MATHEMATICAL LOGIC FOR COMPUTER SCIENTISTS

by

MICHAEL LEVIN

The work resulting in this publication
was done at M.I.T's Project MAC and at the Stanford
Artificial Intelligence Laboratory, and was supported
in part by IBM

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Cambridge, Massachusetts, 02139

ABSTRACT

This book is an introductory course in mathematical logic covering basic topics in quantification theory and recursive function theory, and is intended for the reader who is interested in artificial intelligence, computer linguistics, and other related areas. The text is theoretical, but organized with implementation in mind. Toward the end there are a few experimental subjects aiming toward systems that can examine their own behavior, and toward the semantics of programming languages. The arithmetization of metamathematics is carried out in LISP rather than in the natural numbers, following an axiomatic treatment of LISP.

ACKNOWLEDGEMENTS

To Ed Fredkin, who is not a mathematician, for having the confidence that I could write this book.

To my students at MIT, for their help and interest in this work.

To Delphine Radcliffe, for her expert typing, composing and spelling, and her patience and hard work while preparing two drafts.

PRELIMINARY EDITION

The author will be pleased to read any comments, criticisms, suggestions or errata that you may have. Address correspondence in care of Project MAC, 545 Main Street, Cambridge, Mass. 02139.

[Dhar Lho said], "Logic is the most important science of all learning. If one knows logic, all other studies become secondary. Therefore, I shall first discuss logic with you. Generally speaking, logic is the study of judgement and definitions, of which the most important subjects are the studies of direct experience, of inference and deduction, of 'sufficient reasoning' and 'false reasoning', of 'non-decisive proofs', and of the patterns for constructing propositions. Now, tell me about all these things!"

[Milarepa replied . . .], "What I understand is that all manifestations [consist in] Mind, and Mind is the Illuminating-Voidness without any shadow or impediment. Of this truth I have a decisive understanding; therefore not a single trace of inference or deduction can be found in my mind. If you want me to give some examples of 'false-reasoning', your own knowledge is a good one because it is against the Dharma; and since this 'false reasoning' only enhances your cravings and makes them 'sufficient', it is a good example of 'sufficient reasoning'. Your hypocritical and pretentious priestly manner contains the elements of both 'false' and 'sufficient' reasoning, which in turn stand as a good example of 'non-decisive proof'."

-The Hundred Thousand Songs of Milarepa-

PREFACE

I would like to discuss first the contents of this book, and then the attitudes behind it.

The first two chapters are about LISP. In Chapter One, s-expressions are introduced as a data space, and the basic functions of s-expressions are presented. In Chapter Two, recursive procedures are explained, and the recursive functions of s-expressions are defined as those for which such procedures can be written. There is a discussion of why these appear to include all effectively computable functions (Turing's and Church's theses).

Chapters Three and Four are about propositional logic. Chapter Three introduces the notion of a deduction from given premises leading to a conclusion, and establishes the fact that deductions are mechanical procedures that can be checked for correctness by a computer program. Chapter Four considers theories in propositional logic and models for propositional logic, and contains consistency and completeness theorems. All of this is a dress rehearsal for first order logic, where the same themes will be repeated in a richer setting.

Chapter Five is a brief interruption of the development of deductive systems to discuss the concepts "recursive" and "recursively enumerable", and to demonstrate the existence of undecidable questions such as Turing's halting problem.

The central portion of the book is about first order quantification theory, specifically first order languages with function and predicate names. Chapter Six introduces first order languages, first order models, and the semantic notion of satisfiability. Chapter Seven defines deduction, proves it to be semantically consistent, and presents a number of standard proof-theoretic results, including the deduction theorem, replacement of equivalents, change of bound variables and the choice rule. Chapter Eight contains the

completeness theorem for first order logic in several different forms, together with related results such as compactness and the Skolem-Löwenheim theorem. Chapter Nine is a bundle of loose ends, which includes the extension of first order theories by means of conservative definitions, decidability, and comparisons with other deductive systems, namely Robinson's resolution, and Gentzen-type systems.

The next topic is the theory of arithmetic, which is the arithmetic of the s-expressions. The arithmetic of the natural numbers is treated as a special case of this. The theory is presented informally in Chapter Ten, and as a formal first order theory in Chapter Eleven. In both cases, there is an emphasis on the strong analogy between Peano's postulates for the natural numbers, and the corresponding postulates for s-expressions.

Chapter Twelve is concerned with the representation of recursive functions in the first order theory, which is then used to prove the incompleteness of arithmetic in three different ways: The first way is by constructing the LISP analogue of Gödel's undecidable sentence. The machinery to do this comes naturally, because it is none other than an updated version of the proof-checker discussed in Chapter Three. Representing formulas and deductions by s-expressions is not nearly so strange or impractical as representing them by Gödel numbers. The second method of proving the incompleteness of arithmetic is by representing computation (as distinct from deduction) in the deductive system, and mapping the halting problem into first order arithmetic. The third method uses an "information theoretic complexity" approach due to Chaitin. The argument advanced here is that these incompleteness results are not irrelevant theoretical considerations, but rather that they illustrate the richness of arithmetic, and introduce new (meta) ways of reasoning.

This idea is followed up in Chapter Thirteen, which presents a formal axiomatic "metamathematics" which can be used to reason about formal arithmetic, and to produce proofs of the existence of proofs which are generally much shorter than the original proofs. There is a hierarchy of metamathematical levels, in that one can prove that there is a proof that there is a proof of some formula. This technique also enables one to prove the validity of theorem schemas and derived rules of inference.

Chapter Fourteen is about the recursion theorem, and its usefulness for representing partial recursive functions in first order arithmetic. This chapter is useful as background for studying current research in the semantics of programming languages.

Chapter Fifteen contains some concluding remarks about second order arithmetic and axiomatic set theory.

* * *

Once upon a time, it was believed that the problem of getting a machine to behave intelligently would be solved by starting with a small kernel of intelligence capable of learning, reasoning, and organizing itself as it grew. There were several variants of this idea, and some of them assigned an important role to a "proof procedure" that could create demonstrations of logical propositions.

This approach is now considered naive and simplistic. As researchers have immersed themselves in the task of simulating detailed aspects of human speech and perception, there has been a growing appreciation of the complexity and subtlety of these acts, and the large amount of detailed knowledge that seems to underlie the phenomenon of intelligence. Artificial intelligence research is now detailed and nitty-gritty rather than vague and general. How do light and shadow allow us to find the edges of a block? How does the preformed concept of a block allow us to infer one from some edges? How do we determine the antecedent of a personal pronoun?

Artificial intelligence is also becoming a more structured discipline, not as a universal mathematical theory, but as an epistemological and psychological theory. One of the main developments of the last few years is the recognition of knowledge as being procedural rather than merely factual. Knowledge is not a body of facts but, rather, what one does with one's facts and situations. Such a study cannot help but run into the problem of intentionality. It is no accident that phenomenology, gestalt psychology, and the developmental epistemology of Piaget are now seen by many workers as relevant to artificial intelligence research.

The logistic approach to artificial intelligence is severely and, in my opinion, correctly criticized in Marvin Minsky's currently unpublished "Frames" paper. Almost all of the criticisms are related to the fact that

logical deduction isolates the factual information or axioms from the methods of reasoning or rules of inference. (i) This separation forces one to represent knowledge about the world as a large body of independent statements. Without a structure governing their relations, there is no way of selecting the relevant facts from among all the possible ones, and so attempts at deduction run into a combinatory explosiveness. (ii) Many "facts" are true only when used in a reasonable way. Minsky uses the example of "nearness" which is transitive in the sense that if A is near B, and B is near C, then A is near C. This bit of reasoning works as long as it is not carried too far. In principle, one can always make a more precise formulation of any axiom by adding more parameters. But this seems to be unrealistic and, in any case, people do not make use of deduction beyond the point of common sense. (iii) Deduction is monotonic in the sense that adding new axioms allows one to make new inferences, but does not prevent one from making any of the old ones. If a general rule turns out to have exceptions not foreseen at the time it was postulated, there is little one can do except change the original rule, and recheck everything one has done so far for correctness. The rules of logic do not permit one to make restrictions concerning the inappropriateness of certain deductions. (iv) Consistency and completeness do not appear to be desirable properties of a practical system of reasoning because there is no way to organize a body of real knowledge that is either consistent or complete. For example, human reasoning appears to make use of some of the principles of set theory, but has no specific safeguard that prevents the paradoxes of naive set theory. If someone is informed of Russell's paradox, he may either develop a critique of it or simply ignore it and go about his business. But in no case will the existence of the paradox interfere with his reasoning about ordinary situations. [1]

* * *

---

[1] Minsky writes "I regard the recent demonstration of the consistency of modern set theory, thus, as indicating that set theory is probably inadequate for our purposes--not as reassurance that set theory is safe to use!" Minsky is referring to the work of Yessenin-Volpin, who curiously enough is saying much the same thing. Following a famous result of Gödel, the consistency of ZF (axiomatic set theory) cannot follow from any argument that can be formalized within ZF itself. Since ZF is intended to incorporate all the set theoretic principles that mathematicians need to do their work, this

The question, then, is why study mathematical logic at all and, in particular, why should there be a book organized as if the most important task to be done is to create an automated proof-checker capable of axiomatizing systems of knowledge of almost any kind? (A proof-checker, as distinct from a proof-procedure, doesn't have the smarts to create a proof. It merely forces the intelligent human or other proof-generator to be completely precise, and perhaps it fills in the gaps in the proofs if they are not too difficult.) I think that the answer to this question is not that such a project ought to be undertaken, but that the presumption involved is contained within logic itself, and goes back at least as far as Descartes, if not Aristotle.

The logistic method is an attempt to grab a hold on the world by reducing it to premises, inferences and conclusions. This is not always a healthy way of relating to the world. I think that part of Dhar Lho's error was in not seeing this. Formal logic is the necessary consequence of informal logic, and automated logic is the necessary consequence of formal logic. The nature of the fruit is in the seed, and the mature fruit tells us something about the seed, as well as vice versa.

Formal mathematical logic can be viewed as a structure, interesting in itself. But there is always a motive for one's choice of structures to develop. In the case of first order logical theories, this motive is the notion that, at least in principle, entire areas of mathematics can be formalized axiomatically in first order logic, and their theorems proven within it. Carrying this one step further, there is the ambition to axiomatize "real" situations in the same way.

It is for this reason that the later chapters of this book are aimed in the direction of a large and unsolved problem which the professional logicians have not been overly interested in solving. How can a deductive system incorporate within itself those metamathematical processes which are

---

has discouraged logicians from expecting to be able to prove the consistency of ZF. But Yessenin-Volpin writes that ZF is "not so expressive as is commonly believed". His consistency proof (which is too new and unusual for there to be any adequate professional evaluation at this time) uses techniques that are startling to mathematicians, but possibly relevant to Minsky's discussion, which he calls "tactics of attention", and which relate the deductive process to questions of modality and intention.

necessary to the work of a real mathematician, and do so in such a manner that new mathematical tools are proven to be valid before they are used? If mathematical logic does not investigate this problem soon, it will have failed to mature its most important concept, which is the applicability of the axiomatic method.

In stressing this point, I am guilty of some confusions and inaccuracies which will be evident to any trained mathematical logician. Questions of foundations have been obscured by using axioms and definitions that are too strong. The distinction between finitary and set theoretic reasoning, and the historic context that makes this distinction important have not been made clear enough. My decision to allow definitions into theories has converted them into temporal or developmental entities, which is not as neat as the standard treatment of theories, although it is more practical and realistic. Some of the proofs of theorems are a bit sketchy and occasionally non-existent. This is especially true if the theorem asserts that there is an effective procedure that does such and such. The book is written for people with computational experience to whom such things are self-evident. On the whole, I think that this book is a useful introduction to logic from one point of view. The student who then wishes to continue his study of mathematical logic will have little difficulty in making the transition to the more standard presentation.

* * *

In some sense, then, this book is not about what its contents appear to be. The reader will have to form his own opinion concerning the relevance of logic to artificial intelligence or any other endeavor. If he is interested, this book will lead him through a maze of particulars and details, and will suggest some ways in which to organize this experience. Because logic is so abstract, it generally turns out that anything which is a real problem in logic will present itself elsewhere in some other form. You will have to ask yourself what is the relation between quantification, and space and time, or what is the relation between the deduction theorem, and modalities of speech such as the subjunctive, or whatever else it is that you notice while studying logic. Good luck!

PREREQUISITES

It is assumed that the reader has a background and interest in computer programming, and that he has mathematical aptitude. The text assumes no specific mathematical knowledge other than those fundamental concepts basic to all of theoretical mathematics, but some mathematical sophistication is expected.

LISP is used extensively, and it will help to have programmed in LISP. But this is not essential because LISP is developed in the first two chapters. There is some reference to common programming ideas such as ALGOL, call by value and name, procedures, etc.

The mathematical prerequisites can all be found in [Halmos].[1] They include:

Sets, subsets, membership, union, intersection, complement, power set.

Function, domain, range, argument, value.

Cardinality, finite, infinite, countable, uncountable, diagonalization.

Equality, equivalence relation, partition, coset.

Mathematical induction.

Partial ordering, linear ordering, upper and lower bound, greatest upper bound, and greatest lower bound.

Some familiarity with symbolic logic will be useful, but not essential. References to other mathematical subjects such as analysis, topology or ordinal numbers are mainly used as illustrations that may be skipped over.

---

[1] When a reference is made in square brackets, the complete citation can be found in the bibliography, listed alphabetically by author's name.

