiers at all, and hence are bounded wffs.

The **order relation** $x \leq y$ is defined in $\mathcal{N}$ by the bounded wff

$$(\exists u \leq y)u \doteq x.$$

The **strict order relation** $x < y$ is defined in $\mathcal{N}$ by the bounded wff

$$\neg x \doteq y \wedge (\exists u \leq y)u \doteq x.$$

The **predecessor function** $\mathrm{Pred}(x) = y$, where

$$y = 0 \text{ if } x = 0, \text{ and } y = x - 1 \text{ otherwise},$$

is defined in $\mathcal{N}$ by the bounded wff

$$[x \doteq \mathbf{0} \wedge y \doteq \mathbf{0}] \vee x \doteq \mathbf{s}(y).$$

The **dotminus function** $x \dot{-} y = z$, where

$$z = 0 \text{ if } x \leq y, \text{ and } z = x - y \text{ otherwise},$$

is defined in $\mathcal{N}$ by the bounded wff

$$[x \leq y \wedge z \doteq \mathbf{0}] \vee x \doteq y + z.$$

The **remainder function** $\mathrm{Rem}(x, y) = r$, where $r = 0$ if $y = 0$, and r is the remainder when x is divided by y otherwise, is defined in $\mathcal{N}$ by the bounded wff

$$[y \doteq \mathbf{0} \wedge r \doteq \mathbf{0}] \vee (\exists q \leq x)[x \doteq q * y + r \wedge r < y].$$

(In the last two examples, $x \leq y$ and $r < y$ are abbreviations for the previously given bounded wffs.)

The predecessor function, dotminus function, and remainder function are total functions.

Definition 5.4.3 A wff $\mathbf{A}$ is said to be a Σ_1 **wff** if it has the form $\exists x\, \mathbf{B}$ where $\mathbf{B}$ is a bounded wff. A relation or function is Σ_1 **definable** if it is defined in $\mathcal{N}$ by a Σ_1 wff.

Thus a Σ_1 wff is formed by putting one existential quantifier in front of a bounded wff.

Any bounded wff $\mathbf{A}$ is equivalent to the Σ_1 wff $\exists v\, \mathbf{A}$ where v does not occur in $\mathbf{A}$. Therefore any function or relation which is definable in $\mathcal{N}$ by a bounded wff is Σ_1 definable. In particular, the constant, successor, addition, multiplication, predecessor, dotminus, and remainder functions and the equality, order, and strict order relations are Σ_1 definable.

The following lemma is helpful in showing that things are Σ_1 definable.

Lemma 5.4.4 *(i) Suppose* $\mathbf{C}$ *and* $\mathbf{D}$ *are* Σ_1 *wffs and* x, y *are distinct variables. Then the relations defined by*

$$\mathbf{C} \vee \mathbf{D}, \qquad \mathbf{C} \wedge \mathbf{D}, \qquad (\exists x \leq y)\mathbf{C}, \qquad (\forall x \leq y)\mathbf{C}, \qquad \exists x \mathbf{C}$$

are Σ_1 *definable in* $\mathcal{N}$.

(ii) If a relation R *is defined in* $\mathcal{N}$ *by a wff which is built from bounded wffs in finitely many steps by repeatedly using* $\vee, \wedge$, *bounded quantifiers, and existential quantifiers in any order, then* R *is* Σ_1 *definable.*

Proof: Part (ii) is proved by repeated application of Part (i). We prove Part (i). Suppose that $\mathbf{C}$ and $\mathbf{D}$ are Σ_1 formulas $\exists u\, \mathbf{A}, \exists v\, \mathbf{B}$ where $\mathbf{A}$ and $\mathbf{B}$ are bounded wffs.

Let u', v' be new variables which do not occur in $\mathbf{C}$ or $\mathbf{D}$ and are distinct from each other and from x, y. Let $\mathbf{A}'$ be the wff obtained from $\mathbf{A}$ by replacing all occurrences of u by u', and let $\mathbf{B}'$ be the wff obtained from $\mathbf{B}$ by replacing all occurrences of v by v'. By Exercise 9 in Chapter 2 (but for full rather than pure predicate logic), the wff $\mathbf{C}$ equivalent to $\exists u'\, \mathbf{A}'$ and $\mathbf{D}$ is equivalent to $\exists v'\, \mathbf{B}'$.

We may therefore simplify the problem by taking $\mathbf{A}$ and $\mathbf{B}$ so that the variables u, v, x, y are all distinct, u does not occur in $\mathbf{B}$, and v does not occur in $\mathbf{A}$.

The wff $\exists x\, \mathbf{C}$ defines the same relation in $\mathcal{N}$ as the Σ_1 wff

$$\exists w\, (\exists x \leq w)(\exists u \leq w)\mathbf{A}$$

where w is a new variable, because

$$\mathcal{N} \models \exists x\, \exists u\, \mathbf{A} \Leftrightarrow [\exists w\, (\exists x \leq w)(\exists u \leq w)\mathbf{A}].$$

The wff $\mathbf{C} \wedge \mathbf{D}$ defines the same relation in $\mathcal{N}$ as the wff

$$\exists u\, \exists v\, [\mathbf{A} \wedge \mathbf{B}],$$

because the wff

$$\exists u\, \exists v\, [\mathbf{A} \wedge \mathbf{B}] \Leftrightarrow [\exists u\, \mathbf{A} \wedge \exists v\, \mathbf{B}]$$

is tableau provable when u does not occur in $\mathbf{B}$ and v does not occur in $\mathbf{A}$. By the preceding existential quantifier case, it follows that the relation defined by $\mathbf{C} \wedge \mathbf{D}$ is Σ_1 definable in $\mathcal{N}$.

The $\mathbf{C} \vee \mathbf{D}$ case is similar.

The wff $(\exists x \leq y)\mathbf{A}$ defines the same relation in $\mathcal{N}$ as the Σ_1 wff

$$\exists u\, (\exists x \leq y)\mathbf{A},$$

because

$$\exists u\, (\exists x \leq y)\mathbf{A} \Leftrightarrow (\exists x \leq y)\exists u\, \mathbf{A}$$

is tableau provable.

Finally, the wff $(\forall x \leq y)\mathbf{A}$ defines the same relation in $\mathcal{N}$ as the Σ_1 wff

$$\exists w\, (\forall x \leq y)(\exists u \leq w)\mathbf{A},$$

because

$$\mathcal{N} \models \exists w\, (\forall x \leq y)(\exists u \leq w)\mathbf{A} \Leftrightarrow (\forall x \leq y)\exists u\, \mathbf{A}.$$

End of Proof.

Our next task is to define the state relation for an RM program.

We shall sometimes write a finite sequence of natural numbers as a "vector",

$$\vec{a} = (a_0, a_1, \ldots, a_k).$$

Definition 5.4.5 Let $\mathbf{P}$ be an RM program and suppose that k is the largest register number appearing in the instruction list of $\mathbf{P}$. The **state** of a computation by $\mathbf{P}$ at a given time is the finite sequence $\vec{s} = (s_0, s_1, \ldots, s_k)$ where the program counter contains the number s_0 and the registers $R_1, \ldots, R_k$ contain the numbers $s_1, \ldots, s_k$.

The **state relation** of $\mathbf{P}$ is the $(2k+3)$-ary relation $STATE_{\mathbf{P}}$ where

$$(\vec{a}, t, \vec{b}) \in STATE_{\mathbf{P}}$$

means that an RM machine which starts in the state $(a_0, \ldots, a_k)$ and executes the instructions of $\mathbf{P}$ will be in the state $\vec{b}$ after t instructions are executed.

The state relation of $\mathbf{P}$ will be obtained from another relation, the nextstate relation.

Definition 5.4.6 Let $\mathbf{P}$ be an RM program and suppose that k is the largest register number appearing in the instruction list of $\mathbf{P}$. The **nextstate relation** of $\mathbf{P}$ is the $(2k+2)$-ary relation $NXSTATE_{\mathbf{P}}$ where

$$(\vec{a}, \vec{b}) \in NXSTATE_{\mathbf{P}}$$

means that the a_0-th instruction of $\mathbf{P}$ will change state $\vec{a}$ to state $\vec{b}$.

In the above definition, it is to be understood that a halt instruction makes no change in the state.

Lemma 5.4.7 *Let $\mathbf{P}$ be an RM program. Then $NXSTATE_{\mathbf{P}}$ is Σ_1 definable.*

Proof: For convenience we assume that the program $\mathbf{P}$ is regular, so there are no halts before the last nonhalt instruction and no jump targets beyond the first halt instruction. Let k be the largest register number appearing in the instruction list of $\mathbf{P}$.

The action of each single RM instruction I involving register numbers between 1 and k may be expressed by a bounded wff

$$\mathbf{A}_I(\vec{x}, \vec{y})$$

with $2k + 2$ variables, where the instruction I changes a given state $\vec{x}$ to the new state $\vec{y}$. We write down these wffs for each instruction type.

$\mathbf{A}_H$:

$$y_0 \doteq x_0 \wedge \cdots \wedge y_k \doteq x_k.$$

$\mathbf{A}_{(Z,n)}$:

$$y_0 \doteq \mathbf{s}(x_0) \wedge y_1 \doteq x_1 \wedge \cdots \wedge y_n \doteq \mathbf{0} \wedge \cdots \wedge y_k \doteq x_k.$$

$\mathbf{A}_{(S,n)}$:

$$y_0 \doteq \mathbf{s}(x_0) \wedge y_1 \doteq x_1 \wedge \cdots \wedge y_n \doteq \mathbf{s}(x_n) \wedge \cdots \wedge y_k \doteq x_k.$$

$\mathbf{A}_{(T,n,p)}$:

$$y_0 \doteq \mathbf{s}(x_0) \wedge y_1 \doteq x_1 \wedge \cdots \wedge y_p \doteq x_n \wedge \cdots \wedge y_k \doteq x_k.$$

$\mathbf{A}_{(J,n,p,q)}$:

$$[[x_n \doteq x_p \wedge y_0 \doteq \mathbf{q}] \vee [\neg x_n \doteq x_p \wedge y_0 \doteq \mathbf{s}(x_0)]] \wedge y_1 \doteq x_1 \wedge \cdots \wedge y_k \doteq x_k.$$

Now let

$$I(0), I(1), \ldots, I(m)$$

be the instruction list for $\mathbf{P}$, where $I(m)$ is the last nonhalt instruction. Then the nextstate relation

$$(\vec{x}, \vec{y}) \in NXSTATE_{\mathbf{P}}$$

is defined in $\mathcal{N}$ by the following bounded wff:

$$[x_0 \doteq \mathbf{0} \wedge \mathbf{A}_{I(0)}] \vee [x_0 \doteq \mathbf{1} \wedge \mathbf{A}_{I(1)}] \vee \cdots \vee [x_0 \doteq \mathbf{m} \wedge \mathbf{A}_{I(m)}] \vee [\mathbf{m} < x_0 \wedge \mathbf{A}_H].$$

End of Proof.

We now wish to show that the state relation of each RM program is Σ_1 definable. In order to determine the state of an RM computation at some time t, one must go through the entire sequence of states at all times less than t. For this reason, we will need a Σ_1 definable way of "coding" sequences of natural numbers. We cannot use our Gödel numbering scheme for this purpose, because it depends on the exponential function $y = 10^x$, and we do not yet know that this function is Σ_1 definable. Another coding scheme is needed– one which is easier to define within arithmetic. Gödel found a way to do this using the following function, called the Gödel beta function. We must take a short detour in our development to give this coding scheme.

Definition 5.4.8 The **Gödel beta function** is defined by

$$\beta(x, y, z) = \mathrm{Rem}(x, y * (z + 1) + 1).$$

Lemma 5.4.9 *The Gödel beta function is Σ_1 definable.*

Proof: We have seen that the remainder function $\mathrm{Rem}(x, y) = r$ is defined in $\mathcal{N}$ by a Σ_1 wff $\mathbf{R}(x, y, z)$. The Gödel beta function $\beta(x, y, z) = v$ is then defined in $\mathcal{N}$ by the Σ_1 wff

$$\mathbf{R}(x, \mathbf{s}(y * (\mathbf{s}(z))), v).$$

End of Proof.

To use the Gödel beta function for coding finite sequences, we need a classical theorem in number theory called the Chinese Remainder Theorem. Since this theorem can be found in most number theory texts, we shall state it without proof.

Theorem 5.4.10 (Chinese Remainder Theorem) *Suppose that $m_1, \ldots, m_n$ are positive integers such that m_i and m_j are relatively prime whenever $1 \leq i < j \leq n$. If $0 \leq a_i < m_i$ for $i = 1, \ldots, n$, there exists x such that*

$$Rem(x, m_i) = a_i \text{ for } i = 1, \ldots, n.$$

The next lemma uses the Chinese Remainder Theorem to show that the Gödel beta function can code finite sequences.

Lemma 5.4.11 *For each finite sequence $(a_1, \ldots, a_n)$ of natural numbers, there exist b, c such that*

(1) $$\beta(c, d, i) = a_i \text{ for } i = 1, \ldots, n.$$

Thus the pair (c, d) "codes" the finite sequence $(a_1, \ldots, a_n)$ using the Gödel beta function.

Proof: Let M be such that $n \leq M$ and $a_i \leq M$ for $i = 1, \ldots, n$. Let $d = M!$. For $i = 1, \ldots, n$, let $m_i = d * (i+1) + 1$. Then

$$0 \leq a_i \leq M \leq d < m_i \text{ for } i = 1, \ldots, n.$$

Moreover, for each x and $i = 1, \ldots, n$, we have

$$\beta(x, d, i) = \text{Rem}(x, d * (i+1) + 1) = \text{Rem}(x, m_i). \tag{2}$$

We claim that whenever $1 \leq i < j \leq n$, the numbers m_i and m_j are relatively prime. Suppose not. Then some prime p divides both m_i and m_j. Therefore p divides their difference $m_j - m_i = (j-i) * c$. Hence either p divides $j - i$ or p divides d. But p divides $m_i = d * (i+1) + 1$, so p cannot divide d. Therefore p divides $j - i$. But $j - i < n \leq M$, so $p < M$ and hence p divides $d = M!$. This contradiction proves the claim.

By the Chinese Remainder Theorem 5.4.10, there exists c such that

$$\text{Rem}(c, m_i) = a_i \text{ for } i = 1, \ldots, n. \tag{3}$$

The desired conclusion (1) follows from (2) and (3). **End of Proof.**

Theorem 5.4.12 *For each RM program* $\mathbf{P}$*, the state relation* $STATE_{\mathbf{P}}$ *is* Σ_1 *definable.*

Proof: For simplicity we again assume that $\mathbf{P}$ is a regular program. Let k be the largest register number occurring in an instruction of $\mathbf{P}$. We must find a Σ_1 wff which defines the state relation

$$(\vec{a}, t, \vec{b}) \in STATE_{\mathbf{P}}$$

in $\mathcal{N}$.

The idea is to write a wff which says that there exists a finite sequence of states $(S_0, \ldots, S_t)$ such that $S_0 = \vec{a}$, $(S_u, S_{u+1}) \in NXSTATE_{\mathbf{P}}$ for all $u < t$, and $S_t = \vec{b}$. In order to do this with a Σ_1 wff, we replace the finite sequence of states by a pair of natural numbers which codes a finite sequence of states via the Gödel beta function.

Let $\mathbf{A}(\vec{a}, \vec{b})$ be a Σ_1 wff which represents the nextstate relation $NXSTATE_{\mathbf{P}}$ in $\mathcal{N}$. Let $\mathbf{B}(c, d, z, v)$ be a Σ_1 wff which represents the

Gödel beta function $\beta(c, d, z) = v$ in $\mathcal{N}$. Since each state has $k+1$ coordinates, it will be convenient to combine $k+1$ values of z together. Let $j = k + 1$ and let $\mathbf{C}(c, d, z, \vec{v})$ be the wff

$$\mathbf{B}(c, d, \mathbf{j} * z, v_0) \wedge \mathbf{B}(c, d, \mathbf{j} * z + \mathbf{1}, v_1) \wedge \cdots \wedge \mathbf{B}(c, d, \mathbf{j} * z + \mathbf{k}, v_k)$$

which defines the relation

$$\beta(c, d, jz) = v_0 \wedge \beta(c, d, jz + 1) = v_1 \wedge \cdots \wedge \beta(c, d, jz + k) = v_k.$$

Then the state relation

$$STATE_{\mathbf{P}}(\vec{a}, t, \vec{b})$$

is defined in $\mathcal{N}$ by the wff

$$\exists c \exists d [\mathbf{C}(c, d, \mathbf{0}, \vec{a}) \wedge \mathbf{C}(c, d, t, \vec{b}) \wedge$$

$$(\forall u \leq t) \exists \vec{x} \exists \vec{y} [u \doteq t \vee \mathbf{A}(\vec{x}, \vec{y}) \wedge \mathbf{C}(c, d, u, \vec{x}) \wedge \mathbf{C}(c, d, \mathbf{s}(u), \vec{y})]].$$

This wff is built from Σ_1 wffs using $\wedge, \vee$, bounded quantifiers, and existential quantifiers. By Lemma 5.4.4, the state relation is Σ_1 definable. **End of Proof.**

Theorem 5.4.13 *Every computable function is* Σ_1 *definable.*

Proof: Let F be a computable (partial) function of n variables. There is an RM program P which neatly computes F. By Theorem 5.4.12, the relation $STATE_{\mathbf{P}}$ is Σ_1 definable. It is defined in $\mathcal{N}$ by some Σ_1 wff

$$\mathbf{A}(\vec{y}, t, \vec{z}).$$

We may break the sequence of variable $\vec{y}$ into parts $\vec{y} = (\vec{x}, \vec{u})$ where $\vec{x}$ consists of the first n variables in $\vec{y}$. Let p be the number of the first halt instruction of $\mathbf{P}$. The program $\mathbf{P}$ halts when the instruction number is p. Then the graph $F(\vec{x}) = v$ of the partial function F computed by $\mathbf{P}$ is defined in $\mathcal{N}$ by the wff

$$\exists t \exists \vec{u} \exists \vec{z} [\mathbf{A}(\vec{x}, \vec{u}, t, \vec{z}) \wedge z_0 \doteq \mathbf{p} \wedge v \doteq z_1].$$

Thus by Lemma 5.4.4, F is Σ_1 definable. **End of Proof.**

We now make the final step, from Σ_1 definability to representability.

Lemma 5.4.14 *Each bounded wff* $\mathbf{A}$ *represents the relation defined by* $\mathbf{A}$ *in* $\mathcal{N}$.

Proof: Let $\mathcal{S}$ be the set of all wffs $\mathbf{A}$ such that the relation defined by $\mathbf{A}$ in $\mathcal{N}$ is represented by $\mathbf{A}$ in Weak Arithmetic. We must show that every bounded wff belongs to $\mathcal{S}$.

We have seen that the atomic wffs

$$x \doteq y, \mathbf{0} \doteq y, \mathbf{s}(x) \doteq y, x + y \doteq z, x * y \doteq z$$

belong to $\mathcal{S}$. Using this fact, it can be shown by induction on terms that any wff of the form $\tau \doteq y$ belongs to $\mathcal{S}$, where y is a variable which does not occur in τ.

It then follows that any atomic wff, i.e. equation between two terms, belongs to $\mathcal{S}$. For if σ, τ are terms with all variables replaced by numerals, there are a and b such that $\mathcal{N} \models \sigma \doteq \mathbf{a}$ and $\mathcal{N} \models \tau \doteq \mathbf{b}$. One can then check that if $a = b$ then $\mathbf{WA} \vdash \sigma \doteq \tau$ and otherwise $\mathbf{WA} \vdash \neg\sigma \doteq \tau$.

It is a routine matter to check that the set $\mathcal{S}$ is closed under each propositional connective.

We now show that the set $\mathcal{S}$ is closed under bounded quantifiers. We assume $\mathbf{A} \in \mathcal{S}$ and prove that $(\exists x \leq y)\mathbf{A} \in \mathcal{S}$. The trick is to use Lemma 5.2.6. By that lemma, for each b, it can be proved in $\mathbf{WA}$ that the wff

$$(\exists x \leq \mathbf{b})\mathbf{A}(x, \vec{z})$$

is equivalent to

$$\mathbf{A}(\mathbf{0}, \vec{z}) \vee \cdots \vee \mathbf{A}(\mathbf{b}, \vec{z}).$$

The latter wff is a finite disjunction of members of $\mathcal{S}$, and thus belongs to $\mathcal{S}$ by the preceding paragraph. It then follows that $(\exists x \leq y)\mathbf{A} \in \mathcal{S}$.

The bounded universal quantifier case is similar. This shows that every bounded wff belongs to $\mathcal{S}$. **End of Proof.**

We first take up weak representability, and then representability.

Theorem 5.4.15 *Each* Σ_1 *wff* $\mathbf{C}$ *weakly represents the relation defined by* $\mathbf{C}$ *in* $\mathcal{N}$. *A relation is weakly representable if and only if it is* Σ_1 *definable.*

Proof: Suppose first that a relation R is defined in $\mathcal{N}$ by a Σ_1 wff $\exists u\, \mathbf{A}(u, \vec{x})$, where $\mathbf{A}$ is a bounded wff. We show that $\exists u\, \mathbf{A}(u, \vec{x})$ weakly represents R. Suppose $\vec{a} \in R$. Then

$$\mathcal{N} \models \exists u\, \mathbf{A}(u, \vec{\mathbf{a}}).$$

Then for some $b \in \mathbf{N}$,

$$\mathcal{N} \models \mathbf{A}(\mathbf{b}, \vec{\mathbf{a}}).$$

By the preceding lemma,

$$\mathbf{WA} \vdash \mathbf{A}(\mathbf{b}, \vec{\mathbf{a}}),$$

and hence

$$\mathbf{WA} \vdash \exists u\, \mathbf{A}(u, \vec{\mathbf{a}}).$$

Now suppose

$$\mathbf{WA} \vdash \exists u\, \mathbf{A}(u, \vec{\mathbf{a}}).$$

Then

$$\mathcal{N} \models \exists u\, \mathbf{A}(u, \vec{\mathbf{a}}),$$

so $\vec{a} \in R$. Therefore R is weakly representable.

Now suppose that R is weakly represented by a wff $\mathbf{B}(\vec{x})$. Let F be the function such that $F(\vec{a}) = 0$ if $\vec{a} \in R$ and $F(\vec{a})$ is undefined otherwise. Then $F(\vec{x}) = y$ is weakly represented by the wff $\mathbf{B}(\vec{x}) \wedge y \doteq \mathbf{0}$. By the first half of Theorem 5.3.3, which was proved in the last section using Church's Thesis, F is computable. Therefore by Theorem 5.4.13, $F(\vec{x}) = y$ is defined in $\mathcal{N}$ by a Σ_1 wff $\mathbf{C}(\vec{x}, y)$. Then R is defined in $\mathcal{N}$ by the wff $\exists y\, \mathbf{C}(\vec{x}, y)$, and by Lemma 5.4.4, R is Σ_1 definable. **End of Proof.**

This gives us the second half of Theorem 5.3.3.

Corollary 5.4.16 *Every computable (partial) function is weakly representable.*

Proof: Suppose F is computable. By Theorem 5.4.13, F is Σ_1 definable, so by the preceding theorem, F is weakly representable. **End of Proof**

Theorem 5.4.17 *If a relation R has the property that both R and $\neg R$ are Σ_1 definable, then R is representable.*

Proof: Suppose $R(\vec{x})$ is defined in $\mathcal{N}$ by the wff $\exists u\, \mathbf{A}(u, \vec{x})$ and $\neg R(\vec{x})$ is defined in $\mathcal{N}$ by the wff $\exists v\, \mathbf{B}(v, \vec{x})$, where $\mathbf{A}$ and $\mathbf{B}$ are bounded wffs. Then

$$\mathcal{N} \models \exists u\, \mathbf{A}(u, \vec{x}) \Leftrightarrow \forall v \neg \mathbf{B}(v, \vec{x}).$$

It follows that $R(\vec{x})$ is also defined in $\mathcal{N}$ by the wff

$$\mathbf{C}(\vec{x}): \quad \exists u\, [\mathbf{A}(u, \vec{x}) \wedge (\forall v \leq u) \neg \mathbf{B}(v, \vec{x})],$$

and $\neg R(\vec{x})$ is defined in $\mathcal{N}$ by the wff

$$\mathbf{D}(\vec{x}): \quad \exists v\, [\mathbf{B}(v, \vec{x}) \wedge (\forall u \leq v) \neg \mathbf{A}(u, \vec{x})].$$

Both $\mathbf{C}$ and $\mathbf{D}$ are Σ_1 wffs. We show that $\mathbf{C}$ represents R.

If $\vec{a} \in R$, then by Theorem 5.4.15,

$$\mathbf{WA} \vdash \mathbf{C}(\vec{\mathbf{a}}).$$

Now suppose that $\vec{a} \notin R$. Then

$$\mathbf{WA} \vdash \mathbf{D}(\vec{\mathbf{a}}).$$

Exercise 10 shows that the three sentences

$$\mathbf{C}(\vec{\mathbf{a}}), \mathbf{D}(\vec{\mathbf{a}}), \forall u\, \forall v\, [u \leq v \vee v \leq u]$$

are tableau confutable. The third sentence above is Axiom 9 of **WA**. Therefore

$$\mathbf{WA} \vdash \mathbf{D}(\vec{\mathbf{a}}) \Rightarrow \neg \mathbf{C}(\vec{\mathbf{a}}),$$

and it follows that

$$\mathbf{WA} \vdash \neg \mathbf{C}(\vec{\mathbf{a}}).$$

This shows that $\mathbf{C}$ represents R. **End of Proof.**

Theorem 5.4.18 *Every total function F which is Σ_1 definable is representable.*

Proof: Let the graph $F(\vec{x}) = v$ of F be defined in $\mathcal{N}$ by a Σ_1 wff

$$\exists z\, \mathbf{A}(z, \vec{x}, v)$$

where $\mathbf{A}$ is a bounded wff. Since F is total, the complement $\neg F(\vec{x}) = v$ of the graph of F is defined in $\mathcal{N}$ by the wff

$$\exists z\, \exists w\, [\mathbf{A}(z, \vec{x}, w) \wedge \neg w \doteq v].$$

By Lemma 5.4.4, the complement of the graph of F is Σ_1 definable. By the preceding lemma, the function F is representable. **End of Proof.**

Putting everything together, we have now completed the proof of the Equivalence Theorem, showing that every computable total function is Σ_1 definable, and hence representable.

5.5 First Incompleteness Theorem

In this section we prove a theorem of Tarski which shows that the set of sentences which are true in the standard model $\mathcal{N}$ of arithmetic is not definable in $\mathcal{N}$. We shall then use Tarski's Theorem to give a proof of Gödel's First Incompleteness Theorem, which shows that **PA** is not complete.

Let us first review the notions of a consistent theory and of a complete theory, which were discussed informally in Chapter 3.

Definition 5.5.1 A theory $\mathbf{H}$ in the language of arithmetic is **consistent** if $\mathbf{H}$ does not have a tableau refutation. $\mathbf{H}$ is **complete** if $\mathbf{H}$ is consistent and for every sentence $\mathbf{A}$ in the language of arithmetic, either $\mathbf{H} \vdash \mathbf{A}$ or $\mathbf{H} \vdash \neg \mathbf{A}$.

The proof of Tarski's Theorem is based on the liar paradox,

This sentence is false.

The idea is to show that if the set of codes of true sentences were definable, then one could find a sentence which asserts its own falsehood, as in the liar paradox.

Here are the main steps of the proof that **PA** is not complete. Using the Equivalence Theorem we will show that the set of all codes of sentences which are provable from **PA** is definable in $\mathcal{N}$. Then by Tarski's Theorem the set of sentences provable from **PA** cannot be the same as the set of sentences true in $\mathcal{N}$. Since every sentence provable from **PA** is true in $\mathcal{N}$, it follows that there is a sentence **B** which is true in $\mathcal{N}$ but is neither provable nor disprovable from **PA**.

Gödel's original incompleteness proof, which will be given in the next section, is somewhat harder than the proof in this section but gives important additional information. It not only shows that **PA** is not complete, but actually produces an example of a sentence **B** which is neither provable nor disprovable from **PA**.

We introduce two more properties of theories.

Definition 5.5.2 A theory **H** in the language of arithmetic is **sound** if $\mathcal{N} \models \mathbf{H}$, that is, every sentence in **H** is true in the standard model of arithmetic.

Definition 5.5.3 By an **axiomatized theory** we mean a set of sentences **H** in the language of arithmetic such that the set of codes of elements of **H** is computable.

Every sound theory is consistent because it has the model $\mathcal{N}$. If a theory **H** is consistent but not complete, it will have an extension **H'** which is consistent but not sound (Exercise 4).

We saw in Chapter 3 that Weak Arithmetic is sound but not complete, and that Peano Arithmetic is sound. Any finite theory such as **WA** is obviously axiomatized, and we showed earlier in this chapter that **PA** is axiomatized. In this section we shall see that **PA** is not complete. In fact, we shall show even more, that no sound axiomatized theory is complete.

It will be convenient to introduce a name for the set of all sentences which are true in $\mathcal{N}$.

Definition 5.5.4 The set of all sentences which are true in a model $\mathcal{M}$ is denoted by $Th(\mathcal{M})$, and called the **theory of** $\mathcal{M}$. In particular, $Th(\mathcal{N})$ is called **complete arithmetic**.

For any model $\mathcal{M}$, the theory $Th(\mathcal{M})$ is automatically complete. A theory **H** is sound if and only if it is a subset of $Th(\mathcal{N})$.

Given a wff $\mathbf{A}(v)$ in the language of arithmetic with one free variable v and code a, the sentence $\mathbf{A}(\mathbf{a})$ will be called the **diagonal sentence** for $\mathbf{A}(v)$. Thus the diagonal sentence for a wff **A** is the sentence formed by replacing each free occurrence of v by the numeral representing the code of $\mathbf{A}(v)$. The diagonal sentence will be used in this section to form a sentence which asserts its own falsehood, and in the next section to form a sentence which asserts its own unprovability. To construct such sentences, we need the following definition:

Definition 5.5.5 The **diagonal relation** is the binary relation D on **N** consisting of those pairs (a, b) for which a is the code of a wff $A(v)$ in the language of arithmetic with one free variable v and b is the code of the diagonal sentence $A(\mathbf{a})$.

Lemma 5.5.6 *The diagonal relation D is computable.*

Proof: We outline an algorithm which, given an input (a, b), outputs a 1 if $(a, b) \in D$ and a 0 otherwise. First check whether a is the code of a wff $\mathbf{A}(v)$ with one free variable v. If not, output 0 and stop. Otherwise, compute the code of the sentence $\mathbf{A}(\mathbf{a})$. Output a 1 if this code is equal to b and output 0 otherwise, and stop. By Church's Thesis, the diagonal relation D is computable. **End of Proof.**

We need one more lemma before proving Tarski's Theorem.

Lemma 5.5.7 *In the language of arithmetic, for any wff $\mathbf{B}(x)$ with one free variable x, there is a sentence **C** such that*

$$\mathcal{N} \models \mathbf{C} \Leftrightarrow \mathbf{B}(\mathbf{c})$$

where c is the code of **C**.

Proof: Let $\mathbf{D}(v,x)$ be a wff which defines the diagonal relation D in $\mathcal{N}$. Let $\mathbf{E}(v)$ be the wff

$$\forall x\,[\mathbf{D}(v,x) \Rightarrow \mathbf{B}(x)].$$

Let e be the code of $\mathbf{E}$. Let $\mathbf{C}$ be the diagonal sentence $\mathbf{E(e)}$ of $\mathbf{E}(v)$. In expanded form, $\mathbf{C}$ is

$$\forall x\,[\mathbf{D(e},x) \Rightarrow \mathbf{B}(x)].$$

Let c be the code of $\mathbf{C}$. The sentence $\mathbf{C} \Leftrightarrow \mathbf{B(c)}$ in expanded form is

$$\forall x\,[\mathbf{D(e},x) \Rightarrow \mathbf{B}(x)] \Leftrightarrow \mathbf{B(c)}. \tag{1}$$

Since $\mathbf{C}$ is the diagonal sentence of $\mathbf{E}$, $(e,c) \in D$. Therefore c is the unique number such that

$$\mathcal{N} \models \mathbf{D(e,c)}.$$

It follows that the sentence (1) is true in $\mathcal{N}$, as required. **End of Proof.**

Theorem 5.5.8 (Tarski's Theorem) *Let TR denote the set of all codes of sentences true in $\mathcal{N}$. Then TR is not definable in $\mathcal{N}$.*

Proof: Assume TR is definable in $\mathcal{N}$ by a wff $\mathbf{TR}(v)$ with one free variable v. By the preceding lemma there is a sentence $\mathbf{P}$ with code p such that

$$\mathcal{N} \models \mathbf{P} \Leftrightarrow \neg\mathbf{TR(p)}.$$

Thus,

$$\mathcal{N} \models \mathbf{P} \text{ if and only if } \mathcal{N} \not\models \mathbf{TR(p)}$$

But since $\mathbf{TR}$ defines TR in $\mathcal{N}$, the right-hand side above is equivalent to $\mathcal{N} \not\models \mathbf{P}$. We are left with the contradiction that $\mathbf{P}$ is true in $\mathcal{N}$ if and only if $\mathbf{P}$ is not true in $\mathcal{N}$. We conclude that TR is not definable after all. **End of Proof.**

In Section 1 we defined the proof relation $\mathrm{PRF}_{\mathbf{H}}$ for a set $\mathbf{H}$ of sentences in the language of arithmetic to be the set of all pairs of natural numbers (x,y) such that x is the code of a wff $\mathbf{A}$ and y is the code of a tableau proof of $\mathbf{A}$ from $\mathbf{H}$. Using the proof relation, we can carry out the incompleteness proof sketched at the beginning of this section.

Theorem 5.5.9 (First Incompleteness Theorem) *Let $\mathbf{H}$ be a sound axiomatized theory. Then $\mathbf{H}$ is not complete.*

Proof: Theorem 5.1.9 showed that for each axiomatized theory $\mathbf{H}$, the proof relation $\mathrm{PRF}_{\mathbf{H}}$ is computable. By the Equivalence Theorem 5.3.1, $\mathrm{PRF}_{\mathbf{H}}$ is representable, and therefore definable in $\mathcal{N}$ by a formula $\mathbf{P}_{\mathbf{H}}(x,y)$. Therefore the set of codes of sentences which are provable from $\mathbf{H}$ is definable in $\mathcal{N}$ by the formula $\exists y\,\mathbf{P}_{\mathbf{H}}(x,y)$. Then by Tarski's Theorem and the soundness of $\mathbf{H}$, the set of sentences provable from $\mathbf{H}$ must be a proper subset of $Th(\mathcal{N})$. Thus there is a sentence $\mathbf{B}$ which is true in $\mathcal{N}$ but not provable from $\mathbf{H}$. Moreover, $\neg\mathbf{B}$ is not provable from $\mathbf{H}$ because it is false in $\mathcal{N}$ and $\mathbf{H}$ is sound. Therefore $\mathbf{H}$ is not complete. **End of Proof.**

Corollary 5.5.10 *The complete theory $Th(\mathcal{N})$ of arithmetic is not axiomatized.*

5.6 Gödel's Original Incompleteness Proof

In this section we shall give another proof of the First Incompleteness Theorem, using Gödel's original method.

The central idea is to modify the liar paradox by finding a sentence $\mathbf{A}_G$, called a "G sentence." which asserts its own unprovability from **PA**. Thus the Gödelian sentence $\mathbf{A}_G$ says

I am not provable from **PA**.

Now if $\mathbf{A}_G$ is provable from **PA**, then $\mathbf{A}_G$ must be true in $\mathcal{N}$ because **PA** is sound, and therefore $\mathbf{A}_G$ is not provable from **PA**. Thus $\mathbf{A}_G$ cannot be provable from **PA**. It follows that $\mathbf{A}_G$ is true in $\mathcal{N}$, and since **PA** is sound, the negation also is not provable from **PA**. Hence **PA** is incomplete.

With these remarks we have in outline form another proof that **PA** is an incomplete theory, and that there are sentences which are true in $\mathcal{N}$ but not provable from **PA**. The main technical difficulty is to show that the Gödelian sentence exists.

As a starting point we introduce the concept of a **proof formula** for a theory $\mathbf{H}$ in the language of arithmetic.

Definition 5.6.1 Let **H** be an axiomatized theory in the language of arithmetic. A **proof formula for H** is a wff $\mathbf{PRF_H}$ which represents the proof relation $\mathrm{PRF_H}$.

Corollary 5.6.2 *Let* **H** *be an axiomatized theory. Then the proof relation* $PRF_{\mathbf{H}}$ *for* **H** *is representable, i.e.,* **H** *has a proof formula.*

Proof: Theorem 5.1.9 showed that for each axiomatized theory **H**, the proof relation $\mathrm{PRF_H}$ is computable.

By the Equivalence Theorem 5.3.1, $\mathrm{PRF_H}$ is representable. **End of Proof.**

A proof formula for **H** allows us to express a statement like

(1) *The tableau* **T** *is a proof of the sentence* **A** *from* **H**.

formally in arithmetic, by translating it into the wff:

(2) $\mathbf{PRF_H(a, t)}$

where a is the code of the wff **A** and t is the code of the tableau **T**. There are several steps involved in this translation. First, form the proof formula $\mathbf{PRF_H}(x, y)$ for **H**. Then compute the codes a for the wff **A** and t for the tableau **T**. Finally, form the numerals (which are terms) **a** for a and **t** for t, and substitute these terms for the variables x, y in $\mathbf{PRF_H}(x, y)$.

The next result shows that by using an existential quantifier, we can express the statement

(3) The sentence **A** is tableau provable from **H**

formally in arithmetic by the wff

(4) $\exists y\, \mathbf{PRF_H(a}, y)$

where a is the code for the wff **A**.

In the incompleteness proof in the preceding section, we used the fact that the set of codes of sentences which are provable from an axiomatized theory **H** is definable in $\mathcal{N}$. We now prove that this set is also weakly representable.

Theorem 5.6.3 *Let* $\mathbf{PRF_H}$ *be a proof formula for an axiomatized theory* **H**, *and let PV be the set of all codes of sentences which are provable from* **H**, *that is,*

$$PV = \{\#(\mathbf{A}) : \mathbf{A} \in SENT(\mathcal{L}) \text{ and } \mathbf{H} \vdash \mathbf{A}\}$$

where $\mathcal{L}$ *is the vocabulary of arithmetic. Then the wff* $\exists y\, \mathbf{PRF_H}(x, y)$ *weakly represents the relation PV and also defines PV in* $\mathcal{N}$. *That is, for all* $a \in \mathbf{N}$,

(i) $a \in PV$ *if and only if* $\mathbf{WA} \vdash \exists y\, \mathbf{PRF_H(a}, y)$,

(ii) $a \in PV$ *if and only if* $\mathcal{N} \models \exists y\, \mathbf{PRF_H(a}, y)$.

Proof: Since $\mathbf{PRF_H}$ represents $\mathrm{PRF_H}$, $\mathbf{PRF_H}$ defines $\mathrm{PRF_H}$ in $\mathcal{N}$ by Theorem 5.2.3. Both (i) and (ii) are proved by the following list of statements.

If $a \in PV$ then for some $b \in \mathbf{N}$, $(a, b) \in \mathrm{PRF_H}$.

If $(a, b) \in \mathrm{PRF_H}$ then $\mathbf{WA} \vdash \mathbf{PRF_H(a, b)}$.

If $\mathbf{WA} \vdash \mathbf{PRF_H(a, b)}$ then $\mathbf{WA} \vdash \exists y\, \mathbf{PRF_H(a}, y)$.

If $\mathbf{WA} \vdash \exists y\, \mathbf{PRF_H(a}, y)$, then $\mathcal{N} \models \exists y\, \mathbf{PRF_H(a}, y)$.

If $\mathcal{N} \models \exists y\, \mathbf{PRF_H(a}, y)$, then for some $b \in \mathbf{N}$, $\mathcal{N} \models \mathbf{PRF_H(a, b)}$.

If $\mathcal{N} \models \mathbf{PRF_H(a, b)}$, then $(a, b) \in \mathrm{PRF_H}$.

If $(a, b) \in \mathrm{PRF_H}$ then $a \in PV$. **End of Proof.**

Once we see that statements about proofs can be expressed formally, many questions naturally arise about the relationship between this formal version of proof (like (2)) and our usual notion of proof (like (1)). For instance, we will be able to investigate questions like:

(5) If **A** is provable, is it *provable* that **A** is provable?

(6) If it's provable that **A** is provable, must **A** be provable?

(7) Is it provable that if $\mathbf{A}$ and $\mathbf{B}$ are both provable then $\mathbf{A} \wedge \mathbf{B}$ is provable?

(8) If $\mathbf{A} \vee \mathbf{B}$ is provable, must one of $\mathbf{A}$ and $\mathbf{B}$ be provable? Is the answer to this question provable?

We now turn to the First Incompleteness Theorem.

Definition 5.6.4 If $\mathbf{H}$ is an axiomatized theory, a sentence $\mathbf{P}$ of arithmetic is a **Gödelian sentence for H** if

$$\mathbf{H} \vdash \mathbf{P} \Leftrightarrow \neg \exists y\, \mathbf{PRF_H}(\mathbf{p}, y)$$

where p is the code for $\mathbf{P}$.

Thus $\mathbf{P}$ is Gödelian for $\mathbf{H}$ if $\mathbf{H}$ proves that [$\mathbf{P}$ is true if and only if $\mathbf{P}$ is not provable from $\mathbf{H}$]. A Gödelian sentence for $\mathbf{H}$ asserts its own unprovability from $\mathbf{H}$.

The following proposition shows that a Gödelian sentence quickly leads to incompleteness.

Proposition 5.6.5 *Let $\mathbf{H}$ be a sound axiomatized theory and let $\mathbf{P}$ be a Gödelian sentence for $\mathbf{H}$. Then $\mathbf{P}$ is true in $\mathcal{N}$ but not provable from $\mathbf{H}$, and $\mathbf{H}$ is consistent but not complete.*

Proof: Let p be the code for $\mathbf{P}$. Since $\mathbf{P}$ is Gödelian for $\mathbf{H}$ and $\mathbf{H}$ is sound, we have

(1) $$\mathcal{N} \models \mathbf{P} \Leftrightarrow \neg \exists y\, \mathbf{PRF_H}(\mathbf{p}, y)$$

We claim that

(2) $$\mathcal{N} \models \neg \exists y\, \mathbf{PRF_H}(\mathbf{p}, y).$$

Suppose (2) fails. Then there exists n such that

$$\mathcal{N} \models \mathbf{PRF_H}(\mathbf{p}, \mathbf{n}),$$

and by Theorem 5.6.3, $\mathbf{H} \vdash \mathbf{P}$. By soundness, $\mathcal{N} \models \mathbf{P}$. Then by (1), (2) holds. Thus (2) holds in all cases.

By (1) and (2), $\mathcal{N} \models \mathbf{P}$. By (2) and Theorem 5.6.3, $\mathbf{H} \not\vdash \mathbf{P}$. Since $\mathbf{H}$ is sound and $\mathcal{N} \models \mathbf{P}$, we also have $\mathbf{H} \not\vdash \neg\mathbf{P}$. Therefore $\mathbf{H}$ is consistent but not complete. **End of Proof.**

We now show that theories such as **PA** have Gödelian sentences. In the preceding section we defined the diagonal relation D, consisting of those pairs (a, b) for which a is the code of a wff $A(v)$ in the language of arithmetic with one free variable v and b is the code of the diagonal sentence $A(\mathbf{a})$. We showed that D is computable. Since D is computable, it is representable. The next lemma shows that there is a wff $\mathbf{D}$ which does an especially good job of representing D. It will be needed in forming a Gödelian sentence.

Lemma 5.6.6 *There is a wff $\mathbf{D}(v, x)$ such that $\mathbf{D}$ represents the diagonal relation D and for each $(a, b) \in D$,*

$$\mathbf{WA} \vdash \forall x\, [\mathbf{D}(\mathbf{a}, x) \Leftrightarrow x \doteq \mathbf{b}].$$

Proof: By the preceding lemma, D is computable. By the Equivalence Theorem, D is represented in **WA** by some formula $\mathbf{B}(v, x)$. Let $\mathbf{D}(v, x)$ be the wff

(3) $$\mathbf{B}(v, x) \wedge \forall u\, [u < x \Rightarrow \neg \mathbf{B}(v, u)].$$

Intuitively, $\mathbf{D}(\mathbf{a}, \mathbf{b})$ says that b is the first number such that $(a, b) \in D$.

We first check the second half of representability. If $(a, b) \notin D$, then $\mathbf{WA} \vdash \neg\mathbf{B}(\mathbf{a}, \mathbf{b})$, and therefore $\mathbf{WA} \vdash \neg\mathbf{D}(\mathbf{a}, \mathbf{b})$ because $\mathbf{D}$ is the conjunction of $\mathbf{B}$ and another wff.

Suppose that $(a, b) \in D$. To prove (3), work within **WA** and consider each of the three cases $x < \mathbf{b}, x \doteq \mathbf{b}, \mathbf{b} < x$. In the first case, show that $\neg\mathbf{D}(\mathbf{b}, x)$ using the fact that

$$\mathbf{WA} \vdash x < \mathbf{b} \Rightarrow x \doteq \mathbf{0} \vee \ldots \vee x \doteq \mathbf{b} - \mathbf{1}$$

and for each $c < b$,

$$\mathbf{WA} \vdash \neg\mathbf{B}(\mathbf{a}, \mathbf{c}).$$

The second and third cases use the fact that, since $\mathbf{B}$ represents D, $\mathbf{WA} \vdash \mathbf{B}(\mathbf{a}, \mathbf{b})$ but $\mathbf{WA} \vdash \neg\mathbf{B}(\mathbf{a}, \mathbf{c})$ for all $c < b$. This gives us $\mathbf{D}(\mathbf{a}, x)$ in the case $x \doteq \mathbf{b}$ and $\neg\mathbf{D}(\mathbf{a}, x)$ in the case $\mathbf{b} < x$. We have thus proved (3).

The first half of representability follows from (3) and the fact that

$$\vdash \forall x\,[\mathbf{D}(\mathbf{a}, x) \Leftrightarrow x \doteq \mathbf{b}] \Rightarrow \mathbf{D}(\mathbf{a}, \mathbf{b}).$$

End of Proof.

We now prove a stronger form of Lemma 5.5.7.

Lemma 5.6.7 (Diagonalization Lemma) *In the language of arithmetic, for any wff* $\mathbf{B}(x)$ *with one free variable* x*, there is a sentence* $\mathbf{C}$ *such that*

$$\mathbf{WA} \vdash \mathbf{C} \Leftrightarrow \mathbf{B}(\mathbf{c})$$

where c *is the code of* $\mathbf{C}$.

Proof: (To make the idea easier to follow, we shall keep in mind the important case where $\mathbf{B}(x)$ is a wff which says "x is not provable from $\mathbf{H}$".) Let $\mathbf{D}(v, x)$ be the wff of the preceding lemma. Let $\mathbf{E}(v)$ be the wff

$$\forall x\,[\mathbf{D}(v, x) \Rightarrow \mathbf{B}(x)].$$

(Intuitively, $\mathbf{E}(v)$ says that the diagonal sentence of the wff with code v is not provable from $\mathbf{H}$). Let e be the code of $\mathbf{E}$. Let $\mathbf{C}$ be the diagonal sentence $\mathbf{E}(\mathbf{e})$ of $\mathbf{E}(v)$. In expanded form, $\mathbf{C}$ is

$$\forall x\,[\mathbf{D}(\mathbf{e}, x) \Rightarrow \mathbf{B}(x)].$$

(Intuitively, $\mathbf{C}$ says that the diagonal sentence of the wff with code e is not provable from $\mathbf{H}$, that is, $\mathbf{C}$ says that $\mathbf{C}$ is not provable from $\mathbf{H}$!). Let c be the code of $\mathbf{C}$. The sentence $\mathbf{C} \Leftrightarrow \mathbf{B}(\mathbf{c})$ in expanded form is

$$\forall x\,[\mathbf{D}(\mathbf{e}, x) \Rightarrow \mathbf{B}(x)] \Leftrightarrow \mathbf{B}(\mathbf{c}). \tag{4}$$

We must show that (4) is provable from $\mathbf{WA}$. Since $\mathbf{C}$ is the diagonal sentence of $\mathbf{E}$, $(e, c) \in D$. By the preceding lemma, the sentence

$$\forall x\,[\mathbf{D}(\mathbf{e}, x) \Leftrightarrow x \doteq \mathbf{c}] \tag{5}$$

is provable from $\mathbf{WA}$. One can easily check that (4) is tableau provable from (5) (e.g. by using the TABLEAU program), so (4) is also provable from $\mathbf{WA}$ as required. **End of Proof.**

Corollary 5.6.8 *Let* $\mathbf{H}$ *be an axiomatized theory and let* $\mathbf{PRF_H}$ *be a proof formula for* $\mathbf{H}$*. Then there is a sentence* $\mathbf{P}$ *such that*

$$\mathbf{WA} \vdash \mathbf{P} \Leftrightarrow \neg\exists y\,\mathbf{PRF_H}(\mathbf{p}, y) \tag{†},$$

where p *is the code of* $\mathbf{P}$.

Proof: Let $\mathbf{B}(x)$ be the wff $\neg\exists y\,\mathbf{PRF_H}(x, y)$ (which intuitively says that "x is not provable from $\mathbf{H}$"). By the Diagonalization Lemma, there is a sentence $\mathbf{P}$ with code p such that

$$\mathbf{WA} \vdash \mathbf{P} \Leftrightarrow \mathbf{B}(\mathbf{p}).$$

This is (†). **End of Proof.**

Proposition 5.6.9 *Let* $\mathbf{H}$ *be a consistent axiomatized theory which includes* $\mathbf{WA}$*, and let* $\mathbf{P}$ *be a sentence with property (†) from the preceding corollary. Then* $\mathbf{P}$ *is a Gödelian sentence for* $\mathbf{H}$*. Moreover,* $\mathbf{P}$ *is true in* $\mathcal{N}$ *but not provable from* $\mathbf{H}$.

Proof: $\mathbf{P}$ is Gödelian for $\mathbf{H}$ because (†) holds and $\mathbf{H}$ includes $\mathbf{WA}$. We show next that $\mathbf{P}$ is not provable from $\mathbf{H}$. Suppose on the contrary that $\mathbf{H} \vdash \mathbf{P}$. Since $\mathbf{P}$ is Gödelian for $\mathbf{H}$,

$$\mathbf{H} \vdash \neg\exists y\,\mathbf{PRF_H}(\mathbf{p}, y).$$

But $\mathbf{PRF_H}$ is a proof formula for $\mathbf{H}$ and $\mathbf{H} \vdash \mathbf{P}$, so

$$\mathbf{WA} \vdash \exists y\,\mathbf{PRF_H}(\mathbf{p}, y).$$

Since $\mathbf{H}$ includes $\mathbf{WA}$,

$$\mathbf{H} \vdash \exists y\,\mathbf{PRF_H}(\mathbf{p}, y).$$

This contradicts the fact that $\mathbf{H}$ is consistent. We conclude that $\mathbf{P}$ is not provable from $\mathbf{H}$.

It follows that the sentence $\neg\exists y\,\mathbf{PRF_H}(\mathbf{p}, y)$ is true in $\mathcal{N}$. Since $\mathbf{WA}$ is sound, we conclude from (†) that $\mathbf{P}$ is true in $\mathcal{N}$. **End of Proof.**

We can now easily prove the First Incompleteness Theorem.

Theorem 5.6.10 (First Incompleteness Theorem) *No sound axiomatized theory is complete. In particular, Peano Arithmetic is not complete.*

Proof: Suppose **H** is a sound axiomatized theory, and assume that **H** is complete. Since **H** is axiomatized, by Corollary 5.6.8 there is a sentence **P** such that (†) holds.

For each axiom **Q** of **WA**, $\mathcal{N} \models \neg\mathbf{Q}$, hence by soundness, $\mathbf{H} \nvdash \neg\mathbf{Q}$, and by completeness, $\mathbf{H} \vdash \mathbf{Q}$. Thus every axiom of **WA** is provable from **H**, and hence every sentence provable from $\mathbf{H} \cup \mathbf{WA}$ is provable from **H** alone. Since **H** is sound, $\mathbf{H} \cup \mathbf{WA}$ is sound. Then by Proposition 5.6.9, there is a Gödelian sentence **P** for $\mathbf{H} \cup \mathbf{WA}$. By Proposition 5.6.5, **H** is not complete. **End of Proof.**

5.7 Gödel–Rosser Theorem

The First Incompleteness Theorem in the preceding two sections requires the theory in question to be sound. As reasonable as this property may be, it is quite complex from the point of view of computability. To check the soundness of a theory, we must decide whether each of its sentences is true (in $\mathcal{N}$). As we show in Theorem 5.5.8, no procedure which decides the truth of every sentence of arithmetic is even definable in $\mathcal{N}$.

In this section we shall prove an improvement of the First Incompleteness Theorem which does not depend on the notion of soundness, the Gödel–Rosser Theorem: No consistent axiomatized theory which includes **WA** is complete.

We first need some results about the undecidability of some of the relations and theories we have been studying.

Definition 5.7.1 A set of sentences **H** is called a **decidable theory** if the set

$$\{x : \exists y\, \mathrm{PRF}_{\mathbf{H}}(x, y)\}$$

of codes of sentences which are provable from **H** is computable. Theories which are not decidable are called **undecidable**.

The next theorem shows that **PA** and **WA** are undecidable. Thus the set of codes of proofs from **PA** is computable, but the set of codes of provable wffs from **PA** is not computable.

Theorem 5.7.2 *Any consistent theory which includes* **WA** *is undecidable.*

Proof: Assume **H** is consistent, decidable, and includes **WA**. We shall obtain a contradiction.

Since **H** is decidable, the set

$$PV = \{x : \exists y\, \mathrm{PRF}_{\mathbf{H}}(x, y)\}$$

of codes of sentences which are provable from **H** is computable. We may assume that every sentence which is provable from **H** is already an element of the set **H**. Then PV is the set of codes of elements of **H**, so **H** is an axiomatized theory. Let $\mathbf{PRF_H}$ be a proof formula for **H**. By the Equivalence Theorem, PV is represented by some wff **B**. $\mathbf{B} \wedge \mathbf{PRF_H}$ is also a proof formula for **H**, because $\mathrm{PRF}_{\mathbf{H}} = PV \cap \mathrm{PRF}_{\mathbf{H}}$ and $\mathbf{B} \wedge \mathbf{PRF_H}$ represents $PV \cap \mathrm{PRF}_{\mathbf{H}}$. By Corollary 5.6.8 there is a wff **P** such that (†) holds with $\mathbf{B} \wedge \mathbf{PRF_H}$ in place of $\mathbf{PRF_H}$. By Proposition 5.6.9,

$$\mathbf{H} \nvdash \mathbf{P}.$$

Let p be the code of **P**. Then $p \notin PV$. Since **B** represents PV, $\mathbf{WA} \vdash \neg\mathbf{B}(\mathbf{p})$. Therefore

$$\mathbf{WA} \vdash \neg\exists y\, [\mathbf{B} \wedge \mathbf{PRF_H}](\mathbf{p}, y).$$

It now follows from (†) that $\mathbf{WA} \vdash \mathbf{P}$, and since $\mathbf{WA} \subset \mathbf{H}$, $\mathbf{H} \vdash \mathbf{P}$. This is a contradiction and completes the proof. **End of Proof.**

For example, the theories **WA** and **PA** are undecidable because each is a consistent theory which includes **WA**.

$Th(\mathcal{N})$ is thus an example of a consistent theory which includes **WA**, and by the preceding theorem, $Th(\mathcal{N})$ is undecidable. Since every sentence which is provable from $Th(\mathcal{N})$) is true in $\mathcal{N}$ and vice versa, it follows that the set of codes of sentences which are true in $\mathcal{N}$ is not computable.

Theorem 5.7.2 says something about theories containing sentences which are false in $\mathcal{N}$. (Such theories are called **unsound**.) For example, we might try to make **PA** a complete theory by adding to **PA** the axiom $\neg\mathbf{P}$ where **P** is Gödelian for **PA**. Let $\mathbf{PA}^+ = \mathbf{PA} \cup \{\neg\mathbf{P}\}$. Since $\mathcal{N} \models \mathbf{P}$, $\mathbf{PA}^+$ is unsound. But $\mathbf{PA}^+$ is consistent; if not, then every model of **PA** would satisfy **P** and we would have that $\mathbf{PA} \vdash \mathbf{P}$, contradicting Proposition 5.6.5. We can therefore conclude by our last theorem that $\mathbf{PA}^+$ is an undecidable theory.

The next lemma shows that any undecidable axiomatized theory is also incomplete. It leads to the Gödel-Rosser Theorem, which is an improvement of the First Incompleteness Theorem.

Lemma 5.7.3 *Every complete axiomatized theory* **H** *is decidable.*

Proof: We describe an algorithm which determines whether an input a is the code of a sentence which is provable from **H**. First, check whether a is the code of a sentence of arithmetic. If not, output 0 and stop. If a is the code of a sentence **A**, compute the code b of the sentence $\neg\mathbf{A}$. Now for $c = 0, 1, 2, \ldots$, check to see whether $(a, c) \in \mathrm{PRF}_{\mathbf{H}}$, and then check whether $(b, c) \in \mathrm{PRF}_{\mathbf{H}}$. Continue this process until either $(a, c) \in \mathrm{PRF}_{\mathbf{H}}$ or $(b, c) \in \mathrm{PRF}_{\mathbf{H}}$. The process will stop after finitely many steps because **H** is complete, so either $\mathbf{H} \vdash \mathbf{A}$ or $\mathbf{H} \vdash \neg\mathbf{A}$. If $(a, c) \in \mathrm{PRF}_{\mathbf{H}}$ we output 1 and stop, and if $(b, c) \in \mathrm{PRF}_{\mathbf{H}}$ we output 0 and stop. This shows that **H** is decidable. **End of Proof.**

Theorem 5.7.4 (Gödel-Rosser Theorem) *No consistent axiomatized theory which includes* **WA** *is complete.*

Proof: Suppose **H** is an axiomatized theory which includes **WA** and assume that **H** is complete. By Lemma 5.7.3, **H** is decidable, contradicting Theorem 5.7.2. **End of Proof.**

The Gödel-Rosser Theorem, like the First Incompleteness Theorem, is stated entirely in terms of provability, and does not require the notion of a wff being true in $\mathcal{N}$.

The Gödel-Rosser Theorem implies that there is no computable way to add axioms one-by-one to **PA** in order to make it complete – even if we are allowed to add infinitely many axioms. (In other words, we

cannot find a list of such axioms whose codes are computed by an RM program.) To see this, suppose we attempt to add axioms $\mathbf{A}_0, \mathbf{A}_1, \ldots$ using some algorithm. Note that if instead we add the axioms $\mathbf{A}_0, \mathbf{A}_0 \wedge \mathbf{A}_1, \ldots$ we obtain a theory which has the same consequences as the first, only now the codes of the new axioms are arranged in increasing order. By Exercise 2, the set of these codes is computable, and hence so is the set of codes of

$$\mathbf{PA} \cup \{\mathbf{A}_0, \mathbf{A}_0 \wedge \mathbf{A}_1, \ldots\}.$$

Thus, by the Gödel-Rosser Theorem, the new theory is either inconsistent or incomplete.

In contrast to the Gödel-Rosser Theorem, there are several known examples of theories **H** in the language of arithmetic which are consistent, complete, and decidable. By the theorem, no such theory can include **WA**. A trivial example is the theory $Th(\mathcal{M})$ of all sentences true in a finite model $\mathcal{M}$. Two very important examples due to Tarski are the theories $Th(\mathbf{R})$ and $Th(\mathbf{C})$ where **R** is the field of real numbers and **C** is the field of complex numbers. Another important complete decidable theory, due to Presburger, is the theory $Th(\mathcal{N}_+)$ where $\mathcal{N}_+$ is the standard model of arithmetic in the vocabulary $\{\mathbf{0}, \mathbf{s}, +\}$ *without* the multiplication symbol $*$.

Using the results of this section, we can give another proof of Church's Theorem, which was proved in Chapter 4.

Theorem 5.7.5 (Church's Theorem) *The empty theory in the language of arithmetic is undecidable. That is, the set V of all codes of valid sentences in the language of arithmetic is not computable.*

Proof: Suppose the set of codes of valid sentences is decidable and is computed by an RM program which we shall call VAL. We shall show that **WA** would then be decidable, thus obtaining a contradiction. The proof depends on the fact that **WA** is a finite set of sentences. Since **WA** is finite, we may form the sentence **C** which is the conjunction of all sentences in the set **WA**. Then for each sentence **A**, we have

$$\mathbf{WA} \models \mathbf{A} \text{ if and only if } \models \mathbf{C} \Rightarrow \mathbf{A}.$$

We describe an algorithm which would, under our hypothesis, determine whether an input a is the code of a sentence which is a valid

consequence of **WA**. First check whether a is the code of a sentence. If not, output 0 and stop. If a is the code of a sentence $\mathbf{A}$, compute the code c of the sentence $\mathbf{C} \Rightarrow \mathbf{A}$. Then use the hypothetical program VAL to decide whether or not $\mathbf{C} \Rightarrow \mathbf{A}$ is valid. If so, then $\mathbf{WA} \vdash \mathbf{A}$ and we output 1, and otherwise $\mathbf{WA} \nvdash \mathbf{A}$ and we output 0. This shows that **WA** would be decidable and contradicts Theorem 5.7.2. **End of Proof.**

While Church's Theorem shows that the set V of codes of valid sentences is undecidable, Theorem 5.6.3, shows that V is definable in $\mathcal{N}$ and weakly representable. By contrast, the method of truth tables shows that the set of codes of valid sentences in propositional logic is computable.

We have seen that the set of codes of sentences true in the standard model of arithmetic is not computable. Tarski improved this result by showing that the set of codes of true sentences is not even definable in $\mathcal{N}$:

5.8 Provability and Modal Logic

One of the innovations of the 1970's (forty years after Gödel's discovery of the Incompleteness Theorems) was an application of a simple kind of logic – called **modal logic** – to investigate questions of provability in arithmetic. This approach allows one to study the Incompleteness Theorems without the rather involved machinery of Gödel numbering. We shall describe this approach here and use it to prove Gödel's Second Incompleteness Theorem.

The rest of this chapter is organized as follows: In this section we describe modal logic, an interpretation of modal wffs as sentences of arithmetic, and a broad class of theories of arithmetic which are needed to define this interpretation precisely. In 5.9 we describe modal tableau proofs and discuss various axioms for modal logic which express certain essential properties of "provability." In 5.10 we revisit the First Incompleteness Theorem. In 5.11 we prove Gödel's Second Incompleteness Theorem and discuss several related results.

We begin our study of modal logic with a language which has as its primitive symbols those of propositional logic together with a new

symbol $\Box$ which is a formal counterpart of the predicate "is provable from **H**," where **H** is some theory in the language of arithmetic, like **WA** or **PA**. The symbol $\Box$ will allow us to formulate modal axioms which express essential properties of provability without involving us in the details of codes.

We want each propositional symbol in modal logic to stand for a sentence in the language of arithmetic, but we will not be concerned with the inner structure of the sentences. To accomplish this, we shall simply take the sentences in the language of arithmetic themselves to be the propositional symbols of our modal logic. We shall use the capital boldface letters $\mathbf{P}, \mathbf{Q}, \ldots$ to stand for arbitrary propositional symbols of modal logic (we shall stop using them for RM programs). Lower case boldface letters will be used for numerals. Thus in our modal logic, $\mathbf{P}, \mathbf{Q}, \ldots$ will stand for sentences in the language of arithmetic, but we do not have to specify which sentences.

Formally, modal logic is obtained by adding a new symbol $\Box$, called a modal operator, to propositional logic. The vocabulary of modal logic consists of a set $\mathcal{P}$ of proposition symbols, as in propositional logic. The primitive symbols consist of the proposition symbols just described, the connectives and brackets of ordinary propositional logic, and the symbol $\Box$. Any finite sequence of these primitive symbols is a **string**. A **modal wff** is a finite string obtained by finitely many applications of the following rules of formation:

(Modal:$\mathcal{P}$)	Any proposition symbol is a modal wff
(Modal:$\neg$)	If $\mathbf{A}$ is a modal wff, then $\neg\mathbf{A}$ is a modal wff
(Modal:$\Box$)	If $\mathbf{A}$ is a modal wff, then $\Box\mathbf{A}$ is a modal wff.
(Modal:$\wedge, \vee, \Rightarrow, \Leftrightarrow$)	If $\mathbf{A}$ and $\mathbf{B}$ are modal wffs, then $[\mathbf{A}*\mathbf{B}]$ is a modal wff whenever $* \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$.

For the set $\mathcal{P}$ of propositional symbols we take the set $SENT(\mathcal{L})$ of sentences in the vocabulary $\mathcal{L}$ of arithmetic. We shall let $\mathbf{F}$ denote the false sentence $\neg\mathbf{0} \doteq \mathbf{0}$ of arithmetic. Thus $\mathbf{F}$ is a particular proposition

symbol of modal logic as well as a sentence of arithmetic. The set of all modal wffs will be denoted by $WFF(\mathcal{P})$.

Modal wffs and the logic associated with them can be interpreted in a variety of ways; originally modal logic arose (starting as far back as Aristotle) as an attempt to formalize the idea of **necessary truth**: Given a proposition P about the world, if P *happens* to be true, is it *necessarily* true? Around 1910, C.I. Lewis introduced the symbol $\Box$ as a new operator in propositional logic to give formal expression to this notion of necessity. Thus, for any proposition P, $\Box P$ was to be understood as saying "it is necessary that P (holds)." Since then, this operator has been interpreted in a number of different ways and various axiom systems have been developed to formalize these interpretations; $\Box$ has been interpreted as "it is necessary that," "it is provable that," and "it is computable that"; in his popularized treatment of the Incompleteness Theorems, Smullyan [1987] interprets $\Box$ as "it is believable that." In this chapter, we shall interpret $\Box$ as "it is provable from $\mathbf{H}$ that," where $\mathbf{H}$ is a pre-determined set of sentences of arithmetic.

We now take up the question of how to assign meaning to our modal wffs, i.e., the question of semantics. In the logics we considered in earlier chapters, the question was answered by developing a theory of models for the particular logic we were studying. A similar approach (see Boolos [1979] or Smorynski [1985]) could be carried out here but is unnecessary for our purposes; another kind of interpretation is already suggested by the fact that our proposition symbols denote sentences of arithmetic. We shall show that we can inductively assign a sentence of arithmetic to each modal wff in such a way that connectives are preserved and the symbol $\Box$ has our intended meaning "provable from $\mathbf{H}$." We shall call this association an **arithmetical interpretation of modal logic**. This interpretation will depend only on the choice of a proof formula $\mathbf{PRF_H}$ for $\mathbf{H}$. Each modal wff $\mathbf{A}$ will have an interpretation $I(\mathbf{A})$ which is a sentence the language of arithmetic, and the proof formula for $\mathbf{H}$ will play a special role in this interpretation.

Because of the inductive nature of the definition of the arithmetical interpretation, we need an Inductive Definition Principle for modal wffs and, as usual, this requires a Unique Readability Theorem. The proofs of these are virtually the same as their analogues in propositional logic; proofs at the $\Box$-stage of each argument proceed like the $\neg$-stage of the

corresponding proof in propositional logic. We leave the details to the reader; see Exercise 12.

Definition 5.8.1 Let $\mathbf{H}$ be an axiomatized theory with a proof formula $\mathbf{PRF_H}$. By the **arithmetical interpretation** of modal logic by $\mathbf{PRF_H}$ we mean the function

$$I : WFF(\mathcal{P}) \to SENT(\mathcal{L}),$$

where $\mathcal{L}$ is the vocabulary of arithmetic, defined recursively as follows.

Basis For each proposition symbol $\mathbf{P}$, $I(\mathbf{P}) = \mathbf{P}$.

Negation $I(\neg \mathbf{A}) = \neg I(\mathbf{A})$.

Binary connective For each binary connective $*$,

$$I([\mathbf{A} * \mathbf{B}]) = [I(\mathbf{A}) * I(\mathbf{B})].$$

Modal operator $I(\Box \mathbf{A}) = \exists y\, \mathbf{PRF_H}(\mathbf{a}, y)$,

where $a = \#(I(\mathbf{A}))$ is the code of the sentence $I(\mathbf{A})$.

From now on, it will be understood that I is an arithmetical interpretation of modal logic by $\mathbf{PRF_H}$, where $\mathbf{H}$ is a given axiomatized theory and $\mathbf{PRF_H}$ is a given proof formula for $\mathbf{H}$.

Definition 5.8.2 Let $\mathbf{H}$ be an axiomatized theory and let $\mathbf{PRF_H}$ be a proof formula for $\mathbf{H}$. We say that a modal wff $\mathbf{C}$ **holds for** $\mathbf{PRF_H}$ if its arithmetical interpretation $I(\mathbf{C})$ by $\mathbf{PRF_H}$ is true in $\mathcal{N}$. If the proof formula $\mathbf{PRF_H}$ is clear from the context, we say that $\mathbf{C}$ **holds for** $\mathbf{H}$ if it holds for $\mathbf{PRF_H}$.

The following corollary shows that $\Box \mathbf{A}$ has our intended meaning under the arithmetical interpretation by $\mathbf{PRF_H}$.

Corollary 5.8.3 (Arithmetical Interpretation Theorem) *Let* $\mathbf{PRF_H}$ *be a proof formula for* $\mathbf{H}$. *For every modal wff* $\mathbf{A}$, *the following are equivalent:*

(i) $\Box\mathbf{A}$ *holds for* $\mathbf{H}$, *i.e.* $\mathcal{N} \models I(\Box\mathbf{A})$.

(ii) $I(\mathbf{A})$ *is provable from* $\mathbf{H}$, *i.e.* $\mathbf{H} \vdash I(\mathbf{A})$.

(iii) $I(\Box\mathbf{A})$ *is provable from* $\mathbf{WA}$, *i.e.* $\mathbf{WA} \vdash I(\Box\mathbf{A})$.

Proof: Let a be the code of the wff $I(\mathbf{A})$. Then $I(\Box\mathbf{A})$ is the sentence $\exists y\, \mathbf{PRF_H}(\mathbf{a}, y)$. By Theorem 5.6.3, conditions (i)–(iii) are equivalent. **End of Proof.**

The Arithmetical Interpretation Theorem can be used to translate a statement saying that a modal wff holds for $\mathbf{H}$ to a statement about provability from $\mathbf{H}$. If $\mathbf{A}$ is a simple modal wff where there are no boxes within boxes, the translation is done by replacing each $\Box\mathbf{P}$ within $\mathbf{A}$ by "$\mathbf{H} \vdash \mathbf{P}$." To make the translation more readable, we shall sometimes write $\mathbf{H} \vdash \mathbf{P}$ in either of the long forms

"$\mathbf{H}$ proves $\mathbf{P}$"

or

"$\mathbf{P}$ is provable from $\mathbf{H}$."

Example. Consider the modal wff

$$\mathbf{A}: \quad \Box\mathbf{P} \wedge \Box[\mathbf{P} \Rightarrow \mathbf{Q}] \Rightarrow \Box\mathbf{Q}.$$

"$\mathbf{A}$ holds for $\mathbf{H}$" translates into:

$$\text{If } \mathbf{H} \vdash \mathbf{P} \text{ and } \mathbf{H} \vdash [\mathbf{P} \Rightarrow \mathbf{Q}] \text{ then } \mathbf{H} \vdash \mathbf{Q}.$$

This is the Rule of Modus Ponens, which is true for any theory $\mathbf{H}$ by the Completeness Theorem.

If the modal wff $\mathbf{A}$ has nested boxes, the translation is more difficult and involves codes and the proof formula for $\mathbf{H}$.

Example. Consider the modal wff $\Box\Box\mathbf{P}$. Intuitively, $\Box\Box\mathbf{P}$ says "$\mathbf{H}$ proves that $\mathbf{P}$ is provable from $\mathbf{H}$." By the Arithmetical Interpretation Theorem, $\Box\Box\mathbf{P}$ holds for $\mathbf{H}$ if and only if $\mathbf{H}$ proves $I(\Box\mathbf{P})$. We shall compute $I(\Box\mathbf{P})$ and $I(\Box\Box\mathbf{P})$. Let p be the code of $\mathbf{P}$. Then

$$I(\Box\mathbf{P}) = \exists y\, \mathbf{PRF_H}(\mathbf{p}, y).$$

Therefore $\Box\Box\mathbf{P}$ holds for $\mathbf{H}$ if and only if

$$\mathbf{H} \vdash \exists y\, \mathbf{PRF_H}(\mathbf{p}, y).$$

Now let c be the code of $\exists y\, \mathbf{PRF_H}(\mathbf{p}, y)$. Then c is the code of $I(\Box\mathbf{P})$, so

$$I(\Box\Box\mathbf{P}) = \exists y\, \mathbf{PRF_H}(\mathbf{c}, y).$$

5.9 Modal Systems and Tableaus

We now embark on a discussion of those properties which hold for provability, and formulate them as axioms for a modal logic. One property of provability in both propositional and predicate logic is that for any hypothesis set $\mathbf{H}$, if $\mathbf{H} \vdash \mathbf{A}$ and $\mathbf{H} \vdash [\mathbf{A} \Rightarrow \mathbf{B}]$, then $\mathbf{H} \vdash \mathbf{B}$. This is called the Rule of Modus Ponens and follows from the Completeness Theorem. Thus, we treat the following list of modal wffs as axioms:

$$\Box\mathbf{A} \wedge \Box[\mathbf{A} \Rightarrow \mathbf{B}] \Rightarrow \Box\mathbf{B}, \quad \text{for any modal wffs } \mathbf{A}, \mathbf{B}.$$

Another property of provability that we wish to formalize is that all propositional tautologies are provable. Thus, we would like to say that $\Box\mathbf{A}$ is an axiom for each modal wff $\mathbf{A}$ such that $\mathbf{A}$ is a "tautology." In our modal language, what we mean by a tautology is a modal wff having the form of an ordinary tautology of propositional logic. For instance,

$$[\mathbf{P} \wedge \mathbf{Q}] \Rightarrow \mathbf{P}$$

is a tautology of propositional logic, so

$$\Box\mathbf{A} \wedge [\Box\mathbf{A} \vee \Box\Box\mathbf{B}] \Rightarrow \Box\mathbf{A}$$

is a modal tautology. (Here, we have replaced $\mathbf{P}$ with $\Box\mathbf{A}$ and $\mathbf{Q}$ with $\Box\mathbf{A} \vee \Box\Box\mathbf{B}$.)

Here is a formal definition of modal tautology.

Definition 5.9.1 A **modal tautology** is a modal wff **C** such that for some tautology **D** of ordinary propositional logic with the proposition symbols $\mathbf{P}_1, \ldots, \mathbf{P}_n$ and some list of modal wffs $\mathbf{A}_1, \ldots, \mathbf{A}_n$, **C** is obtained by replacing each occurrence of $\mathbf{P}_i$ in **D** by $\mathbf{A}_i$ for $i = 1, \ldots, n$.

We remark that neither of the wffs

$$\Box[\mathbf{P} \vee \neg\mathbf{P}], \quad \Box\mathbf{P} \vee \Box\neg\mathbf{P}$$

is a modal tautology, (although $\Box\mathbf{P} \vee \neg\Box\mathbf{P}$ is). The first of these modal wffs is "true," that is, $I(\Box[\mathbf{P} \vee \neg\mathbf{P}])$ is true for any **P** under any arithmetical interpretation of modal logic, because for any **H**, $\mathbf{P} \vee \neg\mathbf{P}$ is provable from **H**, but the wff does not satisfy the criterion for a modal tautology. The second wff, however, is not even true in general. For example, let us take **H** to be **WA** and **P** to be the proposition symbol which stands for the sentence $\forall x\, \mathbf{0} * x \doteq \mathbf{0}$. We have seen in Chapter 3 that the sentence $\forall x\, \mathbf{0} * x \doteq \mathbf{0}$ is true in some models of **WA** and false in others, so that neither **P** nor $\neg\mathbf{P}$ is provable from **WA**. Thus the sentence $I([\Box\mathbf{P} \vee \Box\neg\mathbf{P}])$ is false for the arithmetical interpretation of modal logic by $\mathbf{PRF}_{\mathbf{WA}}$.

In this and the next section we shall study four axiom systems for modal logic, called **Mod(0)**, **Mod(1)**, **Mod(2)**, and **Mod(3)**. Other systems will be introduced in the exercises. The axioms will express properties which "ought" to be true about provability. The first of these axiom systems, **Mod(0)**, has an axiom expressing the fact that each modal tautology is provable and an axiom expressing the rule of modus ponens. The other systems add more axioms which we shall discuss later on. We shall list all four systems here so they will be easy to look up, even though we shall need only the first system **Mod(0)** at this time.

Definition 5.9.2 [1] The modal system **Mod(0)** has the following two axiom schemes.

(tt) $\Box\mathbf{C}$ for every modal tautology **C**.

[1] In the literature, the modal system having axiom schemes (tt), (mp), (n), and (fmp) is known as (modal system) K. **Mod(3)** is known as K_4.

(mp) $\Box\mathbf{A} \wedge \Box[\mathbf{A} \Rightarrow \mathbf{B}] \Rightarrow \Box\mathbf{B}$ for all modal wffs **A** and **B**.

The modal system **Mod(1)** has as axioms those of **Mod(0)** together with the axiom scheme

(n) $\Box\mathbf{A} \Rightarrow \Box\Box\mathbf{A}$ for every modal wff **A**.

The modal system **Mod(2)** has as axioms those of **Mod(1)** together with the axiom scheme

(s) $\Box\Box\mathbf{A} \Rightarrow \Box\mathbf{A}$ for all modal wffs **A**.

The modal system **Mod(3)** has as axioms those of **Mod(1)** together with the following axiom schemes for all modal wffs **A** and **B**:

(fmp) $\Box[[\Box\mathbf{A} \wedge \Box[\mathbf{A} \Rightarrow \mathbf{B}]] \Rightarrow \Box\mathbf{B}]$.

(fn) $\Box[\Box\mathbf{A} \Rightarrow \Box\Box\mathbf{A}]$.

(tt) stands for **tautology**, (mp) for **modus ponens**, (n) for **normal**, and (s) for **soundness**. (fmp) stands for **formalized modus ponens** and (fn) for **formalized normal**.

Each of the above modal axiom schemes is actually an infinite list of modal wffs. Each of these individual wffs is an axiom of the corresponding modal logic, and is called an **instance** of the axiom scheme. Note that axiom (s) is included among the **Mod(2)** axioms, but is *not* included among the axioms of **Mod(3)**. We have

$$\mathbf{Mod(0)} \subset \mathbf{Mod(1)},\ \mathbf{Mod(1)} \subset \mathbf{Mod(2)},\ \mathbf{Mod(1)} \subset \mathbf{Mod(3)}.$$

Definition 5.9.3 For each modal system **Mod(k)**, $k = 0, 1, 2, 3$, we define a **Type k theory** to be an axiomatized theory **H** in the language of arithmetic with a proof formula $\mathbf{PRF}_{\mathbf{H}}$ such that **Mod(k)** holds for **H**, that is, the arithmetical interpretation of each **Mod(k)** axiom by $\mathbf{PRF}_{\mathbf{H}}$ is true in $\mathcal{N}$.

Proposition 5.9.4 *Every axiomatized theory is a Type* 0 *theory.*

Proof: Suppose that **H** is an axiomatized theory. First let **C** be a modal tautology, so that $\Box\mathbf{C}$ is an instance of axiom scheme (tt). Then **C** is obtained from a tautology **D** of propositional logic by replacing propositional symbols $\mathbf{P}_1, \ldots, \mathbf{P}_n$ by modal wffs $\mathbf{A}_1, \ldots, \mathbf{A}_n$. By the Completeness Theorem for propositional logic, **D** has a propositional tableau proof **T**. By replacing each propositional symbol $\mathbf{P}_i$ by the modal wff $\mathbf{A}_i$ in **T**, we obtain a modal tableau proof $\mathbf{T}'$ of C using only the propositional tableau rules. Replacing each modal wff **B** in $\mathbf{T}'$ by the wff $I(\mathbf{B})$, we obtain a tableau proof of $I(\mathbf{C})$. Thus $I(\mathbf{C})$ is tableau provable from the empty set of hypotheses, and hence tableau provable from **H**. By the Arithmetical Interpretation Theorem, $I(\Box\mathbf{C})$ is true in $\mathcal{N}$.

Now consider an instance $\Box\mathbf{A} \wedge \Box[\mathbf{A} \Rightarrow \mathbf{B}] \Rightarrow \Box\mathbf{B}$ of axiom scheme (mp). We have

$$I([\Box\mathbf{A} \wedge \Box[\mathbf{A} \Rightarrow \mathbf{B}]] \Rightarrow \Box\mathbf{B}) \;=$$

$$[I(\Box\mathbf{A}) \wedge I(\Box[\mathbf{A} \Rightarrow \mathbf{B}])] \Rightarrow I(\Box\mathbf{B}).$$

Suppose that $I(\Box\mathbf{A})$ and $I(\Box[\mathbf{A} \Rightarrow \mathbf{B}])$ are true in $\mathcal{N}$. By the Arithmetical Interpretation Theorem, both $I(\mathbf{A})$ and $I(\mathbf{A} \Rightarrow \mathbf{B})$ are tableau provable from **H**. Moreover,

$$I([\mathbf{A} \Rightarrow \mathbf{B}]) \;=\; [I(\mathbf{A}) \Rightarrow I(\mathbf{B})].$$

Therefore by the Completeness Theorem 3.5.2, $I(\mathbf{B})$ is tableau provable from **H**. Thus by the Arithmetical Interpretation Theorem, $I(\Box\mathbf{B})$ is true in $\mathcal{N}$. This shows that $I([\Box\mathbf{A} \wedge \Box[\mathbf{A} \Rightarrow \mathbf{B}]] \Rightarrow \Box\mathbf{B})$ is true in $\mathcal{N}$ as required. **End of Proof.**

To prove consequences of **Mod(k)**, we again use the tableau method.