CONTENTS

		Page
CHAPTER ONE	SYMBOLIC EXPRESSIONS	
§1.1	S-expressions	1
§1.2	Basic Functions of S-expressions	6
CHAPTER TWO	RECURSIVE DEFINITIONS	
§2.1	Functions	13
§2.2	Recursive Definition	14
§2.3	Partial Recursive Functions	23
§2.4	A Universal LISP Function	24
CHAPTER THREE	PROPOSITIONAL LOGIC - I	
§3.1	Propositional Formulas	32
§3.2	Interpretation	34
§3.3	Deduction	37
§3.4	Proof-Checking	39
CHAPTER FOUR	PROPOSITIONAL LOGIC - II	
§4.1	Proof Theory	42
§4.2	Model Theory	46
§4.3	Consistency and Completeness	50
CHAPTER FIVE	RECURSIVE FUNCTIONS AND SETS	
§5.1	Recapitulation	52
§5.2	Turing's Halting Theorem	53
§5.3	Recursive and Recursively Enumerable Sets	55
CHAPTER SIX	FIRST ORDER LOGIC - INTRODUCTION	
§6.1	Languages, Formulas and Sentences	58
§6.2	First Order Models	61
§6.3	Theories	65
CHAPTER SEVEN	FIRST ORDER LOGIC - DEDUCTION	
§7.1	Substitution	69
§7.2	The Rules of Deduction	72
§7.3	The Consistency Theorem	75
§7.4	Existence of Deductions, Replacement	77
§7.5	The Deduction Theorem	82
§7.6	The Choice Rule	85
CHAPTER EIGHT	FIRST ORDER LOGIC - COMPLETENESS	
§8.1	Completeness	89
§8.2	Equality	93
§8.3	The Skolem-Löwenheim Theorem	95

Page

CHAPTER NINE FIRST ORDER LOGIC - ADDITIONAL TOPICS

§9.1 Definitions 98
§9.2 Herbrand's Theorem 103
§9.3 Substitution and Unification 106
§9.4 Resolution 110
§9.5 Gentzen-Type Systems 115
§9.6 Decidability 116

CHAPTER TEN INFORMAL ARITHMETIC

§10.1 The Postulates of Arithmetic 118
§10.2 Primitive Recursion 120
§10.3 Other Arithmetics 122

CHAPTER ELEVEN FORMAL ARITHMETIC

§11.1 Multi-Typed Logic 125
§11.2 Axioms for the Theory of Arithmetic 127
§11.3 Development of the Theory 133

CHAPTER TWELVE RECURSION AND DEDUCTION

§12.1 Expressibility and Representability 137
§12.2 Primitive Recursion 139
§12.3 The Incompleteness of Arithmetic 140
§12.4 Representability of Recursive Functions 147

CHAPTER THIRTEEN METAMATHEMATICS

§13.1 Truth and Tarski's Theorem 149
§13.2 Metamathematical Deduction 152
§13.3 The Hierarchy of Truth 155

CHAPTER FOURTEEN THE RECURSION THEOREM

§14.1 The Nature of the Problem 156
§14.2 The Recursion Theorem 159
§14.3 Application of the Recursion Theorem 164

CHAPTER FIFTEEN SECOND ORDER ARITHMETIC AND SET THEORY

§15.1 Second Order Arithmetic 170
§15.2 Axiomatic Set Theory 173

BIBLIOGRAPHY

This empty page was substituted for a blank page in the original document.

CHAPTER ONE

SYMBOLIC EXPRESSIONS

Preview of Chapters One and Two

Chapter One introduces the basic data of LISP which are called s-expressions, and a set of basic functions of s-expressions from which one may construct many other LISP functions. Chapter Two introduces a simple language, recursive in nature, in which one can describe precisely how to compute a complicated function from the basic functions. It is important to learn this material thoroughly before proceeding further in this book because LISP will be used in relation to all the subsequent topics of discussion, and because, as we shall see later, LISP itself is the subject of a theory which is as elegant and simple in its postulates as is number theory.

Pedagogically, it makes sense to have some practical experience with a subject before attempting a theory about it. For example, numbers and the use of numbers are taught in elementary school, while number theory is typically a college level subject. Therefore, it is important to make use of these two chapters and their exercises to acquire some basic skill with s-expressions.

If you are already a LISP programmer, just skim through the two chapters and note that some of the definitions used here differ from the programming system you are used to, and that many parts of the language have been omitted.

§1.1 S-expressions

The basic units from which s-expressions are built are called atoms.

We shall define atoms, and then show how to build larger s-expressions from these. Atoms are of two kinds: names and numbers.

A name is any sequence of one or more capital letters and digits which begins with a capital letter.

A positive number is any sequence of one or more digits that does not begin with 0.

Zero (0) is also a number.

The positive numbers together with 0 are called natural numbers. While we could define many other kinds of numbers, in this book we shall always mean natural number when we say number unless we specify otherwise. Therefore:

A number is a positive number or zero.

There are many types of entities which can be and are considered atoms in various LISP systems. Once again we shall restrict ourselves to the minimal structure required by the subject matter of this book. Therefore:

An atom is either a name or a number.

Examples of atoms:

A	ABC3
REDBOX	SAM
6	AQ34500J7
0	CAMBRIDGE

We now proceed to s-expressions which are the main subject of this chapter. An s-expression is a tree-like structure created entirely from atoms placed in a particular arrangement. Parentheses, dots, and the

spaces used to separate one atom from another are used to specify this arrangement.

An s-expression is either an atom or else it is a structure having the form (α . β) where both α and β are s-expressions.

This is an example of an inductive definition. From it, we can infer that A is an s-expression because A is an atom. Similarly, B is an s-expression. Therefore (A . B) is an s-expression. Applying the definition again, since (A . B) and C are s-expressions, ((A . B) . C) is also an s-expression.

Examples of s-expressions:

A	XYZ
(A . XYZ)	((A . XYZ) . A)
(A . (XYZ . A))	(A1 . (A2 . (A3 . NIL)))
((A1 . A2) . (B1 . B2))	((A . A) . (A . A))

Since we shall frequently make use of this kind of definition, it merits some discussion. It is a common practice among mathematicians to limit such a definition by adding: "...and nothing else is an s-expression". We shall always assume this to be the case.

It is possible to conclude from the definition that all s-expressions have the same number of left and right parentheses. This is because (a) all atoms have the same number of left and right parentheses, namely none, and (b) any other s-expression has the form (α . β) where α and β are s-expressions. If this proposition is true for α and β, then it is certainly true for (α . β) which adds one more parenthesis of each type. It is also evident that each left parenthesis is paired with a unique right parenthesis, namely the first right parenthesis encountered by making a left to right scan starting at the given left parenthesis such that all the intervening parentheses are paired.

Notice that both (A . (B . C)) and ((A . B) . C) are s-expressions, and that they are considered to be different s-expressions. The mathematical principle which asserts that algebraically X+(Y+Z) is the same as (X+Y)+Z is called associativity. The composition of s-expressions is not associative.

One more comment, this time on the use of Greek letters. A Greek letter is never part of any s-expression, or for that matter any other type of entity constructed anywhere in this book. It is purely an explanatory device, as in the previous definition where we say "let α and β be any s-expressions".

Problem Set 1

1. Which of the following are s-expressions?

a. ABC
b. 35A
c. (A . B)
d. (A . B))
e. (A . B . C)
f. ((A . B) . C)

2. How many different s-expressions are there that use the atom "A" exactly n times and contain no other atoms? (Call this function $\pi(n)$. Don't try to find an algebraic formula for $\pi(n)$ which may not exist, but learn how to compute $\pi(n)$ when you know the values of π for all numbers less than n.)

The examples of s-expressions which have just been given are all written in what we call dot notation. There is another "shorthand" notation for writing s-expressions called list notation. It is more convenient and is more generally used. However, we are not introducing any new s-expressions. Every s-expression can be written using only dot notation, but many s-expressions are much easier to write in list notation. Some s-expressions cannot be written in list notation.

Although list notation is most commonly used, dot notation is considered more basic. Theoretical properties of s-expressions are resolved by referring to dot notation.

In the list notation, a special status is given the atom NIL as the terminator of lists. A list is an expression having the form $(\alpha_1\ \alpha_2\ \ldots\ \alpha_n)$ where each α_i is an s-expression. In other words, a list is just several s-expressions enclosed between a set of parentheses, with spaces between them. This list is the same s-expression as $(\alpha_1 . (\alpha_2 . \ldots . (\alpha_n . \text{NIL}) \ldots))$.

Some examples of lists (left column) and the equivalent in dot notation (right column):

()	NIL
(A)	(A . NIL)
(A B C)	(A . (B . (C . NIL)))
((A))	((A . NIL) . NIL)
((A B) (XYZ) (U V))	((A . (B . NIL)) . ((XYZ . NIL) . ((U . (V . NIL)) . NIL)))
((A) ((A)))	((A . NIL) . (((A . NIL) . NIL) . NIL))

Some s-expressions cannot be represented without dots, for example (A . B). Mixed notation may also be encountered such as ((A . B) (C . D)). In this case, there is a list at the top level, and dots at a lower level. This is the same s-expression as ((A . B) . ((C . D) . NIL)). In general, we avoid creating s-expressions that require dots, but it is well to keep in mind that the dot notation is the simplest way of explaining the underlying theory of s-expressions.

Problem Set 2

1. Write each of these s-expressions using only dot notation.

a. A	b. (A B)
c. (1 (2) ((3)))	d. ()
e. (A (B ((C))))	f. ((A) 2)

2. Write each of these s-expressions without dots if possible.

a. ((A . NIL) . ((B . NIL) . NIL))
b. ((A . NIL) . (B . NIL))
c. (A . (B . (C . NIL)))
d. (NIL . NIL)
e. ((APPLE . (PIE . NIL)) . ((CHEESE . NIL) . NIL))
f. ((X . NIL) . ((NIL . Y) . NIL))

§1.2 Basic Functions of S-expressions

We are now going to consider a small number of very basic operations that one can perform on s-expressions. These operations are the foundation of all subsequent processing of s-expressions in much the same way that counting up and down is the foundation for all of arithmetic. As you probably know, counting is even more basic than adding and multiplying when we analyse operations from the mechanical viewpoint.

Because we are using a mathematical approach, we describe these operations as being functions. The first function to be discussed is called cons.

The function cons is used to construct bigger s-expressions out of smaller ones. It takes two s-expressions and puts a left parenthesis before the first one, a LISP dot between them, and a right parenthesis after the second one. For example, cons of A and B is (A . B). Also, cons of (A . B) and (X . Y) is ((A . B) . (X . Y)).

We need a reasonable way of writing these assertions other than in English. So we use a notation that looks like this:

cons[A, B] = (A . B)

cons[(A . B), (X . Y)] = ((A . B) (X . Y))

We have said that cons is a function. In the first line above, A and B are arguments of the function cons, and (A . B) is the value of cons associated with these two arguments. It is a common mathematical and scientific notation to write a function followed by a list of its arguments enclosed within parentheses. The arguments, if there are more than one, are separated from each other by commas. This is exactly what we have done here except that we use square brackets instead of parentheses. The reason for this is that when the arguments are s-expressions, this could get confusing since parentheses occur as parts of s-expressions.

Getting back to cons for the moment. Since every s-expression is built from atoms, every s-expression can be put together from atoms using cons. Consider the case of (A . (B . C)). We have cons[B, C] = (B . C), and cons[A, (B . C)] = (A . (B . C)). Putting these together, we have cons[A, cons[B, C]] = (A . (B . C)). This is an extension of our notation, and is

called composition.

Let us look at some examples of cons:

1. cons[BILL, JOE] = (BILL . JOE)
2. cons[A, (B . C)] = (A . (B . C))
3. cons[A, cons[B, C]] = (A . (B . C))
4. cons[A, NIL] = (A)
5. cons[A, (B C)] = (A B C)
6. cons[A, cons[B, (C)]] = (A . (B . (C . NIL))) = (A B C)
7. cons[A, cons[B, cons[C, ()]]] = (A B C)
8. cons[(A), (A)] = ((A) A)

Problem Set 3

1. What is the value of each of the following?

a. cons[B, B]
b. cons[(A . B), (A . C)]
c. cons[(A B), (A C)]
d. cons[Q, (R S)]
e. cons[(A B C), (D E F)]
f. cons[cons[cons[A, NIL], NIL], NIL]

2. What is a commutative operator? Is addition of numbers commutative? Is cons commutative?

3. Describe a necessary and sufficient condition for the value of cons to be expressible without dots.

Next, we consider the pair of functions car and cdr which are used to take apart s-expressions. Car and cdr are unary functions; unlike cons which is a binary function, each takes only a single argument.

car[(A . B)] = A
cdr[(A . B)] = B

Car and cdr are not defined as having values when their arguments are atomic. For example, car[A] has no meaning. Any s-expression which is not an atom we call a composite s-expression. If a composite s-expression is written in dot notation, there is always one main dot. This is the dot which is contained only within the outermost set of parentheses. Then car of

the s-expression is the expression between this dot and the leftmost parenthesis of the whole s-expression, and cdr is the expression between this dot and the rightmost parenthesis.

$$(\underbrace{(A\ .\ (B\ .\ C))}_{\text{car}}\ .\ \underbrace{(D\ .\ D)}_{\text{cdr}})$$

Examples:

car[(A B)] = A
cdr[(A B)] = (B)
cdr[(B)] = ()
cdr[()] is undefined
car[(((A)) (B))] = ((A))
car[cdr[(A B)]] = B
cdr[cdr[(A B)]] = () = NIL
car[((A))] = (A)
car[car[((A))]] = A
car[cons[A, B]] = A
cons[car[(A)], cdr[(B C D)]] = (A C D)

Many people have objected to the names car and cdr, proposing some alternative such as "first" and "rest" which describe the effect of car and cdr on lists. Yet these names have remained around because they compose into sequences of cars and cdrs and remain at least slightly pronounceable. For example, caddr (pronounced CAH-duh-der) means "car of cdr of cdr". So caddr[(A B C)] is the same as car[cdr[cdr[(A B C)]]] which is C. Notice that it is the rightmost a or d in the word which gets performed first, just as it is the rightmost function when we write out the longer form.

Examples:

car[(A B C)] = A
cadr[(A B C)] = B
caddr[(A B C)] = C
cdddr[(A B C)] = ()
cdar[(A B C)] is undefined
cadadr[((A B) (C D) (E F))] = D

Problem Set 4

What is the value of each of the following?

1. car[(A . B)]
2. cdr[(A . B)]
3. car[(A B)]
4. cdr[(A B)]
5. car[cdr[(A B)]]
6. cadr[(A B)]
7. cdar[(A B)]
8. cdar[((A B))]
9. cdar[((A) B)]
10. caaar[(((((A)))))]
11. cons[car[((A))],
    cadr[(A (((B) (C))))]]

Mixed Expressions

We have been discussing LISP expressions such as "cons[x, y]". Arithmetic expressions such as "$3x+y^2$" are familiar to you and need no special explanation. Since numbers are considered atoms and can appear within s-expressions, it is perfectly meaningful to mix LISP and arithmetic.

Example:

car[(2 3 4)] + cadr[(5 7 9)] = 2+7 = 9

Not all such expressions will be meaningful. 3+car[(4 A 10)] = 7, but 3+cadr[(4 A 10)] is undefined. ("A" is a name, and addition is not defined on names. Certainly we would not want to say categorically that 3+A is meaningless. The question of whether A can be considered to be a variable or whether it means only itself is one of interpretation. The question can only be considered in context, and we cannot discuss it adequately here.)

Within LISP, the notions of truth and falsity can be represented by the atoms T and F respectively. A function whose value is always T or F is called a predicate. There is a basic predicate called atom which tells us whether its argument is an atom, that is, it has the value T if its argument is an atom, and F if its argument is a composite s-expression.

Examples:

atom[A] = T
atom[()] = T
atom[(A B C)] = F
atom[cdr[(A B C)]] = F
atom[1974] = T
atom[ABC] = T
atom[car[(A B C)]] = T
atom[cons[A, B]] = F

Equality itself is considered to be a basic predicate of s-expressions. Suppose we give the notion of equality the name equal. Equal is defined as a binary predicate which has the value T if both its arguments are the same, but has the value F if its arguments are different s-expressions.