A **modal tableau of type** k, or **Mod(k) tableau**, is defined as in propositional logic except that we add to the usual list of tableau extension rules the following rule:

$\boxed{Ax_k}$ Any axiom of **Mod(k)** can be added at the end of a branch (where $k = 0, 1, 2,$ or 3).

(We have seen this sort of tableau rule before; the Equality Rule $\boxed{=3}$ similarly allows any node to be extended by an axiom.) As before, we declare a branch of a tableau to be contradictory if for some modal wff **A**, both **A** and $\neg\mathbf{A}$ occur on the branch; a **Mod(k)** tableau proof is then defined in the usual way. If there is a **Mod(k)** tableau proof of the modal wff **A**, we write

$$\vdash_k \mathbf{A}.$$

Likewise, if **J** is a set of modal wffs and there is a **Mod(k)** tableau proof of **A** from **J**, we write

$$\mathbf{J} \vdash_k \mathbf{A}.$$

In this book, we shall only consider finite modal tableaus.

The TABLEAU program is equipped to accept modal wffs and execute the $\boxed{Ax_k}$ rules. To run the modal logic version of the TABLEAU program, choose "start a MODAL tableau" at the title screen. You will then be able to enter wffs of modal logic as hypotheses and as formulas to be proved, and to use the axioms of modal logic in tableau proofs. To enter a $\Box$ as part of a modal wff, you can either hit the # key, type in the word BOX, or hold the Ctrl key down and hit B. To use a modal axiom in a tableau, hit the A key at the end of a branch in Tableau mode, and then choose the desired axiom scheme from the menu.

What information do tableau proofs in modal logic give us? In propositional and predicate logic, the tableau method provided a convenient procedure for checking whether a sentence was a semantic consequence of a given hypothesis set. Modal proofs can also be understood in this way by introducing a notion of a *model* for modal logic. Instead, we shall understand modal proofs by going back to our arithmetical interpretation of modal logic. The following proposition is like the Soundness Theorem for propositional logic, and can be proved by induction on the number of nodes of a tableau.

Proposition 5.9.5 *If there is a tableau proof of a modal wff* C *in the modal system* **Mod(k)**, *then for any Type k theory* **H**, C *holds for* **H**.

Thus a **Mod(k)** tableau proof tells us that a modal wff holds for all Type k theories. By Proposition 5.9.4, a **Mod(0)** tableau proof tells us that a modal wff holds for all axiomatized theories.

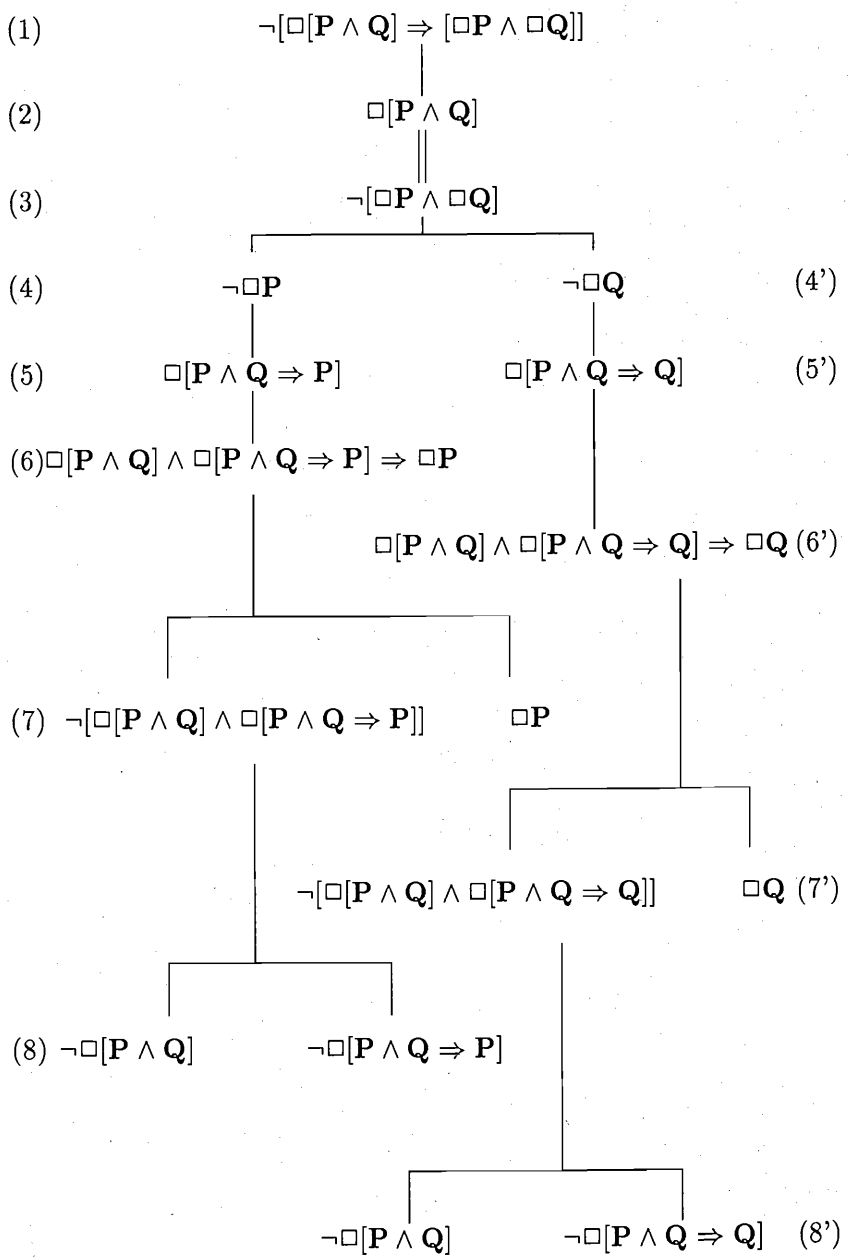
A corresponding completeness theorem can also be formulated; see Exercise 24.

We shall now use modal tableaus to prove some lemmas which will be used later for the incompleteness theorems. In most cases we shall only sketch the main steps of the proof, and leave the construction of a formal tableau proof as a problem using the TABLEAU program.

We state our first lemma formally in modal logic, and then give an English translation in terms of provability from an axiomatized theory **H**. Since this is our first lemma, the full tableau proof will be given in the text.

Lemma 5.9.6 $\vdash_0 \Box[\mathbf{P} \wedge \mathbf{Q}] \Rightarrow [\Box\mathbf{P} \wedge \Box\mathbf{Q}]$
If $\mathbf{H} \vdash \mathbf{P} \wedge \mathbf{Q}$, *then* $\mathbf{H} \vdash \mathbf{P}$ *and* $\mathbf{H} \vdash \mathbf{Q}$.

Proof: Here is an informal proof expressed in terms of provability from **H**. Suppose $\mathbf{H} \vdash \mathbf{P} \wedge \mathbf{Q}$. Since $\mathbf{P} \wedge \mathbf{Q} \Rightarrow \mathbf{P}$ is a tautology, it is provable from **H**. By Axiom Scheme (mp), $\mathbf{H} \vdash \mathbf{P}$. Likewise, since $\mathbf{H} \vdash \mathbf{P} \wedge \mathbf{Q} \Rightarrow \mathbf{Q}$, $\mathbf{H} \vdash \mathbf{Q}$. The formal modal tableau proof appears on the next page.



Justification of nodes: (1): negation of the formula to be proved. (2) and (3): by (1). (4) and (4′): by (3). (5) and (5′): by (tt). (6) and (6′): by (mp). (7): by (6). (7′): by (6′). (8): by (7). (8′):by (7′).
End of Proof.

Remarks (a) First of all, notice that the availability of the $\boxed{Ax_0}$ rule allows us to start with an empty hypothesis set: Since we are allowed to introduce any instance of our axiom schemes at any node, we are saved the inconvenience of having to figure out in advance which axioms to use as our hypothesis set.

(b) The tableau proof above exhibits a pattern which will recur in future proofs: We begin the tableau construction by using all the usual tableau extension rules for *propositional* logic until each node is occupied by a $\Box$-wff (i.e. a modal wff of the form $\Box\mathbf{C}$). Since a $\Box$-wff cannot be broken down further using propositional tableau rules, we must come up with an instance of one (or possibly several) of our modal axiom schemes that can be used in conjunction with further applications of propositional tableau rules to extend the branch in question. We continue extending branches until we reach another $\Box$-wff, and then we repeat the process (unless the branch we are on is contradictory, in which case we move, as usual, to another branch).

Often, the difficult part in the construction is to find the right tautology so that Axiom Scheme (tt) can be used. Recall that in Chapter 1, two methods were developed to show that a propositional wff is a tautology – the truth table method and tableau proofs. These methods can now be used to verify that the (tt) axiom is being used correctly in a modal tableau proof. First, check that the original propositional wff is a tautology either by using truth tables, tableaus, or by finding the wff in one of the lists of particular tautologies developed in Chapter 1. Then make a substitution to get a modal tautology $\mathbf{C}$, and conclude that $\Box\mathbf{C}$ is an instance of Axiom Scheme (tt).

The TABLEAU program makes sure that the (tt) axiom is used correctly. Before adding a wff $\Box\mathbf{C}$ as an instance of (tt) in a modal tableau, you must show that $\mathbf{C}$ is a propositional tautology. The program automatically starts a temporary tableau with root $\neg\mathbf{C}$ for this purpose.

The next lemma is the converse of Lemma 5.9.6.

Lemma 5.9.7 $\vdash_0 [\Box\mathbf{P} \wedge \Box\mathbf{Q}] \Rightarrow \Box[\mathbf{P} \wedge \mathbf{Q}]$
If each of $\mathbf{P}$ *and* $\mathbf{Q}$ *is provable, so is* $\mathbf{P} \wedge \mathbf{Q}$.

Proof: Use the tautology

$$\mathbf{P} \Rightarrow [\mathbf{Q} \Rightarrow [\mathbf{P} \wedge \mathbf{Q}]]$$

to apply Axiom Scheme (tt). Then apply Axiom Scheme (mp) twice, once with $\mathbf{A} = \mathbf{P}$ and $\mathbf{B} = [\mathbf{Q} \Rightarrow [\mathbf{P} \wedge \mathbf{Q}]]$, and once with $\mathbf{A} = \mathbf{Q}$ and $\mathbf{B} = [\mathbf{P} \wedge \mathbf{Q}]$. The Computer Problem 1.TBM asks for a tableau proof.
End of Proof.

The next lemma gives an analogue of Lemma 5.9.6 for the connective $\Rightarrow$, and an analogue of Lemma 5.9.7 for $\vee$.

Lemma 5.9.8 *(a)* $\vdash_0 \Box[\mathbf{P} \Rightarrow \mathbf{Q}] \Rightarrow [\Box\mathbf{P} \Rightarrow \Box\mathbf{Q}]$
If $\mathbf{H} \vdash \mathbf{P} \Rightarrow \mathbf{Q}$ *and* $\mathbf{H} \vdash \mathbf{P}$, *then* $\mathbf{H} \vdash \mathbf{Q}$.

(b) $\vdash_0 [\Box\mathbf{P} \vee \Box\mathbf{Q}] \Rightarrow \Box[\mathbf{P} \vee \mathbf{Q}]$.

If $\mathbf{H} \vdash \mathbf{P}$ *or* $\mathbf{H} \vdash \mathbf{Q}$, *then* $\mathbf{H} \vdash \mathbf{P} \vee \mathbf{Q}$.

Proof: To prove *(a)*, use the tautology

$$[\mathbf{P} \Rightarrow [\mathbf{Q} \Rightarrow \mathbf{R}]] \Leftrightarrow [[\mathbf{P} \wedge \mathbf{Q}] \Rightarrow \mathbf{R}]$$

and Axiom Scheme (mp). The formal tableau proof is left for the student as Computer Problem 2.TBM.

To prove *(b)*, use the tautologies

$$\mathbf{P} \Rightarrow \mathbf{P} \vee \mathbf{Q} \quad \text{and} \quad \mathbf{Q} \Rightarrow \mathbf{P} \vee \mathbf{Q}.$$

The formal tableau proof is computer problem 3.TBM. **End of Proof.**

Example. The converses of the statements in Lemma 5.9.8 do *not* follow from the axioms of **Mod(0)**. We have already seen that if $\mathbf{H}$ is

the axiomatized theory **WA**, **P** is the sentence $\forall x\, \mathbf{0} * x \doteq \mathbf{0}$ and **Q** is $\neg\mathbf{P}$, then the sentence

$$I(\Box[\mathbf{P} \vee \mathbf{Q}] \Rightarrow [\Box\mathbf{P} \vee \Box\mathbf{Q}])$$

is false in $\mathcal{N}$. Thus the modal wff

$$\Box[\mathbf{P} \vee \mathbf{Q}] \Rightarrow [\Box\mathbf{P} \vee \Box\mathbf{Q}]$$

cannot have a **Mod(0)** tableau proof.

The verification that the converse of part *(a)* is also false is left to the reader as Exercise 15.

We plan to use the modal wffs from the preceding lemmas as hypotheses in later modal tableau proofs. To justify this, we need the following theorem, which is the modal form of the Learning Theorem from Chapter 2.

Theorem 5.9.9 (Learning Theorem) *Let* **J** *be a finite set of modal wffs and let* **A** *be a modal wff. For each of our modal systems* **Mod(k)**, *if* $\vdash_k \mathbf{C}$ *for each* $\mathbf{C} \in \mathbf{J}$ *and* $\mathbf{J} \vdash_k \mathbf{A}$ *then* $\vdash_k \mathbf{A}$.

Proof: Let **K** be the set of all **Mod(k)** axioms **B** such that for some $\mathbf{C} \in \mathbf{J}$, either **B** is used in the modal tableau proof for $\vdash_k \mathbf{C}$, or **B** is used in the modal tableau proof for $\mathbf{J} \vdash_k \mathbf{A}$. By moving all the nodes containing these axioms up into the root node, we obtain ordinary propositional tableau proofs for $\mathbf{K} \vdash \mathbf{C}$, all $\mathbf{C} \in \mathbf{J}$, and for $\mathbf{J} \cup \mathbf{K} \vdash \mathbf{A}$. As we saw in Exercise 24 in Chapter 1, there is a propositional tableau proof for $\mathbf{K} \vdash \mathbf{A}$. Moving the modal axioms **K** back down from the root node, we obtain a **Mod(k)** tableau proof for $\vdash_k \mathbf{A}$ as required. **End of Proof.**

The following theorem is often useful in combination with the Learning Theorem as an aid in proving new results from old results.

Theorem 5.9.10 (Modal Substitution Theorem) *Suppose a modal wff* **C** *has a* **Mod(k)** *tableau proof from a set of modal wffs* **J**. *Let* $\mathbf{A}_1, \ldots, \mathbf{A}_n$ *be modal wffs and let* $\mathbf{C}'$, $\mathbf{J}'$ *be formed from* **C** *and* **J** *by replacing each occurrence of the propositional symbols* $\mathbf{P}_1, \ldots, \mathbf{P}_n$ *by the wffs* $\mathbf{A}_1, \ldots, \mathbf{A}_n$. *Then* $\mathbf{C}'$ *has a* **Mod(k)** *tableau proof from* $\mathbf{J}'$.

We leave the proof of this theorem as Exercise 14. The main steps are to check that if **C** is a modal axiom then $\mathbf{C}'$ is a modal axiom, and that if **T** is a modal tableau proof of **C** from **J**, then $\mathbf{T}'$ is a modal tableau proof of $\mathbf{C}'$ from $\mathbf{J}'$.

Most of the modal wffs proved in our lemmas contain one or two propositional symbols **P** and **Q**. The Modal Substitution Theorem shows that the same wffs with **P** and **Q** replaced by arbitrary modal wffs **A** and **B** are also provable in modal logic. For example, Lemma 5.9.6 combined with the Modal Substitution Theorem shows that

$$\vdash_0 \Box[\mathbf{A} \wedge \mathbf{B}] \Rightarrow [\Box\mathbf{A} \wedge \Box\mathbf{B}]$$

for any modal wffs **A** and **B** whatever. Now by the Learning Theorem, any modal wff **C** which has a **Mod(0)** tableau proof with the above wff as a hypothesis also has a **Mod(0)** tableau proof with no hypotheses at all.

We now turn to the modal system **Mod(1)**. Recall that the modal system **Mod(1)** has as axioms those of **Mod(0)** together with the axiom scheme

(n) $\Box\mathbf{A} \Rightarrow \Box\Box\mathbf{A}$ (for every modal wff **A**).

The Axiom Scheme (n) expresses another reasonable property of provability: if a sentence is provable from **H**, it ought to be provable from **H** that it is provable.

The next result shows that **WA** and **PA** are Type 1 theories.

Proposition 5.9.11 *Any axiomatized theory which contains all the axioms of* **WA** *is a Type* 1 *theory.*

Proof: Given an axiomatized theory **H**, we must show that the modal axiom scheme (n) holds for **H**. Let **A** be any modal wff. Thus for each modal wff **A**, we must show that $I(\Box\mathbf{A} \Rightarrow \Box\Box\mathbf{A})$ is true in $\mathcal{N}$. We have

$$I([\Box\mathbf{A} \Rightarrow \Box\Box\mathbf{A}]) = [I(\Box\mathbf{A}) \Rightarrow I(\Box\Box\mathbf{A})].$$

Suppose that $I(\Box\mathbf{A})$ is true in $\mathcal{N}$. By the Arithmetical Interpretation Theorem, the sentence $I(\mathbf{A})$ is tableau provable from $\mathbf{H}$. Let a be the code of $I(\mathbf{A})$. By Theorem 5.6.3,

$$\mathbf{WA} \vdash \exists y\, \mathbf{PRF_H}(\mathbf{a}, y).$$

Under the arithmetical interpretation by $\mathbf{PRF_H}$, we have

$$I(\Box\mathbf{A}) = \exists y\, \mathbf{PRF_H}(\mathbf{a}, y).$$

Since $\mathbf{H}$ contains all the axioms of $\mathbf{WA}$, $\mathbf{H} \vdash I(\Box\mathbf{A})$. Now by the Arithmetical Interpretation Theorem again, $I(\Box\Box\mathbf{A})$ is true in $\mathcal{N}$. Therefore $I(\Box\mathbf{A} \Rightarrow \Box\Box\mathbf{A})$ is true in $\mathcal{N}$, so the modal axiom scheme (n) holds for $\mathbf{H}$. **End of Proof.**

We conclude this section with a discussion of the modal system $\mathbf{Mod(2)}$, which is obtained from $\mathbf{Mod(1)}$ by adding the axiom scheme

(s) $\Box\Box\mathbf{A} \Rightarrow \Box\mathbf{A}$ for all modal wffs $\mathbf{A}$.

This axiom scheme, called the soundness scheme, says that if $\mathbf{H}$ proves that $\mathbf{P}$ is provable then $\mathbf{H}$ proves $\mathbf{P}$. More precisely, $\mathbf{H}$ is a Type 2 theory if and only if $\mathbf{H}$ is a Type 1 theory and for each $\mathbf{A}$, if $\mathbf{H} \vdash I(\Box\mathbf{A})$ then $\mathbf{H} \vdash I(\mathbf{A})$.

Recall that a theory $\mathbf{H}$ is called sound if every sentence which belongs to $\mathbf{H}$ is true in $\mathcal{N}$. Since $\mathbf{WA}$ and $\mathbf{PA}$ are sound, the next result shows that $\mathbf{WA}$ and $\mathbf{PA}$ are Type 2 theories.

Proposition 5.9.12 *Every sound Type* 1 *theory is a Type* 2 *theory.*

Proof: Let $\mathbf{H}$ be a sound Type 1 theory. Consider a modal wff $\mathbf{A}$. Suppose $I(\Box\Box\mathbf{A})$ is true in $\mathcal{N}$. By the Arithmetical Interpretation Theorem, $\mathbf{H} \vdash I(\Box\mathbf{A})$. Since $\mathbf{H}$ is sound, $I(\Box\mathbf{A})$ is true in $\mathcal{N}$. Therefore $I(\Box\Box\mathbf{A} \Rightarrow \Box\mathbf{A})$ is true in $\mathcal{N}$ as required. **End of Proof.**

In Exercise 23, a strengthening (ss) of the axiom scheme (s) is introduced and it is shown that for a Type 1 theory $\mathbf{H}$, $\mathbf{H}$ is sound if and only if (ss) holds for $\mathbf{H}$.

5.10 First Incompleteness Theorem Revisited

In this section we shall revisit the First Incompleteness Theorem from the viewpoint of modal logic. The provability operator $\Box$ in modal logic lets us avoid some of the complicated details involving codes of proofs in formal arithmetic, and for this reason it helps to illuminate the essential ideas in the incompleteness theorems.

To state the incompleteness theorems in modal logic, we need to formalize the statement

$\mathbf{H}$ is consistent

as well as the Gödel sentence $\mathbf{A}_G$. A theory $\mathbf{H}$ is consistent if and only if the false sentence $\mathbf{F}$ is not provable from $\mathbf{H}$. Thus, $\mathbf{Con}_H$ can be formalized by the modal wff

$$\neg\Box\mathbf{F}.$$

A theory $\mathbf{H}$ is consistent if and only if $\neg\Box\mathbf{F}$ holds for $\mathbf{H}$.

As for the Gödel sentence $\mathbf{A}_G$, we can formalize the statement "This sentence is unprovable" by obtaining a modal proposition symbol $\mathbf{P}$ for which $\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}$ holds (intuitively,"$\mathbf{P}$ holds if and only if $\mathbf{P}$ is unprovable"). For $\mathbf{P}$ to be a Gödelian sentence for $\mathbf{H}$, the information that $\mathbf{P}$ asserts its own unprovability must be provable from $\mathbf{H}$. Thus the formal version of our Gödelian sentence becomes:

(*) $$\Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}]$$

"$\mathbf{H}$ proves that $\mathbf{P}$ asserts its own unprovability from $\mathbf{H}$."

Using the Arithmetical Interpretation Theorem, we see that each of the following conditions is equivalent to $\mathbf{P}$ being a Gödelian sentence for an axiomatized theory $\mathbf{H}$ (for a given proof formula $\mathbf{PRF_H}$).

$\Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}]$ holds for $\mathbf{H}$,

$$\mathcal{N} \models I(\Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}]),$$

$$\mathbf{H} \vdash I(\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}),$$
$$\mathbf{H} \vdash \mathbf{P} \Leftrightarrow \neg I(\Box\mathbf{P}).$$

We shall break the modal logic form of the First Incompleteness Theorem into two parts. We begin with Part I. It is similar to Theorem 5.6.10. However, it avoids the soundness assumption and involves only the notion of provability, and thus can be expressed in our modal logic and formalized in arithmetic.

Theorem 5.10.1 (First Incompleteness Theorem, Part I)

$$\vdash_1 \Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}] \Rightarrow [\Box\mathbf{P} \Rightarrow \Box\mathbf{F}].$$

If **P** *is Gödelian for a Type* 1 *theory* **H** *and* **H** *proves* **P**, *then* **H** *is inconsistent.*

Proof: Here is an informal proof. Assume $\Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}]$ and $\Box\mathbf{P}$. Then $\Box\neg\Box\mathbf{P}$ by (mp). Also, $\Box\Box\mathbf{P}$ by (n). Thus $\Box\Box\mathbf{P}$ and $\Box\neg\Box\mathbf{P}$, and so $\Box\mathbf{F}$.

A rigorous proof in modal logic is given by two tableau problems. Problem PART1.TBM gives a **Mod(1)** tableau proof of the desired wff

$$\Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}] \Rightarrow [\Box\mathbf{P} \Rightarrow \Box\mathbf{F}]$$

from the two hypotheses

$$\Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}] \Rightarrow \Box[\mathbf{P} \Rightarrow \neg\Box\mathbf{P}],$$
$$[\Box\Box\mathbf{P} \wedge \Box\neg\Box\mathbf{P}] \Rightarrow \Box[\Box\mathbf{P} \wedge \neg\Box\mathbf{P}].$$

Problem 4.TBM gives a **Mod(0)** tableau proof of the first hypothesis. The second hypothesis is **Mod(0)** tableau provable by Lemma 5.9.7 and the Modal Substitution Theorem. Then by the Learning Theorem, the conclusion is **Mod(1)** tableau provable with no hypotheses. **End of Proof.**

Part II of the First Incompleteness Theorem says that we can replace the strong hypothesis of soundness in Theorem 5.6.10 by the (weaker) axiom scheme (s) and arrive at the same conclusion. Moreover, like Part I, Part II can be formalized in arithmetic; see Exercise 29.

Theorem 5.10.2 (First Incompleteness Theorem, Part II)

$$\vdash_2 \Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}] \Rightarrow [\Box\neg\mathbf{P} \Rightarrow \Box\mathbf{F}].$$

If **P** *is Gödelian for a Type* 2 *theory* **H**, *and* **H** *proves* $\neg$**P**, *then* **H** *is inconsistent.*

Proof: We first give an informal proof.

Assume that $\Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}]$ and $\Box\neg\mathbf{P}$. Since $[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}]$ implies $[\neg\mathbf{P} \Leftrightarrow \Box\mathbf{P}]$ using only propositional logic, $\Box[\neg\mathbf{P} \Rightarrow \Box\mathbf{P}]$. By (mp), $\Box\Box\mathbf{P}$. By Axiom Scheme (s), $\Box\mathbf{P}$, and by Part I of the First Incompleteness Theorem, $\Box\mathbf{F}$.

For a rigorous proof in modal logic, Computer Problem PART2.TBM gives a **Mod(2)** tableau proof of the desired conclusion

$$\Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}] \Rightarrow [\Box\neg\mathbf{P} \Rightarrow \Box\mathbf{F}].$$

from the two hypotheses

$$\Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}] \Rightarrow [\Box\mathbf{P} \Rightarrow \Box\mathbf{F}],$$

$$\Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}] \Rightarrow \Box[\neg\mathbf{P} \Rightarrow \Box\mathbf{P}].$$

The first hypothesis is Part I of the First Incompleteness Theorem, and Computer Problem 5.TBM shows that the second hypothesis has a **Mod(0)** tableau proof. **End of Proof.**

Corollary 5.10.3 *No consistent Type* 2 *theory is complete.*

Proof: Let **H** be a consistent Type 2 theory. Then there is a Gödelian sentence **P** for **H**. Since **H** is consistent, $\Box\mathbf{F}$ does not hold for **H**. Using both parts of the First Incompleteness Theorem, we see that neither $\Box\mathbf{P}$ nor $\Box\neg\mathbf{P}$ holds for **H**, so that $\mathbf{H} \not\vdash \mathbf{P}$ and $\mathbf{H} \not\vdash \neg\mathbf{P}$. Therefore **H** is not complete. **End of Proof.**

5.11 Second Incompleteness Theorem

We now turn to Gödel's Second Incompleteness Theorem. This theorem tells us that one of the sentences which is not provable from **PA** is "**PA** is consistent"! Now since arithmetic is the basis for so much of mathematics, one would hope that **PA** is consistent. Of course, once we know that $\mathcal{N} \models \mathbf{PA}$, we know **PA** is consistent; but Gödel's Second Incompleteness Theorem tells us that the statement "$\mathcal{N} \models \mathbf{PA}$" cannot be formalized and proved within **PA**. But then how does one ever prove "$\mathcal{N} \models \mathbf{PA}$" formally? In particular, how does one construct the model $\mathcal{N}$ *formally*? (Once the model is constructed, it is easy to see that it satisfies **PA**). A reasonable approach is to formalize arithmetic and the notion of a model of arithmetic within set theory, say ZFC. Then the formal statement corresponding to "$\mathcal{N} \models \mathbf{PA}$" can be proved in ZFC; hence, according to ZFC at least, **PA** is consistent. But, is ZFC consistent? Gödel's proof of the Second Incompleteness Theorem can be adapted to show that no proof of the consistency of ZFC can be formalized within ZFC! (See Enderton [1972].) One can, however, work within an even more powerful theory than ZFC to prove formally the consistency of ZFC, but again the consistency of this stronger theory remains problematic. More significantly, the proof of consistency for each of the theories mentioned becomes progressively more difficult and requires more and more machinery.

The moral of these remarks is that the truly endless search for an all-embracing formal system in which all mathematics can be proved consistent is doomed to failure: once a system is rich enough to prove the Peano axioms, it is rich enough for Gödel's Second Incompleteness Theorem to apply.

It would seem that Gödel's incompleteness theorems force us to the viewpoint that any answer to the question

Is mathematics consistent?

must rely in part on non-formal methods. "Mathematical intuition" is an example of such a method: It is a widespread belief among mathematicians that certain mathematical structures are so natural that they need not be formally constructed in order for us to be certain of

their mathematical soundness. Nearly all mathematicians agree that small natural numbers (that can be computed on a computer, say) and computable operations on them can safely be assumed without introducing inconsistency. A slightly stronger claim is that the existence of the standard model $\mathcal{N}$ of arithmetic is a self-evident truth. Nearly all working mathematicians make this assumption in their mathematical practice (whether or not they speak of this philosophical stance, their work reflects this assumption). Once this position is granted, of course, we have the consistency of **PA** given to us – not formally – but by an "*a priori* mathematical intuition" of the model $\mathcal{N}$. Still stronger is the claim that ZFC is consistent; again the justification is the belief in a certain fairly natural model of the ZFC axioms (in Exercises 2.51 and 2.52, the first few levels of this model are constructed). A milder claim is that while ZFC as a whole may be inconsistent, at least that finite fragment of it which has been used to prove the theorems of our present-day mathematics *is* consistent.

We do not raise these issues here with the intention of providing a final answer; philosophies among both mathematicians and philosophers regarding these questions vary widely. Our discussion is intended mainly to offer the reader a sense of the tremendous foundational impact of Gödel's work.

The proof of the Second Incompleteness Theorem is essentially a formalized version of the first part of the First Incompleteness Theorem: In part I of the First Incompleteness Theorem, the wff

$$(*) \qquad \Box\mathbf{P} \Rightarrow \Box\mathbf{F}$$

is proved in **Mod(1)**, assuming **P** is Gödelian. The Main Lemma for the Second Incompleteness Theorem will prove the wff

$$\Box[\Box\mathbf{P} \Rightarrow \Box\mathbf{F}],$$

in the stronger modal system **Mod(3)**, again assuming **P** is Gödelian. Intuitively, this can be accomplished by showing that each step of the proof of $(*)$ can be formalized. For this kind of proof to work, we need to assume as axioms formalized versions of our **Mod(1)** axioms. Thus we are led to postulate formalized versions of Axiom schemes of

Modus Ponens (mp) and Normality (n) as the two new axiom schemes of Formalized Modus Ponens (fmp) and Formalized Normality (fn).

We recall that the axioms for the modal system **Mod(3)** consists of the axioms of **Mod(1)** together with these two axiom schemes,

$$(fmp) \quad \Box[[\Box\mathbf{A} \wedge \Box[\mathbf{A} \Rightarrow \mathbf{B}]] \Rightarrow \Box\mathbf{B}];$$
$$(fn) \quad \Box[\Box\mathbf{A} \Rightarrow \Box\Box\mathbf{A}].$$

Note that **Mod(3)** does not contain the Soundness axiom scheme (s).

It can be shown that **PA** is a Type 3 theory, but the details are beyond the scope of this book. We state without proof a theorem which gives us a rich collection of Type 3 theories.

Theorem 5.11.1 *Any axiomatized theory* **H** *which contains all the axioms of* **PA** *is a Type* 3 *theory.*

One interesting feature of **Mod(3)** is, as Smullyan [1987] describes it, a kind of "self-awareness" - **Mod(3)** "knows" that it satisfies its own axioms in the sense that for each axiom **A** of **Mod(3)**, $\Box\mathbf{A}$ is also provable in **Mod(3)**. This makes **Mod(3)** an especially natural system in which to prove formalized versions of modal theorems. We now prove a theorem showing that **Mod(3)** is even more self aware - it "knows" that each of its theorems is provable. This theorem is a precise form of the intuitive principle that every **Mod(3)** tableau proof can be formalized in **Mod(3)**.

Theorem 5.11.2 (Self-Awareness Theorem) *If* **A** *is a modal wff and* $\vdash_3 \mathbf{A}$, *then* $\vdash_3 \Box\mathbf{A}$.

Proof: We first show that $\vdash_3 \Box\mathbf{K}$ for each axiom **K** of **Mod(3)**.

(mp): Let **K** be the axiom $\Box\mathbf{A} \wedge \Box[\mathbf{A} \Rightarrow \mathbf{B}] \Rightarrow \Box\mathbf{B}$. Then $\Box\mathbf{K}$ is an instance of the Axiom Scheme (fmp) and thus has a **Mod(3)** tableau proof.

(n): Let **K** be the axiom $\Box\mathbf{A} \Rightarrow \Box\Box\mathbf{A}$. Then $\Box\mathbf{K}$ is an instance Axiom Scheme (fn) and hence has a **Mod(3)** tableau proof.

(tt), (fmp) and (fn): Let **K** be an instance of one of these three axiom schemes. In each case, **K** has the form $\Box\mathbf{C}$ for some modal wff

C. Therefore $\Box\mathbf{K}$ is $\Box\Box\mathbf{C}$, which has a **Mod(3)** tableau proof using the two axioms $\Box\mathbf{C}$ and $\Box\mathbf{C} \Rightarrow \Box\Box\mathbf{C}$.

Now let **A** be any modal wff such that $\vdash_3 \mathbf{A}$. Then there is a **Mod(3)** tableau proof of **A**. Let **J** be the finite set of modal axioms which are used in this proof. By moving these axioms up to the root of the tableau, we obtain a tableau proof of **A** from **J** which only uses the tableau rules of propositional logic. Let **E** be the conjunction of all the wffs in the set **J**. Then $\mathbf{E} \Rightarrow \mathbf{A}$ is a modal tautology. Therefore $\Box[\mathbf{E} \Rightarrow \mathbf{A}]$ is a modal axiom (an instance of (tt)). Since each $\mathbf{K} \in \mathbf{J}$ is a modal axiom, we have $\vdash_3 \Box\mathbf{K}$ for each $\mathbf{K} \in \mathbf{J}$. Using 5.9.7 finitely many times, we see that $\vdash_3 \Box\mathbf{E}$. Finally, using the (mp) axiom $\Box\mathbf{E} \wedge \Box[\mathbf{E} \Rightarrow \mathbf{A}] \Rightarrow \Box\mathbf{A}$, we obtain the desired conclusion that $\vdash_3 \Box\mathbf{A}$. **End of Proof.**

The Self-Awareness Theorem may be combined with the Learning and Modal Substitution Theorems to simplify **Mod(3)** tableau proofs. All previous modal lemmas may now be used with a $\Box$ in front. For example, the Self-Awareness Theorem applied to Lemma 5.9.8 *(a)* gives

$$\vdash_3 \Box[\Box[\mathbf{P} \Rightarrow \mathbf{Q}] \Rightarrow [\Box\mathbf{P} \Rightarrow \Box\mathbf{Q}]].$$

(It is provable that if $\mathbf{P} \Rightarrow \mathbf{Q}$ is provable, then whenever **P** is provable, **Q** is also provable.)

The Learning Theorem allows us to add this wff as an extra hypothesis in a **Mod(3)** tableau. Computer Problem 6.TBM asks for a formal **Mod(3)** tableau proof of this wff (without using the Self-Awareness Theorem).

Before proving the main lemma for the Second Incompleteness Theorem, we need the following strengthening of Lemma 5.9.8 *(a)*:

Lemma 5.11.3

$$\vdash_3 \Box[\mathbf{P} \Rightarrow \mathbf{Q}] \Rightarrow \Box[\Box\mathbf{P} \Rightarrow \Box\mathbf{Q}].$$

If a Type 3 *theory* **H** *proves* $\mathbf{P} \Rightarrow \mathbf{Q}$, *then* **H** *also proves that [if* **H** *proves* **P** *then* **H** *proves* **Q***].*

Proof: Here is an informal proof. Assume $\Box[\mathbf{P} \Rightarrow \mathbf{Q}]$. By (n), $\Box\Box[\mathbf{P} \Rightarrow \mathbf{Q}]$. We use the Self-Awareness Theorem to show that Lemma 5.9.8

with a $\Box$ in front is **Mod(3)** tableau provable, so we have

$$\Box[\Box[\mathbf{P} \Rightarrow \mathbf{Q}] \Rightarrow [\Box\mathbf{P} \Rightarrow \Box\mathbf{Q}]].$$

By (mp), it follows that $\Box[\Box\mathbf{P} \Rightarrow \mathbf{Q}]$.

Computer Problem 7.TBM gives a **Mod(1)** tableau proof of the conclusion

$$\Box[\mathbf{P} \Rightarrow \mathbf{Q}] \Rightarrow \Box[\Box\mathbf{P} \Rightarrow \Box\mathbf{Q}]$$

from the two hypotheses

$$\Box[\Box[\mathbf{P} \Rightarrow \mathbf{Q}] \Rightarrow [\Box\mathbf{P} \Rightarrow \Box\mathbf{Q}]],$$

$$\Box[\Box[\mathbf{P} \Rightarrow \mathbf{Q}] \Rightarrow [\Box\mathbf{P} \Rightarrow \Box\mathbf{Q}]] \Rightarrow [\Box\Box[\mathbf{P} \Rightarrow \mathbf{Q}] \Rightarrow \Box[\Box\mathbf{P} \Rightarrow \Box\mathbf{Q}]].$$

The first hypothesis is **Mod(3)** tableau provable by Lemma 5.9.8 and the Self-Awareness Theorem. The second hypothesis is **Mod(0)** tableau provable by Lemma 5.9.8 and the Modal Substitution Theorem, because it is

$$\Box[\mathbf{A} \Rightarrow \mathbf{B}] \Rightarrow [\Box\mathbf{A} \Rightarrow \Box\mathbf{B}]$$

with $\mathbf{A} = \Box[\mathbf{P} \Rightarrow \mathbf{Q}]$ and $\mathbf{B} = [\Box\mathbf{P} \Rightarrow \Box\mathbf{Q}]$. **End of Proof.**

We now come to the Main Lemma.

Lemma 5.11.4 (Main Lemma)

$$\vdash_3 \Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}] \Rightarrow \Box[\Box\mathbf{P} \Rightarrow \Box\mathbf{F}].$$

If **P** *is Gödelian for a Type* 3 *theory* **H**, *then* **H** *proves that if* **H** *proves* **P** *then* **H** *is inconsistent.*

Proof: Here is an informal proof. Assume that $\Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}]$. By Lemma 5.11.3, we have $\Box[\Box\mathbf{P} \Rightarrow \Box\neg\Box\mathbf{P}]$. Axiom Scheme (fn) gives us $\Box[\Box\mathbf{P} \Rightarrow \Box\Box\mathbf{P}]$. Now (fmp) can be used to prove that $\Box[\Box\mathbf{P} \Rightarrow \Box\mathbf{F}]$.

Computer Problem MAIN.TBM gives a rigorous **Mod(0)** tableau proof of the desired conclusion

$$\Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}] \Rightarrow \Box[\Box\mathbf{P} \Rightarrow \Box\mathbf{F}]$$

from the three hypotheses

$$\Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}] \Rightarrow \Box[\mathbf{P} \Rightarrow \neg\Box\mathbf{P}],$$

$$\Box[\mathbf{P} \Rightarrow \neg\Box\mathbf{P}] \Rightarrow \Box[\Box\mathbf{P} \Rightarrow \Box\neg\Box\mathbf{P}],$$

$$\Box[[\Box\mathbf{P} \Rightarrow \Box\neg\Box\mathbf{P}] \Rightarrow [\Box\mathbf{P} \Rightarrow \Box\mathbf{F}]].$$

The first hypothesis is proved in the system **Mod(0)** in Computer Problem 4.TBM. The second hypothesis is Lemma 5.11.3 with $\neg\Box\mathbf{P}$ for **Q**. The wff after the $\Box$ in the third hypothesis is proved in the system **Mod(1)** in Computer Problem 8.TBM. The Self-Awareness Theorem now shows that the third hypothesis has a **Mod(3)** tableau proof. Thus by the Learning Theorem, the conclusion has a **Mod(3)** tableau proof. **End of Proof.**

Now, at long last, we are ready to prove Gödel's Second Incompleteness Theorem.

Theorem 5.11.5 (Second Incompleteness Theorem)

$$\vdash_3 \Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}] \Rightarrow [\Box\neg\Box\mathbf{F} \Rightarrow \Box\mathbf{F}]$$

If **P** *is Gödelian for a Type* 3 *theory* **H**, *and* **H** *proves its own consistency, then* **H** *is inconsistent.*

Proof: Here is an informal proof. Assume **P** is Gödelian and $\Box\neg\Box\mathbf{F}$. By the Main Lemma, $\Box[\Box\mathbf{P} \Rightarrow \Box\mathbf{F}]$. Computer Problem 9.TBM shows that this implies $\Box[\neg\Box\mathbf{F} \Rightarrow \neg\Box\mathbf{P}]$. Then by (mp) we have $\Box\neg\Box\mathbf{P}$. Since **P** is Gödelian, it follows that $\Box[\neg\Box\mathbf{P} \Rightarrow \mathbf{P}]$. By (mp) again, $\Box\mathbf{P}$. By the First Incompleteness Theorem Part I, $\Box\mathbf{P} \Rightarrow \Box\mathbf{F}$. Therefore $\Box\mathbf{F}$ as required.

For a modal tableau proof, Computer Problem SECOND.TBM gives a **Mod(0)** tableau proof of the desired conclusion

$$\Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}] \Rightarrow [\Box\neg\Box\mathbf{F} \Rightarrow \Box\mathbf{F}]$$

from the hypotheses

$$\Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}] \Rightarrow \Box[\Box\mathbf{P} \Rightarrow \Box\mathbf{F}],$$

$$\Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}] \Rightarrow \Box[\neg\Box\mathbf{P} \Rightarrow \mathbf{P}],$$

$$\Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}] \Rightarrow [\Box\mathbf{P} \Rightarrow \Box\mathbf{F}],$$

$$\Box[\Box\mathbf{P} \Rightarrow \Box\mathbf{F}] \Rightarrow \Box[\neg\Box\mathbf{F} \Rightarrow \neg\Box\mathbf{P}].$$

The Main Lemma says that the first hypothesis has a **Mod(3)** tableau proof, the second hypothesis has an easy **Mod(0)** tableau proof similar to the Computer Problems 4.TBM and 5.TBM, the third hypothesis is Part I of the First Incompleteness Theorem, and Computer Problem 9.TBM gives a **Mod(0)** tableau proof of the third hypothesis. Since each hypothesis has a **Mod(3)** tableau proof, the conclusion is **Mod(3)** tableau provable by the Learning Theorem. **End of Proof.**

We conclude this section by mentioning several additional results which are worked out in the exercises.

First, we have shown that if **P** is Gödelian for a Type 3 theory **H** and if **H** is consistent, then **H** neither proves **P** (First Incompleteness Theorem) nor $\neg I(\Box\mathbf{F})$ (Second Incompeteness Theorem). We could have proved the second of these from the first by proving the remarkable fact that for such **H**, **P** and $\neg I(\Box\mathbf{F})$ *are provably equivalent*! That is,

$$\vdash_3 \Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{F}].$$

This tells us that *all Gödelian sentences are equivalent*! In other words, if **P** and **Q** are both Gödelian for a Type 3 theory **H**, then

$$\mathbf{H} \vdash \mathbf{P} \Leftrightarrow \mathbf{Q}.$$

See Exercise 22.

The Second Incompleteness Theorem says that consistent Type 3 theories with a Gödelian sentence cannot prove their own consistency. But what about weaker theories? If we are content to replace consistency with soundness, it can be shown that no sound *axiomatized* theory with a Gödelian sentence can prove its own soundness; see Exercise 24.

We have seen that the First Incompleteness Theorem tells us that for sound Type 1 theories, a sentence which provably asserts its own unprovability is unprovable, but *true*. What can be said about a sentence which provably asserts its own *provability*? In other words, what conclusions can be drawn from the modal wff

$$\Box[\mathbf{P} \Leftrightarrow \Box\mathbf{P}]?$$

Such a wff is called a **Henkin sentence**. The Diagonalization Lemma shows that **PA** (and many other theories as well) has a Henkin sentence,

and a result known as Löb's Theorem demonstrates that such sentences are always provable in **PA** (hence true). These matters are taken up in Exercises 25 and 27.

As we observed earlier, the proof of the Second Incompleteness Theorem is largely a formalization of the proof of the first half of the First Incompleteness Theorem. Can a formalized version of the *second half* of the First Incompleteness Theorem be proved? What about a formalized version of the *Second* Incompleteness Theorem? We investigate these questions in Exercise 29.

Our proof of Gödel's First Incompleteness Theorem depended on the construction of some version of a Gödelian sentence. As we explained earlier, Gödelian sentences express in the formal language of arithmetic the proposition "I am unprovable." The original form of this latter proposition is known as the **Liar Paradox**: "This sentence is false." It has the property that it's true if and only if it's false, and is therefore a primitive version of a Gödelian sentence. It is possible to prove versions of the First Incompleteness Theorem using a formalized translation of another famous paradox – **Berry's Paradox** – quite different in spirit from the Liar Paradox and its variations.

Berry's Paradox arises from the following consideration: Suppose you are asked to make a list of all natural numbers which can be described using fewer than 100 keystrokes on a typewriter. The first few natural numbers could be described by simply typing out the usual base 10 numerals $0, 1, \ldots, 100, 101, \ldots, 10,000, \ldots$. However, once we reach numbers which have 100 or more digits, we might resort to English sentences which describe a procedure that would "compute" these larger numbers. Thus, for example, "1 followed by 99 zeroes" describes a number whose base 10 numeral is too long to type out. Now notice that if we are allowed at most 99 keystrokes in a description, and our typewriter has only, say, 70 keys, then only finitely many descriptions are possible. Thus there is a natural number which cannot be described using fewer than 100 keystrokes; and if there is such a number at all, there must be a *least* such number n. Thus,

(*) n is the smallest natural number which cannot be described using fewer than 100 keystrokes.

But now (*) is a description of n which uses fewer than 100 keystrokes!

The paradox is partially resolved by the fact that we have not been very clear about which expressions count as "descriptions" of natural numbers. The notion of "description" can, however, be made rigorous; in fact, we gave a definition of what it means to "name" a natural number in Exercise 3.9. Using this definition, Berry's Paradox has a formal version which leads to a proof of the First Incompleteness Theorem. In Exercise 30, we outline a proof (due to Boolos – see Boolos [1989]) of the First Incompleteness Theorem which uses this formal version of Berry's Paradox.

For the reader who would like to do further reading in this area, we recommend Smullyan [1987], Boolos [1979], and Smorynski [1985].

5.12 Modal Tableau Problems (TAB7)

In these problems the reader is asked to use the TABLEAU program to work out the indicated proofs. The modal system is given. The problem files are located in directory TAB7 on the distribution diskette, and the install program will put them in a subdirectory called TAB7 on your hard disk.

1.TBM	Hypotheses:	none
	To be proved:	$[\Box\mathbf{P} \wedge \Box\mathbf{Q}] \Rightarrow \Box[\mathbf{P} \wedge \mathbf{Q}]$
	Modal System:	**Mod(0)**
	Can be done in 15 nodes.	
2.TBM	Hypotheses:	none
	To be proved:	$\Box[\mathbf{P} \Rightarrow \mathbf{Q}] \Rightarrow [\Box\mathbf{P} \Rightarrow \Box\mathbf{Q}]$
	Modal System:	**Mod(0)**
	Can be done in 9 nodes.	
3.TBM	Hypotheses:	none
	To be proved:	$[\Box\mathbf{P} \vee \Box\mathbf{Q}] \Rightarrow \Box[\mathbf{P} \vee \mathbf{Q}]$
	Modal System:	**Mod(0)**
	Can be done in 16 nodes.	
4.TBM	Hypothesis:	$\Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}]$
	To be proved:	$\Box[\mathbf{P} \Rightarrow \neg\Box\mathbf{P}]$
	Modal system:	**Mod(0)**
	Can be done in 6 nodes.	
5.TBM	Hypothesis:	$\Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}]$
	To be proved:	$\Box[\neg\mathbf{P} \Rightarrow \Box\mathbf{P}]$
	Modal System:	**Mod(0)**
	Can be done in 6 nodes.	
6.TBM	Hypotheses:	none
	To be proved:	$\Box[\Box[\mathbf{P} \Rightarrow \mathbf{Q}] \Rightarrow [\Box\mathbf{P} \Rightarrow \Box\mathbf{Q}]]$
	Modal system:	**Mod(2)**
	Can be done in 7 nodes.	

7.TBM	Hypotheses:	$\Box[\Box[\mathbf{P} \Rightarrow \mathbf{Q}] \Rightarrow [\Box\mathbf{P} \Rightarrow \Box\mathbf{Q}]]$
	$\Box[\Box[\mathbf{P} \Rightarrow \mathbf{Q}] \Rightarrow [\Box\mathbf{P} \Rightarrow \Box\mathbf{Q}]] \Rightarrow$	$[\Box\Box[\mathbf{P} \Rightarrow \mathbf{Q}] \Rightarrow \Box[\Box\mathbf{P} \Rightarrow \mathbf{Q}]]$
	To be proved:	$\Box[\mathbf{P} \Rightarrow \mathbf{Q}] \Rightarrow \Box[\Box\mathbf{P} \Rightarrow \Box\mathbf{Q}]$
	Modal System:	**Mod(1)**
	Can be done in 9 nodes.	
8.TBM	Hypotheses:	none
	To be proved:	$[\Box\mathbf{P} \Rightarrow \Box\neg\Box\mathbf{P}] \Rightarrow [\Box\mathbf{P} \Rightarrow \Box\mathbf{F}]$
	Modal System:	**Mod(1)**
	Can be done in 22 nodes.	
9.TBM	Hypothesis:	$\Box[\Box\mathbf{P} \Rightarrow \Box\mathbf{F}]$
	To be proved:	$\Box[\neg\Box\mathbf{F} \Rightarrow \neg\Box\mathbf{P}]$
	Modal System:	**Mod(0)**
	Can be done in 6 nodes.	
10.TBM	Hypotheses:	None
	To be proved:	$\Box[\mathbf{P} \Leftrightarrow \mathbf{Q}] \Rightarrow [\Box\mathbf{P} \Leftrightarrow \Box\mathbf{Q}]$
	Modal System:	**Mod(0)**
	Can be done in 28 nodes.	
PART1.TBM	Hypotheses:	$\Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}] \Rightarrow [\Box\mathbf{P} \Rightarrow \neg\Box\mathbf{P}]$
		$\Box\Box\mathbf{P} \wedge \Box\neg\Box\mathbf{P} \Rightarrow \Box[\Box\mathbf{P} \wedge \neg\Box\mathbf{P}]$
	To be proved:	$\Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}] \Rightarrow [\Box\mathbf{P} \Rightarrow \Box\mathbf{F}]$
	Modal System:	**Mod(1)**
	Can be done in 24 nodes.	
PART2.TBM	Hypotheses:	$\Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}] \Rightarrow [\Box\mathbf{P} \Rightarrow \Box\mathbf{F}]$
		$\Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}] \Rightarrow \Box[\neg\mathbf{P} \Rightarrow \Box\mathbf{P}]$
	To be proved:	$\Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}] \Rightarrow [\Box\neg\mathbf{P} \Rightarrow \Box\mathbf{F}]$
	Modal System:	**Mod(2)**
	Can be done in 18 nodes.	

MAIN.TBM	Hypotheses:	$\Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}] \Rightarrow \Box[\mathbf{P} \Rightarrow \neg\Box\mathbf{P}]$
		$\Box[\mathbf{P} \Rightarrow \neg\Box\mathbf{P}] \Rightarrow \Box[\Box\mathbf{P} \Rightarrow \Box\neg\Box\mathbf{P}]$
		$\Box[[\Box\mathbf{P} \Rightarrow \Box\neg\Box\mathbf{P}] \Rightarrow [\Box\mathbf{P} \Rightarrow \Box\mathbf{F}]]$
	To be proved:	$\Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}] \Rightarrow \Box[\Box\mathbf{P} \Rightarrow \Box\mathbf{F}]$
	Modal System:	**Mod(0)**
	Can be done in 11 nodes.	
SECOND.TBM	Hypotheses:	$\Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}] \Rightarrow \Box[\Box\mathbf{P} \Rightarrow \Box\mathbf{F}]$
		$\Box[\mathbf{P} \Rightarrow \neg\Box\mathbf{P}] \Rightarrow \Box[\neg\Box\mathbf{P} \Rightarrow \mathbf{P}]$
		$\Box[\mathbf{P} \Rightarrow \neg\Box\mathbf{P}] \Rightarrow [\Box\mathbf{P} \Rightarrow \Box\mathbf{F}]$
		$\Box[[\Box\mathbf{P} \Rightarrow \Box\mathbf{F}] \Rightarrow \Box[\neg\Box\mathbf{F} \Rightarrow \neg\Box\mathbf{P}]]$
	To be proved:	$\Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}] \Rightarrow [\Box\neg\Box\mathbf{F} \Rightarrow \Box\mathbf{F}]$
	Modal System:	**Mod(0)**
	Can be done in 24 nodes.	

5.13 Exercises

In the exercises for this chapter, all wffs are understood to be in the language of arithmetic.

In Exercises 1 - 3 below, the reader is asked to use Church's Thesis to verify that certain functions associated with syntax are computable.

1. Use Church's Thesis to show that the partial function which sends the code $\#(\mathbf{A})$ to the code $\#(\neg\mathbf{A})$, for each wff $\mathbf{A}$, is computable.

2. Suppose f is a computable function and for all n, $f(n)$ is the code of a wff $\mathbf{A}_n$. Suppose h is defined by

$$h(n) = \#(\mathbf{A}_1 \wedge \cdots \wedge \mathbf{A}_n).$$

Show that h is computable. (Hint: Show that h is obtained from the function $(\#(\mathbf{A}), \#(\mathbf{B})) \mapsto \#(\mathbf{A} \wedge \mathbf{B})$ by primitive recursion and use Church's Thesis to show that the latter is computable.)

3. Use Church's Thesis to prove that the function which takes a pair (m, n), to $\#(\mathbf{A(n)})$ if $m = \#(\mathbf{A}(v))$ and $n \in \mathbf{N}$, and takes (m, n) to 0 otherwise, is computable.

4. Prove that every theory **H** in the language of arithmetic which is consistent but not complete has an extension **H′** which is consistent but not sound.

5.

(a) Suppose that f is a unary weakly representable partial function, and the domain of f is representable (as a unary relation). Prove that f is representable, and that f can be extended to a total computable function.

(b) Give an example of a unary representable partial function f such that f can be extended to a total computable function but the domain of f is not representable.

6.

(a) Show that the unary relation

$$E = \{n : n \text{ is an even natural number}\}$$

is representable. Hint: Show that E is represented by the wff **E**, given by

$$\mathbf{E}(x) = \exists z[z \leq x \wedge x \doteq z + z].$$

(b) Let $\mathbf{B}(x, y)$ be the wff given by

$$\mathbf{B}(x, y) = \neg\mathbf{E}(x) \wedge \mathbf{E}(y) \Rightarrow \neg\mathbf{E}(x + y).$$

Intuitively, **B** says that if x is odd and y is even, then $x + y$ is odd. Show that for all $m, n \in \mathbf{N}$,

$$\mathbf{WA} \vdash \mathbf{B}(\mathbf{m}, \mathbf{n}).$$

(c) Let **C** be the *sentence* given by

$$\mathbf{C} = \forall x \forall y\, [\neg\mathbf{E}(x) \wedge \mathbf{E}(y) \Rightarrow \neg\mathbf{E}(x + y)].$$

Intuitively, **C** also says that an odd plus an even is an odd. However, because no restriction has been placed on how the variables

x and y are interpreted, the sentence **C**–unlike the wff $\mathbf{B}(x, y)$–asserts that this property must hold even for the most bizarre interpretations of x and y in nonstandard models. Not surprisingly, the assertion cannot be proved in **WA**; prove this; i.e., prove that

$$\mathbf{WA} \nvdash \mathbf{C}.$$

(Hint: a counter-model is given in Example 3.7.4.)

7. Show that the following relations are representable:

(a) the binary relation consisting of those pairs (a, b) of natural numbers for which b is divisible by a (assume that 0 is divisible by every number);

(b) the unary relation consisting of all prime numbers (recall that p is prime if $p > 1$ and the only divisors of p are 1 and p itself).

8. Show that the Fibonacci sequence F (considered as a unary (total) function) is representable, where F is given by the following data:

$$F(0) = 1, \quad F(1) = 1$$

$$F(n + 2) = F(n + 1) + F(n).$$

(Thus, F can be expressed as the sequence $1, 1, 2, 3, 5, 8, 13, \ldots$.)

9. In this exercise, we discuss a stronger kind of representability of a relation in a theory than was considered in the text. We will use the results of this exercise in Exercise 30. Suppose R is a finite subset of $\mathbf{N}$, say $R = \{r_1, r_2, \ldots, r_k\}$.

(a) Give an example of a wff $\mathbf{A}(x)$ which represents R.

(b) Suppose we are given a wff $\mathbf{A}(x)$ which represents R. Show that, although it is true that R consists precisely of those natural numbers n for which
$\mathbf{WA} \vdash \mathbf{A}(\mathbf{n})$, this information may not be available from within

WA; i.e., show that there may not be a tableau proof from **WA** of the sentence

$$\forall x\,[\mathbf{A}(x) \Rightarrow [x \doteq \mathbf{r_1} \vee x \doteq \mathbf{r_2} \vee \ldots \vee x \doteq \mathbf{r_k}]].$$

(Hint: Consider the case in which $R = \{1\}$. Design a wff **A** which represents R but for which the sentence

$$\mathbf{B} = \forall x\,[\mathbf{A}(x) \Rightarrow x \doteq \mathbf{s(0)}]$$

is independent of **WA**. Use the standard model $\mathcal{N}$ to show that there is a model of **WA** which satisfies **B**; then use one of the other models of **WA** given in the text to show that $\neg\mathbf{B}$ is also consistent with **WA**.)

(c) In light of part (b), we make the following definition:

Definition. A wff $\mathbf{A}(x)$ with just one free variable x **names** the finite set $R = \{r_1, r_2, \ldots, r_k\}$ in the theory **H** if

$$\mathbf{H} \vdash \forall x\,[\mathbf{A}(x) \Leftrightarrow [x \doteq \mathbf{r_1} \vee x \doteq \mathbf{r_2} \vee \ldots \vee x \doteq \mathbf{r_k}]].$$

In the special case in which R has only one element n, we say that $\mathbf{A}(x)$ **names the natural number** n in **H**. Thus, $\mathbf{A}(x)$ names n in **H** if

$$\mathbf{H} \vdash \forall x\,[\mathbf{A}(x) \Leftrightarrow x \doteq \mathbf{n}].$$

For each n, give a wff which names n in **WA**.

(d) Show that if a wff **A** names n in **WA**, then **A** represents the relation $R = \{n\}$.

10. Prove that for any wffs $\mathbf{A}(u)$ and $\mathbf{B}(v)$ where u is not free in **B** and v is not free in **A**, the following set of three wffs is tableau confutable.

$$\exists u\,[\mathbf{A}(u) \wedge (\forall v \leq u)\neg\mathbf{B}(v)],$$

$$\exists v\,[\mathbf{B}(v) \wedge (\forall u \leq v)\neg\mathbf{A}(u)],$$

$$\forall u \forall v\,[u \leq v \vee v \leq u].$$

11. (Recursively Enumerable Sets).

Definition A subset $A \subset \mathbf{N}$ is **recursively enumerable** (or r.e.) if $A = \emptyset$ or A is the range of a total computable function (i.e. there is a total computable function f such that for each $a \in A$ there is a number n such that $f(n) = a$). The r.e. relations are defined in a similar way.

(a) Show that the following are equivalent for a subset $A \subset \mathbf{N}$:

(i) For some computable binary relation R, $a \in A$ if and only if there is b such that $(a, b) \in R$.

(ii) A is the domain of a computable partial function.

(iii) A is recursively enumerable.

(iv) A is weakly representable.

(b) Prove that a subset $A \subset \mathbf{N}$ is computable if and only if both A and its complement $\mathbf{N}\backslash A = \{x \in \mathbf{N} \mid x \notin A\}$ are recursively enumerable.

(c) We say that a total function $f : \mathbf{N} \to \mathbf{N}$ is **increasing** if whenever $m < n$, $f(m) < f(n)$. Show that a subset $A \subset \mathbf{N}$ is computable if and only if either A is finite or A is the range of an increasing computable function.

12. Formulate and prove a Unique Readability Theorem and an Inductive Definition Principle for modal wffs.

13. Show that the following are modal tautologies.

(a) $\Box[\mathbf{P} \Rightarrow \mathbf{Q}] \Rightarrow [[\Box\mathbf{P} \vee \Box\mathbf{Q}] \Rightarrow \Box[\mathbf{P} \Rightarrow \mathbf{Q}]]$

(b) $\Box[\mathbf{P} \wedge \Box\mathbf{Q} \Leftrightarrow \mathbf{R}] \vee \neg\Box[\mathbf{P} \wedge \Box\mathbf{Q} \Leftrightarrow \mathbf{R}]$

(c) $[\Box[\mathbf{P} \vee \Box\mathbf{Q}] \Rightarrow \Box\mathbf{F}] \wedge [\neg\Box[\mathbf{P} \vee \Box\mathbf{Q}] \Rightarrow \Box\mathbf{F}] \Rightarrow \Box\mathbf{F}$.

14. Prove the Modal Substitution Theorem.

15.

(a) Show that the converse of part (a) of Lemma 5.9.8 is not generally true by finding a suitable axiomatized theory **H** and suitable wffs **P**, **Q**.

(b) Prove or disprove:

(i) $\vdash_0 [\Box\mathbf{P} \Leftrightarrow \Box\mathbf{Q}] \Rightarrow \Box[\mathbf{P} \Leftrightarrow \mathbf{Q}]$

(ii) $\vdash_0 \Box[\mathbf{P} \Leftrightarrow \mathbf{Q}] \Rightarrow [\Box\mathbf{P} \Leftrightarrow \Box\mathbf{Q}]$.

16. Show that if an axiomatized theory **H** is *consistent with* **WA** (i.e., $\mathbf{H} \cup \mathbf{WA}$ is consistent), then **H** is incomplete.

17. Show that any axiomatized theory which is consistent with **WA** is undecidable.

The next two exercises give two alternative proofs that **PA** is incomplete. Note that by Theorem 5.7.3, it suffices to show that **PA** is undecidable.

18.

Definition Suppose A and B are disjoint recursively enumerable sets of natural numbers (see Exercise 11). Then A and B are **recursively inseparable** if there is no computable set C such that $A \subseteq C$ and $B \cap C = \emptyset$.

In this problem, **PA** is shown to be undecidable from the fact that the sets $P_1 = \{\#(\mathbf{A}) : \mathbf{PA} \vdash \mathbf{A}\}$ and $P_0 = \{\#(\mathbf{A}) : \mathbf{PA} \vdash \neg\mathbf{A}\}$ are recursively inseparable.

(a) Show that the sets P_0 and P_1 described above are recursively inseparable.

(Hint: Suppose C is a computable set such that $A \subseteq C$ and $B \cap C = \emptyset$. C is representable by a wff $\mathbf{C}(x)$. By the Diagonalization Lemma, there is a sentence **P** with code p such that

$$\mathbf{PA} \vdash [\mathbf{P} \Leftrightarrow \neg\mathbf{C}(\mathbf{p})].$$

Get a contradiction by considering whether $p \in C$.)

(b) Use part (a) to show that neither P_0 nor P_1 is computable; conclude that **PA** is an undecidable theory.

19. In this exercise, the undecidability of **PA** is proved from the undecidability of the Halting Problem. Let K_0 be the set of all (x, y) such that x is the Gödel number of a program $\mathbf{P}_x$ which halts on input y.

(a) Let B be the set of all quadruples (x, y, z, t) such that x is the Gödel number of a program $\mathbf{P}_x$ which on input y outputs z after $\mathbf{P}_x$ has executed fewer than t steps. Show that B is a computable 4-ary relation.

(b) Using the Equivalence Theorem, we can find a wff $\mathbf{B}(x, y, z, w)$ which represents B. Prove that the wff

$$\mathbf{A}(x, y) = \exists z\, \exists w\, \mathbf{B}(x, y, z, t)$$

weakly represents K_0.

(c) Prove that if **PA** is decidable, so is K_0; i.e., decidability of **PA** implies the decidability of the halting problem. The same argument works for any sound theory $\mathbf{H} \supset \mathbf{WA}$ in place of **PA**.

20. (Another form of the Self-Awareness Theorem.) Prove that if $\mathbf{J} \vdash_3 \mathbf{A}$ then $\Box\mathbf{J} \vdash_3 \Box\mathbf{A}$, where $\Box\mathbf{J}$ denotes the set of wffs

$$\{\Box\mathbf{C} : \mathbf{C} \in \mathbf{J}\}.$$

21. Suppose a Type 2 consistent theory **H** proves a sentence of the form $\mathbf{P} \Leftrightarrow I(\Box\neg\mathbf{P})$ (notice that this sentence is not quite in the form that makes **P** Gödelian for **H**). Show that

(a) $\mathbf{H} \nvdash \mathbf{P}$ and $\mathbf{H} \nvdash \neg\mathbf{P}$;

(b) $\mathbf{P}$ actually is Gödelian for $\mathbf{H}$; i.e. $\mathbf{H} \vdash \mathbf{P} \Leftrightarrow \neg I(\Box\mathbf{P}))$.

22. Prove that if $\mathbf{P}$ is Gödelian for a Type 3 theory $\mathbf{H}$ (not necessarily consistent) then $\mathbf{P}$ is provably equivalent to $\neg\Box\mathbf{F}$; i.e. show

$$\vdash_3 \Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}] \Rightarrow \Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{F}].$$

Then show that it follows that all Gödelian sentences for such a theory are equivalent, i.e.,

$$\vdash_3 [\Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}] \wedge \Box[\mathbf{Q} \Leftrightarrow \neg\Box\mathbf{Q}]] \Rightarrow \Box[\mathbf{P} \Leftrightarrow \mathbf{Q}].$$

23. Suppose $\mathbf{H}$ is a sound axiomatized theory. Then for all modal wffs $\mathbf{A}$,

$$\text{if } \mathbf{H} \vdash I(\mathbf{A}) \text{ then } \mathcal{N} \models I(\mathbf{A}),$$

i.e. provability of $\mathbf{A}$ implies $\mathbf{A}$ is true. Thus, if $\mathbf{P}$ is Gödelian for $\mathbf{H}$, not only is $\mathbf{P} \Leftrightarrow \neg I(\Box\mathbf{P})$ provable in $\mathbf{H}$, but it is actually true. These observations lead us to a somewhat different proof of Part 1 of the First Incompleteness Theorem for sound theories. We begin by defining a modal system $\mathbf{Mod(4)}$: $\mathbf{Mod(4)}$ has as axioms those of $\mathbf{Mod(0)}$ together with the axiom scheme

$$(ss) \qquad \Box\mathbf{A} \Rightarrow \mathbf{A}, \text{ for all modal wffs } \mathbf{A}.$$

(Here, "ss" stands for "strong soundness.")

(a) Prove that an axiomatized theory is Type 4 if and only if it is a sound theory.

(b) Prove

$$\vdash_4 [\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}] \Rightarrow \neg\Box\mathbf{P}.$$

(If $\mathbf{P}$ truly asserts its own unprovability from $\mathbf{H}$, then $\mathbf{P}$ is unprovable from $\mathbf{H}$.)

(c) Prove

$$\vdash_4 [\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}] \Rightarrow \neg\Box\neg\mathbf{P}.$$

(If $\mathbf{P}$ truly asserts its own unprovability from $\mathbf{H}$, then $\neg\mathbf{P}$ is unprovable from $\mathbf{H}$.)

Parts (a) – (c) together show that no Type 4 theory is complete.

(d) Show that (ss) is really a strengthening of (s) by proving that for all $\mathbf{A}$,

$$\vdash_4 \Box\Box\mathbf{A} \Rightarrow \Box\mathbf{A},$$

and noting that there is a Type 4 theory for which (ss) does not hold. (Hint: For the second half, try an inconsistent theory.)

Putting (a) – (d) together, we conclude:

Theorem. *A theory* $\mathbf{H}$ *is incomplete whenever* $\mathbf{H}$ *is sound and there is a sentence* $\mathbf{P}$ *such that*

$$\mathcal{N} \models \mathbf{P} \text{ if and only if } \mathbf{H} \nvdash \mathbf{P}.$$

24. While $\mathbf{PA}$ is Type 3 and satisfies (ss), $\mathbf{PA}$ does *not* satisfy a formalized version of (ss):

$$(fss): \quad \Box[\Box\mathbf{A} \Rightarrow \mathbf{A}] \text{ for all modal wffs } \mathbf{A}.$$

In fact, as is shown in this exercise, no sound axiomatized theory which satisfies Axiom Scheme (fss) has a Gödelian sentence. Prove this by carrying out the following steps:

(a) Prove

$$\mathbf{J} \vdash_0 \Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}] \Rightarrow \Box\mathbf{P}$$

where $\mathbf{J} = \{\Box[\Box\mathbf{P} \Rightarrow \mathbf{P}]\}$; in other words, $\mathbf{J}$ contains a single instance of (fss) where $\mathbf{A} = \mathbf{P}$.

(b) Show

$$\mathbf{J} \vdash_4 \Box[\mathbf{P} \Rightarrow \neg\Box\mathbf{P}] \Rightarrow \mathbf{F}.$$

(c) Conclude that if **H** is a sound axiomatized theory which satisfies (fss), then **H** has no Gödelian sentence. Hence **PA** does not satisfy (fss).

25. (*Löb's Theorem.*) Although many instances of (fss) must fail in most "reasonable" theories of arithmetic, there are some instances of (fss) which hold in *every* axiomatized theory of arithmetic; for instance, if **P** is a tautology, $\mathbf{H} \vdash I(\Box\mathbf{P}) \Rightarrow \mathbf{P}$. More generally, if $\mathbf{H} \vdash \mathbf{P}$, then $\mathbf{H} \vdash [I(\Box\mathbf{P}) \Rightarrow \mathbf{P}]$.

In this exercise, the reader is asked to show that in **PA**, the *only* sentences **P** for which

$$\mathbf{PA} \vdash I(\Box\mathbf{P}) \Rightarrow \mathbf{P}.$$

are those provable from **PA**, i.e., for all sentences **P**,

(*) $\qquad \mathbf{PA} \vdash I(\Box\mathbf{P}) \Rightarrow \mathbf{P}$ implies $\mathbf{PA} \vdash \mathbf{P}$

where I is the interpretation function relative to $\mathbf{PR}_{\mathbf{PA}}$. The statement (*) is called **Löb's Theorem.** We state this theorem in a more general form and outline the steps of proof in parts (a) - (c) below.

Löb's Theorem. *Suppose* **H** *is a Type* 3 *theory. Then for each sentence* **Q**,

$$\mathbf{H} \vdash I(\Box\mathbf{Q}) \Rightarrow \mathbf{Q}, \textit{ if and only if } \mathbf{H} \vdash \mathbf{Q}.$$

In particular, (*) *holds.*

(a) Prove the implication from right to left for any axiomatized theory using modal tableaus; i.e., show

$$\vdash_0 \Box\mathbf{Q} \Rightarrow [\Box\Box\mathbf{Q} \Rightarrow \Box\mathbf{Q}].$$

(b) Use the Diagonalization Lemma to show that any axiomatized theory **H** which includes **WA** has the following property (L):

(L) for every sentence **Q**, there is a sentence **P** such that $\mathbf{H} \vdash \mathbf{P} \Leftrightarrow [I(\Box\mathbf{P}) \Rightarrow \mathbf{Q}]$.

(c) Prove

$$\mathbf{J} \vdash_3 \Box[\Box\mathbf{P} \Rightarrow \mathbf{Q}],$$

where **J** consists of the modal wffs

$$\Box[\mathbf{P} \Leftrightarrow [\Box\mathbf{P} \Rightarrow \mathbf{Q}]]$$

$$\Box[\Box\mathbf{Q} \Rightarrow \mathbf{Q}].$$

(d) Using the result of part (c), prove

$$\mathbf{K} \vdash_3 \Box\mathbf{Q},$$

where **K** consists of the modal wffs

$$\Box[\mathbf{P} \Leftrightarrow [\Box\mathbf{P} \Rightarrow \mathbf{Q}]]$$

$$\Box[\Box\mathbf{P} \Rightarrow \mathbf{Q}].$$

(e) Put parts (a) - (d) together to prove Löb's Theorem.

Löb's Theorem gives us another property of provability: Let **Mod(5)** be the modal system whose axioms are those of **Mod(3)** together with the axiom scheme

(g) $\Box[\Box\mathbf{A} \Rightarrow \mathbf{A}] \Rightarrow \Box\mathbf{A}$ for all modal wffs **A**

("g" stands for "Gödel.") As the previous exercise shows, the axioms of **Mod(5)** hold for **PA**. Remarkably, if $\not\vdash_5 \Box\mathbf{A}$ for some **A**, then there is a way to assign the modal proposition symbols to sentences of arithmetic so that the translation of **A** as a sentence of arithmetic is not provable from **PA**! Thus **Mod(5)** "captures" **PA** in a modal fashion and is an important modal system for studying arithmetic (see Boolos [1979] for more discussion). In the following two exercises, we use **Mod(5)** to establish several interesting facts about **PA**.

26. In this exercise, the reader is led to a *modal* proof that any Type 5 theory has a Gödelian sentence.

(a) Show that if $\mathbf{A}, \mathbf{B}$, and $\mathbf{C}$ are ordinary propositional wffs, then

$$\mathbf{A} \Rightarrow \mathbf{B} \vdash [\mathbf{A} \Rightarrow \mathbf{C}] \Rightarrow [\mathbf{B} \Rightarrow \mathbf{C}].$$

(b) Show that for any modal wff $\mathbf{A}$,

$$(*) \qquad \vdash_5 \Box[[\Box\mathbf{A} \Rightarrow \mathbf{A}] \Leftrightarrow [\Box[\Box\mathbf{A} \Rightarrow \mathbf{A}] \Rightarrow \mathbf{A}]].$$

(Note: Because we have a new axiom (g) in **Mod(5)**, the Self-Awareness Theorem is not guaranteed to hold; it *can* be proved, however, and the reader may wish to assume it in working this problem. The more thorough reader, after proving $(*)$ with the leftmost '$\Box$' removed, will want to check that each step of his proof can be formalized, so that $(*)$ is established without assuming the Self-Awareness Theorem.)

(c) Show that

$$\vdash_5 \Box[\mathbf{B} \Leftrightarrow \neg\Box\mathbf{B}]$$

where $\mathbf{B}$ is the modal wff $\Box\mathbf{F} \Rightarrow \mathbf{F}$.

27.

(a) Assume that **PA** is a **Mod(5)** theory. Suppose $\mathbf{P}$ provably asserts its own provability, or, somewhat more generally, assume

$$\mathbf{PA} \vdash \mathbf{P} \Rightarrow I(\Box\mathbf{P}).$$

Show that $\mathbf{PA} \vdash \mathbf{P}$ and hence that $\mathbf{P}$ is true. (Use Löb's Theorem.)

Such a sentence is called a **Henkin sentence** for **PA**.

(b) Show that any axiomatized theory including **WA** has a Henkin sentence. (Hint: Use the Diagonalization Lemma.)

28.

(a) Give an example of a axiomatized theory $\mathbf{H}$ and a sentence $\mathbf{P}$ to demonstrate that

$$\nvdash_0 \neg\Box\mathbf{P} \Rightarrow \Box\neg\mathbf{P}.$$

(b) As in (a), show

$$\nvdash_0 \Box[\neg\Box\mathbf{P} \Rightarrow \Box\neg\mathbf{P}].$$

(c) Show that if $\mathbf{H}$ is a sound Type 3 theory satisfying (s) and having a Gödelian sentence $\mathbf{P}$, and if $\mathbf{Q} = [\neg\Box\mathbf{P} \Rightarrow \Box\neg\mathbf{P}]$, then

$$\neg\Box\mathbf{Q} \Rightarrow \Box\neg\mathbf{Q}$$

is false (in $\mathcal{N}$).

29. (Formalizations.) In this exercise, we present the formalized versions of several of the important theorems discussed in the text.

(a) (*Part II of the First Incompleteness Theorem*) Show that

$$\vdash_3 \Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}] \Rightarrow \Box[[\neg\Box\mathbf{F} \wedge [\Box\Box\mathbf{P} \Rightarrow \Box\mathbf{P}]] \Rightarrow \neg\Box\neg\mathbf{P}].$$

(Hint: First prove $\vdash_3 \Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}] \Rightarrow [\neg\Box\neg\mathbf{P} \Rightarrow \neg\Box\Box\mathbf{P}]$).

(b) (*Second Incompleteness Theorem*) Prove

$$\vdash_3 \Box[\mathbf{P} \Leftrightarrow \neg\Box\mathbf{P}] \Rightarrow \Box[\Box\neg\Box\mathbf{F} \Rightarrow \Box\mathbf{F}].$$

(c) (*Löb's Theorem*) Formulate a formalization of Löb's Theorem and prove it in the modal system **Mod(5)**.

30. This exercise outlines Boolos' proof [1989] of (a version of) Gödel's First Incompleteness Theorem, namely, that no sound recursively enumerable theory of arithmetic is complete.

Recall from Exercise 3.9 that a wff $\mathbf{A}(x)$ having x as its only free variable **names** a natural number n in an axiomatized theory $\mathbf{H}$ if

$$\mathbf{H} \vdash \forall x\,[\mathbf{A}(x) \Leftrightarrow x \doteq \mathbf{n}].$$

(a) Let **H** be a sound axiomatized theory. Show that for each natural number m, there is a least natural number n_m which cannot be named in **H** by any wff having $\leq m$ symbols and whose variables (bound or free) lie in the set $\{x_1, x_2, \ldots, x_m\}$. (In the present context, the number of symbols in a wff is the length of the sequence obtained by thinking of the wff as simply a string of symbols; more formally, if **A** is a wff and $z = \#(\mathbf{A})$, then the number of symbols of **A** equals $Terms(z)$.

(b) Suppose **H** is a sound axiomatized theory. Show that the following relations are r.e.

$$P_{\mathbf{H}} = \left\{(n, a) : \begin{array}{l}\text{the number } n \text{ is named in } \mathbf{H} \\ \text{by the wff coded by } a\ .\end{array}\right\}$$

$$Q_{\mathbf{H}} = \left\{(n, b) : \begin{array}{l}\text{the number } n \text{ is named by a wff having} \\ \text{exactly } b \text{ symbols and whose variables} \\ \text{(bound or free) are among } x_1, x_2, \ldots, x_b\end{array}\right\}.$$

(c) Recall from part (e) of Exercise 11 that the relation $Q_{\mathbf{H}}$ in part (b) – being r.e. – is weakly represented by some wff **A**. Use **A** to build another wff **B** which weakly represents the following relation:

$$R_{\mathbf{H}} = \left\{(n, d) : \begin{array}{l}n \text{ is the least natural number not named by} \\ \text{any wff that contains fewer than } d \text{ symbols} \\ \text{and whose variables (bound or free) are} \\ \text{among } x_1, x_2, \ldots, x_d\end{array}\right\}.$$

(d) Let **B** be as in (c); let k be the number of symbols in **B**. Why may we assume that all variables (bound or free) which occur in **B** are among $x_1, x_2, \ldots, x_k$ and that the only variables which occur free in **B** are x_1 and x_2?

(e) Continuing part (d), define the following wff $\mathbf{C}(x_1)$:

$$\mathbf{C}(x_1) \doteq \mathbf{B}(x_1, x_2//\mathbf{10} * \mathbf{k}).$$

Using part (a), let $n = n_m$ where $m = 10 * k$. First show that $\mathbf{C}(x_1)$ does *not* name the number n in **H**, i.e., that

$$\mathbf{H} \not\vdash \forall x_1 [\mathbf{C}(x_1) \Leftrightarrow x_1 \doteq \mathbf{n}].$$

(Hint: Count the number of symbols in **C**.)

Then show that

$$\mathcal{N} \models \forall x_1 [\mathbf{C}(x_1) \Leftrightarrow x_1 \doteq \mathbf{n}].$$

Thus, show that there is a sentence, true in $\mathcal{N}$, which can neither be proved nor disproved from **H**.

In addition to giving a new proof of the First Incompleteness Theorem, this problem suggests a resolution of Berry's Paradox (as formulated at the end of Section 5.9): As we mentioned in the text, the paradoxical nature of the fact that the sentence

(*) n is the smallest natural number which cannot be described using fewer than 100 keystrokes.

"describes" the number n may hinge on a lack of preciseness in our account of which strings of keystrokes actually count as "descriptions." Indeed, in the above problem the paradox dissolves once we make it clear that a natural number n is "described" by a formula if and only if n is *named* by the formula in the theory at hand, say **PA**; for then the formal version of (*) – namely, $\mathbf{C}(x_1)$ – does not actually "describe" (*name*) the number n, although it does *weakly represent* it. It may be that our experience with this problem generalizes to *any* attempt to be precise about the meaning of "description" in Berry's Paradox: Perhaps (*), because it uses the notion of "description" in "describing" n, is a description of an inherently different kind from strings of keystrokes (like '100') which do not refer to the notion of "description" at all. Thus, one might reasonably conjecture at this point that *any* formalization of Berry's Paradox – and in particular, of the notion of "description" – would result in the conclusion that (*) does *not* describe n in the formal sense, and the paradox is thereby resolved.

Appendix A

Sets and Functions

In this section of the appendix we discuss some of the basic notions of – what is sometimes called – *naive set theory*; these include the notions of *set, subset, set operations (union, intersection, etc.), functions, cardinality, finite sequences,* and *permutations.* Although these concepts are fairly easy to grasp, very little of higher level mathematics could be developed without them. Probably because of its simplicity, naive set theory is rarely taught *explicitly* in third and fourth year undergraduate courses; students at this level are generally expected to "pick it up" as they go along. Unfortunately, however, it often happens that areas of confusion in courses like abstract algebra and analysis arise from a too fragile grasp of the ideas to be discussed here. Our intention here is to provide a straightforward development of these concepts, to be used by the reader as necessary to supplement his knowledge.