Examples:

equal[A, A] = T
equal[NIL, ()] = T
equal[A, (A)] = F
equal[car[(2 3 4)]+2, 4] = T
equal[(A B C), (A B C)] = T
equal[(A B), (A . (B . NIL))] = T
equal[(A . (B . C)), ((A . B) . C)] = F

In practice, we shall seldom use the function name equal, but instead use the equal sign to mean the same thing. Instead of writing equal[A, A], we shall write A = A, or when necessary [A = A]. When a function symbol (usually a special symbol rather than a name spelled with letters) is used between two arguments rather than preceding both of them, this is called infix notation. We use it frequently and in fairly obvious ways, but since problems of syntax are not an important part of this book, there will be no formal theory about parsing such grammars. In conclusion, the preceding examples will normally appear as [A = A] = T, [A = (A)] = F, etc.

Cons, car, cdr, atom and equal are the five basic functions for the manipulation of s-expressions.

Suppose we wish to form a list from three constituents. We can describe this construction by writing cons[α, cons[β, cons[γ, NIL]]] where α, β, and γ are the three s-expressions to be listed. This is too long to write, so we introduce the shorter notation using the function list which can have any number of arguments including none. The preceding example can be replaced by list[α, β, γ].

Examples:

list[A, B, C] = (A B C)
list[] = NIL = ()
list[(A B), (C D)] = ((A B) (C D))
list[list[A]] = ((A))
cons[A, list[B, C, D]] = (A B C D)

Another convenience is the predicate null which has a single argument and is true only if that argument is NIL.

Examples:

null[()] = T
null[A] = F
null[cdr[(A)]] = T
null[(NIL)] = F

Atoms can be sorted out into two types, names and numbers, and to do this we introduce two predicates, name and num.

Examples:

name[ABC] = T
name[(A)] = F
name[5] = F
num[5] = T
num[cadr[(A 2 (5))] = T
num[(3)] = F
num[A] = F
num[3 + car[(5)]] = T

There is another function which we shall consider to be basic without any justification at present. Consider the set of all names (not numbers). There are infinitely many of them, but they can be placed in a definite order in an infinite list, that is, they can be enumerated. We list shorter names before longer ones, arranging the finitely many names of any particular length in alphabetical order, putting 0 thru 9 at the end of the alphabet.

The function enum is only defined when its argument is a number. The value is always a name, and if we form the list enum[0], enum[1], enum[2]... we get exactly the enumeration discussed above.

Problem Set 5

1. list[A, 2 + car[(3 4)], B]
2. list[2 + 2, 2 + 2 = 4, 2 + 2 = 5)

3. atom[list[A, A = B]]

4. enum[2 + 2]

5. name[enum[2 + 2]]

6. enum[A]

7. cons[name[A], num[A]]

8. (A . B) = (B . A)

9. list[(A B), list[(C D), (E F)]]

10. cadadr[((A B) ((C D) (E)))]

CHAPTER TWO

RECURSIVE DEFINITIONS

Preview of Chapter Two

The chapter begins with some comments on functions, and the terminology concerning them, from the viewpoint of naive set theory. After this, a very simple language is defined which, together with the basic functions discussed in Chapter One, will allow us to define every function of s-expressions that is in any reasonable sense calculable.

Having now completed Chapter One, it is as if you had learned arithmetic but not algebra. We can handle 2+2=4, but not x+y=z. What we need among other things are variables, and a way to use them so that we can describe general instead of particular computations.

§2.1 Functions

The concept of a "function" in set theory is synonymous with "mapping" or "correspondence". Suppose we have two sets, A and B, and for every object in A, there is an object (or element) of B associated with it. Then this correspondence is called a function, and A is the domain of the function, and B is the range of the function. Although every element of A has a corresponding element in B, it is not the case that every element in B must correspond to an element in A. A given element of B may correspond to more than one element of A, or to none at all.

From the point of view of set theory, the function itself is viewed as a set. If f is a function from A to B (we write this as $f:A \rightarrow B$) then f itself is a set of ordered pairs $\langle a, b \rangle$ such that $a \in A$ (a is a member of A) and $b \in B$, and

such that for every $a \in A$ there is exactly one such ordered pair which is a member of f.

Some functions have more than one argument. If f is a function of n arguments, then f has n domains, A_1 thru A_n, and one range, B, and we may specify this information by writing $f: A_1 \times \ldots \times A_n \rightarrow B$. The function f is then a set of n+1-tuples $\langle a_1, \ldots a_n, b \rangle$ where each $a_i \in A_i$, and $b \in B$, and such that for every combination of one a_i from each A_i, there is exactly one such n+1-tuple in f. For example, cons: $S \times S \rightarrow S$ is the infinite set of triples $\{\langle A, A, (A \ . \ A)\rangle \ldots\}$ containing every possible pair of s-expressions together with their cons.

As was mentioned in Chapter One, members of the domain(s) of a function are called arguments, and members of the range are called values.

The set {T, F} is called #. A function whose range is # is called a predicate.

Any subset of a function is called a partial function. That is, a partial function is a function that may not have values for all its arguments. Sometimes we are a bit sloppy and use the word "function" when we mean "partial function". Then, when we want to emphasize the completeness of a function, we are led to use the term total function. We also speak of partial and total predicates.

All of this may seem extremely obvious, but it is important to stress that when we talk of a function we are not referring to a procedure or a subroutine. The distinction is important, and is roughly analogous to the difference between a loaf of bread and a recipe for baking a loaf of bread. Recipes, like procedures, can be published in books and journals. No one has ever published a loaf of bread. Similarly, the function cons can be discussed, and subroutines written to compute it, but the function itself is an infinite set and is therefore a conceptual object only, never a printed object. It is also important not to confuse the name of a function with the function itself.

§2.2 Recursive Definition

> Recursive Definition: A definition of a function permitting values of the function to be computed systematically in a finite number of steps; esp: a mathematical definition in which the first case

is given and the nth case is defined in terms of one or more previous cases and esp. the immediately preceding one.

Webster's Third New
International Dictionary

It would be hard to improve on this definition. We shall start this discussion by illustrating that many ordinary functions of arithmetic may be defined recursively starting only with the functions successor and predecessor as given. The meaning of the notation being used will be explained in English. After this, we shall define more formally the language we have been using.

The successor of a number is one more than that number. For example, the successor of 5 is 6. Our notation for the successor of n is n'. So $5' = 6$, and $5'' = 7$.

The predecessor of a number is the next smallest number. The predecessor of zero is not defined. Our notation for the predecessor of n is n^-. So $7^- = 6$, and $7^{--} = 5$, and 1^{--} is undefined.

Starting with only these two functions, and equality, we proceed to define addition and multiplication:

$$(1)\ m+n \leftarrow [n=0 \rightarrow m, T \rightarrow m'+n^-]$$
$$(2)\ m \times n \leftarrow [n=0 \rightarrow 0, T \rightarrow m+m \times n^-]$$

Translated into English, the first definition reads: "The sum of m and n is m if n is 0; otherwise it is the same as the sum of the successor of m and the predecessor of n." This fits the dictionary definition perfectly. We say that we are recursing downward on n. When we count n down to 0, then the process is over and we have an answer. For example, $5+3 = 5'+3^- = 6+2 = 6'+2^- = 7+1 = 7'+1^- = 8+0 = 8$. The recursive definition is applied over and over again until the second argument (called n in this definition) is 0. As long as n is greater than 0, the second part of the definition applies and the computation proceeds. When $n = 0$, then the first part of the definition applies and the computation is over. It never becomes necessary to take the predecessor of 0, and therefore an undefined condition will never arise.

We call line (1) a recursive definition. It provides an explicit method of computing the function "+" given the successor and predecessor functions.

This particular recursive definition gives a value which is a number every time it is applied to a pair of arguments which are numbers. But not all recursive definitions are this way. A recursive definition may not compute a value for a variety of reasons. The fact that (a) this particular recursive definition computes a total function, and (b) this total function is the familiar function "+" are particulars which are obvious in this case, but in general the correspondence between the function computed by a recursive definition and a function understood or specified in some other way must not be assumed without good and sufficient reason.

Line (2) is a recursive definition for multiplication. It can be translated into English as "The product of m and n is 0 if n is 0; otherwise it is the sum of m and the product of m and the predecessor of n." This definition invokes the previous definition of addition. So the recursive definition of multiplication is really both lines.

It will be our general habit when making recursive definitions to build up more complicated functions from simpler ones. From the definition of multiplication we see that $5 \times 3 = 5+5 \times 2 = 5+5+5 \times 1 = 5+5+5+5 \times 0 = 5+5+5+0 = 15$.

Recursive definition is also used to specify the computation of predicates. The numerical relation "is greater than" is an example of such. It can be defined by:

$$m > n \leftarrow [m = 0 \rightarrow F, n = 0 \rightarrow T, T \rightarrow m^- > n^-]$$

which expresses the English definition: "If m is zero, then m is not greater than n; if m is not zero and n is zero then m is greater than n; and if neither is zero then m being greater than n depends on m^- being greater than n^-." When m and n are numbers, the value of ">" will always be either T or F. The predicate ">" can now be used to define the function max[m, n], whose value is the larger of its two arguments.

$$max[m, n] \leftarrow [m > n \rightarrow m, T \rightarrow n]$$

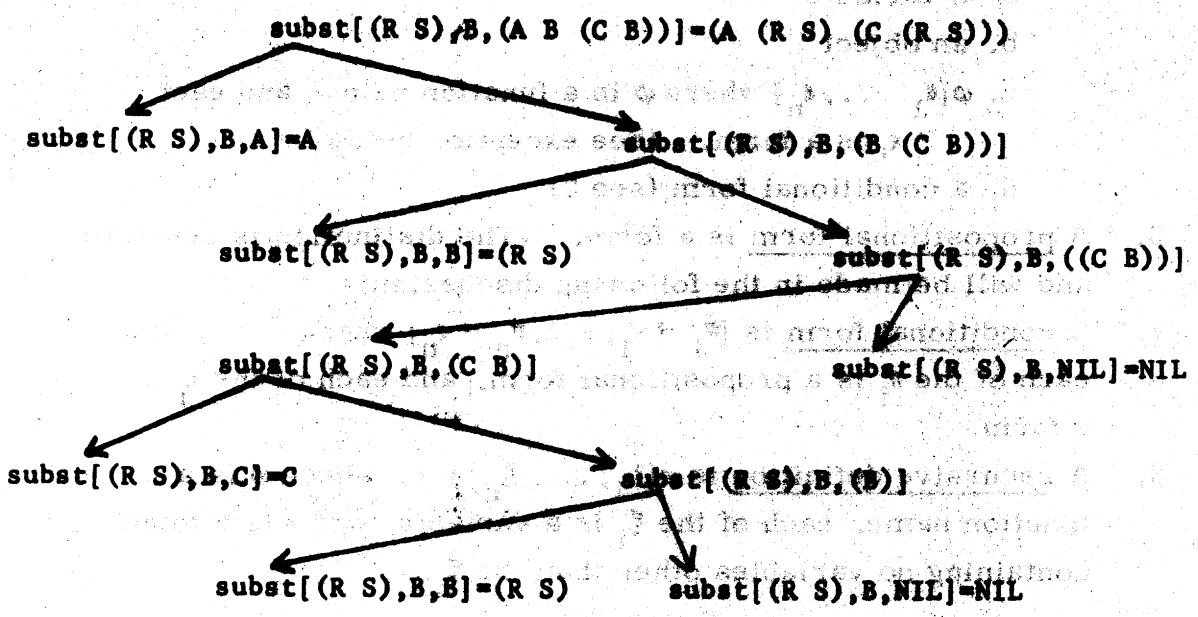
Recursive definition is used to define functions of s-expressions other than numbers in a similar way. An important LISP function is subst[x, y, z], whose value is the s-expression resulting from substituting the s-expression x for all occurrences of the atom y in the s-expression z. For example, the

result of substituting (R S) for all occurrences of B in (A B (C B)) is (A (R S) (C (R S))), i.e., subst[(R S), B, (A B (C B))] = (A (R S) (C (R S))). Subst is defined recursively by:

```
subst[x, y, z] ← [atom[z] → [y = z → x, T → z], T → cons[subst[x, y,
      car[z]], subst[x, y, cdr[z]]]]
```

which translates: "If z is an atom, then if y is the same as z the value is x, otherwise the value is z; but if z is not an atom then the value is obtained by first computing subst of x and y and car of z, and subst of x and y and cdr of z, and then taking the cons of these two s-expressions."

This example is more complex than the preceding ones in two ways. It contains a choice nested within a choice in that if z is an atom, then there is still another decision to be made. Also, the recursion generates a tree-structure of subproblems rather than a linear sequence as in the preceding numerical examples. The recursion on the argument z may require computing subst of car[z] and cdr[z], which may in turn require computing subst with the third argument being car[car[z]], cdr[car[z]], car[cdr[z]], and cdr[cdr[z]]. The larger the s-expression z, the larger this tree of subproblems will grow.



cons[A,cons[(R S),cons[cons[C,cons[(R S),NIL]],NIL]]]=(A (R S) (C (R S)))

We have been making use of the language of recursive definitions in an informal way; we now proceed to define the language more formally. First we give a concise grammar, and then define the semantics of the language as a set of instructions for computing complex functions from given basic functions.

Grammar:

1. An identifier is a sequence of one or more lower case letters and digits. It must begin with a letter. (This is the same as the definition of a name except that names have upper case letters. Identifiers and names are in one-to-one correspondence by merely changing the case of the letters.)
2. A variable is an identifier.
3. An object is an s-expression.
4. A function name is an identifier. (See exception below.)
5. A form can be any of the following:
   a. a variable
   b. an object
   c. $\varphi[\epsilon_1, \ldots, \epsilon_n]$ where φ is a function name, and each ϵ_i is a form. (See exception below.)
   d. a conditional form (see 7)
6. A propositional form is a form. (The distinction is semantic and will be made in the following discussion.)
7. A conditional form is $[\pi_1 \rightarrow \epsilon_1, \ldots, \pi_n \rightarrow \epsilon_n]$ where $n \geq 1$, and each of the π_i is a propositional form, and each of the ϵ_i is a form.
8. A recursive definition is $\varphi[\xi_1, \ldots, \xi_n] \leftarrow \epsilon$, where φ is a function name, each of the ξ_i is a variable, and ϵ is a form containing no variables other than the ξ_i.

Exception: Rules 4 and 5 above permit forms such as cons[car[x], cdr[y]] but not forms such as $3 \times m + n$. It is convenient to have functions specified by conventional symbols such as "+", "x" and "÷" as well as by identifiers, and it is also convenient and conventional to use certain of these symbols as infixes (m+n), prefixes (-m)

and suffixes (m′). Because we are not concerned with writing compilers we shall gloss over the syntactic problems of such notation in the following way: every function and predicate which is specified by a conventional symbol has an identifier type name also. As long as the parsing of any form that we write is clear to the reader, we can pretend in certain theoretical situations that the only official notation is that of identifier names followed by arguments in brackets, i.e., $\varphi[\ldots]$. For example, "3 x x - y > z" is simply a convenient notation so that we do not have to write out "greater [difference[times[3, x], y], z]".

Semantics:

1. A recursive definition has meaning because it is an explicit algorithm for computing a partial recursive function. To the left of the left arrow in the recursive definition is the name given to the partial function being defined, and a list of variables. The list of variables specifies the number of arguments the function has, and assigns these variables as the temporary names of the arguments while the computation is in progress. This temporary identification of arguments with variables is called a binding of the variables. The value of the function is obtained by evaluating the form to the right of the left arrow, using the rules given below, with this binding of the variables in effect.
2. The value of an object is itself.
3. The value of a variable is obtained from the binding as specified in rule 1.
4. The value of a form of the type $\varphi[\epsilon_1, \ldots, \epsilon_n]$ is computed by first evaluating each of the forms ϵ_i using these rules, and using the resulting sequence of values as arguments for the function φ. If φ is one of the basic functions or predicates, its value is obtained immediately. However, if φ is itself specified by a recursive definition, then the current computation must be set aside, and the computation to obtain

the value of φ for these arguments must be performed. When this is completed, the current computation is then resumed, and neither the bindings or the variables, nor any other partial results in process will have been changed from what they were before invoking the definition of φ. This process can occur nested to a considerable depth, with many levels of computation interrupted and incomplete. When the definition of φ is invoked from within the definition of φ, this process is called recursion. The example of subst is a good case to study.

5. A propositional form is a form whose expected value is T or F. Typically, it is either one of the objects T or F, or it is a form $\phi[\epsilon_1, \ldots, \epsilon_n]$ where ϕ is a predicate. It may also be a conditional form.

6. The value of a conditional form $[\pi_1 \rightarrow \epsilon_1, \ldots, \pi_n \rightarrow \epsilon_n]$ is obtained by evaluating the propositional forms π_i from left to right until one is found whose value is T. Then, no more π_i are evaluated, but the corresponding ϵ_i is evaluated, and its value is the value of the conditional form. An important property of a conditional form is that nothing gets evaluated beyond what is necessary to select and evaluate the proper ϵ_i. For example, if π_1 evaluates to F, then ϵ_1 is not evaluated but passed over, and π_2 gets evaluated. If the value of π_2 is T, then ϵ_2 is evaluated to provide the value of the conditional form, and everything to the right of ϵ_2 is ignored.

There are a variety of reasons why the process of evaluating a recursive function may not produce a value:

1. A variable on the right side of a definition does not occur on the left side.
2. A function referred to in the definition has not been defined.
3. A function is given an incorrect number of arguments.
4. In the process of evaluating a conditional form, one of the π_i

evaluates to something that is neither T nor F.

5. In the process of evaluating a conditional form, all the π_i evaluate to F and the end of the form is reached.

6. A function called in the computation is given an argument for which no value is defined, such as car applied to an atom, or addition applied to non-numbers.

7. The computation continues forever without encountering any of the errors mentioned above, but without ever terminating.

Reasons 1 thru 6 above are simply programming errors that can be avoided by correct procedure. Reason 7 is a fundamental property of computation having important logical consequences. There is no possibility of eliminating it from any programming language powerful enough to do general purpose computation.

Problem Set 6

In each of these problems you may assume any of the definitions made thus far, including all the problems preceding the one you are working on. Sometimes it is necessary to define a helping function first before defining the function you want.

1. Are the functions "-" and "÷" defined here total or partial?

$$m - n \leftarrow [n = 0 \rightarrow m, T \rightarrow m^- - n^-]$$
$$m \div n \leftarrow [n > m \rightarrow 0, T \rightarrow 1 + [m - n] \div n]$$

2. Define expt[m, n] or m^n. (Let expt[0, 0] = 1.)

3. Define remainder[m, n].

4. Define m|n, which means m goes into n an integer number of times. (It is a predicate.)

5. Define prime[n], a predicate which is true if n is a prime number. (The first prime number is 2.)

6. Define gcd[m, n] (greatest common divisor), and lcm[m, n] (least common multiple).

7. Define nthprime[m], which gives the nth prime. Use the convention that nthprime[0] = 1 and nthprime[1] = 2.

8. Define the predecessor function using only successor and equality,

thereby demonstrating that all the functions in this problem set require only successor and equality as their basis.

Let us adopt the coding convention that a finite set of s-expressions will be represented in LISP by a list of these s-expressions. For instance, the set {A B C} can be represented by the list (A B C) or by the list (B C A) or by any other permutation of the members. The list must have no repetitions. We can then define LISP functions that perform basic set-theoretic operations. For example, the relation of membership is represented by the LISP predicate member, defined by

member[x, y] ← [null[y] → F, x = car[y] → T, T → member[x, cdr[y]]]

The operation of taking the union of two sets is represented by the function union:

union[x, y] ← [null[x] → y, member[car[x], y] → union[cdr[x], y],
T → cons[car[x], union[cdr[x], y]]]

Problem Set 7

1. Define the function intersection[x, y].

2. Define the predicate sequiv[x, y] which means "equivalent" in the sense of representing the same set. Two lists representing sets are sequiv if they differ only in the order of their elements, e.g., sequiv[(A B C), (A C B)] = T.

3. Define the function reverse, whose value is the same list as its argument, but in reverse order, e.g., reverse[((A B) (C D))] = ((C D) (A B)).

4. Define the function length, which computes the length of the list x. length[NIL] = 0.

5. Define the function size, where size[x] is the number of atoms occurring in x, counting each atom as many times as it occurs, e.g., size[(A (A))] = 4.

6. Define the function vocab, where vocab[x] is the set of atoms occurring in the s-expression x. Vocab[(A (B C) C)] = (A B C NIL), or any list which is sequiv to this.

§2.3 Partial Recursive Functions

The basic functions for computing with s-expressions (including numbers) are: car, cdr, cons, equal, atom, num, name, enum and successor.

We have seen that predecessor can be defined from successor. Name can also be defined from atom and num if we are sure that there will never be any other type of atom. We prefer to leave this unspecified.

The function enum is peculiar. Without enum, we would not be able to define those functions which depend on the spelling of names, but would be limited to functions that only take note of two names as being the same or different. But using enum, we can define concat, which concatenates two atoms (e.g., concat[A, X3] = AX3), and explode, which lists the letters and digits in a name, e.g., explode[AX3] = (A X 3). These two functions, in turn, form the basis for any other manipulation of the characters that make up names.

The basic functions of s-expressions together with the language of recursive definition lead to the concept of a partial recursive function.

Lemma 2.1

Consider a finite sequence of recursive definitions:

$$\varphi_1[\xi_1, \ldots, \xi_{m_1}] \leftarrow \epsilon_1$$

$$\varphi_n[\xi_1, \ldots, \xi_{m_n}] \leftarrow \epsilon_n$$

where each φ_i is a distinct function name, and each ϵ_i contains only the names of basic functions and names from the sequence $\varphi_1, \ldots, \varphi_n$; then associated with each φ_i there is a procedure for computing a function of m_i arguments. This procedure, when performed with any given sequence of m_i s-expressions as arguments, either produces a value, encounters an undefined situation, or fails to terminate. Thus, each function-name φ_i is associated with an m_i-ary (partial) function, namely that function defined for exactly those arguments for which the computation terminates with a value (the value of the function).

The entire preceding section is sufficient proof that such a well-defined

computational procedure exists.

Definition 2.2

A partial recursive function (of s-expressions) is any function for which at least one computational procedure as defined above exists.

It is well to keep in mind the distinction between a function, a function-name, and a procedure. It is the process of writing down recursive definitions that associates names and procedures with functions. A function is independent of any procedure used to specify it. However, the concept of recursiveness is absolute; a function is either recursive or it is not recursive. It is recursive if there is at least one way to compute it (and it is easy to see that there are then many ways to compute it), and it is not recursive if there is no way to compute it. When a function is specified in some way that does not imply a computational procedure, this does not tell us whether or not it is recursive.

§2.4 A Universal LISP Function

It is natural to want a theory of recursive functions. We may ask questions such as: How large is the class of recursive functions? Are there functions that are well defined but not recursive? If we add new computational techniques or more basic functions, are we able to compute more functions? The idea of an interpreter or universal function is central to such a theory.

The importance of lemma 2.1 is that the procedure for computing partial functions is effective. This means that we can program a general purpose computer so that when we give it a sequence of recursive definitions, and a set of arguments for one of the functions, the computer then computes the value of the function applied to these arguments if the value exists, and if the computer has enough storage and time. Such a program is called a LISP interpreter, and has been written for many computers. What is of great significance for the theory of recursive functions, is that such an interpreter can be written in LISP itself.

We define a universal LISP function called apply. Apply[fa, args] has

two arguments. The first argument is a sequence of recursive definitions as in the statement of lemma 2.1. Since an argument for apply must be an s-expression, we must code such a sequence of recursive definitions into a single s-expression. The first function defined in the sequence will be the one we wish to compute, and the other definitions necessary to it may follow in any order. The second argument for apply is a list of the arguments for this function.

We first define a translation whereby a sequence of recursive definitions, as in the schema of Lemma 2.1, becomes a single s-expression, the argument fa of apply. We shall call this translation process "*", so that, for example, if ϵ is a form in the language of recursive functions, then ϵ^* is its translation into an s-expression.

Rules for translating recursive definitions into s-expressions:

1. If ϵ is a variable, then ϵ^* is the atom obtained by making all of its letters upper case.
2. If ϵ is a number, then ϵ^* is just ϵ.
3. If ϵ is T, F, or NIL, then ϵ^* is just ϵ.
4. If ϵ is any other object (s-expression), then ϵ^* is (QUOTE ϵ).
5. If φ is a function-name, then φ^* is the atom obtained by making all its letters upper case.
6. If ϵ is a form of the type $\varphi[\epsilon_1, \ldots, \epsilon_n]$, then ϵ^* is $(\varphi^* \ \epsilon^* \ \ldots \ \epsilon_n^*)$. (Forms using infix, prefix, or suffix operators are translated as if they were in standard form. There are names for each such operator. Also, some functions have an indefinite number of arguments. They are LIST, PLUS, and TIMES.)
7. If ϵ is a conditional form $[\pi_1^* \rightarrow \epsilon_1, \ldots, \pi_n \rightarrow \epsilon_n]$, then ϵ^* is $(\text{COND } (\pi_1^* \ \epsilon_1^*) \ \ldots \ (\pi_n^* \ \epsilon_n^*))$.
8. A recursive definition $\varphi[\xi_1, \ldots, \xi_n] \leftarrow \epsilon$ is translated as $(\varphi^* \ (\xi_1^* \ \ldots \ \xi_n^*) \ \epsilon^*)$.
9. The argument "fa" of apply is a <u>list</u> of translated recursive definitions as described in step 8, with the function to be applied coming first on the list, and all functions that it uses, except for basic functions, appearing in any order on the list.

The translation process "*" is the LISP equivalent of a technique known to logicians as "Gödel numbering" which we shall discuss later. However, Gödel numbering is a theoretical concept which is impossible to use in any practical sense, whereas the use of s-expressions to define recursive functions is standard practice for LISP programmers.

Examples of translation:

Rule	e	e*
1.	x	X
	a3	A3
2.	25	25
3.	F	F
4.	A	(QUOTE A)
	(A B)	(QUOTE (A B))
	(3)	(QUOTE (3))
5.	car	CAR
6.	car[x]	(CAR X)
	cons[u, cdr[A]]	(CONS U (CDR (QUOTE A)))
6a	m + n × p + q	(PLUS M (TIMES N P) Q)
7.	[x = 0 → 1, x = 1 → 1, T → fibb [x - 1] + fibb[x - 2]]	(COND ((EQUAL X 0) 1) ((EQUAL X 1) 1) (T (PLUS (FIBB (DIFFERENCE X 1)) (FIBB (DIFFERENCE X 2)))))
8.	fibb[x] ← [x = 0 → 1, x = 1 → 1, T → fibb[x - 1] + fibb[x - 2]]	(FIBB (X) (COND ((EQUAL X 0) 1) ((EQUAL X 1) 1) (T PLUS (FIBB (DIFFERENCE X 1)) (FIBB (DIFFERENCE X 2)))))
9.	foo[x, y] ← list[x, glitch[y]] glitch[x] ← list[x, C] foo[A, B] = (A (B C)) apply[((FOO (X Y) (LIST X (GLITCH Y))) (GLITCH (X) (LIST X (QUOTE C)))), (A B)] = (A (B C))	

The partial recursive function apply is defined via a number of auxiliary functions.

Definition of apply:

```
apply[fa, args] ← app[caar[fa], args, fa]

app[fn, args, fa] ← [
        fn = CAR → caar[args],
        fn = CDR → cdar[args],
        fn = CONS → cons[car[args], cadr[args]],
        fn = LIST → args,
        fn = ATOM → atom[car[args]],
        fn = NUM → num[car[args]],
        fn = NAME → name[car[args]],
        fn = NULL → null[car[args]],
        fn = ENUM → enum[car[args]],
        fn = SUCCESSOR → car[args]',
        fn = PLUS → applus[args],
        fn = TIMES → aptimes[args],
        fn = NOT → [car[args] = T → F, car[args] = F → T],
        T → apd[assoc[fn, fa], args, fa]    ]

eval[e, a, fa] ← [
        num[e] → e,
        e = T → e,
        e = F → e,
        e = NIL → e,
        name[e] → cadr[assoc[e, a]],
        car[e] = QUOTE → cadr[e],
        car[e] = COND → evcon[cdr[e], a, fa],
        car[e] = AND → evand[cdr[e], a, fa],
        car[e] = OR → evor[cdr[e], a, fa],
        T → app[car[e], evlis[cdr[e], a, fa], fa]    ]

apd[fd, args, fa] ← eval[caddr[fd], pair[cadr[fd], args], fa]

applus[a] ← [null[a] → 0, T → car[a] + applus[cdr[a]]]

aptimes[a] ← [null[a] → 1, T → car[a] x aptimes[cdr[a]]]

assoc[e, a] ← [e = caar[a] → car[a], T → assoc[e, cdr[a]]]

pair[x, y] ← [null[x] → [null[y] → NIL], T → cons[list[car[x], car[y]],
        pair[cdr[x], cdr[y]]]]

evlis[e, a, fa] ← [null[e] → NIL, T → cons[eval[car[e], a, fa],
        evlis[cdr[e], a, fa]]]

evcon[e, a, fa] ← [eval[caar[e], a, fa] → eval[cadar[e], a, fa],
        T → evcon[cdr[e], a, fa]]

evand[e, a, fa] ← [null[e] → T, eval[car[e], a, fa] →
        evand[cdr[e], a, fa], T → F]

evor[e, a, fa] ← [null[e] → F, eval[car[e], a, fa] → T, T → evor[cdr[e], a, fa]]
```

The reader who finds this piece of coding dense may either puzzle through it himself, study one of the texts on LISP programming, or simply take it on faith that it does what we claim it does. The reader familiar with one or more LISP dialects should note that this interpreter differs considerably from the apply operator of any computer implementation. Its arguments are different, it does not handle LAMBDA or functional arguments, it does not evaluate free variables, it treats T, F, NIL and conditionals in a non-standard fashion and it has no PROG feature.