A.1 Sets

Intuitively speaking, a set X divides the mathematical universe into two parts: those objects x which **belong** to X and those which don't. The notation $x \in X$ means x belongs to X, the notation $x \notin X$ means that x does not belong to X. The objects which belong to X are called the **elements** of X or the **members** of X. Other words which are roughly synonymous with the word *set* are *class*, *collection*, and *aggregate*. These longer words are often used simply to avoid using the

word *set* twice in one sentence. (The situation typically arises when an author wants to talk about sets whose elements are themselves sets; he might say " the collection of all finite sets of integers" rather than "the set of all finite sets of integers.") Authors typically try to denote sets by capital letters (e.g. X) and their elements by the corresponding small letters (e.g. $x \in X$) but are not required to do so by any commonly used convention.

The reader who has worked through Section 2.12 should be aware that technically speaking, a *set* is, by definition, a member of a model of ZFC (or of some other axiomatic theory of sets); and while the words *collection* and *aggregate* do not have technical definitions, a *class* is defined to be a collection defined by a predicate. Thus, every set is a class, but not conversely. For example, the collection of all even natural numbers is (by the Axiom of Comprehension) a set, and therefore a class; but the collection of *all sets* is a class (defined by the predicate $x = x$) which is not a set. In general, unless there is some danger that the collection of objects at hand is "too big" to be a set (and this does happen in some areas of mathematics), the collections referred to by mathematicians are to be understood as *sets*.

The simplest sets are **finite** and these are often defined by simply listing (**enumerating**) their elements between curly brackets. Thus if $X = \{2, 3, 8\}$ then $3 \in X$ and $7 \notin X$. Often an author uses dots as a notational device to mean "etcetera" and indicate that the pattern continues. Thus if

$$A = \{a_1, a_2, \ldots, a_n\} \tag{A.1}$$

then for any object b, the phrase "$b \in A$" and the phrase "$b = a_i$ for some $i = 1, 2, \ldots$" have the same meaning; i.e., one is true if and only if the other is. Having defined A by (A.1) we have

$$b \in A \Longleftrightarrow b = a_1 \text{ or } b = a_2 \text{ or } \ldots \text{ or } b = a_n,$$

where the symbol $\Longleftrightarrow$ means *if and only if*. In other words, the shorter phrase "$b \in A$" has the same meaning as the more cumbersome phrase "$b = a_1$ or $b = a_2$ or $\ldots$ $b = a_n$."

The device of listing some of the elements with dots between curly brackets can also be used to define infinite sets provided that the context makes it clear what the dots stand for. For example we can define the

set of **natural numbers** by

$$\mathbf{N} = \{0, 1, 2, 3, \ldots\}$$

and the set of **integers** by

$$\mathbf{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$$

and hope that the reader understands that $0 \in \mathbf{N}$, $5 \in \mathbf{N}$, $-5 \notin \mathbf{N}$, $\frac{3}{5} \notin \mathbf{N}$, $0 \in \mathbf{Z}$, $5 \in \mathbf{Z}$, $-5 \in \mathbf{Z}$, $\frac{3}{5} \notin \mathbf{Z}$, etc..

Certain sets are so important that they have names:

$\emptyset$	(the empty set)
$\mathbf{N}$	(the natural numbers)
$\mathbf{Z}$	(the integers)
$\mathbf{Q}$	(the rational numbers)
$\mathbf{R}$	(the real numbers)
$\mathbf{C}$	(the complex numbers)

These names are almost universally used by mathematicians today, but in older books one may find other notations. Here are some true assertions: $0 \notin \emptyset$, $\frac{3}{5} \in \mathbf{Q}$, $\sqrt{2} \notin \mathbf{Q}$, $\sqrt{2} \in \mathbf{R}$, $x^2 \neq -1$ for all $x \in \mathbf{R}$, and $x^2 = -1$ for some $x \in \mathbf{C}$ (namely $x = \pm i$).

If X is a set and $P(x)$ is a property which either holds or fails for each element $x \in X$, then we may form a new set Y consisting of all $x \in X$ for which $P(x)$ is true. This set Y is denoted by

$$Y = \{x \in X : P(x)\} \tag{A.2}$$

and[1] called "the set of all $x \in X$ such that $P(x)$." For example, if $Y = \{x \in \mathbf{N} : x^2 < 6 + x\}$, then $2 \in Y$ (as $2^2 < 6 + 2$), $3 \notin Y$ (as $3^2 \not< 6 + 3$), and $-1 \notin Y$ (as $-1 \notin \mathbf{N}$).

This is a very handy notation. Having defined Y by (A.2) we may assert that for all x

$$x \in Y \Longleftrightarrow x \in X \text{ and } P(x)$$

[1]The symbol | is sometimes used instead of : here

and that for all $x \in X$

$$x \in Y \Longleftrightarrow P(x).$$

where the symbol $\Longleftrightarrow$ means *if and only if*. Since the property $P(x)$ may be quite cumbersome to state, the notation $x \in Y$ is both shorter and easier to understand. The reader who has worked through Section 2.12 will recognize that the collection Y is guaranteed by ZFC to be a *set* (by the Axiom of Comprehension).

Example A.1.1 Using these notations, the set E of even natural numbers may be denoted by any of the following three notations:

$$\begin{aligned} E &= \{0, 2, 4, \ldots\} \\ &= \{m \in \mathbf{N} : m \text{ is divisible by } 2\} \\ &= \{2n : n \in \mathbf{N}\} \end{aligned}$$

A set Y is a **subset** of a set X, written

$$Y \subset X$$

iff every element of Y is an element of X.

For example,

$$\{1, 3, 4, 7\} \subset \{0, 1, 2, 3, 4, 7, 9\}$$

since every element on the left appears on the right. On the other hand,

$$\{1, 3, 4, 7\} \not\subset \{0, 1, 2, 4, 7, 9\}$$

since $3 \in \{1, 3, 4, 7\}$ but $3 \notin \{0, 1, 2, 4, 7, 9\}$.

Note the following inclusions:

$$\mathbf{N} \subset \mathbf{Z}$$

(every natural number is an integer),

$$\mathbf{Z} \subset \mathbf{Q}$$

(every integer is a rational number),

$$\mathbf{Q} \subset \mathbf{R}$$

(every rational number is a real number), and

$$\mathbf{R} \subset \mathbf{C}$$

(every real number is a complex number).

The empty set is a subset of every set:

$$\emptyset \subset X$$

for every set X. This is because *every* element x of the empty set lies in X – or indeed satisfies any other property – since there are no such elements x. However, while it is true that the empty set is a *subset* of every set, it is certainly not an *element* of every set: for instance, the set $\{1, 2\}$ contains only the numbers 1 and 2 and hence does not contain $\emptyset$ as a member. Also, do not confuse the empty set with the set whose only element is 0:

$$\emptyset \neq \{0\}$$

since $0 \in \{0\}$ but $0 \notin \emptyset$.

Let Y and X be two sets. Two sets are **equal**, written $X = Y$, if $X \subset Y$ and $Y \subset X$, i.e., if every element of X is an element of Y and every element of Y is an element of X.

Example A.1.2 Let $X = \{n \in \mathbf{N} : n^2 + 7 < 6n\}$ and $Y = \{2, 3, 4\}$. Then $X = Y$. In other words, the natural numbers n which satisfy the inequality

$$n^2 + 7 < 6n$$

are precisely $n = 2, 3, 4$. (This may be proved by graphing the function $y = x^2 + 7 - 6x$.)

It follows from the definitions that a set defined by an enumeration is unaffected by the order of the enumeration and by any repetitions in the enumeration. Thus

$$\{1, 3, 7\} = \{3, 1, 7\} = \{3, 1, 7, 1, 3\}.$$

The reader who has read Section 2.12 may wish to verify that the statement "The sets X and Y are equal if and only if X and Y have the same elements" is a theorem of ZFC – it follows from the Axiom of Extensionality.

A.2 Boolean Operations

The **intersection**, $X \cap Y$, of two sets X and Y is the set of objects in both of them:

$$X \cap Y = \{z : z \in X \text{ and } z \in Y\}.$$

X and Y are said to be **disjoint** if they have an empty intersection, i.e., if

$$X \cap Y = \emptyset.$$

The **union**, $X \cup Y$, of two sets X and Y is the set of objects in one or the other of them:

$$X \cup Y = \{z : z \in X \text{ or } z \in Y\}.$$

There is a notation resembling the sigma notation for sums, for the intersection and union of a collection of sets. If $\{X_i\}_{i \in I}$ is a family of sets indexed by some index set I, then the intersection of the family is

$$\bigcap_{i \in I} X_i = \{z : z \in X_i \text{ for all } i \in I\}$$

and the union is

$$\bigcup_{i \in I} X_i = \{z : z \in X_i \text{ for some } i \in I\}.$$

For example, if $I = \{1, 2, 3\}$, then

$$\bigcap_{i \in I} X_i = X_1 \cap X_2 \cap X_3, \text{ and } \bigcup_{i \in I} X_i = X_1 \cup X_2 \cup X_3.$$

Two special cases of taking unions of indexed collections that occur frequently in this book are **increasing unions** and **disjoint unions**.

Increasing Unions Suppose $X_0, X_1, \ldots, X_i, \ldots$ denote sets such that $X_0 \subset X_1 \subset \ldots \subset X_i \subset \ldots$, so that each set is included in the next as a subset. Suppose

$$X = \bigcup_{i \in \mathbf{N}} X_i.$$

Then X is called the **increasing union** of the X_i. X has the property that for each $x \in X$, there is a natural number k such that for all $j \geq k$, $x \in X_j$. In other words, not only is each x in X a member of some X_k, but x is in *every* X_j for $j \geq k$ as well.

Disjoint Unions Suppose now that $\{X_i : i \in I\}$ is any collection of sets, indexed by a set I. Sometimes it is useful to think of the elements in all of the X_i as collected together in a single set in such a way that members of X_i are distinguishable from members of X_j whenever $i \neq j$. If the X_i already happen to be pairwise disjoint (i.e., for all $i \neq j$, $X_i \cap X_j = \emptyset$), our goal is easily accomplished by simply taking the union of the X_i, as above. But if there is some object x for which $x \in X_i \cap X_j$, and $i \neq j$, then once we take the union of the X_i, x must be thought of as coming from both X_i and X_j, and possibly other sets. This situation can be undesirable; for instance, when we described the set of all function symbols to be used in full predicate logic, we wanted to collect all elements of the $\mathcal{F}_n$, $n = 0, 1, \ldots$. But if a particular function symbol $\mathbf{F}$ occurs in both $\mathcal{F}_n$ and $\mathcal{F}_m$ (and $n \neq m$) then this lack of uniqueness causes ambiguity in the use of $\mathbf{F}$ in our logic, since, for instance, we don't have a unique arity associated with it (it's both n-ary and m-ary). To avoid this complication, we required that the union of the $\mathcal{F}_n$ be a *disjoint* union.

In practice, "taking the disjoint union" of a collection usually means that we insist that the sets whose union we will take are already disjoint. Thus, in our example above, we can simply *define* the sets $\mathcal{F}_n$ so that function symbols occurring in one of these sets do not occur in any other. Occasionally however, one is presented with sets whose elements one does not wish to redefine. For such (rare) occasions, we define the disjoint union as follows:

Given sets X_i as above, we first replace each X_i by the set $\tilde{X}_i$ of all pairs (i, x) for which $x \in X_i$. Now $\tilde{X}_i$ is essentially the same as X_i, only now every member x of X_i is "tagged" with the index i. The **disjoint**

union of the X_i is then defined to be the union of the $\tilde{X}_i$:

$$\bigcup_{i \in I} \tilde{X}_i.$$

One other set operation which is often used is the **difference** $X \setminus Y$ of two sets X and Y, defined by

$$X \setminus Y = \{x | x \in X \text{ and } x \notin Y\}.$$

When $Y \subset X$, this is also called the **complement** of Y in X.

Problem A.2.1 Prove that $Y \subset X$ if and only if $Y \setminus X = \emptyset$.

Problem A.2.2 Prove DeMorgan's Laws:

$$Y \setminus \left(\bigcup_{i \in I} X_i \right) = \left(\bigcap_{i \in I} Y \setminus X_i \right)$$

$$Y \setminus \left(\bigcap_{i \in I} X_i \right) = \left(\bigcup_{i \in I} Y \setminus X_i \right)$$

Problem A.2.3 Write out the elements of the disjoint union of the sets $\{0, 1\}$ and $\{1, 2\}$. How many elements are in the disjoint union? How many are in the (ordinary) union of these two sets?

A.3 Functions

A **function** is a mathematical object f consisting of a set X called the **domain** of f, a set Y called the **codomain** of f, and an operation which assigns to every element $x \in X$ a unique value $f(x) \in Y$. This is summarized by the notation

$$f : X \to Y$$

(Note that the arrow goes from domain to codomain.) Other words which are roughly synonymous with the word *function* are **map**, **mapping**, and **transformation**[2]. When a function f is defined without explicit mention of X as above, the domain of f is denoted (in this book at least)

$$Dom(f).$$

The unique value assigned by a function f to $x \in X$ is usually denoted by $f(x)$ but in some contexts other notations such as f_x or fx are customary. $f(x)$ is sometimes called the **value** of f for **argument** x. In the context of computability theory in which computable functions are treated like computing machines (see Chapter 4) $f(x)$ is called the **output** of f for **input** x. We will use this latter terminology frequently in this appendix because (we feel) it helps the beginner form a clearer picture of the concept of function and its properties.

The set of all values (outputs) $f(x)$ of a function f is called its **range** and is denoted $Ran(f)$:

$$Ran(f) = \{f(x) : x \in Dom(f)\}.$$

Any numerical expression involving a real variable defines a function. For example, the equation

$$f(x) = \frac{1}{1 - x}$$

defines a function $f : X \to \mathbf{R}$ whose domain is given by

$$X = \{x \in \mathbf{R} : x \neq 1\}$$

and whose range is given by

$$Ran(f) = \{x \in \mathbf{R} : x \neq 0\}.$$

(In elementary algebra and calculus texts, the domain of a function defined by an explicit formula in this fashion is always assumed to be

[2]It should be mentioned that in some areas of mathematics, the word "map" (or "mapping") is reserved for "structure preserving" functions. For instance, in group theory, a *map* is often understood to be a *group homomorphism*; in topology, it is often taken to be a *surjective continuous* function.

the largest set where the formula is meaningful and the codomain is assumed to be the set $\mathbf{R}$ of real numbers. In more advanced books it is customary to specify domain and codomain as part of the definition.)

Sometimes one wishes to refer to a function without giving it a name. A good way to do this is with the symbol $\mapsto$. Thus one could refer to the function f defined above as the function

$$\{x \in \mathbf{R} : x \neq 1\} \to \mathbf{R} : \; x \mapsto \frac{1}{1-x}.$$

One might call this the function which maps the number x to the number $1/(1-x)$.

Two functions $f_1 : X_1 \to Y_1$ and $f_2 : X_2 \to Y_2$ are **equal** if their domains and codomains are equal ($X_1 = X_2$ and $Y_1 = Y_2$), and they return the same output for any input: $f_1(x) = f_2(x)$ for all $x \in X_1$. This may be summarized symbolically by:

$$f_1 = f_2 \quad \Longleftrightarrow \quad \begin{cases} X_1 = X_2, \, Y_1 = Y_2, \text{ and} \\ f_1(x) = f_2(x) \text{ for all } x \in X_1. \end{cases}$$

We caution the reader that according to this definition of equality the two functions $f : \mathbf{N} \to \mathbf{N}$ and $g : \mathbf{N} \to E = \{n : n \text{ is an even natural number}\}$ defined by

$$f(n) = g(n) = 2n$$

for $n \in \mathbf{N}$ are not equal since their codomains are not equal. It may seem like nit-picking to distinguish these two (and indeed until recently most authors did not) but failure to make the distinction sometimes leads to confusion. For instance, as we shall see below, the function g is *onto* (since its codomain and range are equal) while f is not (3 is in the codomain of f, but not in its range). Hence, even in this simple example, it makes sense to distinguish these functions. (In more advanced areas of mathematics, such as algebraic topology and model theory, this distinction is at times crucial.)

Beginners often confuse the function with the formula which defines it. This leads to confusion because

- Not every function is defined by a single formula. For example, the function $g : \mathbf{R} \to \mathbf{R}$ defined by

$$g(x) = \begin{cases} x^2 & \text{if } x > 0 \\ 7 & \text{if } x = 0 \\ x^3 & \text{if } x < 0 \end{cases}$$

requires three formulas to define it: the formula to use in evaluating the output $g(x)$ depends on the value of the input x.

- Different formulas can define the same function. For example, the formulas

$$g(x) = x^2$$

and

$$h(t) = t^2$$

define the same function from $\mathbf{R}$ into $\mathbf{R}$. As another example, the function $f_1 : \mathbf{R} \to \mathbf{R}$ defined by

$$f_1(x) = (x+1)^2$$

is the same as the function $f_2 : \mathbf{R} \to \mathbf{R}$ defined by

$$f_2(x) = x^2 + 2x + 1.$$

The reason that $f_1 = f_2$ is that the domain of f_1 is the same as the domain of f_2 (namely $\mathbf{R}$), the codomain of f_1 is the same as the codomain of f_2 (namely $\mathbf{R}$), and $f_1(x) = f_2(x)$ for every $x \in \mathbf{R}$. The point is that the *formulas* $(x+1)^2$ and $x^2 + 2x + 1$ are different (simply because they look different) but their *values* are the same for all x.

Suppose $f : X \to Y$. For any subset $A \subset X$ the set

$$f(A) = \{f(x) : x \in A\}$$

is called the **image** of A by f. For any subset $B \subset Y$ the set

$$f^{-1}(B) = \{x : f(x) \in B\}$$

is called the **preimage** of B by f. The image $f(X)$ of the whole space X by f is called the **range** of f (the reader may wish to check that this definition is equivalent to the definition of "range" given earlier in the appendix).

Problem A.3.1 Show that

$$\begin{aligned} f(\bigcup_i A_i) &= \bigcup_i f(A_i) \\ f(\bigcap_i A_i) &\subset \bigcap_i f(A_i) \\ f^{-1}(\bigcup_i A_i) &= \bigcup_i f^{-1}(A_i) \\ f(\bigcap_i A_i) &= \bigcap_i f^{-1}(A_i) \end{aligned}$$

Give an example which shows that the second inclusion need not be an equality.

A notation which is often used to define sets is

$$Y = \{f(x) : x \in X\}$$

(where f is some function whose domain is the set X) which is to be understood as an abbreviation for

$$Y = \{y : y = f(x) \text{ for some } x \in X\}$$

so that for any y

$$y \in Y \Longleftrightarrow y = f(x) \text{ for some } x \in X.$$

A.4 Composition and Restriction

Given functions $f : X \to Y$ and $g : Y \to Z$ the **composition** of f and g is denoted $g \circ f$ (read " *g after f*") and defined by $g \circ f : X \to Z$ with

$$(g \circ f)(x) = g(f(x))$$

for $x \in X$. The operation of composition is associative:

$$(h \circ g) \circ f = h \circ (g \circ f).$$

Suppose we are given a function $f : X \to Y$ and a subset $X_0 \subset X$. The **restriction of f to X_0**, denoted $f|X_0$, is defined by

$$Dom(f|X_0) = X_0,$$

$$(f|X_0)(x) = f(x) \quad \text{for all } x \in X_0.$$

For example, if $f : \mathbf{R} \to \mathbf{R}$ is a function whose graph is the straight line given by $f(x) = 2x$, and if $[0,1]$ denotes the unit interval, then $f|[0,1]$, the restriction of f to $[0,1]$, is a function whose graph is the closed line segment from the $(0,0)$ to $(1,2)$.

The opposite of *restricting* a function to a smaller domain is *extending* a function to a larger domain. Suppose $g : X \to Y$ is a function and $X \subset Z$. Then any function $h : Z \to Y$ is called an **extension of g to Z** if $h|X = g$, i.e., if

$$h(x) = g(x) \quad \text{for all } x \in X.$$

Thus, for example, if g is the function defined earlier by $g : X \to \mathbf{R} : x \mapsto \frac{1}{1-x}$ with domain $X = \{x \in \mathbf{R} : x \neq 1\}$, then g has an extension $\tilde{g}$ defined by

$$\tilde{g}(x) = \begin{cases} \frac{1}{1-x} & \text{if } x \neq 1 \\ 0 & \text{if } x = 1. \end{cases}$$

The reader may recall from a calculus course that the function g described above is *continuous* on its domain X, but *has no continuous extension* to $\mathbf{R}$. In particular, $\tilde{g} : \mathbf{R} \to \mathbf{R}$ is not continuous.

Problem A.4.1 Suppose Y and $X_0, X_1, \ldots, X_i, \ldots$ are sets such that $X_0 \subset X_1 \subset \ldots \subset X_i \subset \ldots$ and for each i, $f_i : X_i \to Y$ is a function such that for all $j < i$,

$$f_i|X_j = f_j$$

i.e., for each $j < i$, f_j is the restriction of f_i to X_j. Let X be the increasing union of the X_i. Show that there is a *unique* function $f : X \to Y$ which extends each f_i, $(i = 0, 1, \ldots)$ to X.

A.5 Identity, One-one, and Onto Functions

A function whose domain and codomain are equal and which returns its argument unchanged is called an *identity function*; more precisely the function

$$I_Y : Y \to Y$$

defined by

$$I_Y(y) = y$$

for $y \in Y$ is called the **identity function** of Y. It satisfies the identities

$$I_Y \circ f = f$$

for $f : X \to Y$ and

$$g \circ I_Y = g$$

for $g : Y \to Z$.

A function $f : X \to Y$ is called **one-one** if its output determines its input uniquely; [3] i.e., if for all $x_1, x_2 \in X$ we have $x_1 = x_2$ whenever $f(x_1) = f(x_2)$. A function $f : X \to Y$ is called **onto** if every point of Y is the output of some input; i.e., if for every $y \in Y$ there is an $x \in X$ such that $f(x) = y$. A function is called **one-one and onto**, or a **bijection**, if it is both one-one and onto.

Think of the equation $y = f(x)$ as a problem to be solved for x. Then:

$$\text{the function } f \text{ is } \left\{ \begin{array}{c} \text{one-one} \\ \text{onto} \\ \text{one-one and onto} \end{array} \right\}$$

if and only if for *every* $y \in Y$ the equation

$$y = f(x) \text{ has } \left\{ \begin{array}{c} \text{at most} \\ \text{at least} \\ \text{exactly} \end{array} \right\} \text{ one solution } x \in X.$$

The function

$$\mathbf{R} \to \mathbf{R} : x \mapsto x^3$$

is both one-one and onto since the equation

$$y = x^3$$

possesses the unique solution $x = y^{\frac{1}{3}} \in \mathbf{R}$ for every $y \in \mathbf{R}$. In contrast, the function

$$\mathbf{R} \to \mathbf{R} : x \to x^2$$

[3] Of course, for *any* function its *input* determines its *output* uniquely; that is the definition of a function.

is not one-one since the equation

$$4 = x^2$$

has *two* distinct solutions, namely $x = 2$ and $x = -2$. It is also not onto since $-4 \in \mathbf{R}$ but the equation

$$-4 = x^2$$

has *no* solution $x \in \mathbf{R}$.

The equation $-4 = x^2$ does have a complex solution $x = 2i \in \mathbf{C}$ but that is not relevant to the question of whether the function $\mathbf{R} \to \mathbf{R} : x \mapsto x^2$ is onto. The functions $\mathbf{C} \to \mathbf{C} : x \mapsto x^2$ and $\mathbf{R} \to \mathbf{R} : x \mapsto x^2$ are different: they have a different domain and codomain. The function $\mathbf{C} \to \mathbf{C} : x \mapsto x^2$ *is* onto.

The concepts of one-one and onto can be formulated in other ways. For instance, a function $f : Y \to Z$ is called **left cancellable** if for all sets X and all pairs of functions $g_1 : X \to Y$, $g_2 : X \to Y$,

$$\text{whenever } f \circ g_1 = f \circ g_2, \text{ we have } g_1 = g_2.$$

Likewise, a function $f : Y \to Z$ is **right cancellable** if for all sets W and all pairs of functions $h_1 : Z \to W$, $h_2 : Z \to W$,

$$\text{whenever } h_1 \circ f = h_2 \circ f, \text{ we have } h_1 = h_2.$$

The next proposition demonstrates the connection between these concepts; we leave its proof to the reader.

Proposition A.5.1 *Suppose $f : X \to Y$ is a function.*

1. *The following are equivalent:*
   - (a) *f is one-one.*
   - (b) *For all $y \in Ran(f)$, the set $f^{-1}(\{y\})$ has exactly one element.*
   - (c) *For all subsets $X_0 \subset X$, $f^{-1}(f(X_0)) = X_0$.*
   - (d) *f is left-cancellable.*
2. *The following are equivalent:*

(a) *f is onto.*

(b) $f(X) = Y$.

(c) *The range and codomain of f are equal.*

(d) *For all subsets $Y_0 \subset Y$, $f(f^{-1}(Y_0)) = Y_0$.*

(e) *f is right-cancellable.*

A.6 Cardinality

Two sets X and Y are said to have the same **size**, or **cardinality**, if there is a one-one, onto function $f : X \to Y$. This notion is familiar when X and Y are finite sets: Given that $X = \{x_1, x_2, \ldots, x_k\}$ and $Y = \{y_1, y_2, \ldots, y_m\}$, then X and Y have the same cardinality if $k = m$, i.e., if they have the same number of elements.

On the other hand, two infinite sets which may at first appear to have different sizes may in fact have the same cardinality. For instance, if $E = \{n : n \text{ is an even natural number}\}$, then E and $\mathbf{N}$ have the same cardinality since the function $g : \mathbf{N} \to E$ defined by

$$g(n) = 2n$$

is one-one and onto.

Formally speaking, a set X is said to be **finite** if there is a natural number n such that X and $\{0, 1, \ldots, n\}$ have the same cardinality. X is **infinite** if X is not finite. X is called **countable** or **denumerable** if X and $\mathbf{N}$ have the same cardinality.

Is every set finite or countable? Or is there some enormous (*uncountable*) infinite set X for which X and $\mathbf{N}$ do not have the same cardinality? This question plagued Georg Cantor at the end of the 19th century; many mathematicians and philosophers of the time found this, and related questions about infinite sets, to be outside the proper domain of mathematics. Even in present-day universities, some students, when confronted with this question, feel somewhat disturbed since, after all, how could one *infinite* set be bigger than another?

Despite the controversy surrounding this and related questions, and despite the apparent unlikeliness of the result, Cantor was able to show that uncountable sets exist. We give a proof of this fact below:

For any set X let $\mathcal{P}(X)$ denote the set of all subsets of X:

$$S \in \mathcal{P}(X) \text{ iff } S \subset X.$$

The set $\mathcal{P}(X)$ is called the **power set** of X. Note that if X is a finite set having n elements, then $\mathcal{P}(X)$ is a finite set having 2^n elements.

Theorem A.6.1 (Cantor's Theorem) *There is no onto function*

$$f : X \to \mathcal{P}(X).$$

Proof: Suppose such a function f exists. We will derive a contradiction. Define a subset $S \subset X$ by

$$S = \{x \in X : x \notin f(x)\}.$$

Since f is onto and $S \in \mathcal{P}(X)$ there must be an element $y \in X$ with $S = f(y)$. Now either $y \in S$ or $y \notin S$. If $y \in S$, then $y \in f(y)$ (as $S = f(y)$) so $y \notin S$. If $y \notin S$, then $y \notin f(y)$ so $y \in S$. Either way we get a contradiction, so no such function f exists. **End of Proof.**

The method used in the preceding proof, called Cantor's diagonal method, resembles several other arguments in this book. (See the discussion following the Halting Problem in Section 4.11). Using the diagonal method, Cantor was also able to show that the set $\mathbf{R}$ of real numbers is uncountable. In fact, he showed that $\mathbf{R}$ and $\mathcal{P}(\mathbf{N})$ have exactly the same cardinality!

Problem A.6.2 Is the set $\mathbf{Q}$ of rationals countable or uncountable?

One question which Cantor was unable to answer is whether there is an infinite set X whose size is strictly between that of $\mathbf{N}$ and that of $\mathbf{R}$; more technically, is there a set X together with one-one functions $f : \mathbf{N} \to X$ and $g : X \to \mathbf{R}$ such that $\mathbf{N}$ and X do *not* have the same cardinality and X and $\mathbf{R}$ do *not* have the same cardinality? The assertion that no such set exists – or, stated more positively, that every infinite set of reals either has the same cardinality as $\mathbf{N}$ or as $\mathbf{R}$ – is known as the **Continuum Hypothesis**. It was shown nearly 80 years after Cantor's time that the Continuum Hypothesis is neither provable nor disprovable from any known (reasonable) set theory (i.e., it's an independent sentence for ZFC and many other set theories).

A.7 Inverses

Let $f : X \to Y$. A **left inverse** to f is a function $g : Y \to X$ such that

$$g \circ f = I_X.$$

Proposition A.7.1 *A function $f : X \to Y$ is one-one if and only if there is a left inverse $g : Y \to X$ to f. If f is one-one but not onto, the left inverse is not unique.*

Proof: If $g : Y \to X$ is a right inverse to f the problem $y = f(x)$ has at most one solution for if $y = f(x_1) = f(x_2)$ then $g(y) = g(f(x_1)) = g(f(x_2))$ whence $x_1 = x_2$ since $g(f(x)) = I_X(x) = x$. Conversely, if the problem $y = f(x)$ has at most one solution, then any function $g : Y \to X$ which assigns to $y \in Y$ a solution x of $y = f(x)$ (when there is one) is a left inverse to f. (It does not matter what value g assigns to y when there is no solution x.) **End of Proof.**

Let $f : X \to Y$. A **right inverse** to f is a function $g : Y \to X$ such that

$$f \circ g = I_Y.$$

Proposition A.7.2 *A function $f : X \to Y$ is onto if and only if there is a right inverse $g : Y \to X$ to f. If f is onto but not one-one the right inverse is not unique.*

Proof: If $g : Y \to X$ is a right inverse to $f : X \to Y$ then $x = g(y)$ is a solution to $y = f(x)$ since $f(g(y)) = I_Y(y) = y$. The converse assertion that there is a right inverse $g : Y \to X$ to any onto function $f : X \to Y$ may not seem obvious to someone who thinks of a function as a computer program: even though the problem $y = f(x)$ has a solution x, it may have many, and how is a computer program to choose? (If $X \subset \mathbf{N}$ one could define $g(y)$ to be the smallest $x \in X$ which solves $y = f(x)$ but this will not work if $X = \mathbf{Z}$ for in this case there may not be a smallest x.) In fact, this converse assertion is generally taken as an axiom: the so called *axiom of choice*, and cannot be proved from the other axioms of mathematics. **End of Proof.**

Let $f : X \to Y$. A **two-sided inverse** to f is a function which is both a left inverse to f and a right inverse to f:

$$g \circ f = I_X, \quad f \circ g = I_Y.$$

The word *inverse* unmodified means two-sided inverse.

The following easy proposition explains why the two-sided inverse is necessarily unique.

Proposition A.7.3 *If $f : X \to Y$ has both a left inverse and a right inverse, then it has a two-sided inverse $f^{-1} : Y \to X$, and f^{-1} is the only left inverse of f and the only right inverse of f*

Proof: Let $g : Y \to X$ be a left inverse to f and $h : Y \to X$ be a right inverse. Then

$$g \circ f = I_X$$

and

$$f \circ h = I_Y.$$

Compose on the right by h in the first equation to obtain

$$g \circ f \circ h = I_X \circ h$$

and use the second to obtain

$$I_Y \circ h = I_X \circ h.$$

Now composing a function with the identity (on either side) does not change the function so we have

$$g = h$$

i.e., $g\ (= h)$ is a two-sided identity. Now if g_1 is another left inverse to f then this same argument shows that

$$g_1 = h$$

(i.e., $g_1 = g$). Similarly h is the only right inverse to f. **End of Proof.**

Proposition A.7.4 *The function $f : X \to Y$ is one-one and onto if and only if there is a (necessarily unique) two-sided inverse $f^{-1} : Y \to X$ to f. This inverse function $f^{-1} : Y \to X$ is characterized by the equivalence*

$$y = f(x) \Longleftrightarrow x = f^{-1}(y)$$

for $x \in X$ and $y \in Y$. (The symbol $\Longleftrightarrow$ means if and only if.)

Proposition A.7.5 **(1)** *The identity transformation $I_X : X \to X$ is one-one and onto and is its own inverse:*

$$I_X^{-1} = I_X$$

(2) *If $f : X \to Y$ is one-one and onto, then so is its inverse $f^{-1} : Y \to X$, and the inverse of f^{-1} is given by*

$$(f^{-1})^{-1} = f.$$

(3) *If the function $f : X \to Y$ is one-one and onto and the function $g : Y \to Z$ is one-one and onto, then the composite $g \circ f : X \to Z$ is one-one and onto and its inverse $(g \circ f)^{-1} : Z \to Y$ is given by*

$$(g \circ f)^{-1} = f^{-1} \circ g^{-1}$$

Proof of (1): We have $I_X \circ I_X = I_X$ since $(I_X \circ I_X)(x) = I_X(I_X(x)) = I_X(x)$ for all $x \in X$. **End of Proof.**

Proof of (2): The same formulas

$$f^{-1} \circ f = I_X, \quad f \circ f^{-1} = I_Y$$

which say that f^{-1} is the inverse of f also say that f is the inverse of f^{-1}. **End of Proof.**

Proof of (3):

$$(g \circ f) \circ (f^{-1} \circ g^{-1}) = g \circ I_Y \circ g^{-1} = g \circ g^{-1} = I_Z$$

and

$$(f^{-1} \circ g^{-1}) \circ (g \circ f) = f^{-1} \circ I_Y = f^{-1} \circ f = I_X.$$

End of Proof.

Example A.7.6 Let $\mathbf{R}^+$ denote the set of non-negative real numbers:

$$\mathbf{R}^+ = \{x \in \mathbf{R} : x \geq 0\}$$

and consider the four functions:

$$\begin{array}{ll} f_1 : \mathbf{R} \to \mathbf{R} & f_1(x) = x^2 \text{ for } x \in \mathbf{R}; \\ f_2 : \mathbf{R} \to \mathbf{R}^+ & f_2(x) = x^2 \text{ for } x \in \mathbf{R}; \\ f_3 : \mathbf{R}^+ \to \mathbf{R} & f_3(x) = x^2 \text{ for } x \in \mathbf{R}^+; \\ f_4 : \mathbf{R}^+ \to \mathbf{R}^+ & f_4(x) = x^2 \text{ for } x \in \mathbf{R}^+. \end{array}$$

Then

1 f_1 is neither one-one nor onto. It is not one-one since $f_1(3) = f_1(-3) = 9$ but $3 \neq -3$. It is not onto since $f_1(x) \neq -4$ for all $x \in \mathbf{R}$.

2 f_2 is onto but not one-one. It is not one-one for the same reason that f_1 is not. The reason that f_1 is not onto does not apply to f_2 since the negative numbers are not in the codomain of f_2. The function

$$g_2 : \mathbf{R}^+ \to \mathbf{R}$$

given by

$$g_2(y) = \sqrt{y}$$

is a right inverse to $f_2 : \mathbf{R} \to \mathbf{R}^+$ since $f_2(g_2(y)) = y$ for $y \geq 0$. It is not a left inverse for f_2 since $g_2(f_2(x)) = |x|$ for $x \in \mathbf{R}$ and $x \neq |x|$ if $x < 0$. The function

$$\tilde{g}_2 : \mathbf{R}^+ \to \mathbf{R}$$

given by

$$\tilde{g}_2(y) = -\sqrt{y}$$

is a different right inverse to f_3.

3 f_3 is one-one but not onto. It is not onto for the same reason that f_1 is not. The reason that f_1 is not one-one does not apply to f_3 since the negative numbers are not in the domain of f_3. The function

$$g_3 : \mathbf{R} \to \mathbf{R}^+$$

defined by

$$g_3(y) = \begin{cases} \sqrt{y} \text{ for } y \geq 0; \\ 7 \text{ for } y < 0 \end{cases}$$

is a left inverse to the function f_3; namely, $g_3(f_3(x)) = x$ for $x \in \mathbf{R}^+$. It is not a right inverse for $f_3(g_3(y)) = 49 \neq y$ for $y < 0$. (Replacing 7 by some other constant gives a different left inverse to f_3.)

4 f_4 is one-one and onto. The function

$$g_4 : \mathbf{R}^+ \to \mathbf{R}^+$$

given by

$$g_4(y) = \sqrt{y}$$

is the (two-sided) inverse to the function f_4

Example A.7.7 Let $Y = \{y \in \mathbf{R} : -1 \leq y \leq 1\}$ and define $f : \mathbf{R} \to Y$ and $g : Y \to \mathbf{R}$ by $f(\theta) = \sin(\theta)$ and $g(y) = \sin^{-1}(y)$. Then $f(g(y)) = y$ for $y \in Y$. Thus f is a left inverse for g, g is a right inverse for f, f is onto, and g is one-one. However f is not one-one (since $f(2\pi) = f(0)$ although $2\pi \neq 0$) and g is not onto (since $g(y) \neq 2$ for all $y \in Y$).

Problem A.7.8 What is the value of $g(f(\theta)) = \sin^{-1}(\sin(\theta))$ for $\theta \in \mathbf{R}$?

The example and exercise point up the fact that it is very important to specify the domain and codomain when defining a function. In order to define inverses for common functions one often restricts their domains.

Here are some common functions and their inverses. Note how carefully the source and codomain are specified.

1. The linear function

$$\mathbf{R} \to \mathbf{R} : x \mapsto ax + b$$

is one-one and onto if $a \neq 0$; its inverse is the function

$$\mathbf{R} \to \mathbf{R} : y \mapsto (y - b)/a.$$

For $x, y \in \mathbf{R}$:

$$y = ax + b \iff x = (y - b)/a.$$

2. The cube function

$$\mathbf{R} \to \mathbf{R} : x \mapsto x^3$$

is one-one and onto; its inverse is the cube root function

$$\mathbf{R} \to \mathbf{R} : y \mapsto y^{\frac{1}{3}}.$$

For $x, y \in \mathbf{R}$:

$$y = x^3 \iff x = y^{\frac{1}{3}}$$

3. The exponential function

$$\mathbf{R} \to \mathbf{R}^+ \setminus \{0\} : x \mapsto e^x$$

is one-one and onto; its inverse is the natural logarithm

$$\mathbf{R}^+ \setminus \{0\} \to \mathbf{R} : y \mapsto \ln(y).$$

For $x, y \in \mathbf{R}$ with $y > 0$:

$$y = e^x \iff x = \ln(y).$$

4. The restricted sine function

$$\sin : \{\theta \in \mathbf{R} : -\frac{\pi}{2} \leq \theta \leq \frac{\pi}{2}\} \to \{y \in \mathbf{R} : -1 \leq y \leq 1\}$$

is one-one and onto; its inverse is the inverse sine function

$$\sin^{-1} : \{y \in \mathbf{R} : -1 \leq y \leq 1\} \to \{\theta \in \mathbf{R} : -\frac{\pi}{2} \leq \theta \leq \frac{\pi}{2}\}.$$

For $-\frac{\pi}{2} \leq \theta \leq \frac{\pi}{2}\}$ and $-1 \leq y \leq 1$:

$$y = \sin(\theta) \iff \theta = \sin^{-1}(y).$$

The inverse sine function is sometimes called the *arcsine* and denoted arcsin.

A.8 Cartesian Product

Let X and Y be sets. The **Cartesian product** of X and Y is the set of all ordered pairs (x, y) with $x \in X$ and $y \in Y$:

$$X \times Y = \{(x, y) : x \in X,\ y \in Y\}.$$

The Cartesian product is also called the **direct product.**

In certain contexts the word *operation* is often used in place of the word *function*; thus a **unary operation** on a set X is a function with domain and codomain X and a **binary operation** on X is a function with domain $X \times X$ and codomain X.

An example of a unary operation is the operation of *negation* of a real number:

$$\mathbf{R} \to \mathbf{R} : x \mapsto -x$$

and an example of a binary operation is the operation of *addition* of real numbers:

$$\mathbf{R} \times \mathbf{R} \to \mathbf{R} : (x, y) \mapsto x + y.$$

Sometimes the value of a function for given inputs is denoted in other ways. For example, we write $x + y$ rather than $+(x, y)$ Here, parentheses play the crucial role of indicating the order in which the operations are performed $(x-(y+z) \neq (x-y)+z)$ and when parentheses are omitted this order is determined according to some *convention* (e.g. $x - y + z$ means $(x - y) + z$ and not $x - (y + z)$).