Although we must normally define any function with a fixed number of arguments, this interpreter provides three specific exceptions: PLUS, TIMES and LIST.

It also provides for three logical operators: OR, AND and NOT. NOT is a function defined only on the domain T. Its behavior is completely explained by noting that not[T] = F and not[F] = T. The prefix symbol for "not" is "¬". AND and OR are slightly more complex. They are variants of the conditional form. Mathematically, "∧" and "∨" (which stand for "and" and "or", respectively) are functions on the domain T having two arguments. They are completely specified by the following table:

x	y	x ∧ y	x ∨ y
T	T	T	T
T	F	F	T
F	T	F	T
F	F	F	F

If these were evaluated in LISP in the same manner as other functions, then a form such as "$\epsilon_1 \vee \epsilon_2$" would require first evaluating ϵ_1 and ϵ_2 with the expectation of getting T or F in each case and then using the above table to get the value of the form. What we actually do is quite different: First ϵ_1 is evaluated. If this is T, then we conclude that $\epsilon_1 \vee \epsilon_2$ is T and do not evaluate ϵ_2 at all. But if ϵ_1 is F, then we have to evaluate ϵ_2. We treat "∧" similarly; if ϵ_1 is F, we conclude that $\epsilon_1 \wedge \epsilon_2$ is F and do not evaluate ϵ_2.

The form "$\epsilon_1 \vee \epsilon_2$" is completely equivalent to the conditional form $[\epsilon_1 \rightarrow T, T \rightarrow \epsilon_2]$, and the form "$\epsilon_1 \wedge \epsilon_2$" is completely equivalent to the

conditional form $[\epsilon_1 \rightarrow \epsilon_2, T \rightarrow F]$.

This makes possible such definitions as the following alternative definition of member, which would not work if these functions were evaluated in the standard way. (Why?)

$$\text{member}[x, y] \leftarrow \neg\text{null}[y] \wedge [x = \text{car}[y] \vee \text{member}[x, \text{cdr}[y]]]$$

The interpreter we have defined conveniently provides for AND and OR to have an indefinite number of arguments. The form "$\epsilon_1 \wedge \ldots \wedge \epsilon_n$" is translated into (AND $\epsilon_1^* \ldots \epsilon_n^*$), and similarly for "$\vee$" with OR. AND evaluates its arguments from the left to right until it either finds a false one, whereupon it concludes F without further evaluation, or else if they are all true, then the value is T. (AND), that is, AND of no arguments, will be T because none of its arguments are false. OR evaluates its arguments from left to right until one of them is true, whereupon it concludes T without further evaluation, or else if they all evaluate false, then the value is F. The value of (OR) is F because it does not have at least one true argument.

Theorem 2.3 (Interpreter Theorem)

Let $\delta_1, \ldots, \delta_n$ be a self-contained sequence of recursive definitions (in the sense of Lemma 2.1), let φ be the name of the function defined by δ_1, and let $\alpha_1, \ldots, \alpha_m$ be any sequence of s-expressions where m is the number of arguments for φ_1. Then either

$$\varphi_1[\alpha_1, \ldots, \alpha_m] = \text{apply}[\text{list}[\delta_1^*, \ldots \delta_n^*], \text{list}[\alpha_1, \ldots, \alpha_m]]$$

or else both sides of this equation are undefined (i.e., both computations produce the same value, or both fail to produce values).

The starting point for all theoretical study of computation is the fact that any one formulation of a sufficiently general class of effectively computable functions always turns out to be equivalent to all other such formulations. Historically, A. M. Turing defined a class of conceptualized machines of very simple design having an infinitely long tape on which to read and write. Any function that can be computed on such a Turing machine is called computable. Turing then gave very convincing arguments to show that the most elaborate computers that he could think of could not compute anything that

these very simple machines could not compute, given enough time. Turing also showed that there is a universal Turing machine that could interpret any other Turing machine, given a description of the other machine.

It is possible to write a recursive function that simulates a universal Turing machine. It is also possible to design a Turing machine that simulates the interpreter "apply". This not very surprising fact is the basis for a proof that the partial recursive functions are the same as the partial computable functions on a Turing machine.

Turing's Thesis

Any function which can be effectively computed can be computed by a Turing machine.

From this we may conclude that any function of s-expressions that is effectively computable is recursive. This is the converse of lemma 2.1, and is known as "Church's thesis".

Problem 8

Write an interpreter that evaluates arithmetic forms only. It will be called areval, and has two arguments. The first argument is a form to be evaluated, for example, (PLUS 3 (TIMES X Y)). The second argument is an "a-list" which defines the values of the variables occurring within the first argument, for example, ((X 2) (Y 7)). So areval[(PLUS 3 (TIMES X Y)), ((X 2) (Y 7))] = 3+2 ×7 = 17. It does not handle conditional forms or function definitions.

Further Reading

For the reader wishing to learn LISP as a programming language, there are two books: [McCarthy et al.] and [Weissman]. Additional information on particular LISP implementations is usually available at each installation. There is also a set of graded LISP problems with answers [Hart and Levin], which is useful as a teach-yourself aid.

There is an excellent discussion of the validity of Turing's Thesis in [Kleene, §70]. In [Davis], Turing machines are used as the starting point for the development of recursive function theory.

CHAPTER THREE

PROPOSITIONAL LOGIC I

Preview of Chapters Three and Four

This chapter begins our study of logic as a tool for making formal deductions. Propositional logic is the logic dealing with the compounding of sentences or propositions using connectives such as "and", "or", "not", and "implies". It is not an adequate logical language for making inferences because it deals with entire clauses and does not consider their internal structure. We study propositional logic because it is the ground floor of the two-story edifice of first order logic, which is our main subject. The terminology and organization of our study of propositional logic will carry over directly to first order logic.

Chapter Three introduces the language of propositional logic, the technique of making logical propositions, and the feasibility of mechanically checking deductions to determine if they are correct. Chapter Four presents the mathematical theory of propositional logic.

§3.1 Propositional Formulas

Making use of a fairly loose analogy, we can say that propositional variables correspond to simple declarative English sentences, and that propositional formulas correspond to compound sentences.

A It will not rain tomorrow.

B We shall go to the beach.

$A \supset B$ If it does not rain tomorrow, we shall go to the beach.

A and B are propositional variables, "$\supset$" means "implies", and $A \supset B$

is a propositional formula.

We shall make the assumption that unlike English sentences, propositional variables can always be interpreted as being either true or false assertions. There is no middle ground such as "too ambiguous" or "doesn't make sense". Propositional logic is also cruder than English in that the truth of a compound proposition depends only on the truth of its components, and the way they are connected by logical operators, and not on the way that they might meaningfully be related. For example, the English sentence "If two plus two is five then the world will end next Monday," can be considered as nonsense. Suppose we let A mean "Two plus two is five," and B mean "The world will end next Monday." If A and B are both false, then $A \supset B$ is considered true. This is part of the definition of "$\supset$", which simply requires that if A is true then B must be true. Since A is not true, B doesn't have to be true for $A \supset B$ to be true. The saying "If wishes were deeds, then beggars would be kings," captures the essence of this type of thinking.

Definition 3.1

A propositional variable is a name. (It begins with a capital letter.)

A propositional formula is:

(i) a propositional variable
or (ii) $\neg(\alpha)$
or (iii) $(\alpha) \vee (\beta)$
or (iv) $(\alpha) \wedge (\beta)$
or (v) $(\alpha) \supset (\beta)$
or (vi) $(\alpha) \equiv (\beta)$

where α and β are themselves propositional formulas.

The names of the propositional connectives are:

$\neg$	not
$\vee$	or
$\wedge$	and
$\supset$	implies
$\equiv$	equivalent

It follows from this definition that propositional formulas can be constructed of arbitrary size and depth of parenthesization. Sometimes we do not write all of the parentheses because they are not needed.

Examples:

A	$A \supset B$
$A \vee (B \wedge C)$	$\neg(A \supset (B \supset C))$
$(A \wedge B) \equiv (B \wedge A)$	$(A1 \wedge (A2 \wedge A3))$

§3.2 Interpretation

The following truth table is designed to interpret propositional formulas for truth or falsity. To interpret a formula we must first decide on a truth value (T or F) for each propositional variable. This cannot be inferred from the truth tables and for the moment at least must be considered as given. Having done this, we can then assign a truth value to each sub-formula starting with the innermost ones and ending with the entire given formula.

Truth Values of the Propositional Connectives:

A	B	$A \wedge B$	$A \vee B$	$\neg A$	$A \supset B$	$A \equiv B$
T	T	T	T	F	T	T
T	F	F	T	F	F	F
F	T	F	T	T	T	F
F	F	F	F	T	T	T

Example:

Evaluate $(A \wedge B) \equiv (B \vee C)$ when A is T and B and C are F. From the table, we see that if A is T and B is F, then $A \wedge B$ is F. If B is F and C is F, then $B \vee C$ is F. So the formula becomes $F \equiv F$, which according to the table is T.

Problem Set 9

Evaluate each formula, using the following table of values for variables.

A1: T	B1: F
A2: T	B2: T
A3: F	B3: F

1. $A3 \supset B3$
2. $A3 \supset A2$
3. $A1 \vee B1$
4. $A2 \wedge B1$
5. $\neg A3$
6. $\neg\neg\neg B3$
7. $\neg(A3 \vee B2)$
8. $\neg(B1 \supset B1)$

To process propositional formulas in LISP we shall have to translate them into s-expressions. The s-expression form should come as no surprise: propositional variables undergo no change, and the other forms translate into (NOT α), (OR α β), (AND α β), (IMPLIES α β), and (EQUIV α β). Thus $(A \wedge B) \equiv (B \vee C)$ translates into (EQUIV (AND A B) (OR B C)).

Problem Set 10

1. Write a LISP predicate wff[x] which is T if x is a well-formed formula of the propositional logic and F otherwise; wff itself should never be undefined.

2. An interpretation for the propositional variables of a formula is a list (in any order) pairing each name with T or F. For example, ((A T) (B F) (C F)) is the interpretation used in the example preceding problem set 9. Write a LISP predicate propeval[e, a], where e is a propositional formula and a is an interpretation for it. Propeval should interpret the formula as T or F.

If a propositional formula has exactly n different variables in it, then there are 2^n different interpretations for the formula. This is the number of different ways to assign T or F to n things.

Definition 3.2

If every interpretation of a formula is T, then the formula is called a tautology.

If at least one interpretation of a formula is T, then the formula is called satisfiable.

If no interpretation of a formula is T, then the formula is called inconsistent.

Corollary 3.3

Every tautology is satisfiable, but not vice versa. If α is a tautology, then $\neg\alpha$ is inconsistent. If α is inconsistent, then $\neg\alpha$ is a tautology. If α is satisfiable and is not a tautology, then $\neg\alpha$ is also satisfiable and is not a tautology.

Problem Set 11

1. Which of the following are tautologies? Which of the rest are satisfiable or inconsistent?

a. $A \vee \neg A$
b. $A \wedge \neg B$
c. $A \supset B$
d. $A \supset (A \vee B)$
e. $A \supset \neg A$
f. $(\neg A \vee \neg B) \supset \neg(A \wedge B)$
g. $(A \wedge B) \equiv (A \vee B)$
h. $\neg(A \supset B) \wedge B$

2. Write a LISP function vars[x] such that if x is a formula of the propositional calculus, then vars[x] is the set of all the propositional variables that occur in x.

3. Write a LISP function tabs[x] such that if x is a set of propositional variables as is generated above, then the value of tabs[x] is a list of all 2^n interpretations for these variables. For example, consider the formula (IMPLIES (AND A C) (OR B C)). Then vars of this formula is (A B C) or some permutation thereof, and tabs of (A B C) is some permutation of (((A T) (B T) (C T)) ((A T) (B T) (C F)) ((A T) (B F) (C T)) ((A T) (B F) (C F)) ((A F) (B T) (C T)) ((A F) (B T) (C F)) ((A F) (B F) (C T)) ((A F) (B F) (C F))).

4. Write a LISP predicate taut[x] that is T if x is a tautology, and F otherwise.

5. Write a LISP predicate sat[x] which is T if x is satisfiable, and F otherwise.

Two propositional formulas α and β are said to be equivalent formulas if $\alpha \equiv \beta$ is a tautology. The following table of equivalences contains many

well-known properties of propositional formulas. They are given in the form of schemas where the Greek letters are used to represent any formulas. So $\alpha \wedge \beta$ being equivalent to $\beta \wedge \alpha$ means more than $A \wedge B$ being equivalent to $B \wedge A$. It means, for example, that $(A \supset B) \wedge C$ is equivalent to $C \wedge (A \supset B)$.

Equivalences of Propositional Formulas:

1.	$\alpha \vee \beta$	$\beta \vee \alpha$	commutativity of "or"
2.	$\alpha \vee (\beta \vee \gamma)$	$(\alpha \vee \beta) \vee \gamma$	associativity of "or"
3.	$\alpha \wedge \beta$	$\beta \wedge \alpha$	commutativity of "and"
4.	$\alpha \wedge (\beta \wedge \gamma)$	$(\alpha \wedge \beta) \wedge \gamma$	associativity of "and"
5.	$\neg\neg\alpha$	α	elimination of double negation
6.	$\neg(\alpha \vee \beta)$	$\neg\alpha \wedge \neg\beta$	DeMorgan's Laws
7.	$\neg(\alpha \wedge \beta)$	$\neg\alpha \vee \neg\beta$	DeMorgan's Laws
8.	$\alpha \vee (\beta \wedge \gamma)$	$(\alpha \vee \beta) \wedge (\alpha \vee \gamma)$	distributive law
9.	$\alpha \wedge (\beta \vee \gamma)$	$(\alpha \wedge \beta) \vee (\alpha \wedge \gamma)$	distributive law
10.	$\alpha \vee \alpha$	α	idempotency
11.	$\alpha \wedge \alpha$	α	idempotency
12.	$\alpha \supset \beta$	$\neg\alpha \vee \beta$	elimination of "implies"
13.	$\alpha \equiv \beta$	$(\alpha \supset \beta) \wedge (\beta \supset \alpha)$	elimination of "equiv"

From equivalences 1, 2, and 10, we see the justification for regarding "or" as having an indefinite number of arguments in any order without repetitions. From equivalences 3, 4, and 11, the same holds for "and". So we can write $A \vee B \vee C$ without showing which way it associates, and in s-expression language we can write (OR A B C), permitting AND and OR to have an indefinite number of arguments. It is consistent with this practice to assume that (OR) = F and that (AND) = T.

Problem 12

Rewrite wff, propeval, taut and sat to handle AND and OR with an indefinite number of arguments.