The notation where the name of a binary function is placed between (rather than in front of) the arguments is called **infix** notation. Occasionally, the name of the function is placed after the operation – one writes $(x, y)f$ rather than $f(x, y)$ – this is called **postfix** notation. The notation $f(x, y)$ is thus called **prefix** notation. It is possible to omit parentheses unambiguously when using postfix (or prefix notation) and some calculators (e.g., those made by Hewlett-Packard) and programming languages (e.g., APL) do this. (Thus $x-y+z$ is denoted $xy-z+$ in postfix notation.) [4]

[4]The observation that parentheses are not needed with prefix (or postfix) notation is due to a Pole named Lukasiewicz so parentheses-free notation is sometimes called Polish (or reverse Polish) notation.

A.9 Graphing Functions

For any function

$$f : X \to Y$$

we may define its **graph** to be the set

$$G(f) = \{(x, y) \in X \times Y : y = f(x)\}$$

of all pairs (x, y) such that $y = f(x)$.

(1) A subset $G \subset X \times Y$ is the graph of some function f if and only if for every $x \in X$ there is a unique $y \in Y$ (namely $y = f(x)$ with $(x, y) \in G(f)$.

(2) The function f is one-one if and only if for every $y \in Y$ there is at most one $x \in X$ with $(x, y) \in G(f)$.

(3) The function f is onto if and only if for every $y \in Y$ there is at least one $x \in X$ with $(x, y) \in G(f)$.

(4) The function f is one-one and onto if and only if for every $y \in Y$ there is exactly one $x \in X$ with $(x, y) \in G(f)$.

Suppose that both sets X and Y are **intervals** in the set $\mathbf{R}$ of real numbers. For example,

$$X = \{x \in \mathbf{R} : a_1 \leq x \leq a_2\}, \quad Y = \{y \in \mathbf{R} : b_1 \leq x \leq b_2\}.$$

We may plot points in the usual fashion with the set X represented by an interval on the horizontal axis and the set Y represented by an interval on the vertical axis. The set $X \times Y$ will be a rectangle and the graph $G(f)$ of f will be a subset of the rectangle $X \times Y$. Then

(1) A subset $G \subset X \times Y$ is the graph of some function f if and only if every vertical line through X intersects G in exactly one point.

(2) The function f is one-one if and only if every horizontal line through Y intersects $G(f)$ in at most one point.

(3) The function f is onto if and only if every horizontal line through Y intersects $G(f)$ in at least one point.

(4) The function f is one-one if and only if every horizontal line through Y intersects $G(f)$ in exactly one point.

Problem A.9.1 For each of the following sets G specify whether or not it is the graph of a function $f : X \to Y$.

1. $G = \{(x,y) \in \mathbf{R}^2 : x^2 + y^2 = 1\}$,
   $X = \{x \in \mathbf{R} : -1 \le x \le 1\}$,
   $Y = \{y \in \mathbf{R} : -1 \le y \le 1\}$.

2. $G = \{(x,y) \in \mathbf{R}^2 : x^2 + y^2 = 1,\ y \ge 0\}$,
   $X = \{x \in \mathbf{R} : -1 \le x \le 1\}$,
   $Y = \{y \in \mathbf{R} : -1 \le y \le 1\}$.

3. $G = \{(x,y) \in \mathbf{R}^2 : x^2 + y^2 = 1,\ x \ge 0\}$,
   $X = \{x \in \mathbf{R} : -1 \le x \le 1\}$,
   $Y = \{y \in \mathbf{R} : -1 \le y \le 1\}$.

4. $G = \{(x,y) \in \mathbf{R}^2 : x^2 + y^2 = 1,\ y \ge 0\}$,
   $X = \{x \in \mathbf{R} : -1 \le x \le 1\}$,
   $Y = \{y \in \mathbf{R} : 0 \le y \le 1\}$.

5. $G = \{(x,y) \in \mathbf{R}^2 : y = x^3 - x,\ \ -1 \le x \le 1\}$,
   $X = \{x \in \mathbf{R} : -1 \le x \le 1\}$,
   $Y = \{y \in \mathbf{R} : -9 \le y \le 9\}$.

6. $G = \{(x,y) \in \mathbf{R}^2 : y = x^3 + x,\ \ -1 \le x \le 1\}$,
   $X = \{x \in \mathbf{R} : -1 \le x \le 1\}$,
   $Y = \{y \in \mathbf{R} : -9 \le y \le 9\}$.

7. $G = \{(x,y) \in \mathbf{R}^2 : x = y^3 - y,\ \ -1 \le y \le 1\}$,
   $X = \{x \in \mathbf{R} : -9 \le x \le 9\}$,
   $Y = \{y \in \mathbf{R} : -1 \le y \le 1\}$.

8. $G = \{(x,y) \in \mathbf{R}^2 : x = y^3 + y \quad -1 \le y \le 1\}$,
   $X = \{x \in \mathbf{R} : -9 \le x \le 9\}$,
   $Y = \{y \in \mathbf{R} : -1 \le y \le 1\}$.

Problem A.9.2 Graph each of the following functions $f : X \to Y$ and specify whether or not it is one-one or onto or both. If the function is not one-one, draw a horizontal line which intersects its graph at least twice. If the function is not onto, draw a horizontal line which does not intersect its graph.

1. $X = \{x \in \mathbf{R} : -1 \le x \le 1\}$,
   $Y = \{y \in \mathbf{R} : -1 \le y \le 1\}$,
   $f(x) = \sqrt{1 - x^2}$.

2. $X = \{x \in \mathbf{R} : -1 \le x \le 1\}$,
   $Y = \{y \in \mathbf{R} : 0 \le y \le 1\}$,
   $f(x) = \sqrt{1 - x^2}$.

3. $X = \{x \in \mathbf{R} : 0 \le x \le 1\}$,
   $Y = \{y \in \mathbf{R} : -1 \le y \le 1\}$,
   $f(x) = \sqrt{1 - x^2}$.

4. $X = \{x \in \mathbf{R} : 0 \le x \le 1\}$,
   $Y = \{y \in \mathbf{R} : 0 \le y \le 1\}$,
   $f(x) = \sqrt{1 - x^2}$.

5. $X = \{x \in \mathbf{R} : -1 \le x \le 1\}$,
   $Y = \{y \in \mathbf{R} : -2 \le y \le 2\}$,
   $f(x) = x^3 - 1$.

6. $X = \{x \in \mathbf{R} : -1 \le x \le 1\}$,
   $Y = \{y \in \mathbf{R} : -2 \le y \le 1\}$,
   $f(x) = x^3 - 1$.

7. $X = \{x \in \mathbf{R} : -1 \le x \le 1\}$,
   $Y = \{y \in \mathbf{R} : -2 \le y \le 2\}$,
   $f(x) = x^3 + 1$.

8. $X = \{x \in \mathbf{R} : -1 \le x \le 1\}$,
   $Y = \{y \in \mathbf{R} : 0 \le y \le 2\}$,
   $f(x) = x^3 + 1$.

A.10 Finite Sequences

A **sequence of length** n is a list

$$x = (x_1, x_2, \ldots, x_n)$$

of objects; x_i is called the i-th element of the finite sequence x. Two sequences

$$f = (x_1, x_2, \ldots, x_n)$$

and

$$g = (y_1, y_2, \ldots, y_m)$$

are **equal** if $m = n$ and $x_i = y_i$ for $i = 1, 2, \ldots, n$.

It is important to remember that for sequences the order is important. Thus the sequences $x = (4, 7, 9)$ and $y = (7, 4, 9)$ are distinct (for $x_1 = 4 \neq 7 = y_1$) while the sets $\{4, 7, 9\}$ and $\{7, 4, 9\}$ are the same. Similarly, for sequences repetition matters, whereas this is not so for sets. Thus

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

but

$$(1, 2, 3, 1, 2) \neq (1, 2, 3)$$

since the two sequences have different length.

The set of all finite sequences of elements of X of length n is denoted by X^n so that

$$X^n = \{(x_1, x_2, \ldots, x_n) : x_1, x_2, \ldots, x_n \in X\}.$$

It is also customary not to distinguish between a sequence of length one and its sole element: $(x) = x$. In other words, we identify the set X^1 of sequences of length one of elements of X with the set X itself:

$$X^1 = X.$$

A sequence of length n is also called an **n-tuple**. Thus a 2-tuple is a **pair**, a 3-tuple is a **triple**, a 4-tuple is a **quadruple**, etc..

Proposition A.10.1 *Suppose f is an n-tuple of elements of $\{1, 2, \ldots, m\}$, that is,*

$$f : \{1, 2, \ldots, n\} \to \{1, 2, \ldots, m\}.$$

1. *If f is one-one, then $n \leq m$.*
2. *If f is onto, then $n \geq m$.*
3. *If f is one-one and onto, them $n = m$.*

There are m^n functions

$$f : \{1, 2, \ldots, n\} \to \{1, 2, \ldots, m\}$$

from a finite set with n elements to a finite set with m elements: in other words, there are m^n ways to form a sequence of length n (possibly with repetitions) from a set of m objects. For example there are $8 = 2^3$ functions

$$f_j : \{1, 2, 3\} \to \{1, 2\} \text{ for } j = 1, 2, \ldots, 8$$

from a three element set to a two element set. Let's list them and their values in a table:

i	1	2	3
$f_1(i)$	1	1	1
$f_2(i)$	1	1	2
$f_3(i)$	1	2	1
$f_4(i)$	1	2	2
$f_5(i)$	2	1	1
$f_6(i)$	2	1	2
$f_7(i)$	2	2	1
$f_8(i)$	2	2	2

None of these is one-one since $2 < 3$. For example f_4 is not one-one since $f_4(2) = f_4(3) = 2$ but $2 \neq 3$. On the other hand all but f_1 and f_8 are onto. For example f_2 is onto since $f_2(1) = 1$ and $f_2(3) = 2$. There are two right inverses g and h to f_2; one of them is defined by $g(1) = 1$, $g(2) = 3$ and the other by $h(1) = 2$, $h(2) = 3$: $f(g(y)) = f(h(y)) = y$ for $y = 1, 2$. On the other hand, f_1 is not onto since the equation $f_1(x) = 2$ has no solution $x \in \{1, 2, 3\}$.

Problem A.10.2 For each $f : \{1, 2, 3\} \to \{1, 2\}$ which is onto, give all of its right inverses.

Problem A.10.3 Make a table of the $9 = 3^2$ functions

$$f : \{1,2\} \to \{1,2,3\}.$$

For each f say whether it is one-one. If it is give all its left inverses. If it is not, find x_1, x_2 with $f(x_1) = f(x_2)$ but $x_1 \neq x_2$.

A.11 Permutations

Now we deal with the case where $m = n$. Let

$$f : \{1,2,\ldots,n\} \to \{1,2,\ldots,n\}$$

be a function from a finite set with n elements to itself. The function f is called a **permutation** if f is one-one and onto.

Proposition A.11.1 *Suppose f is a function from a finite set with n elements to itself. Then the following conditions are equivalent:*

- *f has a left inverse.*
- *f is one-one.*
- *f has a right inverse.*
- *f is onto.*
- *f is a permutation.*
- *f has a two-sided inverse f^{-1}; i.e., there is a function*

$$f^{-1} : \{1,2,\ldots,n\} \to \{1,2,\ldots,n\}$$

satisfying $f(f^{-1}(y)) = y$ and $f^{-1}(f(x)) = x$ for $x, y = 1,2,\ldots,n$.

Moreover, if f satisfies any of these conditions, f^{-1} is the only left inverse to f and f^{-1} is the only right inverse to f.

Of the n^n functions from $\{1,2,\ldots,n\}$ to itself exactly $n! = n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1$ of these are permutations. (This is the number of ways we can rearrange n things without repetitions.)

Problem A.11.2 There are $27 = 3^3$ functions

$$f : \{1,2,3\} \to \{1,2,3\}$$

and 6 of them are one-one and onto. For each of these 6 give its inverse. Select one of the remaining 21, by specifying the three values $f(1)$, $f(2)$, $f(3)$. Show that this f is not one-one by finding $x_1, x_2 \in \{1,2,3\}$ with $x_1 \neq x_2$ and $f(x_1) = f(x_2)$. Show that this f is not onto, finding $y \in \{1,2,3\}$ such that $f(x) \neq y$ for $x = 1,2,3$.

A.12 Induction

The set of natural numbers

$$\mathbf{N} = \{0,1,2,3,\ldots\}$$

is one of the starting points for mathematics.

An **infinite sequence** is a function whose domain is the set of natural numbers. Infinite sequences are often written with the three dot notation,

$$A = (A_1, A_2, A_3, \ldots).$$

The **induction principle** is a basic property of the natural numbers, and plays a central role in many of the proofs in this course.

INDUCTION PRINCIPLE

To prove that all natural numbers have a given property R:

(Basis step) *Show that 0 has the property R.*

(Successor step) *Show that for every natural number n, if n has property R, then $n+1$ has property R.*

We may write $R(n)$ to mean that n has property R. The induction principle is intuitively plausible because one can prove $R(k)$ for a particular natural number k by first using the basis step to prove $R(0)$, then using the successor step to prove $R(1)$, then using the successor step again to prove $R(2)$, and repeating the process k times to form a proof of $R(k)$.

Infinite sequences are often defined by induction on the natural numbers. Such definitions are justified by the following principle.

INDUCTIVE DEFINITION PRINCIPLE

A sequence $A = (A_1, A_2, A_3, \ldots)$. can be defined uniquely by giving:

(Basis rule) *A value A_0;*

(Successor rule) *For each natural number n, a rule for computing a value A_{n+1} given n and values $A_0, \ldots, A_n$.*

Appendix B

Listings

B.1 Simple GNUMBER Programs

In this section the simple register machine programs on the distribution diskette are reproduced.

ADD.GN

```
0: Z   3          -ADD-  let count = 0
1: J   3   2   5  -LOOP-  if count = b, jump to DONE
2: S   1          let a = a+1
3: S   3          let count = count+1
4: J   1   1   1  jump to LOOP
5: H              -DONE-
```

MULT.GN

```
0: Z   3          -MULT-  let accum = 0
1: Z   4          let i = 0
2: J   2   4  10  -LOOP-  if b = i, jump to DONE
3: S   4          let i = i + 1
4: Z   5          -ADD(accum,a)-  let count = 0
```

```
 5: J   5   1   9  -ALOOP-  if count = a, jump to ADONE
 6: S   3          let accum = accum+1
 7: S   5          let count = count+1
 8: J   1   1   5  jump to ALOOP
 9: J   1   1   2  -ADONE-  jump to LOOP
10: T   3   1      -DONE-  let a = accum
11: H
```

PRED.GN

```
 0: Z   2          -PRED-  let prev = 0
 1: J   1   2   8  if a = prev, jump to DONE
 2: Z   3
 3: S   3          let next = 1
 4: J   1   3   8  -LOOP- if a = next, jump to done
 5: S   3          let next = next + 1
 6: S   2          let prev = prev + 1
 7: J   1   1   4  jump to LOOP
 8: T   2   1      -DONE- let a = prev
 9: H
```

DOTMINUS.GN

```
 0: Z   3          -DOTMINUS-   let count = 0
 1: J   3   2  13  -LOOP-  if count = b, jump to DONE
 2: Z   5          -PRED(a)-  let prev = 0
 3: J   1   5  10  if a = prev, jump to PDONE
 4: Z   4
 5: S   4          let next = 1
 6: J   1   4  10  -PLOOP- if a = next, jump to PDONE
 7: S   4          let next = next+1
 8: S   5          let prev = prev+1
 9: J   1   1   6  jump to PLOOP
```



```
10: T   5   1      -PDONE-  let a = prev
11: S   3          -END PRED-  let count = count+1
12: J   1   1   1  jump to LOOP
13: H              -DONE-
```

DIVREM.GN

```
 0: Z   5          -DIVREM-  let count = 0
 1: J   2   5   1  -HANG- if b = count, jump to HANG
 2: Z   3          let q = 0
 3: Z   4          let r = 0
 4: J   4   2   9  -TEST-  if r = b, jump to INCQ
 5: J   5   1  12  if count = a, jump to DONE
 6: S   4          let r = r+1
 7: S   5          let count = count+1
 8: J   1   1   4  jump to TEST
 9: S   3          -INCQ-  let q = q+1
10: Z   4          let r = 0
11: J   1   1   4  jump to TEST
12: T   3   1      -DONE-  let a = q
13: T   4   2      let b = r
14: H
```

B.2 Advanced RM programs

This section contains pseudocode and register machine listings for the advanced RM programs on the distribution diskette.

FIVE.GN

```
 0: Z  20          -FIVE-  let zero = 0
 1: T  20  21
 2: S  21          let one = 1
```

```
 3: T  21  22
 4: S  22          let two = 2
 5: T  22  23
 6: S  23          let three = 3
 7: T  23  24
 8: S  24          let four = 4
 9: T  24  25
10: S  25          let five = 5
11: H
```

TERMS.GN

```
program TERMS(a)
 input:  a = x, the G.N. of a sequence S
 output:  a = number of terms in S

   let count = 0
   let b = a
   do until not b = a
     let c = a[count]
     let b[count] = c
     let count = count + 1
   loop
   let a = count
end of program TERMS
```

```
 0: Z   4          -TERMS-  let count = 0
 1: T   1   2      let b = a
 2: E   1   4   3  -LOOP-  let c = a[count]
 3: P   3   4   2  let b[count] = c
 4: J   1   2   6  if a = b, jump to NEXT
 5: J   1   1   8  jump to DONE
 6: S   4          -NEXT-  let count = count + 1
 7: J   1   1   2  jump to LOOP
 8: T   4   1      -DONE- let a = count
```



```
 9: H
```

JOIN.GN

```
program JOIN(p,q,psize,qsize)
input:  p = a G.N. of a program P
        q = a G.N. of a program Q
        psize = number of instructions of P
        qsize = the number of instructions of Q
        zero = 0,..., five = J = 5
output:  a = the G.N. of the join PQ

  let ans = p
  let pos = psize
  let count = 0
  do until qsize = count
    let quad = q[count]
    let op = quad[zero]
    if op = J then
      let quad[three] = quad[three] + psize
    let ans[pos] = quad
    let pos = pos + 1
    let count = count + 1
  loop
  let  a = ans
end of program JOIN
```

```
 0: T   1   5      -JOIN-  let ans = a
 1: T   2   6      let q = b
 2: T   3   7      let psize = c
 3: T   4   8      let qsize = d
 4: T   3   9      let pos = psize
 5: Z  10          let count = 0
```

```
 6: J   8  10  23  -MAIN- if qsize = count, jump to DONE
 7: E   6  10  11  let quad = q[count]
 8: E  11  20   1  let op = quad[zero]
 9: J   1  25  14  if op = 5, jump to SETJUMP
10: P  11   9   5  -CONTINUE-  let ans[pos] = quad
11: S   9          let pos = pos + 1
12: S  10          let count = count + 1
13: J   1   1   6  jump to MAIN
14: E  11  23   1  -SETJUMP-  let a = quad[three]
15: T   7   2      let b = psize
16: Z   3          let c = 0
17: J   2   3  21  -LOOP- let a = a + psize, jump to AFTER
18: S   1          let a = a + 1
19: S   3          let c = c + 1
20: J   1   1  17  jump to LOOP
21: P   1  23  11  -AFTER-  let quad[three] = a
22: J   1   1  10  jump to CONTINUE
23: T   5   1      -DONE-  let a = ans
24: H
```

B.3 Pseudocode for PARAM

The PARAM program is an example of how the simpler programs FIVE, TERMS, and JOIN can be combined to form programs which build Gödel numbers of new programs from Gödel numbers of old programs.

```
program PARAM(a,b)
 input:  a = x, the G.N in standard form of an
         ARM program P which neatly computes
         a function f(.,.) of two variables,
         b = y, which goes in the first place of f
 output:  a = the G.N. of an ARM program Q
          which neatly computes the function
          g(.) = f(y,.).
```



```
  FIVE
  let p = a
  let n = b
  let count = 0
  let const = 0
  let instr = G.N. of (Z,2)
  let const[0] = instr
  let instr = G.N. of (S,2)
  do until n = count
    let count = count + 1
    let const[count] = instr
  loop
  let d = Terms(p)
  let a = const
  let b = p
  let c = n + 1
  JOIN(a,b,c,d)
end of program PARAM
```

B.4 PARAM.GN listing

```
 0: Z  20          -PARAM- -FIVE-  let zero = 0
 1: T  20  21
 2: S  21          let one = 1
 3: T  21  22
 4: S  22          let two = 2
 5: T  22  23
 6: S  23          let three = 3
 7: T  23  24
 8: S  24          let four = 4
 9: T  24  25
10: S  25          let five = 5
11: T   1   5      -end of FIVE-  let p = a, G.N. of P
12: T   2   6      let n = second input b
```

```
13: Z   7              let count = 0
14: Z   8              let const = 0
15: Z   9              let instr = 0
16: P  22  20   9      let instr[zero] = Z
17: P  22  21   9      this makes instr = G.N. of (Z,2)
18: P   9  20   8      let const[zero] = instr
19: P  23  20   9      this makes instr = G.N. of (S,2)
20: J   6   7  24      -LOOP-  if n = count, jump to TERMS
21: S   7              let count = count + 1
22: P   9   7   8      let const[count] = instr
23: J   1   1  20      jump to LOOP. const = GN of (Z,2), n (S,2)'s
24: Z   4              -TERMS-  let d = 0
25: T   1   2          let b = a
26: E   1   4   3      -TLOOP-  let c = a[d]
27: P   3   4   2      let d[b] = c
28: J   1   2  31      if a = b, jump to NEXT
29: T   4   1          let a = d
30: J   1   1  33      jump to TDONE
31: S   4              -NEXT-  let d = d + 1
32: J   1   1  26      jump to TLOOP, a = # terms of P
33: T   1   4          -TDONE-  d = # terms of P
34: T   8   1          let a = const, the GN of (Z,2), n (S,2)'s
35: T   5   2          let b = p, the G.N. of program P
36: T   6   3          let c = n
37: S   3              let c = n + 1 = size of const
38: T   1   5          -JOIN-  let ans = a
39: T   2   6          let q = b
40: T   3   7          let psize = c
41: T   4   8          let qsize = d
42: T   3   9          let pos = psize
43: Z  10              let count = 0
44: J   8  10  61      -MAIN- if qsize = count, jump to DONE
45: E   6  10  11      let quad = q[count]
46: E  11  20   1      let op = quad[zero]
47: J   1  25  52      if op = 5, jump to SETJUMP
48: P  11   9   5      -CONTINUE-  let ans[pos] = quad
49: S   9              let pos = pos + 1
```



```
50: S  10              let count = count + 1
51: J   1   1  44      jump to MAIN
52: E  11  23   1      -SETJUMP-  let a = quad[three]
53: T   7   2          let b = psize
54: Z   3              let c = 0
55: J   2   3  59      -JLOOP- let a = a + psize, jump to AFT
56: S   1              let a = a + 1
57: S   3              let c = c + 1
58: J   1   1  55      jump to JLOOP
59: P   1  23  11      -AFT-  let quad[three] = a
60: J   1   1  48      jump to CONTINUE
61: T   5   1          -DONE-  let a = ans
62: H
```

B.5 Pseudocode for NXSTATE and UNIV

Section 4.9 contains a pseudocode description of UNIV, the universal ARM program for two variables. Here we shall give an alternate explanation of this program. We first describe the program NXSTATE0 which is on the distribution disk. Once we have NXSTATE0, UNIV is built in a very simple way by calling NXSTATE0 repeatedly in a loop. The program NXSTATE on the distribution disk starts with 14 steps where the registers R_{20} through R_{27} are given the constant values 0 through 7. UNIV will use the following simpler program NXSTATE0 where these 14 initial steps are left out.

Let **P** be an ARM program which uses at most the registers R_1 through R_n. By the state of an ARM machine at a particular stage in a computation on **P** we mean the finite sequence of length $n+1$ consisting of the contents of the program counter and the registers R_1 through R_n. NXSTATE0 takes as inputs a Gödel number *inst* of a simulated ARM program **P** and a Gödel number *reg* of the state of **P**. (These Gödel numbers do not have to be in standard form). Its output is the Gödel number of the next state of **P** which results after the execution of the instruction of **P** given by the program counter.

The variable pc stands for the simulated program counter, which is the number of the instruction to be executed. quad stands for a

Gödel number of a simulated instruction. op stands for the zeroth term of the sequence coded by quad and is an opcode for one of the instruction letters H,Z,S,T,J,E,P. s1,s2,s3 are the remaining terms of the sequence coded by quad, and v1,v2,v3 are the values of the simulated registers numbered s1,s2,s3.

To make NXSTATE0 easier to use in a loop, we regard the opcode variable op as a second output which will be used by UNIV to determine whether the simulated program has halted.

Since NXSTATE0 and UNIV are ARM programs, the Extract and Put commands are available. To emphasize the meaning of these commands for sequences, we write Extract(x,y,z) as z=x[y] and Put(x,y,z) as z[y]=x.

By replacing the Extract and Put commands by the simple RM programs EXTRACT and PUT, NXSTATE0 and UNIV can, in principle, be written as programs for the simple RM. Such programs would be long and too slow to use in practice.

```
program NXSTATE0(inst,reg,op)
(Nextstate with constants 0-7 aready given)
 input: inst = a G.N. of instruction list,
        reg = a G.N. of old state
        zero = 0, one = 1,two = 2, three = 3
        H = 1, Z = 2, S = 3, T = 4, J = 5, E = 6, P = 7
 output: reg = the G.N. of new state,
         op = opcode of instruction

    let pc = reg[zero]
    let quad = inst[pc]
    let op = quad[zero]
    let s1 = quad[one], s2 = quad[two], s3 = quad[three]
    let v1 = reg[s1], v1 = reg[s2], v3 = reg[s3]
    if op = Z then
        let reg[s1] = zero, pc = pc + 1
    else if op = S then
        let v1 = v1 + 1, reg[s1] = v1, pc = pc + 1
    else if op = T then
        let reg[s2] = v1, pc = pc + 1
```



```
    else if op = J then
        if v1 = v2 then let pc = s3
        else let pc = pc + 1
    else if op = E then
        let v3 = v1[v2], reg[s3] = v3, pc = pc + 1
    else if op = P then
        let v3[v2] = v1, reg[s3] = v3, pc = pc + 1
    else let op = H
    let reg[zero] = pc
end of program NXSTATE0
```

Here is a pseudocode description of the universal program UNIV which uses NXSTATE0 in its main loop. UNIV has three inputs, a Gödel number of an ARM program **P** (not necessarily in standard form), and two numbers x and y. The output of UNIV is the output of **P** computing on inputs x and y.

```
program UNIV(a,b,c)
(Universal ARM program)
 input: a = a G.N. of an ARM program P
        b = x, c = y
 output: a = P(x,y)

  let zero = 0, one = 1, ... , seven = 7
  let time = 0
  let reg = 0
  let reg[one] = b, reg[two] = c
  let op = 0
  do until op = H
    NXSTATE0(a,reg,op)
    let time = time + 1
  loop
  let a = reg[one]
end of program UNIV
```


B.6 NXSTATE0.GN listing

```
 0: E   4  20   5  -NXSTATE0- let pc = reg[zero]
 1: E   1   5   6  let quad = instr[pc]
 2: E   6  20   7  let op = quad[zero]
 3: E   6  21   8  let s1 = quad[one]
 4: E   6  22   9  let s2  quad[two]
 5: E   6  23  10  let s3 = quad[three]
 6: E   4   8  11  let v1 = reg[s1]
 7: E   4   9  12  let v2 = reg[s2]
 8: E   4  10  13  let v3 = reg[s3]
 9: J   7  22  17  if op = two, jump to ZERO
10: J   7  23  19  if op = three, jump to SUCC
11: J   7  24  22  if op = four, jump to TRANS
12: J   7  25  24  if op = five, jump to JUMP
13: J   7  26  26  if op = six, jump to EXTR
14: J   7  27  29  if op = seven, jump to PUT
15: T  21   7      let op = one
16: J   1   1  35  jump to DONE
17: P  20   8   4  -ZERO-  let reg[s1] = 0
18: J   1   1  32  jump to NEXT
19: S  11          -SUCC-  let v1 = v1 + 1
20: P  11   8   4  let reg[s1] = v1
21: J   1   1  32  jump to NEXT
22: P  11   9   4  -TRANS-  let reg[s2] = v1
23: J   1   1  32  jump to NEXT
24: J  11  12  34  -JUMP-  if v1 = v2, jump to SETPC
25: J   1   1  32  jump to NEXT
26: E  11  12  13  -EXTR-  let v3 = v1[v2]
27: P  13  10   4  let reg[s3] = v3
28: J   1   1  32  jump to NEXT
29: P  11  12  13  -PUT- let v3[v2] = v1
30: P  13  10   4  let reg[s3] = v3
31: J   1   1  32  jump to NEXT
32: S   5          -NEXT- let pc = pc + 1
33: J   1   1  35  jump to DONE
34: T  10   5      -SETPC- let pc = s3
```

```
35: P   5  20   4  -DONE- let reg[zero] = pc
36: H
```

B.7 UNIV.GN listing

```
 0: Z  20          -UNIV- let zero = 0
 1: T  20  21
 2: S  21          let one = 1
 3: T  21  22
 4: S  22          let two = 2
 5: T  22  23
 6: S  23          let three = 3
 7: T  23  24
 8: S  24          let four = 4
 9: T  24  25
10: S  25          let five = 5
11: T  25  26
12: S  26          let six = 6
13: T  26  27
14: S  27          let seven = 7
15: Z  15          set time counter to 0
16: Z   4          simulated register list, let reg = 0
17: P   2  21   4  let reg[one] = b
18: P   3  22   4  let reg[two] = c
19: Z   5          let pc = 0
20: Z   7          let op = 0
21: J   7  21  59  -LOOP-  if op = one, jump to EXIT
22: E   1   5   6  -NXSTATE0-  let quad = instr[pc]
23: E   6  20   7  let op = quad[zero]
24: E   6  21   8  let s1 = quad[one]
25: E   6  22   9  let s2 = quad[two]
26: E   6  23  10  let s3 = quad[three]
27: E   4   8  11  let v1 = reg[s1]
28: E   4   9  12  let v2 = reg[s2]
29: E   4  10  13  let v3 = reg[s3]
30: J   7  22  38  if op = two, jump to ZERO
```



```
31: J   7  23  40  if op = three, jump to SUCC
32: J   7  24  43  if op = four, jump to TRANS
33: J   7  25  45  if op = five, jump to JUMP
34: J   7  26  47  if op = six, jump to EXTR
35: J   7  27  50  if op = seven, jump to PUT
36: T  21   7      let op = one
37: J   1   1  56  jump to DONE
38: P  20   8   4  -ZERO-  let reg[s1] = 0
39: J   1   1  53  jump to NEXT
40: S  11          -SUCC-  let v1 = v1 + 1
41: P  11   8   4  let reg[s1] = v1
42: J   1   1  53  jump to NEXT
43: P  11   9   4  -TRANS-  let reg[s2] = v1
44: J   1   1  53  jump to NEXT
45: J  11  12  55  -JUMP-  if v1 = v2, jump to SETPC
46: J   1   1  53  jump to NEXT
47: E  11  12  13  -EXTR-  let v3 v1[v2]
48: P  13  10   4  let reg[s3] = v3
49: J   1   1  53  jump to NEXT
50: P  11  12  13  -PUT- let v3[v2] = v1
51: P  13  10   4  let reg[s3] = v3
52: J   1   1  53  jump to NEXT
53: S   5          -NEXT- let pc = pc + 1
54: J   1   1  56  jump to DONE
55: T  10   5      -SETPC-  let pc = s3
56: P   5  20   4  -DONE-  let reg[zero] = pc
57: S  15          increment time counter
58: J   1   1  21  jump to LOOP
59: E   4  21   1  -EXIT- let output = reg[one]
60:  H
```

Appendix C

The Logiclab Package

Logiclab is a package of four programs which are included with and keyed to the book. The diskette has the main programs, documentation files, problem files, and DOS and Windows (R) setup programs.

The DOS setup program, SETUPDOS.EXE, will create a directory of your choice for the Logiclab package on the hard disk, copy the DOS versions of the programs to the directory, and copy each problem set to a different subdirectory. The Windows setup program, SETUPWIN.EXE, does the same thing but works within Windows and copies the Windows versions of the programs instead of the DOS versions. Neither setup program does anything else.

To install the DOS version of Logiclab on a hard disk, put the diskette in a drive slot (say drive A:), type A:SETUPDOS, and follow the directions on the screen.

To install the Windows version of Logiclab on a hard disk, put the diskette in a drive slot (say A:), run A:SETUPWIN from within Windows, and follow the directions on the screen.

The programs can be used for the problem sets and for classroom demonstrations. There are two versions of each program, one for DOS and one for Windows. The DOS versions of the four programs are TABLEAU, COMPLETE, PREDCALC, and GNUMBER. The Windows versions, which work with either Windows 3.0 or later, or Windows 95, are TABWIN, COMPWIN, PREDWIN, and GNUMWIN. These Windows versions have built-in tutorials which can be selected when the program is started and give quick introductions. The follow-

ing appendices are manuals which tell you how to use these programs. The programs are designed so that you can load in problems, save your solutions in files on a diskette, and hand them in to the instructor.

It may be helpful for the instructor to demonstrate the programs to the class before assigning problem sets, perhaps using a screen projector. The student will want to know which key has been pressed at each step. For this purpose, the DOS versions of all the programs have the option of displaying the last key pressed in the upper right corner of the screen. This option is turned on or off by hitting the control key and the K key together.

TABLEAU is a Semantic Tableau Editor. It helps you to construct a formal tableau proof in either propositional or predicate logic. You have to do the thinking and tell the program which proof steps and substitutions to make at each time. The program takes care of the routine tasks of writing down formulas, making sure that each step is legal, and checking for contradictory branches. The problems give you a list of hypotheses and a formula to be proved, and your task is to build a tableau proof.

The COMPLETE program automatically extends a propositional tableau to a finished tableau. It is designed for classroom demonstrations and experimentation by the student, and illustrates the main part of the proof of the Completeness Theorem for propositional logic.

PREDCALC illustrates the semantics of predicate logic. It acts like a reverse Polish notation calculator, but operates on wffs instead of numbers, and displays both the formulas and their graphs at each step. The problems show you the graph of an unknown wff, and your task is to build a wff which has that graph.

GNUMBER simulates either a simple or advanced register machine, and has extra capabilities which let you experiment with Gödel numbers of wffs. The problems describe computable functions, and your task is to build register machine programs which compute them. The GNUMBER program can be used to check your program, and also to experiment with advanced programs such as the universal ARM program.

The programs are copyrighted by the authors of this book and are part of the public domain. The package fits on one high density 3.5" diskette. Here is a list of the files which are included with the book.

READ.ME: Installation instructions and list of files on the diskette

SETUPDOS.EXE: The program for installing the DOS package on a hard disk

SETUPWIN.EXE: The program for installing the Windows package on a hard disk

TABLEAU.EXE: The Semantic Tableau Editor

TABLEAU.DOC: Manual for the TABLEAU program

TABWIN.EXE: The Windows version of TABLEAU

TABWIN.DOC: Manual for the TABWIN program

***.TBU:** Problem files for TABLEAU or TABWIN, located in the directories TAB1, TAB3, and TAB4

***.TBM:** Modal logic problem files for TABLEAU or TABWIN, located in the directory TAB7

TABPROB.DOC: Discussion of TABLEAU problems

COMPLETE.EXE: Tableau Completer for Propositional Logic

COMPLETE.DOC: Manual for the COMPLETE program

COMPWIN.EXE: Windows version of COMPLETE

COMPWIN.DOC: Manual for the COMPWIN program

PREDCALC.EXE: The Predicate Calculator program

PREDCALC.DOC: Manual for the PREDCALC program

PREDWIN.EXE: The Windows version of PREDCALC

PREDWIN.DOC: Manual for the PREDWIN program

***.PRC:** Problem files for the PREDCALC or PREDWIN, located in the directory PRED2

PREDPROB.DOC: Discussion of PREDCALC problems

GNUMBER.EXE: Register Machine Program with Godel Numbers

GNUMBER.DOC: Manual for the GNUMBER program

GNUMWIN.EXE: Windows version of GNUMBER

GNUMWIN.DOC: Manual for the GNUMWIN program

*.GN: Sample register machine programs for GNUMBER or GNUMWIN, located in the directories GNUM5 and GNUM6

GNUMPROB.DOC: Discussion of GNUMBER problems

Appendix D

TABLEAU – Tableau Editor for DOS

D.1 Introduction

TABLEAU, the Tableau Editor for Predicate Logic, helps you write down a tableau proof. It will run on an IBM PC or compatible computer with at least 320K memory and one disk drive. With more memory, you will have room to build a larger tableau, up to 1500 nodes.

TABLEAU has a top level title screen and three modes of operation: *Hypothesis Mode* builds the formula to be proved and a list of hypotheses. The program will only allow well-formed formulas to be entered. *Tableau Mode* builds a semantic tableau, and shows the current branch and its two neighbors. The program will only allow trees which follow the rules for a semantic tableau. *Map Mode* shows the whole semantic tableau but with abbreviated formulas.

There are two forms of the TABLEAU program, which you select at the title screen. The ordinary form of the program builds tableaus for propositional and predicate logic, and is used for the first three TABLEAU problem sets. The alternative MODAL form of the program builds tableaus for modal logic, and is used in the last problem set which deals with the modal logic forms of the Gödel incompleteness theorems. The modal form of the tableau program will be discussed at the end of this manual.

The program starts with the title screen, and then goes to the Hypothesis Mode. You can change from one mode to another with the commands H, T, and M. The command Q is used to quit the program. To protect against accidental quitting, the first Q returns you to the title screen, and the program asks you to type Q a second time to be sure you really meant to quit. At the title screen you can return to the previous tableau or start a new tableau instead of quitting.

D.2 Getting Started

The program can be run from either a floppy diskette or a hard disk. With a diskette, put a diskette with the TABLEAU.EXE program file in the currently active drive. With a hard disk, either install the program as part of the Logiclab package by typing SETUPDOS.EXE at the DOS prompt, or copy the TABLEAU.EXE file and the TAB1, TAB3, TAB4, and TAB7 subdirectories to a hard disk directory entitled LOGICLAB (or another name of your choice.) If you have a color display, type TABLEAU and hit Enter at the DOS prompt. If you have a monochrome display, type TABLEAU M and hit Enter.

D.3 Title Screen

The title screen appears when you initially start the program and when you use the Q command from within the program. At the initial title screen, you have the following choices:

Enter key or T : Start the TABLEAU program.

D : Change the Drive or Directory from which the problems and solutions are loaded and saved.

If you plan to work on problem files in the TAB1, TAB3, TAB4, or TAB7 directories, you should type D at the title screen, and then specify the problem directory when you see "Enter new path". For example, if you are working from a diskette in the A: drive and wish to work on the problems in TAB1, type `A:\TAB1` and hit Enter. If you are working from hard drive C and wish to work on the problems in TAB1, type `C:\LOGICLAB\TAB1` and hit Enter.

M : Start a MODAL tableau.

Q : QUIT the program.

When you return to the title screen from within the program, you have the following choices:

Enter key : Return to the current tableau without change.

D : Change the drive or directory from which the problems and solutions are loaded and saved.

M : Start a new MODAL tableau.

T : start a new TABLEAU.

Q : Quit the program.

If your current work has not been filed, you will be given a warning and another chance to file the tableau by hitting the F key.

D.4 Hypothesis Mode

In this mode you can enter the formula to be proved and/or a list of hypotheses. You can either type these formulas in at the keyboard or load them from the problem disk. Use the up and down Arrow keys and the PageUp and PageDown keys to move the cursor among the lines on the screen.

D.4.1 Commands in Hypothesis Mode

The currently available commands are listed in the window at the bottom of the screen. Capital or lowercase letters may be used interchangeably. The E(dit), K(ill), and P(ull) commands are available only before a tableau has been started. The F(ile), L(oad), Q(uit), and ?(help) commands are available in all three modes.

E : Edit. Before starting the tableau, you may add new formulas to the hypothesis list and edit old formulas. Go to the line you want to change and hit the E key. The computer will say "new" or "here" if the line is empty, "wff" if the line contains a Well Formed Formula, and "bad" otherwise. When you have a wff or an empty line, you go back to the main program or to another line by hitting the Enter key, the up or down arrow key, or the PageUp or PageDown key . When the computer says "bad," the ? or F1 key will tell you what is wrong with the formula.

F : File. Saves the hypothesis list and tableau in its present state into a file on the disk. A box will appear with either a blank file name or with the name you used last time you filed the current tableau. The file name has the form XXXXXXXX.TBU. Use the keyboard to enter or change the file name. (You should not enter the suffix ".TBU"; the computer will add it automatically). When you have the name you want, hit the Enter key to save the tableau. You are warned if you try to use a file name which already exists. The Esc key cancels the File command, and goes back to the program without saving.

The F command can also be used to erase an unwanted TBU file. To ERASE a TBU file, Quit and start an empty tableau (no hypotheses and no formula to be proved), hit F for the File command, and type the name of the file you want to erase.

K : Kill. The formula in the present line will be removed from the hypothesis list.

L : Load. Use this command to load a problem set or a previously saved tableau. The list of *.TBU files in the current directory is displayed. Type the name of one of these files and press the Enter key. You should not enter the "TBU" suffix, only the name as it appears in the window. If you wish to change directories, you must hit Q to return to the title screen and follow the instructions in Section D.3 above.

P : Pull. The hypothesis in the current line is pulled from its present position and put at the end of the hypothesis list. You can use this command to easily change the order in which the hypotheses are listed.

conjunction	& or /\	AND
disjunction	| or \/	OR
negation	~ or ¬ (Ctrl N)	NOT
implication	->	IMPLIES
equivalence	<->	IFF
universal quantifier	Å (Ctrl A)	ALL
existential quantifier	É (Ctrl E)	EXIST
infix relations	= < <= > >=	
infix functions	+ - *	

Table D.1: Symbols Used by the Tableau Program

M : Change to Map Mode.

Q : Quit. This command returns you to the title screen. From the title screen you can either quit the program, return to the current state without change, change the drive or directory where the tableau files are loaded and saved, or start a new tableau.

T : Change to Tableau Mode.

? or F1 key : Brings up a HELP screen which summarizes the commands.

D.4.2 Propositional Logic

The symbols which are allowed in formulas are the propositional connectives shown in table D.1 and the brackets [and] for punctuation. The computer will accept either the symbols or words as shown in the table.[1] Any other string of letters and numbers which begins with a letter can be used as a propositional symbol.

[1]The words IFTHEN and ONLYIF may also be used as synonyms for IMPLIES.

D.4.3 Predicate Logic

In addition to the symbols used in propositional logic, the quantifiers, infix relation symbols, and infix function symbols shown in table D.1 are allowed. The parentheses and comma are also used for punctuation. Any other string of letters and numbers which begins with a letter can be used as a variable, relation symbol, or function symbol. The type of symbol and the number of argument places are determined by the first use of the symbol. A string which begins with a number can be used only as a constant symbol.

Note: You must put a space between a variable in a quantifier and an atomic formula to tell the computer where the variable ends. For example, `Åx p(x)` is a wff, but `Åxp(x)` is bad because the computer will interpret `xp` as a single variable.

D.4.4 Moving Within a Formula

You can move within a formula using the Right and Left arrow keys. New symbols are inserted at the cursor position. The Backspace and Del keys can be used to erase symbols as usual. The Home key will jump to the beginning of the line and the End key will jump to the end of the line. The Esc key will erase the entire line.

D.4.5 Size Limit for Formulas

The size limit for a formula typed in at the keyboard is 70 characters, which reaches to the end of the line on the screen. Additional characters beyond this limit will be ignored. Propositional symbols, predicate symbols, function symbols, and constant symbols have a maximum length of 20 characters. If a longer symbol is entered, the computer will use only the first 20 characters.

D.5 Tableau Mode

In this mode you can build a semantic tableau. The tableau is a tree which has a formula at each node. The top node has the negation of the formula to be proved, and the next nodes have the hypotheses. If

every branch through a node is contradictory, the formula is shown in red (or enclosed in ":" symbols on a monochrome screen). When every node of the tableau is red, the tableau is a completed proof. On a color display, colored text will often be used. These colors will not be visible on a monochrome display, except for the red formulas which are enclosed in ":" symbols.

Your current location in the tableau is the node which has the blinking cursor and blue background (or reversed text on a monochrome screen). The tableau is built one step at a time. To extend the tableau, you move the cursor to a formula, type G to Get the formula into a box in the window at the bottom of the screen, move the cursor to the end of a branch, and then type E to Extend the tableau. The program will only allow tableau extensions which are legal according to the formal definition of a tableau in the text.

D.5.1 Moving Within the Tableau

The screen shows the current branch of the tableau and the neighboring branches to the right and left. If the tree is too large, only part of the tableau can be seen on the screen at one time. The cursor can be moved within the tableau using the arrow keys in the following ways:

Up arrow : Move up one line.

Down arrow : Move down one line along the current branch.

Right arrow : Move down one line and bear to the right.

Left arrow : Move down one line and bear to the left.

Home : Move to the top of the tableau.

End : Move to the end of the current branch.

PageUp : Move up one screen (9 lines) .

PageDown : Move down one screen (9 lines).

D.5.2 Mouse

The program checks to see whether a mouse is installed. If a mouse is installed, you have can use either the mouse ball or the arrow keys to move within the tableau. The left mouse button can be used instead of the G key to get a formula into the box, and the right button can be used instead of the E key to extend the tableau.

D.5.3 Commands in Tableau Mode

The list of commands is shown in the window at the bottom of the screen. (Only the currently available commands are listed.)

D.5.4 Propositional Logic

E : Extend. The tableau is extended using the tableau rule for the formula in the "Get" box. This command is available only when the cursor is at the bottom of a branch. Nothing happens if the "Get" formula is atomic or negated atomic.

F : File. This is the same as the F command in Hypothesis Mode.

G : Get. The formula at the cursor is put into the green "Get" box in the bottom window. (If the formula is not red, it is also shown in green in the tableau). If you later change branches above that formula, it will drop out of the box. This makes sure that the formula can only be used below the place where it appears in the tableau.

H : Change to Hypothesis Mode.

K : Kill. This command erases everything below the cursor, and is used to correct mistakes.

L : Load. This is the same as the L command in Hypothesis Mode.

M : Change to Map Mode.

P : Print the current branch of the tableau. The printer must be connected and turned on. The logical symbols ALL, EXIST, and NOT will be printed as A, E, and ~.

Q : Quit. Same as the Q command in Hypothesis Mode.

U : Undo. This command undoes the last Kill or Extend command, and goes back to the previous position.

W : Why. This command tells you which formula was used to add the current formula to the tableau. It does this by putting the formula which was used into the "Get" box, and writing the formula in green in the tableau.

? or F1 : Brings up a HELP screen which summarizes the commands.

D.5.5 Predicate Logic

When the tableau is extended using a quantified formula, the variable in the quantifier is replaced by a term. In this program, you must tell the computer which term to use. This is taken care of by an extra provision in the Extend command.

E : Extend (continued). If the formula in the "Get" box starts with a quantifier or negated quantifier, the bottom window turns red (on a color display) and asks you for a term to substitute for the quantified variable. The rules for entering terms here are the same as the rules for entering formulas in Hypothesis Mode. The computer will not let you enter a bad term and will explain what is wrong when you hit the ? or F1 key. Hit the Enter key when you are finished entering the term.

D.5.6 Predicate Logic with Equality

A second box, the "Sub" box, is added in the bottom window to provide for the equality substitution rule. A new command is added which puts a formula into this box.

S : Substitution. This command is available only when the current formula is an equation. The bottom window turns red and you are asked to either accept the equation as given (Enter key), or to reverse it (Right arrow key). The equation will then appear in the "Sub" box. If the formula is not red, will be written in cyan (blue-green) in the tableau.

E : Extend (continued). If the formula in the "Get" box is an atomic or negated atomic formula, the equality substitution rule will be used. To do this the "Sub" box must contain an equation between two terms. The new formula is formed by taking the "Get" formula and replacing the first term in the "Sub" box by the second term in the "Sub" box. Nothing will happen if there is no possible substitution. If there is exactly one possible substitution, the computer will highlight the substitution position, and you will be asked to accept (Enter key) or cancel (Esc key). If there is more than one place to substitute, the computer will highlight the first one and ask you to accept, cancel, or go to the next place (Right arrow key).

W : Why (continued). If the current formula was added to the tableau by an equality substitution, the substitution equation will be put into the "Sub" box and written in cyan in the tableau, and the target of the substitution will be put into the "Get" box and written in green in the tableau.

= : Equality rule. Extend the formula by adding an equation of the form t = t. The bottom window will ask you to type in the term t followed by the Enter key. You can cancel this command by typing the Esc key. This command is available only if there is an = symbol in the hypothesis list.

D.5.7 Size Limit for Substitutions

The maximum length of a term entered at the keyboard during a substitution is 70 characters, which reaches to the end of the line on the screen. Additional characters beyond this limit will be ignored. The length of a formula can increase when a term is substituted for a variable. The program will accept a substitution which results in a formula up to 152 characters long. Beyond that limit, it will give a "long formula" message.

D.6 Map Mode

This mode displays the tableau in a smaller scale by showing only the main connective of the formulas. If the tableau is too large to fit on the screen in Tableau Mode, use the Map Mode to see the big picture. The current location in the tableau is again shown by the blinking cursor and blue background, and the current formula is displayed in full in the bottom window. The current branch is connected by white lines, and other branches are connected by yellow lines. You can still use the arrow and page keys to move within the tableau. However, you cannot change the tableau in Map Mode. Sometimes the tableau is so complex that it will not fit on the screen even in Map Mode. A sharp symbol, #, is used to indicate a portion of the tableau which is too complicated to fit on the screen. The Zoom command can be used to enlarge a portion of the tableau to see what is inside the # symbol.

The commands, shown in the bottom window, are as follows.

F : File. This is the same as the F command in Hypothesis Mode.

H : Change to Hypothesis Mode.

L : Load. This is the same as the L command in Hypothesis Mode.

Q : Quit. Same as the Q command in Hypothesis Mode.

T : Change to Tableau Mode.

Z : Zoom. This command redraws part of the tableau in a larger scale with the present cursor position at the top of the screen. This

command is useful when the tableau is so large that it will not fit on the screen even in Map Mode, so that # symbols appear on the screen. It is best to use this command with the cursor at a node which is below the point where the central branch splits and above the # symbol.

? or F1 : Brings up a HELP screen which summarizes the commands.

If a mouse is installed, you may use the mouse ball instead of the cursor keys to move around the tableau in Map Mode.

D.7 The Modal Logic Option

The Modal logic form of the tableau program is chosen by hitting the M key at the title screen. (Modal Logic is used in Chapter 5 of the text to develop the Gödel Incompleteness Theorems.) The menus at the bottom of the screen will now say "MODAL." In the modal logic form of the program, only propositional symbols are allowed, and there are no variables or quantifiers. In addition to the logical connectives, there is one extra rule of formation for wffs:

If **A** is a wff, then $\Box\mathbf{A}$ is a wff.

The symbol $\Box$ is called a modal operator. Intuitively, $\Box$ means "provable" in Chapter 5. It is shown on the computer screen as a solid box. It can be entered at the keyboard in either of three ways: type #, hold down the Ctrl key and hit B, or type the word **BOX**.

There is one new rule for extending a tableau, called Axiom, which adds a modal axiom at the end of a branch. In Tableau Mode, if the cursor is at the end of a branch and you hit the A key, a menu will appear with the following six choices:

(1) (mp) $\Box\mathbf{A} \wedge \Box[\mathbf{A} \Rightarrow \mathbf{B}] \Rightarrow \Box\mathbf{B}$
(2) (n) $\Box\mathbf{A} \Rightarrow \Box\Box\mathbf{A}$
(3) (fmp) $\Box[\Box\mathbf{A} \wedge \Box[\mathbf{A} \Rightarrow \mathbf{B}] \Rightarrow \Box\mathbf{B}]$
(4) (s) $\Box\Box\mathbf{A} \Rightarrow \Box\mathbf{A}$
(5) (tt) $\Box\mathbf{A}$, where **A** is a modal tautology
(6) Other modal axiom.

If you choose (6), the computer will let you enter any modal wff whatsoever at the end of the branch. This provides flexibility, but is never needed in the problem set assigned in the text. If you choose any of (1) through (5), you will be asked to type in the wff **A**, and if needed, the wff **B**. As usual, the line will be labeled "bad" if the string is not yet a wff, and "wff" if it is one. At any point, hitting the Esc key will cancel the process of adding an axiom, and return to the previous tableau position. When you have a wff, you may finish the line by hitting the Enter or down arrow key.

If you had selected one of the axiom schemes (1) – (4), the wffs which you entered for **A** and **B** will then by substituted into the axiom scheme, and the resulting formula will be displayed at the bottom of the screen for your approval. Hit the Enter key to accept the axiom and add it to the tableau, and the Esc key to cancel.

If the axiom scheme which you had selected is number (5), Axiom (tt), the computer will ask you to show that **A** is a tautology by building a separate tableau proof of **A**. If you succeed, then $\Box\mathbf{A}$ will be added to the main tableau and displayed at the bottom of the screen for your approval. But if you fail or give up by hitting the Q key, the main tableau will remain unchanged.

In the Modal form of the TABLEAU program, the letters **A** and **B** should not be used as proposition symbols, because they are the place holders for wffs when entering modal axioms.

The window at the bottom of the screen in Tableau Mode will show which modal axiom schemes have been used in the current tableau. In some problems, you will be allowed to use only some of the modal axiom schemes. When the "Why" command is invoked at a modal axiom, it will tell you which modal axiom scheme was selected when the wff was added to the tableau.

In the Modal form of the TABLEAU program, the tableaus will be filed and loaded with names of the form XXXXXXXX.TBM, (with a .TBM ending instead of .TBU). The .TBM files are found in the directory TAB7. (If a *.TBM file is renamed to *.TBU and loaded into the regular TABLEAU program, the program will switch to the Modal Logic form and continue to run, and vice versa).

D.8 Changing Directories

When you start the program by typing TABLEAU and hitting the Enter key, the program will use the currently active drive or directory for loading the problem files and saving solutions. You can change drives or directories using the D option within the program at the title screen. A period "." can be used for the current directory, and a double period ".." for the parent of the current directory.

You can start the program with another drive or directory for problem files by typing TABLEAU followed by the desired path name and hitting the Enter key. For example, to automatically load the TAB1 directory of .TBU problem files, you would type at the DOS prompt TABLEAU TAB1 and hit the Enter key. This feature may be useful in a computer lab setting. The path and the M option for a monochrome display can be combined or used separately.

For example, the instructor may create a batch file called TAB.BAT which has the single line

TABLEAU M A:

If the student types TAB [Enter key], the program will run with a monochrome display and will use diskette drive A: for the problems.

Appendix E

TABWIN - Tableau Editor for Windows (R)

E.1 Introduction

TABWIN is a version of the TABLEAU program which works under Microsoft (R) Windows 3.0 or later, and under Windows 95. The TABWIN program can only be started after Windows is running. It can be operated with a mouse or with the keyboard, and works like other Windows applications.

The TABWIN.EXE program and the TAB1, TAB3, TAB4, and TAB7 problem directories can either be copied to your hard disk into a directory named LOGICLAB (or any other name you choose), installed using the SETUPWIN.EXE program, or accessed directly from the diskette. In all of these cases, access the Windows File Manager (in Windows 3.0 or later) or My Computer (in Windows 95), select the disk drive and directory that contains the program, and then select TABWIN.EXE.

The program will begin with a welcome message in a small window with two buttons labeled "Start" and "Tutorial". Click the mouse on the "Tutorial" button to get a quick introduction. Click the mouse on the "Start" button or hit the Enter key to begin the program in the normal way.

The program helps you write down a tableau proof in either pred-

icate or modal logic. A set of hypotheses and a formula to be proved can either be entered at the keyboard or loaded from the disk. Ordinary tableaus use .TBU files, found in the directories TAB1, TAB3, and TAB4, and modal logic tableaus use .TBM files found in the directory TAB7. Following your instructions, the computer will display a tableau of formulas and inform you when the proof is complete.

The main menu at the top of the screen has a File menu, a View menu, a Nodes command, a Help menu, and other commands which are available at different times.

There are three modes of display which can be selected in the View menu: the Hypothesis mode, the Tableau mode, and the Map mode. The program starts in Hypothesis mode, where you can enter hypotheses and a formula to be proved from the keyboard. The tableau proof tree is built and the current branch is displayed in Tableau mode. The Map mode displays the entire tableau in a smaller scale.

A Tableau problem is a set of hypotheses (possibly empty) and an optional formula to be proved. Problems can either be entered at the keyboard in Hypothesis mode, or loaded from the disk using the Open... command in the Files menu.

Your objective is to solve a tableau problem by building a tableau proof. This is done in the Tableau mode. When a tableau proof is complete, every node in the tableau will be displayed in red. Along the way, a single node in the tableau will be shown in red in a color display, or enclosed within two % signs in a monochrome display, when every branch through that node is contradictory.

You can see how large your tableau is by selecting the **Nodes** command in the main menu. A box will appear showing the number of hypotheses, the number of nodes (not counting the root node), and the amount of remaining free space in nodes available for extending the tableau.

The **Help menu** can be reached by using the mouse or the F1 key. It contains five lists of topics and an **About** command, which shows the version number, copyright notice, and icon.

E.2 File Menu

The **New** command will start a new tableau. The current tableau and hypothesis set will be cleared. You will be warned if your current tableau has not yet been saved on disk.

The **Open...** command is used to load a .TBU or .TBM file from the disk, containing a tableau problem or solution. You will be warned if your current tableau has not yet been saved on disk. Directories TAB1, TAB3, and TAB4 contain .TBU files, while directory TAB7 contains .TBM files. You may either choose a file or new directory from a list, or type in the name of a file. If you only give the first part of a file name, the .TBU or .TBM extension will be added automatically.

The **Print** command will print the current branch and the two neighboring branches of the tableau. Red nodes will be printed with ! signs before and after the formula, negation signs will be printed as ~ symbols, and quantifiers will be printed as A and E.

The **Save** command will save the current tableau under the current name. The Save command is disabled if there is no name, or if the current name is a problem file. In these cases you should use the Save As... command instead.

The **Save As...** command will save the current tableau under a name which you will supply. If you only give the first part of a file name, the .TBU extension (or the .TBM extension when the modal logic option is in effect) will be added automatically.

The **Exit** command will quit the Tableau Editor Program and return to Windows. You will be given a warning and a chance to save the current tableau if it has not been saved on disk since the last change.

E.3 View Menu

In this menu you can choose between three display modes: Hypothesis mode, Map mode, and Tableau mode, and select different options.

The **Hypothesis mode** has an editor for adding or changing hypotheses and formulas to be proved. Only full or correctly abbreviated well-formed formulas (wffs) will be accepted. After a tableau has been extended, hypotheses cannot be changed or removed, but may still be

added. You can use the arrow, home, and end keys or the mouse to select a hypothesis. Use the Enter key, e key, doubleclick on a radio button, or choose Edit in the main menu to start editing a hypothesis.

The **Map mode** displays as much of the tableau as possible in a small scale. Only the main symbol of a formula is shown at the node, but the current formula is shown in full in a box at the top of the window. You can use the arrow, home, and end keys or the mouse to move around the tableau.

The **Tableau mode** displays the current and neighboring branches of the tableau, and has the tools needed to build the tableau. The program allows only correct uses of the tableau extension rules. Use the arrow, home, and end keys or the mouse to move among the nodes of the tableau.

In the View menu you can also choose between Ordinary logic and Modal logic, and between Color and Monochrome.

In the **Ordinary Logic** option the program accepts formulas of the full first order predicate logic with equality. Tableaus should be saved in .TBU files. The program starts out in this option.

In the **Modal Logic** option the program accepts formulas of modal propositional logic. Tableaus should be saved in .TBM files.

Use the **Color** option if you have a color monitor. A node will be displayed in red if every branch through it is contradictory.

Use the **Monochrome** option if you have a monochrome monitor. A node will be enclosed in a pair of % symbols if every branch through the node is contradictory.

E.4 Entering Hypotheses

Hypotheses and formulas to be proved can be entered from the keyboard in the Hypothesis mode. In this mode the Edit, Cut, and Paste commands are available on the main menu.

There are several ways to start the hypothesis editor: hit the Enter key, hit the e key, doubleclick on a radio button, or choose the **Edit** command on the main menu. Use the Ok button or Enter key when you are done typing in a hypothesis. Only full or correctly abbreviated wffs will be accepted. If you do not yet have a wff, the help button in the

hypothesis editor window will tell you why not. Any of the following will be accepted as binary connectives:

`and, &, /\; or, |, \/; implies, ifthen, ->; iff, <->.`

Negation and the quantifiers can be entered using the buttons or by typing the words `not, all, exist`. The modal operator is entered by typing #. Strings beginning with a letter can be either variables, predicate symbols, or function symbols, depending on first use. All strings beginning with a digit are constant symbols.

The **Cut** command will delete the current formula and save it for later pasting.

The **Paste** command will paste the formula saved by the last Cut command into the selected hypothesis line.

E.5 Viewing Tableaus

In the Map and Tableau modes, you can look at and move within a tableau. You can use the mouse, the arrow keys, the Home, End, PageUp, and PageDown keys in the natural way. The Why command is available on the main menu in both of these modes, and the Zoom command is available in the Map mode.

The **Why** command shows the node or nodes which were used to put the current node in the tableau, by placing them in the Get and Sub boxes and coloring them green and blue respectively in the tableau.

The **Zoom** command places the current node at the top of the display and shows the portion of the tableau below the current node in a larger scale.

E.6 Building Tableaus

The tableau can be built in the Tableau mode. In this mode, the Axiom, Extend, Kill, Undo, and Why commands are available on the main menu. The Get and Sub boxes, directly below the main menu, tell the computer which formulas to use in extending the tableau. When the Get box contains a nonbasic formula and the current node is terminal (at the end of a branch), the Extend command will use the formula in

the Get box to extend the tableau. When the Get box contains a basic formula, the Sub box contains an equation, and the current node is terminal, the Extend command will perform the indicated substitution on the formula in the Get box and add the result below the current node.

To fill the Get box, click on the Get button with the mouse or type Alt G at the keyboard. The formula at the current node will be colored green and placed in the Get box, and can be used to extend the tableau. The Get box is emptied if you move to a branch which does not contain the formula in the box, or if you fill the Get box when the current node is the HYPOTHESES or TABLEAU label.

To fill the Sub box, click on the Sub button with the mouse or type Alt S at the keyboard. The formula at the current node must be an equation. You will be asked to accept or reverse the substitution. The equation will be colored blue and placed in the Sub box, and can be used to extend the tableau. The Sub box is emptied if you move to a branch which does not contain the formula in the box.

The **Axiom** command will add a tableau axiom at the current position. This command is available only at a terminal node. In ordinary logic, a box will appear asking you to type in a term t, and the equation $t = t$ will be added to the tableau.

In modal logic, a box will appear asking you to choose a modal axiom scheme and type in the required formulas.

The **Extend** command will extend the tableau at the current position, using the formulas in the Get and Sub boxes. This command is available only at a terminal node. There must either be a nonbasic formula in the Get box, or a basic formula in the Get box and an equation in the Sub box. Connective rules will be applied automatically. For quantifier and equality substitution rules, a box will appear asking for additional information.

The **Kill** command deletes all nodes below the current node in the tableau. An exception: the next node below the current node will be retained if it is connected by a double line because the two nodes were built at the same time. If you Kill by mistake, you can undo it right away using the Undo menu command.

The **Undo** command undoes the most recent change in the tableau, which resulted from either an Axiom, Extend, or Kill menu command.

Appendix F

COMPLETE – Tableau Completer for DOS

The COMPLETE program is designed for student experimentation and classroom demonstrations of finished tableaus in propositional logic. Finished tableaus are a key concept in the proof of the Completeness Theorem. The program automatically extends a given tableau by extending every noncontradictory branch in all possible ways, and ends up with either a finished or a contradictory tableau.

If you have a color display, type COMPLETE and hit the Enter key at the DOS prompt. If you have a monochrome display, type COMPLETE M and hit the Enter key. The title screen will appear. Hit S to start the program.

On a color display, nodes are shown in three colors. A node is shown in *red* if every branch through the node is contradictory (as in the TABLEAU program). A node is shown in *blue* if it is not red and either (1) the node is an atomic or negated atomic formula, or (2) the node has previously been used to add nodes below it. All other nodes are shown in *yellow*. The yellow nodes are the nodes which can be used to form further extensions of the tableau in a useful way. A tableau is *finished* if it is noncontradictory and has no yellow nodes.

On a monochrome display, the "red" nodes are enclosed by ":" symbols, the "blue" nodes are enclosed by "—" symbols, and the "yellow" nodes are shown in high intensity text.

The program works by using the first yellow node it finds, or a

yellow node chosen by you, to extend every branch through the node. This forms a larger tableau, but the yellow node just used is now red or blue. This process is repeated, forming larger and larger tableaus. If the computer has enough memory, the process must end after finitely many steps with a finished tableau.

COMPLETE works like the TABLEAU program and uses .TBU files created by the TABLEAU program, but with the following differences.

(1) There is no Hypothesis mode, only Tableau and Map modes.

(2) Only propositional rules are recognized. Wffs beginning with quantifiers and atomic wffs are treated as propositional symbols.

(3) The E(xtend) command uses the current wff to extend every noncontradictory branch through the current node.

(4) There is no G(et) command.

(5) The Enter key moves the cursor to the next yellow node the computer finds.

(6) On a color display, contradictory nodes are shown in red, noncontradictory nodes which are atomic, negated atomic, or already used are shown in blue, and other nodes are shown in yellow.

(7) TBU files can be loaded but cannot be filed.

(8) There is no Modal logic option.

Appendix G

COMPWIN - Tableau Completer for Windows (R)

G.1 Introduction

COMPWIN.EXE is the Windows version of the COMPLETE.EXE program. It works with Microsoft (R) Windows, Version 3.0 or later, and with Windows 95. It can only be started after Windows is running, and is accessed in the same way as the TABWIN program. The program will begin with a welcome message in a small window with two buttons labeled "Start" and "Tutorial." Click the mouse on the "Tutorial" button to get a quick introduction. Click the mouse on the "Start" button or hit the Enter key to begin the program in the normal way.

The COMPWIN Program automatically produces finished tableaus for propositional logic. A set of hypotheses and formula to be proved can be created by the Tabwin program and saved as a .TBU file, which may then be loaded by the Compwin Program. The Completer Program works like the Tableau Editor, but the current formula is extended on every noncontradictory branch through it.

The program can be operated with a mouse or with the keyboard, and works like other Windows applications. The main menu at the top of the screen has a File menu, a View menu, Extend, Continue, Kill, Undo, and Zoom commands, and a Help menu.

There are two modes of display which can be selected in the View

menu. The program starts out in Tableau mode, where the current branch is displayed. The Map mode displays the entire tableau in a smaller scale.

A Tableau problem is a set of hypotheses (possibly empty) and an optional formula to be proved. Problems can be loaded as *.TBU files, which are created by the TABWIN or TABLEAU program, using the Open... command in the File menu. The COMPWIN program uses only the tableau extension rules for propositional logic. In a formula for predicate logic, the scope of a quantifier is treated as if it were an atomic formula.

Your objective is to build a finished tableau, that is, a tableau in which every node is either basic (atomic or negated atomic), or is used in every noncontradictory branch through it. On a color display, nodes which can still be used are shown in yellow, contradictory nodes are shown in red, and the remaining nodes are shown in blue.

In Tableau mode, the boxes at the top of the screen show the formula which was used to get the current wff, the number of hypotheses, the number of nodes (not counting the root node), the amount of remaining free space in nodes available for extending the tableau, and the current status of the tableau (Finished, Unfinished, or Contradictory).

The **Help menu** can be reached by using the mouse or the F1 key. It contains four lists of topics and an **About** command, which shows the version number, copyright notice, and icon.

G.2 File Menu

The **Open...** command will load a .TBU file from the disk, containing a tableau problem or solution. You may either choose a file or new directory from a list, or type in the name of the file from the keyboard. The .TBU extension will be added by the computer if you omit it.

The **Exit** command will Quit the Compwin Program and return to Windows.

G.3 View Menu

In this menu you can choose between two display modes, Map mode and Tableau mode. The tableau can be extended in either the Map or the Tableau mode. You can also choose between Color and Monochrome options.

The **Map mode** will display as much of the tableau as possible in a small scale. Only the main symbol of a formula is shown at the node, but the current formula is shown in full in a box at the top of the window. You can use the arrow, home, and end keys or the mouse to move around the tableau.

The **Tableau mode** will display the current and neighboring branches of the tableau. Use the arrow, home and end keys or the mouse to move among the nodes of the tableau.

Use the **Color** option if you have a color monitor. A node will be displayed in red if every branch through it is contradictory. A node is shown in yellow if it is noncontradictory, is not a basic formula, and can still be used to extend the tableau. All other nodes are shown in blue.

Use the **Monochrome** option if you have a monochrome monitor. A node will be enclosed in a pair of % symbols if every branch through the node is contradictory. A node is will be enclosed in a pair of ! signs if it is noncontradictory, is not a basic formula, and can still be used to extend the tableau.

G.4 Building a Finished Tableau

The finished tableau can be built using the Extend and Continue commands in either the Map or Tableau mode.

The **Extend** command will extend the tableau on every noncontradictory branch through the current position. The current node must be a nonbasic formula. A message box will appear if there is not enough room in memory to perform the tableau extensions.

The **Continue** command will move to the next unused formula (shown in yellow on a color monitor). This command is available only when the tableau is unfinished.

G.5 Other Commands

The remaining commands on the main menu are the Kill, Undo, and Zoom command. The Zoom command is available only in Map mode, while the other commands are available in both the Map and Tableau modes.

The **Kill** command will delete all nodes below the current node in the tableau. An exception: the next node below the current node will be retained if it is connected to the current node by a double line because it was added to the tableau at the same time. If you use this command by mistake, you can undo it right away using the Undo menu command.

The **Undo** command will undo the most recent change in the tableau, which resulted from either an Extend or Kill menu command.

The **Zoom** command is available only in the Map display mode. It places the current node at the top of the display and shows the portion of the tableau below the current node in a larger scale.

Appendix H

PREDCALC – Predicate Calculator for DOS

H.1 Introduction

PREDCALC, the predicate calculator, is a program which demonstrates the rules of formation for formulas of first order predicate logic, and the corresponding inductive definition of the truth value of a formula. It works like a reverse Polish notation calculator, but operates on formulas of predicate logic instead of numbers. There are always four formulas in a stack which are visible on the screen, and four additional formulas in storage locations not visible on the screen. The formulas in the stack are shown both as strings of symbols and as graphs. By the graph of a formula we mean the set of all valuations for which the formula is true. The Calculator Pad has 15 "buttons" which allow you to add an atomic formula to stack location 1, to make a new formula by applying a logical connective or quantifier to formulas in the stack, or to move a formula to a new location. Problems loaded from the disk produce one more formula, called the Goal formula. In some problems you can only see the graph of the goal formula, and in others you can see both the graph and the string of symbols. Your task is to use the calculator pad to make the formula in stack location 1 match the goal formula.

The program runs on an IBM PC or compatible computer with at

least 320K memory and one disk drive. It works best with a graphics monitor.

H.2 Getting Started

The program can be run from either a floppy diskette or a hard disk. With a diskette, put a diskette with the PREDCALC.EXE program file and the PRED2 problem directory in the currently active drive. With a hard disk, either install the program as part of the Logiclab package by typing SETUPDOS.EXE at the DOS prompt, or copy the PREDCALC.EXE file and the PRED2 subdirectory to a hard disk directory called LOGICLAB (or another name of your choice.) If you have a color display, type PREDCALC and hit Enter at the DOS prompt. If you have a monochrome display, type PREDCALC M and hit Enter. The title screen will appear.

H.3 Title Screen

The title screen appears when you initially start the program and when you use the Q command from within the program. As a default, the initial universe is the set $\{0,1,2,3,4,5\}$. At the initial title screen, you have the following choices:

S : Start the PREDCALC program.

D : Change the drive or directory from which the problems and solutions are loaded and saved. If you plan to work on problem files in the PRED2 directory, you should type D at the title screen, and then specify the problem directory when you see "Enter new path". For example, if you are working from a diskette in the A: drive, type `A:\PRED2` and hit the Enter key. If you are working from hard drive C, type `C:\LOGICLAB\PRED2` and hit the Enter key.

U : Change the universe. You may select a universe size between 1 and 8.

Q : Quit the program.

When you return to the title screen from within the program, you have the following choices:

Enter key : Return to the current session without change.

N : Start a new PREDCALC session.

D : Change the drive or directory from which the problems and solutions are loaded and saved.

U : Change the universe. You may select a universe size between 1 and 8.

Q : Quit the program.

If your current work has not been filed, you will be given a warning and another chance to file the session by hitting the F key.

H.4 Display Modes

At all times you can switch back and forth between three display modes, called the *Text Mode*, the *Graphics Mode*, and the *Both Mode*. The program works identically in all three modes. The program starts out in the Text Mode. On a text-only monitor, only the Text Mode is available, and the program will not let you change to the Graphics or Both modes.

In the Text Mode the formulas in the stack are shown in the usual way as strings of symbols. Free and bound variables are shown in different colors to make them easier to identify. The graphs of the formulas are not visible in the main Text Mode display, but they can be seen in tabular form by using the View command described below. The Text Mode corresponds to the syntax of predicate logic.

The Graphics Mode displays the graphs of the formulas in the stack, and the graph of the goal formula if there is one. The graphs on the screen have three dimensions, and for this reason only the three variables x, y, and z are allowed in a formula. On a color display, the

graphs are shown in 4 colors. The Graphics Mode corresponds to the semantics of predicate logic.

The Both Mode displays the formulas in the stack in both text and graphics forms at the same time, but the graphs are shown in a smaller scale. The screen will be in color on an enhanced or better color display. Otherwise the screen will be monochrome.

H.5 Goals

When you load a problem file from the disk (using the L command described later), a universe set and a goal formula are selected. There are two types of goals, text goals and graphics goals.

A text goal is visible both as a string of symbols and as a graph. The object of the problem is to match the goal formula in position one of the stack by using the calculator pad. In order to do this, you must begin with atomic formulas and build up to the goal formula through a parsing sequence. When you succeed, you will be rewarded by the appearance of the word "DONE" on the screen. If the formula in stack position 1 has the same graph as the goal formula but is a different string, you will get partial credit and the word "PART" will appear.

A graphics goal is visible as a graph but hidden as a string of symbols. The object of the problem is to think of a formula which has the required graph and to get the formula into position one of the stack by using the calculator pad. This type of problem is more difficult and requires an understanding of the interpretation of quantifiers in a model. If the letters "NC" appear after the word "GOAL" on the screen, you are asked to find a formula which has no constant symbols, and will only get partial credit if you do use constant symbols. When you succeed in matching the goal formula graph in stack position 1, you will be rewarded by the word "DONE" or "PART" on the screen.

No goal is shown until a problem file is loaded from the disk. Even without a goal, you can use the calculator pad to build and look at formulas in either text or graphics form.

H.6 The Calculator Pad

In all three display modes, a Calculator Pad with 15 buttons and a time counter are shown on the screen. The currently active button on the Calculator Pad is enclosed in a double border. The program starts with the Atomic button active. To PUSH a calculator button, hit the Enter key when the button is active.

Each button with one or more periods requires additional information, either a variable, a constant, or a stack or storage location. If you push the button, the computer will wait for you to either cancel the push by hitting the Esc key, or to give the required information and hit the Enter key.

H.6.1 The Time Counter

The number 0 in the lower left corner of the calculator pad is a Time Counter. Each time you use a calculator button to change the formulas in the stack, the time counter increases by one. A problem loaded from the disk starts with the double border around the time counter. However, nothing happens when you push the time counter "button."

H.6.2 Moving Within the Calculator Pad

The four arrow keys can be used to move vertically or horizontally to another button. The Home, PgUp, End, and PgDn keys move diagonally to another button.

H.6.3 The Help Window

In Text mode, the lower right part of the screen has a help window explaining what the currently active button does. By moving within the calculator pad and reading the help messages, you can discover what all the calculator buttons do.

H.6.4 Mouse

The program checks to see whether a mouse is installed. If a mouse is installed, you can use either the mouse ball or the arrow keys to move within the calculator pad. Either mouse button can be used instead of the Enter key to push the currently active calculator button.

H.6.5 Using the Calculator Buttons

The three buttons labeled "Atomic," "R(...)," and ".=h(..)" are used to enter an atomic formula in stack location 1. You will usually begin a session with the Atomic button. The periods represent argument places which must be replaced by variables or constants. The five buttons labeled "&," "Or," "– >," "< – >," and "Not" can be used to put a new formula in the stack by combining old formulas with logical connectives. The "All." and "Exi." buttons can be used to make a new formula by applying a quantifier to the formula at location 1 in the stack. The "Dup," "Pik.," and "Put." buttons are used to rearrange the formulas which are already in the stack. The "Sto." and "Rcl." buttons store and recall the formula in stack location 1 from a storage location.

When you push the Atomic button, the word "Atomic" disappears and the computer waits for additional information. You must type an atomic formula of length 3 or 5 which has variables among x, y, and z, constants from the universe, the relation symbols =, <, >, and the operation symbols +, -, * which stand for addition, subtraction, and multiplication modulo the universe size. The second symbol in the formula must be a relation symbol. Examples of atomic formulas are $x = 3$, $x < z+2$. Hit Enter (or the mouse button) to enter the formula, or hit the escape key to cancel.

The Random Relation button labeled "R(...)" introduces a new predicate symbol with one, two, or three argument places each time it is pushed, starting with "A." After pushing the button, you must enter one, two, or three arguments, which must be different variables from among x, y, and z. When you are done, hit Enter to enter the formula. The graph of the relation will be chosen randomly by the computer, with the variables you enter. The Random Function button ".=h(..)"

introduces a new operation symbol with one or two arguments each time it is pushed, starting with "a." Again, its graph is chosen randomly with the variables you enter.

The binary connective buttons, "&," "Or," "– >," and "< – >," combine the formulas in stack locations 1 and 2 and place the result in stack location 1. The formulas in locations 3 and 4 are moved to locations 2 and 3, and a false formula is placed in location 4.

The "Not" button and the quantifier buttons "All." and "Exi." change only the formula in stack location 1. For the quantifier buttons you must select a variable from among x, y, and z and then hit the Enter key.

The "Dup" button copies the formula in stack location 1 into stack location 2, and moves the formulas in stack locations 2 and 3 into locations 3 and 4. The formula originally in location 4 is discarded. The "Pik." button asks you for a number n between 2 and 4. It moves the formula n into stack location 1 and moves formulas 1 through $n-1$ into stack locations 2 through n. The effect is to cycle the formulas in locations 1 through n. The "Put" button also asks you for a stack location n and cycles the formulas in the opposite direction, so that formula 1 is moved into stack location n.

The "Sto." button asks you for a number n between 1 and 4. It copies the formula in stack location 1 into storage location n. The "Rcl." button also asks for a number n between 1 and 4. It copies the formula in storage location n into stack location 1, and moves the formulas in stack locations 1 through 3 into stack locations 2 through 4. The formula originally in stack location 4 is discarded.

H.7 The Letter Commands

On the screen display there is a list of commands to the left of the calculator pad. Each of these commands is invoked by hitting a single key. You may use either upper- or lowercase letters.

B : Change to the Both display mode.

C : Clear. The current session is cleared and the time counter is set back to 0.

F : File. Saves the current PREDCALC session into a file on the disk. A box will appear with either a blank file name or with the name you used last time you filed the current session. The file name has up to eight characters followed by the suffix .PRC. Use the keyboard to enter or change the file name. (You should not enter the suffix ".PRC"; the computer will add it automatically). When you have the name you want, hit the Enter key to save the session. You are warned if you try to use a file name which already exists. The Esc key cancels the File command, and goes back to the program without saving.

The F command can also be used to erase an unwanted PRC file. To erase a PRC file, Quit and start an empty session (no goal and time 0), hit F for the File command, and type the name of the file you want to erase.

G : Change to the Graphics display mode.

H or ? or F1 : Help. Brings up a help screen which summarizes the commands.

L : Load. This command displays a list of files in the current directory which contain problems or previously saved PREDCALC sessions. If you type the name of one of these files and hit the Enter key, a new universe and goal or session will be loaded. If you hit Enter without a file name, you will return to the program with no change. The files have names up to eight characters long followed by the suffix ".PRC." You should not enter the suffix, only the name as it appears in the window. You can use the File and Load commands to save and return to a partially solved problem.

P : Print. Prints the formulas in the stack and storage locations, the goal formula, the steps on the Calculator Pad up to the current time, and the graphs of the goal formula and the formula in stack position 1. The printer must be installed and turned on.

Q : Quit. This command returns you to the Title Screen. You can then quit the program by hitting Q again, change the directory

where the PRC files are filed and loaded, change the universe size, start a new session, or return to the current state.

R : Replay. Starts a replay of the current PREDCALC session beginning with time 0. The calculator pad is now under the control of the computer, and the next button is highlighted. You have the following five options.

Hit N or the Enter key to see the Next step in the replay. When you reach the end of the replay, you will regain control of the calculator pad.

Hit P or the backspace key to go back to the Previous step.

Hit H or the Home key to start the replay over at time 0.

Hit E or the End or Esc key to jump to the last step before the end of the replay. If you now hit N or the Enter key, you will regain control of the calculator pad.

Hit K to Kill the remaining steps of the replay and regain control of the calculator pad at the currently displayed time.

S : Storage. Shows the contents of the four storage locations, and the Goal formula if there is one. In the Text Mode, the graph of one formula is displayed in tabular form, and you can see other graphs by hitting the appropriate number key. In the Graphics or Both mode, the graphs of the formulas in the storage locations are displayed on the top half of the screen, and one level of one graph is enlarged in the bottom half of the screen. The arrow keys can be used to control which level of which graph is enlarged.

T : Change to the Text display mode.

U : Undo. This command undoes the last step from the Calculator Pad and decreases the time by one, going back to the previous state. It will also undo the Clear command if used immediately.