§3.3 Deduction

If we are given "It will not rain tomorrow," and "If it does not rain

tomorrow we shall go to the beach," then we may draw the conclusion "Tomorrow we shall go to the beach." This is called a deduction.

$$\frac{A \qquad A \supset B}{B}$$

Ultimately, we want to be able to make use of deductions of considerable length, and to arrive at conclusions that are not immediately obvious from the given statements. The rules for making deductions in propositional logic are extremely simple. A deduction consists of a sequence of numbered lines. Each line is a propositional formula, and the last one is the desired conclusion. There must be a reason or justification for writing each line, and there are only three kinds of justification. A line is justified (a) because it is given, (b) because it is an axiom, or (c) because it follows from previous lines by using a rule of inference.

As <u>axioms</u> for propositional logic, we shall allow any formula that is a tautology.

The only rule of inference for propositional logic is <u>modus-ponens</u>. This rule states that if there is a line in the deduction which is the formula α, and if there is another line in the deduction which is the formula $\alpha \supset \beta$, then we may deduce the formula β. We call α and $\alpha \supset \beta$ the <u>antecedents</u> of the inference, and β the <u>consequent</u> of the inference. Either antecedent may appear before the other in the deduction, but the consequent must follow the antecedents.

The following deduction shows that if we assume the formulas numbered 1 thru 6 below, the formula numbered 21 can be deduced.

1.	$A \supset B$	given
2.	$B \supset C$	given
3.	$C \supset D$	given
4.	$A \vee E$	given
5.	$D \supset G$	given
6.	$E \supset G$	given
7.	$(A \supset B) \supset ((B \supset C) \supset (A \supset C))$	tautology
8.	$(B \supset C) \supset (A \supset C)$	Mp 1, 7

9.	$A \supset C$	Mp 2, 8
10.	$(A \supset C) \supset ((C \supset D) \supset (A \supset D))$	tautology
11.	$(C \supset D) \supset (A \supset D)$	Mp 9, 10
12.	$A \supset D$	Mp 3, 11
13.	$(A \vee E) \supset ((A \supset D) \supset (D \vee E))$	tautology
14.	$(A \supset D) \supset (D \vee E)$	Mp 4, 13
15.	$D \vee E$	Mp 12, 14
16.	$(D \vee E) \supset ((D \supset G) \supset (G \vee E))$	tautology
17.	$(D \supset G) \supset (G \vee E)$	Mp 15, 16
18.	$G \vee E$	Mp 5, 17
19.	$(G \vee E) \supset ((E \supset G) \supset G)$	tautology
20.	$(E \supset G) \supset G$	Mp 18, 19
21.	G	Mp 6, 20

§3.4 Proof-Checking

We are now in a position to attempt a miniature proof-checker for propositional logic. It is a predicate of three arguments, proofchk[g, c, d], where g is a list of given formulas, c is a conclusion, and d is a deduction. If all the arguments have the correct format, and d is a valid deduction proving c starting with g, then the value of proofchk is T. Otherwise it is F.

We have already specified an s-expression language for propositional formulas allowing AND and OR to have an indefinite number of arguments. The format of the arguments of proofchk is as follows:

g: A list of propositional formulas.

c: A single propositional formula.

d: A list of steps. Each step is a list of three items. The first item is a number. The steps are numbered consecutively, 1, 2, 3 ... The second items is a formula. The third item is the justification for the formula. It can be (i) GIVEN, (ii) TAUT, or (iii) (MP m n), where m and n are numbers of previous lines.

Using the previous example, g = ((IMPLIES A B) (IMPLIES B C) (IMPLIES C D) (OR A E) (IMPLIES D G) (IMPLIES E G)), c = G and d = ((1 (IMPLIES A B) GIVEN) (2 (IMPLIES B C) GIVEN) ... (20 (IMPLIES (IMPLIES E G) G) (MP 18 19)) (21 G (MP 6 20))).

For a deduction to be valid, it must have the correct syntax and in addition:

1. If the justification for a step is TAUT, then the body of the step must be a tautology.

2. If the justification for a step is GIVEN, then the body of the step must be a member of the list g.

3. If the justification for a step is (MP m n), then letting the body of the step be β, and letting the body of step m be α, the body of step n must be (IMPLIES α β). Furthermore both m and n must be less than the number of the step being justified.

4. The body of the last step must be c, the conclusion.

A recursive definition of proofchk follows:

```
proofchk[g, c, d] ← wfflis[g] ∧ wff[c] ∧ wfsteplis[d] ∧ ¬null[d] ∧
      steporder[d] ∧ caadr[last[d]] = c ∧ proofchk1[g, d, d]

wfflis[x] ← [null[x] → T, atom[x] → F, T → wff[car[x]] ∧ wfflis[cdr[x]]]

s1[x] ← ¬atom[x] ∧ null[cdr[x]]

s2[x] ← ¬atom[x] ∧ s1[cdr[x]]

s3[x] ← ¬atom[x] ∧ s2[cdr[x]]

wfsteplis[x] ← [null[x] → T, atom[x] → F, T → wfstep[car[x]] ∧
      wfsteplis[cdr[x]]]

wfstep[x] ← s3[x] ∧ num[car[x]] ∧ wff[cadr[x]] ∧ wfjust[caddr[x]]

wfjust[x] ← x = GIVEN ∨ x = TAUT ∨ [s3[x] ∧ car[x] = MP ∧
      num[cadr[x]] ∧ num[caddr[x]]]

steporder[x] ← [s1[x] → T, T → caar[x] + 1 = caadr[x] ∧ steporder[cdr[x]]]

proofchk1[g, d, q] ← [null[q] → T, T → stepchk[g, d, car[q]] ∧
      proofchk1[g, d, cdr[q]]]

stepchk[g, d, s] ← [caddr[s] = TAUT → taut[cadr[s]], caddr[s] = GIVEN →
      member[cadr[s], g], T → mpchk[d, s, caddr[s]]]

mpchk[d, s, j] ← caar[d] ≤ cadr[j] ∧ cadr[j] < car[s] ∧ caar[d] ≤ caddr[j] ∧
      caddr[j] < car[s] ∧ cadr[fetch[caddr[j], d]] = list[IMPLIES,
      cadr[fetch[cadr[j], d]], cadr[s]]

last[x] ← [s1[x] → car[x], T → last[cdr[x]]]

fetch[n, x] ← [null[x] → NIL, n = caar[x] → car[x], T → fetch[n, cdr[x]]]
```

Problem 13

If you have access to an interacting LISP system, program a more practical proofchecker in which you can specify the given and the desired conclusion, and then enter lines of proof. The program should give diagnostics when it does not accept lines offered to it, and let you try again.

There is a difficulty with this method of proof that prevents us from making deductions in a reasonable length of time in certain cases where we would expect to be able to do so. Suppose α is a formula containing 40 different propositional variables. It should be easy to show that $\alpha \vee \neg\alpha$ is a tautology. But if we set taut to check whether this is a tautology, then it will try to form a list of 2^{40} interpretations and will fail on any existent computer.

One way around this difficulty is to make use of the idea of substitution instances. If α is any formula, then replacing some of its propositional variables with formulas generates a substitution instance of the original formula. If a particular propositional variable is to be replaced, then all occurrences of it must be replaced, and must be replaced by the same formula. For example, a substitution instance of $A \wedge (B \supset A)$ could be $(C \vee D) \wedge ((B \equiv D) \supset (C \vee D))$.

A substitution instance of a tautology is always a tautology. So we can add to our deductive system for the propositional logic one more rule of inference:

Substitution rule:

> A line in a deduction is justified if it is a substitution instance of a previous line, and that previous line is a tautology.

Problem 14

Modify proofchk to allow for substitution instances of tautologies. The justification for such a line will have the form (INST n), where n refers to a previous line, whose justification is TAUT.

CHAPTER FOUR

PROPOSITIONAL LOGIC II

Preview of Chapter Four

This chapter develops a theory of propositional logic. The conceptual framework, and even many of the definitions will carry over directly to the theory of first order logic.

The theory naturally divides itself into two aspects. The first of these, called proof theory, concerns itself with propositional formulas, and deductions, viewed as formal objects to be manipulated, without any concern for what they are intuitively supposed to mean. The second aspect of the theory is model theory whose purpose is to validate the logic with respect to its intended meaning. The most important theorems for our purpose are those that relate proof theory to model theory.

§4.1 Proof Theory

At any given time, it is useful to limit the discussion of propositional logic to those formulas that contain only a particular set of propositional variables.

Definition 4.1

A vocabulary is any non-empty set of propositional variables.

A language (of propositional logic) is the set of all formulas containing only variables from a particular vocabulary.

A vocabulary may be finite or infinite. Every vocabulary defines a unique language. All languages are infinite sets even when based on a finite vocabulary. If the formula α is a member of the language L, then the formula $\neg\alpha$ is also a member of L and vice versa. If the formulas α and β are both members of L, then $\alpha \wedge \beta$, $\alpha \vee \beta$, etc. are also members of L. Conversely, if a compound formula is a member of L, then its constituents are members of L.

When using logic as a deductive tool, we frequently select some set of formulas belonging to a language as the axiomatization of our subject matter. Such a set of axioms can be called a theory. We then want to discuss those formulas that can be deduced within the theory. These are sometimes called theorems. This motivates the following definitions:

Definition 4.2

If L is a language, then a theory is any subset of L. If T is a theory of L, and α is any formula of L, then the notation "$T \vdash \alpha$" means that there exists a deduction (as specified in Chapter Three) such that every given formula of the deduction is in T, and the conclusion is α. We can read this as "α is deducible from T". The set of all α in L such that $T \vdash \alpha$ is the set of theorems of T for which we write Th(T).

Definition 4.3

The theory of T is said to be inconsistent if there is some formula α such that $T \vdash \alpha$, and $T \vdash \neg\alpha$. Otherwise T is consistent.

Corollary 4.4

If $T \subset L$ is the empty theory, then Th(T) includes all tautologies of L. If $T \subset L$ is inconsistent, then Th(T) = L. If T and R are theories of L, then $(Th(T) \cup Th(R)) \subset Th(T \cup R)$.

Definition 4.5

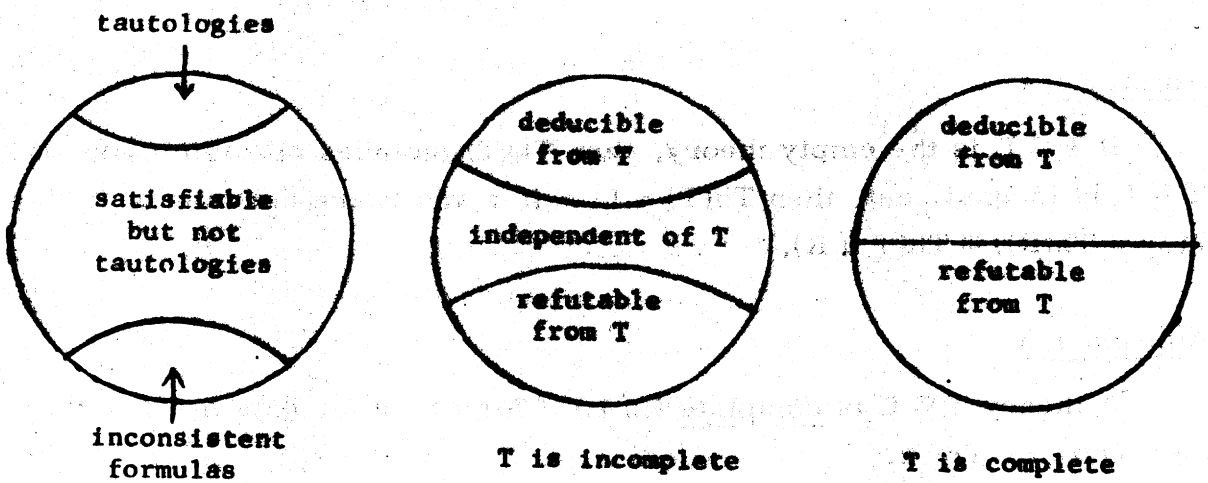
A theory $T \subset L$ is complete (in L) if for every formula $\alpha \in L$ either either $T \vdash \alpha$, or $T \vdash \neg\alpha$.

It is important to observe that the completeness of a theory is relative to the language of which it is a part. The theory is complete if all the formulas of L are provable or refutable from T, and none of them are left undecided. But the same theory may be incomplete with respect to a larger language.

Definition 4.6

If $T \subset L$ is a theory, and $\alpha \in L$, then if neither $T \vdash \alpha$, nor $T \vdash \neg\alpha$, then α is said to be independent of T.

We have seen that formulas may be divided into three classes, tautologies, inconsistent formulas, and those that are satisfiable but not tautologies. Given any consistent theory T in the language L, the formulas of L can then be divided into three disjoint classes relative to T: (i) those that are deducible from T, which includes the tautologies, as a subset, (ii) those whose negations are deducible from T, which we can call the formulas refutable from T and which includes all the inconsistent formulas as a subset, and (iii) those that are independent of T. If T is a complete theory, this last class is empty.



Problem Set 15

1. Which of the following theories are inconsistent? Which are complete within the smallest language containing them?

a. A
$\neg A \vee \neg B$
$\neg B \supset C$
$\neg C$

b. $A \vee B$

c. $A \vee B$
$\neg B$

d. A
$\neg B$
C

2. Prove that if T is any consistent theory in L, there is a theory T' in L which is complete and consistent, and such that $T \subset T'$.

3. Show that every complete theory has a canonical form.

The main theorem of this section is known as the deduction theorem (for propositional logic). It is the formalization of the intuitive proof technique whereby when we want to prove a result having the form "A implies B", we assume A and then derive B.

Both the statement of the theorem, and the method of proof are typical of proof theory. The statement of the theorem is simply that if a certain deduction exists (and a deduction is itself a formal object as defined in Chapter Three), then a certain other deduction must also exist. The proof of the theorem makes no appeal to the meaning of propositional logic, but merely describes how to obtain the second deduction if the first one is given. This is known as a constructive proof.

Theorem 4.7 (Deduction Theorem)

If $T \cup \{\alpha\} \vdash \beta$, then $T \vdash \alpha \supset \beta$.

Proof: The assumption of this theorem is that there is a deduction of the formula β in which only the formulas of T and the formula α are justified as given. Let this deduction be the sequence of formulas $\beta_1, \ldots, \beta_n$ where $\beta_n = \beta$. We shall use the method of mathematical induction to show that for each i, where $1 \leq i \leq n$, it is the case that $T \vdash \alpha \supset \beta_i$. This is sufficient to prove the theorem, because $T \vdash \alpha \supset \beta_n$ is the desired result.