V : View. Shows the contents of the four stack locations, and the Goal formula if there is one, in the same format as the Storage command above. If you have a text only monitor, you can still

see the graphs of the formulas in the stack in tabular form using this command.

The following two special commands are intended for instructors preparing problems for students. These commands work in the same way as the File command, and create PRC files which can be loaded as problem files with the L command.

Ctrl G : File a GRAPHICS goal. This command is called by holding the Ctrl key down and hitting the G key. It saves the formula which is in stack location 1 as a graphics goal.

Ctrl T : File a TEXT goal. This command is called by holding the Ctrl key down and hitting the T key. It saves the formula which is in stack location 1 as a text goal.

H.8 Changing Directories

When you start the program by typing PREDCALC and hitting the Enter key, the currently active drive or directory will be used for loading the problem files and saving solutions. You can change drives or directories using the D option within the program at the title screen. A period "." can be used for the current directory, and a double period ".." for the parent of the current directory.

You can start the program with another drive or directory for problem files by typing PREDCALC followed by the desired path name and hitting the Enter key. For example, to automatically load the PRED2 directory of PRC files, type PREDCALC PRED2 at the DOS prompt and hit the Enter key. This feature may be useful in a computer lab setting. The path and the M option for a monochrome display can be combined or used separately.

For example, the instructor may create a batch file called PRC.BAT which has the single line

PREDCALC M A:

If the student types PRC [Enter key], the program will run with a monochrome display and will use diskette drive A: for the problems.

Appendix I

PREDWIN - Predicate Calculator for Windows (R)

I.1 Introduction

PREDWIN is a version of the PREDCALC program which runs in Microsoft (R) Windows, version 3.0 or later, and in Windows 95.

The PREDWIN program can only be started after Windows is running. It can be operated with a mouse or with the keyboard, and works like other Windows applications. The PREDWIN.EXE program and the PRED2 directory can either be copied to your hard disk into a directory called LOGICLAB (or another name of your choice), installed using the SETUPWIN.EXE program, or accessed directly from the diskette. In all of these cases, access the Windows File Manager (in Windows 3.0 or later) or My Computer (in Windows 95), select the disk drive and directory that contains the program, and then select PREDWIN.EXE.

The program will begin with a welcome message in a small window with two buttons labeled "Start" and "Tutorial." Click the mouse on the "Tutorial" button to get a quick introduction. Click the mouse on the "Start" button or hit the Enter key to begin the program in the normal way.

The program demonstrates the rules of formation for formulas of first- order predicate logic, and the corresponding inductive definition

of the truth value of a formula. It works like a reverse Polish notation calculator, but operates on formulas of predicate logic instead of numbers. There are always four formulas in a stack which are visible on the screen, and four additional formulas in storage locations not visible on the screen. The Calculator Pad in the upper left corner of the screen has 15 buttons which allow you to add an atomic formula to location 1 of the stack or to make a new formula by applying a logical connective or quantifier to formulas already in the stack.

The formulas in the stack are shown in the upper right window in text form. The universe of the current model is shown in the title bar of the window. You can choose a universe size U between 1 and 8. The elements of the universe are natural numbers beginning with 0 and ending with U-1.

The five lower windows display the graphs of the four formulas in the stack and a Goal. (By the graph of a formula we mean the set of all valuations for which the formula is true.) The graphs on the screen have three dimensions, and for this reason only the three variables x, y, and z are allowed in a formula.

The formula window and each graph window can be moved individually to any screen location.

The program runs on an IBM PC (TM) or compatible computer under Microsoft Windows (TM) 3.0 or higher. A mouse is not required, but will make the program easier to use. Microsoft Windows must be running before you start the program.

The program starts with a Welcome box. When you hit the OK button with the mouse, or hit the Enter key, the program will begin.

The main menu at the top of the screen has labels for a File menu, a View menu, an Options menu, an Undo command, and a Help menu.

I.2 Goals

If you wish to load problems from a diskette, put the diskette in drive slot A. To load a problem file from the diskette or the hard drive, select the File Menu, select the **Open...** command, and choose the correct directory or problem file from the list shown in the dialog box on the screen. When you load a problem file from the diskette, a Goal graph

will be shown in a window in the lower right corner of the screen. The object of the problem is to think of a formula which has the required graph and to get the formula into location 1 of the stack by using the buttons on the calculator pad. In order to do this, you must begin with atomic formulas and build up to the goal formula. When you succeed, you will be informed by a message on the screen.

No goal is shown until a problem file is loaded from the disk. You can use the calculator pad buttons to build and look at formulas without a goal.

I.3 The Help Menu

The Help Menu has five categories, each with a list of topics, and an **About** command which displays a box with the program name, version, and copyright information. Each topic has a brief note which you can view at any time while running the program. The General category gives an overview of the program. The Calculator Pad category has one topic for each button on the pad. The File category has one topic for each command on the File Menu, the View category has one topic for each command on the View Menu, and the Options category has one topic for each command on the Options Menu. When you choose one of the five categories, you are shown a list of topics and four buttons. The **Next List** and **Previous** buttons change to another help category. The **Help** button shows a small dialog box containing the help note for the current topic and new buttons labeled **Topics**, **Next**, **Previous**, and **Cancel**. You can move through the topics in the current list using the Next and Previous buttons. The Cancel button or Escape key removes the help box and returns you to the main program. At any time in the program, the F1 key immediately brings up the General Category Help Menu.

I.4 The Calculator Pad

You call a calculator pad command by hitting ENTER when the button with the command is marked with a dark border, or clicking the left

mouse button when the pointer is at the button. Here is a brief overview of the calculator pad commands. The three buttons labeled **Atomic**, `R(...)` , and `.=h(..)` are used to enter an atomic formula in stack location 1. You will usually begin a session with the Atomic button. The periods represent argument places which must be replaced by variables or constants. The five buttons labeled `&`, `\/`, `->`, `<->`, and **Not** can be used to put a new formula in the stack by combining old formulas with logical connectives. The **All..** and **Exi..** buttons can be used to make a new formula by applying a quantifier to a formula in the stack. The **Dup**, **Pik..**, and **Put..** buttons are used to rearrange the formulas which are already in the stack. The **Sto..** and **Rcl..** buttons store and recall the formula in stack location 1 from a storage location.

When you push the Atomic button, a dialog box appears asking for additional information. You can enter an atomic formula of length 3 or 5 which has variables among x, y, z; constants from the universe; the relation symbols $=, <, >$; and the operation symbols $+, -, *$ which stand for addition, subtraction, and multiplication modulo the universe size. The second symbol in the formula must be a relation symbol. Examples of atomic formulas are $x = 3, x < z+2$. Press the OK button to enter the formula, or the Cancel button to cancel the command.

The random relation button labeled `R(...)` introduces a new predicate symbol with one, two, or three argument places each time it is called, starting with A. A dialog box will appear asking you to enter the arguments, which must be different variables from among x, y, z. The graph of the relation will be chosen randomly by the computer, with the variables you enter. The random function button `.=h(..)` introduces a new function symbol with one or two arguments each time it is called, starting with a. Again, its graph is chosen randomly with the variables you enter. The binary connective buttons, `&`, `\/`, `->`, and `<->`, combine the formulas in stack locations 1 and 2 and place the result in stack location 1. The formulas in locations 3 and 4 are moved to locations 2 and 3, and a false formula is placed in location 4.

The **Not** button and the quantifier buttons **All..** and **Exi..** change only the formula in stack location 1. A dialog box will appear for the quantifier buttons asking you to select a variable x, y, or z.

The **Dup** button copies the formula in stack location 1 into stack location 2, and moves the formulas in stack locations 2 and 3 into

locations 3 and 4. The formula originally in stack location 4 is discarded. The **Pik..** button asks you for a stack location $n = 2, 3$, or 4. It moves formula n into stack location 1 and formulas 1 through $n-1$ into stack locations 2 through n. The effect is to cycle the formulas in locations 1 through $n-1$. the **Put..** button also asks you for a stack location n and cycles the formulas in the opposite direction, so that formula 1 is moved into stack location n.

The **Sto..** button asks you for a number $n = 1, 2, 3$, or 4. It copies the formula in stack location 1 into storage location n. The **Rcl..** button also asks for a number $n = 1, 2, 3$, or 4. It copies the formula in storage location n into stack location 1, and moves the formulas in stack locations 1 through 3 into stack locations 2 through 4. The formula originally in stack location 4 is discarded.

The $t = 0$ display in the lower left corner of the calculator pad is a TIME COUNTER. Each time you call a calculator pad command, the time counter increases by one.

The **Undo** command undoes the last calculator pad command and decreases the time by one, going back to the previous state. It will also undo a **Clear** command if used immediately. The Alt+Backspace key combination will invoke the Undo command at any time.

I.5 The File Menu

The **New** command starts a new Predcalc session. All the stack and storage locations are cleared, and you are asked to choose the size of the universe, a number between 1 and 8. The previous session will be lost, and you are warned if it has changed since being saved on disk.

The **Open...** command loads a problem file or a previously saved Predcalc session from the disk. You are shown a list of all files in the current directory with the .PRC extension, and all subdirectories, including PRED2, which contains the problem files provided on the diskette. A file or directory may be selected from the list or typed in at the keyboard. You can change directories by selecting a directory from the list or typing the name of a file in a new directory. The name of the new file will appear on the main title bar. Again, the previous session will be lost, and you are warned if it has changed since being

saved on disk.

The **Save** command saves the current Predcalc session on the disk. If the session was previously loaded or saved, it will be saved with the same name in the current directory. Otherwise you will be asked for a file name. A name with the .PRC extension, such as CUBE.PRC, is recommended. When the file is later loaded, the present goal, stack contents, and storage contents will reappear, so you can resume working on the problem or show your solution to the instructor.

The **Save As...** command asks you for a file name and then saves the current Predcalc session on the disk.

The **Save Goal...** command is used to create a new problem file. It asks you for a file name and then saves the current graph in stack location 1 as a goal. When the file is later loaded in, the graph will appear in the goal window and all stack and storage locations will contain false formulas.

The **Exit** command quits the Predcalc program. You are warned if the current session has changed since being saved on disk.

I.6 The View Menu

The **Arrange Windows** command moves all the windows in the Predwin program to their original positions.

The **Clear** command starts a new session. It clears all the stack and storage locations and sets the time back to 0.

The **Replay** command starts a replay of the current Predcalc session beginning with time 0, with no change in the random relations and functions. Instead of the calculator pad, you see a box with five buttons. The **Next** button causes the replay to go forward one step. After the last step, the calculator pad reappears and the replay is over. The **Previous** button causes the replay to go back one step. The **Home** button chooses new random relations and functions, and jumps back to the beginning at time 0. The **End** button jumps to the end of the replay and the calculator pad reappears. The **Forget** button quits the replay in the middle and brings back the calculator pad, forgetting the later steps in the session.

The **View Storage** command shows the contents of the four storage

locations. They are shown temporarily in place of the stack locations, and a dialog box replaces the calculator pad. Nothing else can be done until you push the OK button or hit ENTER, at which time the contents of the stack locations and the calculator pad will reappear.

I.7 The Options Menu

The **New Universe** command changes the universe. It preserves the time and wffs in the current session, but erases the goal graph if there is one. You will be asked for a new universe size between 1 and 8. A constant which is outside the new universe will be interpreted as the largest element of the universe.

The three options Capitalize Bound Vars, Monochrome, and Small can be turned on and off. The program starts with each option turned off. When one of these options is turned on, it is indicated by a check mark in the menu.

When the **Capitalize Bound Vars** option is on, the wffs are displayed with all occurrences of bound variables shown as capital letters X, Y, Z.

You should use the **Monochrome** option if you do not have a color display. When the Monochrome option is on, the graphs of the wffs use white for true and black for false. When the option is off, the graphs use green for true and red for false.

When the **Small** option is on, the graphs of the wffs are shown in a smaller size. The original large size will fit on a VGA display with the graphs in their original position.

Appendix J

GNUMBER – Gödel Numberer for DOS

J.1 Introduction

GNUMBER, the Register Machine Program with Gödel numbering, simulates a register machine, the basic tool in the study of computable functions. The title screen asks you to select either the Simple or the Advanced form of GNUMBER. The simple form can be used to enter instructions and register values and watch a register machine program run. The advanced form has additional features which let you manipulate Gödel numbers of register machine programs and get a close look at register machine programs which refer to themselves. Programs which refer to themselves lead to the striking results of Gödel which show that some problems are unsolvable.

The program runs on an IBM PC or compatible computer with at least 320K memory and one disk drive. If there is more memory, the program will have room for larger registers.

GNUMBER has a top level title screen and four modes of operation: Instruction editor, Program mode, Register mode, and Execution mode.

The program starts with the title screen, and then goes to the Program mode. You can change from one mode to another with the commands E, I, P and S. The command Q is used to quit the program.

To protect against accidental quitting, the first Q returns you to the title screen, and the program asks you to type Q a second time to be sure you really meant to quit. At the title screen you can return to the previous state or start a new tableau instead of quitting.

J.2 Getting Started

The program can be run from either a floppy diskette or a hard disk. With a diskette, put a diskette with the GNUMBER.EXE program file and the GNUM5 and GNUM6 example directories in the currently active drive. With a hard disk, either install the program as part of the Logiclab package by typing SETUPDOS.EXE at the DOS prompt, or copy the GNUMBER.EXE file and the GNUM5 and GNUM6 subdirectories to a hard disk directory entitled LOGICLAB (or another name of your choice). If you have a color display, type GNUMBER and hit Enter at the DOS prompt. If you have a monochrome display, type GNUMBER M and hit Enter. The title screen will appear.

J.3 Title Screen

The title screen appears when you initially start the program and when you use the Q command from within the program. At the initial title screen, you have the following choices:

S : Start the Simple GNUMBER program.

A : Start the Advanced GNUMBER program.

D : Change the drive or directory from which the examples and solutions are loaded and saved.

If you plan to use the example files in the GNUM5 or GNUM6 directories, you should type D at the title screen, and then specify the problem directory when you see "Enter new path". For example, if you are working from a diskette in the A: drive and wish to use the examples in GNUM5, type `A:\GNUM5` and hit the Enter key. If you are working from hard drive C and wish to use the examples in GNUM5, type `C:\LOGICLAB\GNUM5` and hit the Enter key.

Q : Quit the program.

When you return to the title screen from within the program, you have the following choices:

Enter key : Return to the current state without change.

A : Change to the Advanced GNUMBER program.

S : Change to the Simple GNUMBER program.

D : Change the drive or directory from which the examples and solutions are loaded and saved.

Q : Quit the program.

If your current work has not been filed, you will be given a warning and another chance to file the current program by hitting the F key.

The next few pages first explain what you can do in each mode with the simple form of GNUMBER. Then the additional features of the advanced form are described. If you are only using the simple form, you can skip the material on the advanced form.

J.4 Execution Mode

You can get to the Execution mode from the Program or Register modes with the E command. The Execution mode is the place where you run a register machine program. It has a variety of commands which allow you to start and stop the register machine program and control its speed. Two columns of 23 instructions without comments and one column of 15 registers are visible on the screen.

The register machine program is started by hitting either the space bar or the Enter key. As the register machine program runs at slow or one-step speed, the register contents and next instruction number are updated at each step. At the same time, the time counter at the top of

the screen shows the current number of steps in the run, and the next instruction label is highlighted with a white background. The motion of the highlighted label will give an indication of what the program is doing.

While the register machine program is running, it can be interrupted by pressing any key. You can go to another mode and make changes or explore the contents of a long register, and then return to the Execution mode and continue running the program.

At fast speed, the time counter is updated every 100 steps, and everything is updated when the program is stopped or interrupted.

J.4.1 Viewing More Instructions or Registers

The Execution mode initially shows the two columns of instructions from 0 to 45. To see the next column of instructions, hold the Ctrl key down and hit the right arrow key. To see the previous column of instructions, hold the Ctrl key down and hit the left arrow key. To go back to the first two columns of instructions, hold the Ctrl key down and hit the Home key. In this way you can focus on the part of the program where the action is and watch the program execute step by step.

The Execution mode initially shows one column of registers, from 1 to 15. You can view additional registers by hitting the PageUp or PageDown key.

J.4.2 Execution Mode Commands

I : Go to the INSTRUCTION Editor.

P : Go to the PROGRAM Mode.

R : Go to the REGISTER Mode.

Enter key or Space Bar : Run the current register machine program.

Q : QUIT. This command returns you to the title screen. From the title screen, you can quit the program by hitting Q again, change the drive or directory where the example files are loaded and saved, change to the simple or advanced GNUMBER program, or return to the current state.

O : Make the register machine program run ONE step at a time.

S : Make the register machine program run at SLOW speed. At slow speed you can see the changes in the register contents and the next instruction marker at each step.

F : Make the register machine program run at FAST speed. This is 6 to 20 times faster than slow speed. If you press the U (update) key while the program is running at fast speed, all the registers will be brought up to date and then the program will continue. If you press any other key while running at fast speed, the program will stop. (It can then be restarted by pressing the SPACE key). At fast speed you cannot see the changes in the register contents until you press the U (update) key, the program halts, a register overflows, or you stop the program. The time counter is updated every 100 steps while running at fast speed, but the next instruction marker is not changed.

T : Set the TIME counter and next instruction number to zero. Use this to start a program from the beginning.

J.5 Program Mode

You always start out in the Program mode, and can get there from the Execution mode or Register mode with the P command. In the Program mode, you can delete a single line or a whole register machine program, open a line for the Instruction editor, load a sample register machine program from the disk, or save a register machine program. There are places for 501 instructions, numbered from 0 to 500. Any instruction beyond 500 is assumed to be a Halt. The next instruction number is shown at the top of the screen. On the right side of the screen is a help window which has a list of the available commands. The letters C,D,E,F,H,I,L,O,P,Q,R, and U are used for these commands.

J.5.1 Moving Within the Screen

The Up, Down, Right, and Left arrow keys, the PageUp and PageDown keys, and the Home and End keys can be used to move within the register machine instruction area of the screen. The PageUp key goes up 23 lines. If you hold the Ctrl key down and press the PageUp key the cursor will move to instruction 0. The PageDown key moves down 23 lines. If you hold the Ctrl key down and press the PageDown key the cursor will move to the last nonhalt instruction. The Home key moves to the beginning of the current line, and the End key moves to the end of the current line.

J.5.2 Commands in the Program Mode

C : CLEAR all instructions to H, and set the time counter and next instruction number to zero.

D : DELETE the current instruction line, move all later lines up one, and adjust all J (Jump) instructions accordingly.

E : Change to the EXECUTION Mode.

F : File. Saves the current register machine program in a file on the disk. A box will appear with either a blank file name or with the name you used last time you filed the current RM program. The file name has the form XXXXXXXX.GN. Use the keyboard to enter or change the file name. (You should not enter the suffix ".GN"; the computer will add it automatically). When you have the name you want, hit the Enter key to save the program. You are warned if you try to use a file name which already exists. The Esc key cancels the File command, and goes back to the GNUMBER program without saving.

The F command can also be used to erase an unwanted GN file. To Erase a GN file, Quit and start an empty program (with no instructions), hit F for the File command, and type the name of the file you want to erase.

H : HALT instruction. This command erases the current instruction line and replaces it by an H for the Halt instruction.

I : Change to the INSTRUCTION Editor. The cursor will remain in its current position.

L : LOAD a register machine program. In the bottom window of the screen you will see the message

LOAD A REGISTER MACHINE PROGRAM AT LINE nn

where nn is the current line of the cursor in the Instruction Menu. The computer will show you a list of all files on the disk in the current directory whose names have the suffix .GN, and ask you to type in a file name and hit the Enter key. The register machine program described in the file will then be put into the instruction list, starting at the line nn. All old instructions from line nn to the end will be moved ahead to the end of the new program, and all jump instructions will be adjusted in the correct way. The next instruction number and time counter will be set to 0. You can get back to the Program Menu without loading a new program by hitting the Enter key without a file name. After you load a register machine program, its name will be displayed at the top of the screen. The name will stay there until you change a program instruction, file a program, or load a new program. If the file name you type is not on the diskette, or if there is not enough room to load the new program starting at line nn, you will be informed by a message and will return to the Program Menu with no change.

This command can be used either to load an RM program by itself, or to load an RM program somewhere in the middle of an old program. To load a program by itself, first press Home to get to instruction line 0, then press C to clear out the old instruction list, and then press L. To load a new program in the middle or at the end of an old program, move the cursor to the line where you want the new program to begin and then press the L key.

O : OPEN a line. This command moves all instructions below the current line down one, adjusts all J (Jump) instructions accordingly, and writes an H in the current line. Use this command when you want to insert a new instruction at the current line.

P : PRINT the current instruction list. (Ignored if no printer is installed).

Q : QUIT. Same as the Q command in the Execution Mode.

R : Go to the REGISTER Mode.

U : UNDOES the most recent change in the instruction list. The instruction list is returned to what is was before the most recent use of one of the commands C, D, H, I, L, or O. Use this instruction to recover if you accidentally press the wrong key.

J.6 Instruction Editor

You can get to the Instruction editor from any of the other modes with the I command. You leave the Instruction Editor by hitting the Enter key, which takes you to the Program mode. The Instruction Editor is used to type in or change register machine instructions and comments. The window on the right of the screen will list the available register machine instructions, H, J, S, T, and Z. The Esc key will undo the changes on the current line and return it to its previous state. When you are finished typing in or changing instructions, press the Enter key to return to the Program mode.

J.6.1 Register Machine Instruction Letters

The following register machine instructions can be entered in your programs. The table shows what each instruction does when r, s, and t are the numbers following the instruction letter and [r] is the number in register r.

INSTRUCTION		EFFECT
`H`	(Halt)	Stop.
`Z r`	(Zero)	[r] := 0.
`S r`	(Successor)	[r] := [r]+1.
`T r s`	(Transfer)	[s] := [r].
`J r s t`	(Jump)	if [r] = [s], jump to instruction t.

J.6.2 Entering Register Machine Instructions

When you start the GNUMBER program, there is an H command for Halt at every position. When you move the cursor to a new line in the Editor, the H disappears. You may type in a new instruction letter, a number for each place, and a comment of up to 40 characters. In the third place of the J command, an instruction number between 0 and 501 is needed. At any other place, a register number between 1 and 45 is needed. You can also place "break points" after the instruction letter by typing the symbol !. This will cause the register machine program to stop when you run the program.

To finish an instruction line, hit the Enter key to return to the Program mode, or the Up or Down arrow key, the PageUp or PageDown key, or the Ctrl key with the PageUp or PageDown key, to move to a new line. If your instruction is illegal, the computer will give you an error message. When you get an error message, you have three choices: 1) Correct the error. 2) Press the Esc key, which will undo your changes. 3) Press the Enter key, which will make an H instruction followed by your illegal instruction as a comment, and return to the Program mode.

J.6.3 Register Machine Program Files

There are two ways to create a register machine program file. You can either type in the program with the Instruction Editor and save it with the F command in the program mode, or you can use an ordinary word processor outside the GNUMBER program. If a word processor is used, each line must contain one program instruction letter (capital or lower case) and the required register or instruction numbers. A line number at the beginning and comments at the end are optional. The file must be given a name of the form XXXXXXXX.GN. When a program file is loaded with the L command, any illegal instructions will be replaced by an H command with a ! symbol and the original command and error message as a comment.

J.6.4 Advanced Instruction Letters

Two new 3-placed instruction letters, E and P, are available in the advanced form of GNUMBER. These instructions allow the manipulation of finite sequences of natural numbers.

By means of a Gödel numbering scheme, each natural number is also the code of a finite sequence of natural numbers. The Gödel numbering scheme uses the even decimal positions (starting from 0 on the left) as markers to show where a new term begins, and uses the odd decimal positions for the digits of the terms in the sequence to be coded. A 2 marker means that a new term is beginning, and a 1 marker means that the old term is continuing.

For example, the Gödel number of the sequence 5034 6 217 is (with the original digits underlined)

$$2\underline{5}1\underline{0}1\underline{3}1\underline{4}2\underline{6}2\underline{2}1\underline{1}1\underline{7}.$$

This is a Gödel number in standard form. In order to make every number a Gödel number of some sequence, the initial marker can be any digit except 0, a marker > 2 is identified with a 2, a 0 marker is identified with a 1, and an extra digit at the end is ignored. Any single digit number is a Gödel number of the empty sequence.

The E command EXTRACTS the [s]-th term from the sequence coded by [r] and places it in register t. (All terms beyond the last term of the sequence are considered to be 0). The P command PUTS the number [r] into the [s]-th term of the sequence coded by register t. The effect of these commands may be summarized symbolically, where (r) denotes the sequence with Gödel number [r].

INSTRUCTION	EFFECT
E r s t (Extract)	[t] := the [s]-th term of (r).
P r s t (Put)	The [s]-th term of (t) := [r].

It is possible to change back and forth between the Simple and Advanced forms of GNUMBER at the title screen without losing the current instruction list. It is also possible to enter the advanced E and P instructions even within the Simple Instruction Editor. The simple form of GNUMBER will treat an advanced RM program with the E and/or P instructions in the following way: all E and P instructions will be displayed as blinking characters and will be skipped when the RM program is executed.

J.7 Register Mode

You can get to the Register mode from the Execution or Program mode with the R command. In the Register mode you can put numbers into the registers. There are 45 registers, numbered 1 through 45. GNUMBER starts with 0 in every register. The help window below the registers lists the available commands.

The screen display of the Execution mode and the Register mode are the same except for the help window at the bottom of the screen, beginning with instructions 0 to 45 and registers 1 to 15. In the Register mode, you can view additional instructions by holding the Ctrl key down and hitting the left or right arrow or Home key.

J.7.1 Moving Within the Registers

The PageDown and PageUp keys display the next or previous group of 15 registers, 1-15, 16-30, and 31-45. The Up and Down arrow keys move the cursor up and down one row, and the Home key moves the cursor to register one. You can also get to the NEXT INSTRUCTION REGISTER (register 0) by going to register 1 and pressing the up arrow key.

J.7.2 Entering a Number into a Register

A number is entered into a register by typing the digits 0,...,9 as usual. You can enter a number into the Next Instruction Register as well as the ordinary registers. The backspace key works in the usual way. When you are finished entering the number, hit the Enter key, an Up or Down arrow, Home, PageUp, PageDown, or one of the commands E, I, P, or Q. You can enter up to 1,000 digits. While you are entering a number which is more than one line long, the screen shows you how many digits have scrolled off the left edge of the window.

J.7.3 Exploring a Register

After you or the computer finish entering a number, its total length (measured in digits) is shown at the extreme right of the screen. If a register contains more than a full line of digits, you can explore the contents of the register by using the right and left arrow keys and the End key (which displays the last 39 digits). This will cause the number to scroll horizontally and be displayed in white. The Enter, Up, Down, Home, PageUp, and PageDown keys and the I, M, and Q commands will leave the register and behave in the usual way.

J.7.4 Register Mode Commands

C : CLEAR all registers (put a zero in every register).

E : Go to the EXECUTION Mode.

I : Go to the INSTRUCTION Editor.

P : Go the the PROGRAM Mode.

Q : QUIT. Same as the Q command in the Execution Mode.

J.7.5 Advanced Register Mode Commands

There are four new commands which involve Gödel numbers.

A Gödel number is assigned to a register machine program in the following way. Each register machine instruction is a sequence consisting of a letter and from 0 to 3 numbers. The instruction letters H,Z,S,T,J,E,P are assigned the codes 1 through 7 respectively. This makes each register machine instruction a sequence of from 1 to 4 numbers, and this sequence is assigned its Gödel number. The instruction list is considered to end at the last nonhalt instruction. The register machine program is a finite sequence of instructions, which gives rise to a finite sequence of Gödel numbers that in turn has a Gödel number.

G : Put the GODEL number of the register machine program shown in the current instruction list into the current register. The Gödel number will be in standard form. The RM program is taken to be the list of all instructions through the last nonhalt instruction. All the subsequent halt instructions are ignored in computing the Gödel number. A program which has only halt instructions has Gödel number zero. This command also sets the time counter and next instruction to 0, and leaves all other registers unchanged.

U : (UNGODEL) Put the register machine program whose Gödel number (not necessarily in standard form) is in the current register into the instruction list. A term in the current register sequence which is not a Gödel number of an instruction is treated as the end of the instruction list, and all later terms will be ignored. This command also sets the time counter and next instruction register to 0, and leaves all other registers unchanged.

S : Change to SEQUENCE display. This command causes any number which has more than 3 digits and is the Gödel number of a sequence in standard form to be displayed as a sequence of numbers separated by commas. All other numbers will still be displayed in the ordinary way. When you explore a register containing a sequence, the right and left arrow keys move to the beginning of the next or preceding term of the sequence, and the End key moves to the beginning of the last term of the sequence.

In the Advanced Register mode you can enter numbers into registers in sequence form as well as in number form. To enter a sequence into a register, first type the left parenthesis "(" and then type in the terms of the sequence separated by commas. When you are finished entering the sequence, type either the right parenthesis ")", the Enter key, an Up or Down arrow, Home, PageUp, PageDown, or one of the commands E, I, P, or Q. The register will contain the Gödel number of the sequence.

N : Change to NUMBER display. This command causes the numbers in all registers to be displayed in the usual way as ordinary numbers.

The last line in the help window tells you whether the Number or Sequence display is being used.

The next to the last line in the help window displays more information about the current register. If the computer has enough memory, the first few registers will have room for 15,000 digits instead of 1,000 digits. The amount of room in the current register is reported. If you are exploring a register in a number display, the number of digits and the place of the first visible digit are reported. If you are exploring a register in a sequence display, the number of terms, the place of the first visible term, and the length (number of digits) of the first visible term are reported.

J.8 Changing Directories

When you start the program by typing GNUMBER and hitting the Enter key, the program will use the currently active drive or directory for loading the example files and saving solutions. You can change drives or directories from within the program at the title screen by following the directions in Section J.3. A period "." can be used for the current directory, and a double period ".." for the parent of the current directory.

You can start the program with another drive or directory for example files by typing GNUMBER followed by the desired path name and hitting the Enter key. For example, to load the GNUM5 directory of GN files, you would type GNUMBER GNUM5 M at the DOS prompt and then hit the Enter key. This feature may be useful in a computer lab setting. The path and the M option for a monochrome display can be combined or used separately.

For example, the instructor may create a batch file called GNU.BAT which has the single line

GNUMBER M A:

If the student types GNU [Enter key], the program will run with a monochrome display and will use diskette drive A: for the example files.

Appendix K

GNUMWIN - Gödel Numberer for Windows (R)

K.1 Introduction

GNUMWIN is the version of the GNUMBER program which works with Microsoft (R) Windows, Version 3.0 or later, and with Windows 95.

The GNUMWIN program can only be started after Windows is running. It can be operated with a mouse or with the keyboard, and works like other Windows applications. The GNUMWIN.EXE program and the GNUM5 and GNUM6 directories can either be copied to your hard disk into a directory called LOGICLAB (or another name or your choice), installed using the SETUPWIN.EXE program, or accessed directly from the diskette. In all of these cases, access the Windows File Manager (in Windows 3.0 or later) or My Computer (in Windows 95), select the disk drive and directory that contains the program, and then select GNUMWIN.EXE.

The program will begin with a welcome message in a small window with two buttons labeled "Start" and "Tutorial". Click the mouse on the "Tutorial" button to get a quick introduction. Click the mouse on the "Start" button or hit the Enter key to begin the program in the normal way.

There are two main windows, a program window with instructions

0 to 999, and a register window with registers 0 to 99. Register 0 is the program counter, and the others are data registers. Each data register can hold a natural number with up to 20,000 digits.

The main menu at the top of the screen has a File menu, a Program menu, a Registers menu, a Windows menu, an Options menu, a Step command, a Go menu, and a Help menu.

The program counter, register 0, holds the number of the next instruction to be executed. It can always be seen at the top of the registers window. A register machine (RM) program starts with a 0 in the program counter. During the execution of an RM program the contents of the program counter and the data registers change but the instructions remain fixed.

The labels of the next instruction (in the program counter) and one register are marked in reverse video. They can be changed by clicking the mouse button on a new label, or by using the arrow, Page Up or Down, Home, or End keys. The tab key switches the arrow key action between the two windows. The scroll bars move the window up or down without changing the marked label. The Enter key moves both windows to the marked label.

The simple register machine has 5 instruction types, H (halt), Z (zero), S (successor), T (transfer), and J (jump). When one of these instructions (other than halt) is executed, it causes a change in the contents of the program counter holding the next instruction, and may change the contents of a data register.

The advanced register machine has two additional instructions, E (extract) and P (put). These instructions manipulate the contents of the data registers as Gödel numbers of finite sequences.

The time display at the top of the Program window is set to 0 at the start of execution, and increases by 1 at each RM program step. It shows you how long an RM program has been running. An RM program will run until a halt instruction is encountered, a data register overflows, or the user intervenes.

The **Help menu** can be reached by using the mouse or the F1 key. It contains five lists of topics and an **About** command, which shows the version number, copyright notice, and icon.

K.2 Program Execution

The initial values in the data registers can be entered by hand or with the Gödel command. The RM instruction list can be entered by hand, loaded from the disk, or created with the unGödel command. You can then begin execution. The step command on the main menu executes a single instruction. The Go menu has several choices for starting automatic execution.

You can stop the automatic execution of an RM program by hitting any key except U, or by hitting the left mouse button in either the Program or Registers window. Before starting automatic execution, you can set a stopping time, or set a break point by entering a ! sign after an instruction letter in the Program window. You cannot exit the GNUMWIN program while automatic execution is in progress.

The standard Gödel number of a finite sequence of natural numbers is a single natural number which codes the whole sequence. Example: The sequence (345,8008,7) has Gödel number 2314152810101827. (Every other digit is a marker.) A single instruction is a sequence of at most four numbers, (replacing the letters H, Z, ...,P by 1 through 7). Each instruction thus has a Gödel number. The Gödel number of a whole RM program is the Gödel number of the sequence of Gödel numbers of its instructions.

K.3 Register Machine Instructions

Halt: H stops the RM program.

Zero: [Z i] places a 0 in register i and increments the program counter by 1.

Successor: [S i] increments register i by 1 and increments the program counter by 1.

Transfer: [T i j] places the contents of register i into register j and increments the program counter by 1.

Jump: [J i j k] places k into the program counter if the contents of registers i and j are equal, and otherwise increments the program counter by 1.

Extract: [E i j k] If register i contains the Gödel number of a

sequence I and register j contains the number J, then the Jth term of the sequence I is placed in register k. (If I has no Jth term, 0 is placed in register k.) Then the program counter is incremented by 1.

Put: [P i j k] If registers i and j contain numbers I and J, and register k contains the Gödel number of a sequence K, then the standard Gödel number of the sequence L formed from K by replacing the Jth term of K by I is placed in register k. (If K has fewer than J - 1 terms, extra 0 terms are added to the end of K before forming L. Then the program counter is incremented by 1.

K.4 File Menu

The **New** command starts a new RM program. The current instruction list and all registers will be cleared. You will be warned if the current RM program has not yet been saved on disk since the last change.

The **Open...** command loads a .GN file from the disk, containing an RM instruction list. You will be warned if the current RM program has not yet been saved on disk. You will see a list of .GN files and of directories, which may include the parent directory called ".." and the GNUM5 and GNUM6 directories which contain example files. Choose a file or new directory from the list, or type it on the screen. The loaded instructions will be joined with the current RM program at the marked instruction. The window takes the loaded file name as its title if you load at 0 over an empty instruction list, but will drop this title when the instruction list is changed.

The **Save** command will save the current RM program under the current title which appears in the main window caption. If the program is [untitled] you should use the Save As command instead of the Save command.

The **Save As...** command will save the current RM program under a name which you will supply. The window takes this name as its title, which will be used for subsequent save commands. If you only give the first part of a file name, the .GN extension will be added automatically.

The **Print** command will print the current RM program. The beginning and end of the RM program will be marked with a row of dots. You will be warned if the printer is not ready.

The **Exit** command will quit the GNUMWIN Program and return to Windows. You will be given a warning and a chance to save the current RM program if it has not been saved on disk since the last change.

K.5 Program Menu

The **Edit an Instruction** command opens a box in which you can type in an RM instruction and a comment to be placed at the marked position in the instruction list. You can also do this by double clicking with the left mouse button on the instruction label.

The **Clear an Instruction** command replaces the marked instruction by a Halt.

The **Delete an Instruction** command removes the marked instruction, and closes up the gap by moving all lower instructions up one position and fixing the jump targets as necessary.

The **Insert an Instruction** command moves the marked instruction and all later instructions down one position, fixing the jump targets as necessary, and inserts a Halt at the marked place.

The **Clear All Instructions** command replaces all instructions in the current list by Halts. You will be warned if the current list has not been saved since the last change, and given a chance to save it. This command is useful when you want to load a new RM program on a clean slate.

The **Undo** command undoes the last change in the instruction list and moves the marked instruction and program counter to the position at the time of the last change. This command does not affect the contents of the data registers.

K.6 The Registers Menu

The **Edit a Register** command opens a box in which you can enter or change the contents of the marked data register (or the program counter). You can also do this by double clicking the left mouse button on the register label.

The **View a Register** command opens a box with a horizontal scroll bar in which you can view the entire contents of the marked register.

The **Clear a Register** command places a 0 in the marked register.

The **Clear All Registers** command places a 0 in every register.

The **Gödel** command places the Gödel number of the current RM program in the marked register. This command can be used to test RM programs which take Gödel numbers of other RM programs as inputs.

The **UnGödel** command replaces the current instruction list by the RM program whose Gödel number is in the marked register. This command can be used to test the action of RM programs which compute Gödel numbers of other programs.

K.7 Windows Menu

The **Vertical Tile** command arranges the windows with the Program window on the left with its current width, and the Registers window taking up the remaining space on the right. Each window gets at least 1/5 of the total available space.

The **Horizontal Tile** command arranges the windows with the Program window on the top with its current height, and the Registers window taking up the remaining space on the bottom. Each window gets at least 1/5 of the total available space.

The **Move Program** command lets you move the Program Window with the keyboard or mouse.

The **Size Program** command lets you change the size of the Program Window with the keyboard or mouse.

The **Move Registers** command lets you move the Registers Window with the keyboard or mouse.

The **Size Registers** command lets you change the size of the Registers Window with the keyboard or mouse.

K.8 Options Menu

In this menu you can choose between the Numbers and Sequences options, and between the Show Comments and Compress Instructions options.

With the **Numbers** option, the data registers are displayed in the usual way as numbers. The GNUMWIN program starts out with this option.

With the **Sequences** option, the data registers which contain standard Gödel numbers of sequences, and at least 3 digits, are displayed as sequences. The other data registers are displayed as numbers in the usual way.

With the **Show Comments** option, the comments are shown next to the instructions in the Program window. The GNUMWIN program starts out with this option. The **Compress Instructions** option hides the comments in the Program window and instead uses the space to show as many columns of instructions as possible.

K.9 Step Command and Go Menu

The **Step** command will execute the next RM instruction only.

The Go menu controls automatic execution of the current RM program. The Slow, Medium, and Fast commands start automatic execution, and any key or the left mouse button will stop it. You must stop automatic execution to change from one speed to another, or to exit the GNUMWIN program. During automatic execution, you can minimize the GNUMWIN program and turn to other tasks. You will see a changing icon which displays the first 6 digits of time during automatic execution. The normal icon reappears when automatic execution stops.

The **Restart** command will change the time and program counter to 0. This prepares for restarting the current RM program at instruction 0.

The **Stopping Time** command will let you enter a stopping time. The next automatic execution will stop when the time reaches the stopping time.

The **Slow** command will begin automatic execution of the current RM program at the slow rate of about two steps per second.

The **Medium** command will begin automatic execution of the current RM program as fast as possible while still displaying all changes in the data registers and the marked next instruction label. The speed will depend on the capabilities of your computer.

The **Fast** command will begin automatic execution, gaining speed by not updating the Registers window, hiding the next instruction mark, and showing the time only in multiples of 1000. The Registers window will be updated without stopping execution when the U key is hit. The commands in the Program menu are disabled during fast automatic execution. Again, the speed depends on your computer.

Bibliography

Boolos, George

[1979] *The Unprovability of Consistency* (Cambridge University Press).

[1989] A New Proof of the Gödel Incompleteness Theorem, *Notices of the American Mathematical Society, Volume 36, Number 4*, 388-390.

Enderton, Herbert

[1972] *A Mathematical Introduction to Logic* (Academic Press).

Gödel, Kurt

[1940] The Consistency of the Axiom of Choice and of the Generalized Continuum Hypothesis with the Axioms of Set Theory, *Annals of Mathematical Studies, Volume 3* (Princeton University Press).

Hofstadter, Douglas

[1979] *Gödel, Escher, Bach: An Eternal Golden Braid*, (Basic Books, Inc.).

Levy, Azriel

[1972] *Basic Set Theory* (Springer Verlag).

Rucker, Rudolf

[1982] *Infinity and the Mind, the Science and Philosophy of the Infinite* (Birkhäuser).

Smorynski, C.

[1985] *Self-Reference and Modal Logic* (Springer-Verlag).

Smullyan, Raymond

[1987] *Forever Undecided, A Puzzle Guide to Gödel* (Alfred A. Knopf, Inc.).

Index