By the induction principle, it is sufficient to show that if $T \vdash \alpha \supset \beta_j$ for $1 \leq j < i$, then $T \vdash \alpha \supset \beta_i$. There are four cases to consider: (i) if β_i is a tautology, then $\alpha \supset \beta_i$ is also a tautology, and so $T \vdash \alpha \supset \beta_i$. (ii) If β_i follows from β_j and β_k by modus-ponens in the given proof, where β_k is $\beta_j \supset \beta_i$, and $j < i$, and $k < i$, then by the induction assumption $T \vdash \alpha \supset \beta_j$ and $T \vdash \alpha \supset (\beta_j \supset \beta_i)$. Then since $(\alpha \supset \beta_j) \supset (\alpha \supset (\beta_j \supset \beta_i)) \supset (\alpha \supset \beta_i)$ is a tautology, by two applications of modus-ponens we get $\alpha \supset \beta_i$. (iii) If β_i is justified as given in the first proof, and $\beta_i \in T$, then $T \vdash \beta_i$, and since $\beta_i \supset (\alpha \supset \beta_i)$ is a tautology, by modus-ponens we get $T \vdash \alpha \supset \beta_i$. (iv) If β_i is justified as a given in the first proof, and β_i is α. Then $T \vdash \alpha \supset \beta_i$ because this formula $(\alpha \supset \alpha)$ is a tautology.

A constructive proof usually tells us more than is required for the theorem. This proof, for example, tells us that the deduction $T \vdash \alpha \supset \beta$ is computable from the deduction $T \cup \{\alpha\} \vdash \beta$. Moreover, the second deduction is at most three times as long as the first. The opposite of a constructive proof is an existential proof.

Problem 16

Let T be the theory $\{A \supset B, B \supset C\}$. Then $T \cup \{A\} \vdash C$, and we write out this deduction in full:

1.	A	given
2.	$A \supset B$	given
3.	B	Mp 1, 2
4.	$B \supset C$	given
5.	C	Mp 3, 4

The deduction theorem tells us that $T \vdash A \supset C$. Obtain this deduction by following the construction given in the proof of the deduction theorem. Is there a shorter deduction for $T \vdash A \supset C$?

§4.2 Model Theory

While proof theory is concerned with the properties of deductions, model theory is concerned with the meaning of the formulas. A formula is a logical compound of propositions, each of which is regarded as true or false

in some context. The purpose of a model is to supply that context; therefore:

Definition 4.8

A model in a language L is a function from the vocabulary of L into π.

If the vocabulary of L is finite with n members, then there are 2^n different models for L. If the vocabulary of L is infinite, then there are infinitely many models for L, in fact, uncountably many.

If M is a model in the language L, and $\alpha \in L$, then M assigns a truth value to each variable occurring in α. Then, using the truth tables for the propositional connectives, or else using some procedure such as propeval of Chapter Three, a truth value can be assigned to α.

Definition 4.9

If α evaluates to the value T using the model M then we say that M satisfies α, and we use the notation "$M \models \alpha$" to express this concept.

Corollary 4.10

If M is a model in L, and $\alpha \in L$, then either $M \models \alpha$, or $M \models \neg\alpha$. If for every M in L, $M \models \alpha$, then α is a tautology. If there is at least one model M such that $M \models \alpha$, then α is satisfiable. If there is no such M, then α is inconsistent.

Definition 4.11

If $T \subset L$ is a theory, and M is a model in L, and if $M \models \alpha$ for every $\alpha \in T$, then we say that M is a model for T, or M satisfies T, and we write $M \models T$.

So far, we have used the symbol "$\models$" to relate models to formulas or theories. We can also use "$\models$" to express the idea that in any context where the theory T is satisfied, the formula α is also satisfied.

Definition 4.12

If $T \subset L$, and $\alpha \in L$, and if every model in L that satisfies T also

satisfies α, then we say that T semantically implies or semantically entails α, and we write $T \models \alpha$.

It is important to realize that $T \models \alpha$, or T semantically implies α, is not the same thing as saying $T \vdash \alpha$, or α is deductible from T, at least not until we have proven this to be the case.

The main result of this section is the compactness theorem, a rather surprising result when first seen. Suppose some infinite theory is not satisfiable by any model. One might think that this is a property of the theory as a whole. But the compactness theorem states that the unsatisfiability can always be localized to some finite portion of the theory.

An unsatisfiable theory is one that has no model. An inconsistent theory is one for which there is a formula α such that both α and $\neg\alpha$ can be deduced from the theory. The former concept is model theoretic, while the latter is proof theoretic. If a theory is inconsistent, then it is obvious that some finite sub-theory is also inconsistent because the deduction of the inconsistency had to come from finitely many given formulas. But we have not yet proved that unsatisfiable and inconsistent are equivalent concepts. The compactness theorem is a result preliminary to proving this.

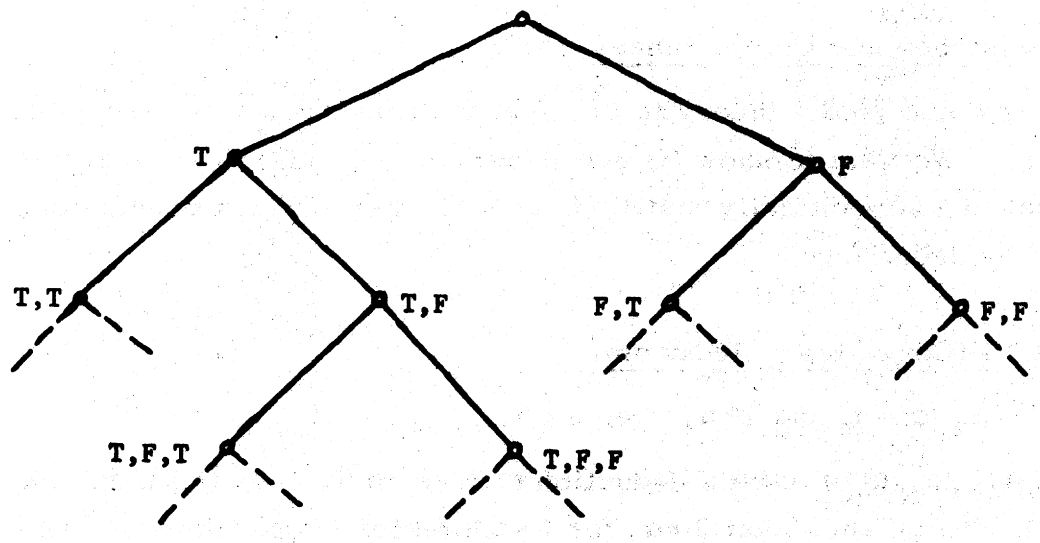
Theorem 4.13 (Compactness Theorem)

If a theory is unsatisfiable, then it has a finite sub-theory which is unsatisfiable.

Proof: If the theory T is finite, then the theorem is trivial because the sub-theory T' is taken to be T. The theorem did not promise that T' was a proper subset. If the vocabulary of T is finite, then there are 2^n models where n is the number of propositional variables in T. None of these models satisfies T, and therefore each one falsifies some formula of T. This set of formulas is not satisfiable, and is the required T'.

Suppose the vocabulary of T is infinite. Let the propositional variables of T be enumerated in some order as the sequence $a_1, a_2, \ldots$ We shall call a function from some initial segment of this sequence into π a "partial model". A partial model assigns truth values to a_1 thru a_n for some $n \geq 0$. We can picture all partial models as nodes on an infinite tree. The

first node is the empty partial model. The next level containing two nodes assigns T and F to a_1, and the third level containing four nodes assigns T and F to a_1 and a_2 in 4 different ways, etc.



A partial model assigns truth values only to those formulas of T whose propositional variables are among those that the particular partial model interprets. If a partial model interprets at least one formula of T as false, it will be called a "terminal". We now "prune" the tree by cutting off all nodes that are descendents of terminals. If the pruned tree has finitely many nodes, then for each terminal we select a formula which is falsified by that terminal. The set of these formulas is the required finite T', because if M is any model, then some initial sequence of M is the same as some terminal. So there is a formula in T' which is not satisfied by M. Therefore T' is an unsatisfiable theory.

Now suppose that the pruned tree after eliminating descendents of terminals is still infinite. Then there must be some infinite descending path passing through infinitely many nodes. This is because if the tree is infinite, then either the left or right half of it is infinite. Then either the left or right half of this half is infinite, etc. But such a path constitutes a model. Furthermore, this model does not falsify any formulas since none of the nodes

it passes through is a terminal. So this model must satisfy the theory T contrary to assumption. Therefore the pruned tree cannot be infinite.

§4.3 Consistency and Completeness

We now use model theory to critique the consistency and completeness of deduction. We want to show (i) that deduction only allows us to obtain conclusions that are semantically justified, and (ii) that all such conclusions can be obtained by deduction.

Theorem 4.14 (Consistency Theorem)

If $T \subset L$, $\alpha \in L$ and $T \vdash \alpha$, then $T \models \alpha$.

Proof: Let $\alpha_1, \ldots, \alpha_n = \alpha$ be a deduction of α from T. By the induction principle, if we can show that $T \models \alpha_j$ for $j < i$ implies $T \models \alpha_i$, then we can conclude that $T \models \alpha_i$ for each i, and, in particular, $T \models \alpha$. There are three cases: (i) If α_i is a tautology, then $T \models \alpha_i$ because all models of L satisfy α_i. (ii) If α_i is given, then $\alpha_i \in T$ and so $T \models \alpha_i$. (iii) If α_i follows from α_j and α_k by modus-ponens, where α_k is $\alpha_j \supset \alpha_i$, then by the induction hypothesis, $T \models \alpha_j$ and $T \models \alpha_j \supset \alpha_i$, so if $M \models T$, then $M \models \alpha_j$ and $M \models \alpha_j \supset \alpha_i$. From the truth table of "$\supset$", it is seen that $M \models \alpha_i$, and so $T \models \alpha_i$.

Lemma 4.15

If $T \not\models \alpha$, then there is a finite subset T' of T such that $T' \not\models \alpha$.

Proof: $T \cup \{\neg\alpha\}$ is an unsatisfiable theory, since every model that satisfies T also satisfies α, and therefore does not satisfy $\neg\alpha$. According to the compactness theorem, there is a finite subset of $T \cup \{\neg\alpha\}$ which is also unsatisfiable. We can always include $\neg\alpha$ in this set, so it can be written $T' \cup \{\neg\alpha\}$ where $T' \subset T$ is finite. If $M \models T'$, then M cannot satisfy $\neg\alpha$, and so $M \not\models \alpha$. Therefore $T' \not\models \alpha$.

Theorem 4.16 (Completeness Theorem)

If $T \subset L$, $\alpha \in L$, and $T \not\models \alpha$, then $T \vdash \alpha$.

Proof: By lemma 4.15 there is a finite $T' \subset T$ such that $T' \not\models \alpha$. Let $T' =$

$\{\beta_1, \ldots, \beta_2\}$. Then $\beta_1 \supset (\beta_2 \supset \ldots (\beta_n \supset \alpha) \ldots)$ is a tautology, and there is a deduction starting from this tautology, introducing each β_i as given, and then detaching it from the tautology using modus-ponens, such that the conclusion of this deduction is α.

Corollary 4.17

If there is at least one formula that cannot be deduced from the theory T, then T is satisfiable.

Problem 17

Prove corollary 4.17.

CHAPTER FIVE

RECURSIVE FUNCTIONS AND SETS

Preview of Chapter Five

This chapter continues from where we left off in Chapter Two. There we formalized the notion of a recursive function, and showed that there was a universal recursive function apply. In this chapter, we continue the discussion of recursive function theory by demonstrating that there are perfectly well defined functions that are not recursive. It is surprisingly easy to get such a result once we have a universal function. The theory goes a bit further by showing that there are functions that are in some sense not even halfway recursive.

§5.1 Recapitulation

To summarize the results of Chapter Two briefly, the following schema represents a sequence of n recursive definitions:

$$\varphi_1[\xi_1, \ldots, \xi_{m_1}] \leftarrow \epsilon_1$$

$$\varphi_n[\xi_1, \ldots, \xi_{m_n}] \leftarrow \epsilon_n$$

Such a sequence defines n partial recursive functions, gives them the names φ_1 thru φ_n, and specifies procedures for computing these functions which terminate with a value whenever the partial function has a value, and are otherwise undefined or fail to terminate. The recursive function specified by the procedure may not be the one we expect, but it must exist because the

behavior of the procedure is taken as its definition.

There is furthermore a well-defined effective method of coding a sequence of recursive definitions into a single s-expression, and there is a partial recursive function called apply such that if β is the coded s-expression just mentioned, and φ_1 has exactly k arguments (that is, $m_1 = k$), and α_1 thru α_k are any s-expressions, then:

$$\varphi_1[\alpha_1, \ldots, \alpha_k] \cong \text{apply}[\beta, \text{list}[\alpha_1, \ldots, \alpha_k]]$$

where the symbol "$\cong$" here means "strong equality" in the sense that either both sides have the same value, or both sides are undefined. (We can also compute any other of the φ_i merely by putting its definition first when coding β.)

§5.2 Turing's Halting Theorem

A. M. Turing first proved this halting theorem using his conceptualized computers now known as Turing Machines. He showed that there is no computer that can always predict whether or not another computer will halt or continue to run indefinitely, after studying the structure and initial state of that other computer. It is assumed here that all computers have access to an unlimited supply of initially blank auxiliary storage. Of course a computer can predict that another computer will halt by simulating its behavior until a halt is encountered. But there is no way to do this without danger that the computer doing the simulation will itself not halt in some cases. A proof along these lines can be found in [Davis].

We define the total binary predicate halt[x, y] as follows: If apply[x, y] is defined, then halt[x, y] is T; otherwise halt[x, y] is F. The predicate halt is certainly meaningful and well defined. But we have not specified any effective means to compute it.

Theorem 5.1 (First Halting Theorem)

The predicate halt is not recursive.

Proof: Suppose, to the contrary, that halt is recursive. Then there is a

sequence of recursive definitions, the first of which computes halt, and the rest of which are auxiliary functions for halt. Then we can define the recursive function diag as follows:

$$\text{diag}[x] \leftarrow [\text{halt}[x, \text{list}[x]] \rightarrow \text{list}[\text{apply}[x, \text{list}[x]]], T \rightarrow \text{NIL}]$$

The function diag is recursive because it has been effectively defined from apply and halt both of which are, or are presumed to be, recursive. Furthermore, diag is a total function because halt is total, and while apply is not total, it only gets to see those arguments certified by halt as producing a value for apply. Because diag is recursive, its definition can be coded into an s-expression which we shall call diag*. This is a list of the translation of the recursive definition of diag written above, followed by the definitions for halt and its auxiliary functions, and apply and its auxiliary functions. Now consider the value of diag[diag*]. This value must exist because diag is total. Therefore, by the interpreter theorem, apply[diag*, list[diag*]] must be defined and have the same value, and hence halt[diag*, list[diag*]] is T. But then from the recursive definition for diag, we have:

$$\text{diag}[\text{diag}^*] = \text{list}[\text{apply}[\text{diag}^*, \text{list}[\text{diag}^*]]] = \text{list}[\text{diag}[\text{diag}^*]]$$

This is a contradiction because it asserts that some s-expression is equal to list of itself. This is like having a number n such that $n = n'$. Since we have arrived at a contradiction using correct reasoning, we must conclude that our original premise that halt is recursive is not true.

This proof is confusing at first sight. If you study it carefully, you will see that it is really no different in its basic technique from Cantor's diagonalization proof that the real numbers are not countable. That is why we have called the self-applicative function "diag". Most undecidability and incompleteness proofs involve some sort of diagonalization technique.

§5.3 Recursive and Recursively Enumerable Sets

Definition 5.2

A set of s-expressions is recursive if there is a total unary recursive predicate which is T for members of the set, and F otherwise.

Since the numbers are a subset of the s-expressions, this definition extends to numbers. For a set of numbers to be recursive, it is sufficient to have a predicate which is defined only for numbers, and is T for members of the set and F for numbers that are not members of the set. This is because the set of all numbers is recursive using the basic predicate num.

If a set is recursive, then there is an effective test for membership in the set which terminates either way. We have just proven that the set of all s-expressions x such that apply[car[x], cadr[x]] is defined is not a recursive set.

There is a weaker condition than recursiveness called recursive enumerability. It applies to sets where there is a membership procedure that always terminates when the answer is yes, but may not terminate when the answer is no.

Definition 5.3

A set of s-expressions is recursively enumerable (abbreviated to r.e.) if it is the set of values for some total unary recursive function defined on the domain of numbers.

This definition creates the picture of a machine that runs continuously, and from time to time prints out some s-expression. Every s-expression that is a member of the set will be printed eventually, and only members of the set will be printed. There may be repetitions. But we cannot always conclude that some s-expression is not a member of the set, because we may not have waited long enough. This is a good intuitive view of recursive enumerability.

Definition 5.4

A predicate is recursively enumerable if the set of arguments for which it is T is r.e.

There are many alternative definitions for a r.e. set, some of which are given in the following corollary.

Corollary 5.5

(a) A set is r.e. if and only if it is the range (set of values) of some total recursive function.

(b) A set is r.e. if and only if it is the range of some partial recursive function.

(c) A set is r.e. if and only if it is the domain of definition for some partial recursive function (i.e., the set of s-expressions on which the partial function is defined).

(d) If a set is r.e. and its complement is also r.e., then both are recursive. (This means complement with respect to the set of all s-expressions, but it is also true if we take a complement with respect to the set of numbers, or any other recursive set.)

Problem Set 18

1. Prove all the parts of corollary 5.5.

2. Show that halt is a r.e. predicate.

3. Using theorem 5.1, and corollary 5.5, part (d), specify some set which is not r.e.

The last result of the chapter is a stronger halting theorem in which we demonstrate the existence of a predicate that is not even r.e. We define the total unary predicate tot[x] to be T if and only if x is a sequence of recursive definitions which is syntactically well formed, and furthermore specifies the computation of a total unary function of the s-expressions. Tot[x] is F if x is not a well-formed sequence of definitions, or if it defines a non-unary function, or if it defines a non-total function. Tot itself is never

undefined.

Theorem 5.6 (Second Halting Theorem)

The predicate tot is not r.e.

Proof: Assume to the contrary that tot is r.e. Then the set of arguments x such that tot[x] is T is a r.e. set, and there is some total recursive numeric function totenum which enumerates this set. Now consider the function diag2 defined by:

diag2[x] ← [num[x] → list[apply[totenum[x], list[x]]], T → NIL]

Given our premises, diag2 is evidently a total unary recursive function. Letting its definition sequence be the s-expression diag2*, we have tot[diag2*] = T. Therefore, there is some number k such that totenum[k] = diag2*. Then diag2[k] = list[apply[diag2*, list[k]]] = list[diag2[k]]. This is a contradiction, so the initial assumption that tot is r.e. must be false.

Once again, the distinction between a rope and a snake had proved too subtle for Western logic.

-The Adamantine Sherlock Holmes-

CHAPTER SIX

FIRST ORDER LOGIC - INTRODUCTION

Preview of Chapters Six, Seven and Eight

Chapter Six introduces the languages, theories and models of first order logic. It contains some basic definitions, and an intuitive exploration of the subject to develop skill in handling formulas and their meanings. No deep theorems are proven.

Chapter Seven defines and develops the theory of deductions. With the exception of the consistency theorem, all of Chapter Seven is proof-theoretic and constructive in nature. It contains all the basic results on provability that we shall need for the rest of the book. Chapter Seven is long and contains many difficult exercises. This seems necessary in order to develop some practical sense about deduction, which theoretical study alone is not likely to do.

Chapter Eight starts with the completeness theorem which is the central topic for the classical study of first order logic. The completeness theorem is then extended to logic with equality, and some consequences of the completeness theorem having philosophical implication are discussed.

§6.1 Languages, Formulas and Sentences

First order logic is much more subtle than propositional logic. In a certain theoretical sense, it is sufficient to represent any completely formalized process of deduction. Let us consider a very trivial deduction: Bowser is a dog. All dogs are mammals. All mammals are vertebrates. Therefore, there is at least one vertebrate. Each of these sentences is simple

rather than compound. If we call them A, B, C and D, respectively, there is no way to deduce D from A, B and C using propositional logic. The internal relations that make this an evident deduction are simply not available. These statements can be formalized in first order logic as follows:

$$\begin{array}{ll} A: & dog[BOWSER] \\ B: & \forall x(dog[x] \supset mammal[x]) \\ C: & \forall x(mammal[x] \supset vertebrate[x]) \\ \hline D: & \exists x(vertebrate[x]) \end{array}$$

When we define deduction in first order logic, it will be seen that there is a deduction of D given A, B and C.

In this example, there is a variable, x, an object, BOWSER, and three predicate names. A slightly more complicated example, containing a function name in addition to a predicate name is: The number three is not even. If a number is not even, then it is odd. If a number is odd, then its square is odd. Therefore there is some number the square of whose square is odd.

$$\begin{array}{l} \neg even[3] \\ \forall n(\neg even[n] \supset odd[n]) \\ \forall n(odd[n] \supset odd[square[n]]) \\ \hline \exists n(odd[square[square[n]]]) \end{array}$$

This is also a valid conclusion in first order logic.

Definition 6.1

A function name is an identifier.

A predicate name is an identifier.

A vocabulary for first order logic is a non-empty set of predicate names, and a (possibly empty) set of function names, together with a number (≥ 0) for each name called the degree of that name.

The purpose of the degree is to specify the number of arguments a predicate or function has.

A <u>term</u> is:

(i) a variable

or (ii) $\varphi[\tau_1, \ldots, \tau_n]$

where φ is a function name of degree n, and each of the τ_i is a term. (Note that this definition allows terms of arbitrary depth. Also, if the degree of φ is 0, then $\varphi[]$ is a term. A 0-ary term is called a constant.)

An <u>atomic formula</u> is $\psi[\tau_1, \ldots, \tau_m]$, where ψ is a predicate name of degree m, and each τ_i is a term. (Note that predicate names occur only outside function names, and that predicate names cannot be nested within each other. Also, if the degree of ψ is 0, then $\psi[]$ is an atomic formula.)

A <u>formula</u> is:

(i) an atomic formula

or (ii) $\neg(\alpha)$

or (iii) $(\alpha) \vee (\beta)$

or (iv) $(\alpha) \wedge (\beta)$

or (v) $(\alpha) \supset (\beta)$

or (vi) $(\alpha) \equiv (\beta)$

or (vii) $\forall \xi(\alpha)$

or (viii) $\exists \xi(\alpha)$

where α and β are formulas, and ξ is a variable. The symbols $\forall$ and $\exists$ are called the <u>universal quantifier</u> and the <u>existential quantifier</u>, respectively, and can be read as "for all" and "there exists".

Informally, we shall relax this grammar in several ways. We may drop some of the parentheses when this does not result in ambiguity for the reader. We do not specify associative grouping for "$\vee$" and "$\wedge$", since this makes no difference. We assume that "$\supset$" associates from the right, so that $p[x] \supset q[x] \supset r[x]$ means $p[x] \supset (q[x] \supset r[x])$. We use terms containing infixes, prefixes and suffixes in the same manner as in Chapter Two. Finally, we use objects as terms, which saves us the trouble of representing each object by a constant.

Throughout this book we shall use the convention that when a quantifier and its quantified variable are followed immediately by a left parenthesis, then the scope of the quantifier extends exactly as far as the matching right

parenthesis. For example, $\exists x(p[x]) \supset q[x]$ means $(\exists x(p[x])) \supset q[x]$, and not $\exists x((p[x]) \supset q[x])$.

Examples of Formulas:

a. $\forall x(p[x] \supset q[x])$

b. $\neg\exists n(n' = 0)$

c. $(n+m) \times p = n \times p + m \times p$

d. $\exists x(p[x]) \supset \forall x(p[x] \supset p[f[x]]) \supset \exists x(p[f[f[x]]])$

e. $(\neg p[] \vee \neg q[]) \equiv \neg(p[] \wedge q[])$

f. $p[x] \vee p[g[x]] \vee p[g[g[x]]]$

Definition 6.3

The set of all formulas using a given vocabulary is called a language.

Definition 6.4

In a formula having the form $\forall\xi(\alpha)$ or $\exists\xi(\alpha)$, every occurrence of the variable ξ is a bound occurrence of ξ. It is bound by the initial quantifier of the formula unless it is bound by some quantifier in α. An occurrence of a variable that is not bound is free.

It is only meaningful to talk about a particular occurrence of a variable being bound or free with respect to a particular formula. For example, within the formula $\exists x(p[x])$, the variable x is bound with respect to the whole formula, but free with respect to the subformula p[x]. In the formula $p[x] \supset \forall x[q[x]]$, the first occurrence of x is free, and the second and third occurrences are bound. In the formula $\forall y(p1[x, y] \vee \exists x(p2[x, y]))$, the only free variable (with respect to the entire formula) is the first occurrence of x.

Definition 6.5

If a formula has no free variables (with respect to itself), then it is called a sentence. A universal closure of a formula α is a sentence $\forall\xi_1 \ldots \forall\xi_n(\alpha)$ where the ξ_i are all the distinct free variables of α in any order.

§6.2 First Order Models

Definition 6.6

Let L be a first order language. Then a model in L is a package

containing the following:

(i) A non-empty set D called the domain of the model.
(ii) For each function name φ of degree n, a function $\tilde{\varphi}: D^n \to D$.
(iii) For each predicate name ψ of degree m, a predicate $\tilde{\psi}: D^m \to \mathbb{T}$.

When we speak of finite, infinite, countable or uncountable models, we are referring to the cardinality of their domains. It is important that the domain be non-empty, and that the functions and predicates that interpret the function and predicate names should be total.

The significance of models is that they specify semantics for formulas. Consider a language L, a formula $\alpha \in L$, and a model M in L. Temporarily, we need another entity called an interpretation. An interpretation I for the formula α, and the model M is a total function from the set of variables occurring in α into the domain of M. Given M, I and α, we can define a valuation for every sub-component of α. The valuation of a term will be a member of D (the domain of M), and the valuation of a formula will be a truth value (member of $\mathbb{T}$), defined as follows:

(i) If ξ is a term which is a variable, then $V(M, I, \xi) = I(\xi)$. That is, the valuation of ξ is the value in D assigned to it by the interpretation I.
(ii) If τ is a term having the form $\varphi[\tau_1, \ldots \tau_n]$, then $V(M, I, \tau) = \tilde{\varphi}(V(M, I, \tau_1), \ldots, V(M, I, \tau_n))$. That is, the valuation of τ is found by first obtaining valuations for the τ_i, which will be members of D, and then using $\tilde{\varphi}$, which is the function modeling the function name φ, to obtain a value in D from these arguments.
(iii) If β is an atomic formula $\psi[\tau_1, \ldots, \tau_m]$, then $V(M, I, \beta) = \tilde{\psi}(V(M, I, \tau_1), \ldots, V(M, I, \tau_m))$. This is a truth value.
(iv) The valuation of a formula having the form $\neg(\beta)$, $(\beta) \vee (\gamma)$, $(\beta) \wedge (\gamma)$, $(\beta) \supset (\gamma)$, or $(\beta) \equiv (\gamma)$ is obtained from $V(M, I, \beta)$ and $V(M, I, \gamma)$ using the truth tables for the propositional connectives.
(v) The valuation $V(M, I, \forall\xi(\beta))$ is T if $V(M, J, \beta)$ is T for

every J which is an interpretation identical to I except possibly for the value it assigns to the variable ξ. Otherwise, the valuation of $\forall\xi(\beta)$ is F.

(vi) The valuation $V(M, I, \exists\xi(\beta))$ is T if $V(M, J, \beta)$ is T for at least one J which is identical to I except possibly for the value it assigns to the variable ξ. Otherwise, the valuation of $\exists\xi(\beta)$ is F.

Proceeding from smaller to larger components in this manner, we see that a valuation $V(M, I, \alpha)$ is eventually defined. It is evident that the choice of the interpretation I is important only for the free variables in α, and that if α has no free variables, the valuation is independent of I. So if α is a sentence, we simply write $V(M, \alpha)$.

Definition 6.7

If M is a model in L, and α is a sentence in L, then if $V(M, \alpha) = T$, we say that M satisfies α, and write $M \models \alpha$.

If α is a sentence in L, and all models in L satisfy α, then α is valid. If at least one model satisfies α, then α is satisfiable. If no models satisfy α, then α is invalid.

The negation of a valid sentence is invalid and vice versa. We could draw the same sort of chart for valid, satisfiable-but-not-valid, and invalid sentences of first order logic, as we draw in Chapter Four for tautologies, satisfiable-but-not-tautological formulas, and inconsistent formulas of propositional logic. In fact, tautologies are a subset of valid formulas, if we define a first order formula that is valid from its propositional structure alone to be a tautology. Similarly, propositionally inconsistent formulas of first order logic are a subset of the invalid formulas.

So far, we have discussed only sentences. What about other formulas? It turns out that there are two ways of regarding a formula with free variables. One way is to see the formula as belonging to some context which supplies interpretations or restricts the meaning of the free variables. For example, in the pair of formulas:

$$2x + 2y = 1$$
$$x^2 + y^2 = 1$$

one probably wants to solve for all interpretations that satisfy both formulas. In the domain of real numbers, there are two of them. The other context for a formula having free variables is to regard the formula as meaning the same thing as its universal closure. For example:

$$\sin^2 x + \cos^2 x = 1$$

Here the meaning is that this assertion is true for all x, i.e., $\forall x(\sin^2 x + \cos^2 x = 1)$.

Definition 6.8

For any formula α, $M \models \alpha$ means that M satisfies a closure of α. A formula is valid, satisfiable, or invalid if its closure is valid, satisfiable or invalid, respectively.

Two formulas α and β are equivalent, if $\alpha \equiv \beta$ is valid.

Problem Set 19

1. Classify each of the following formulas as being either valid, invalid, or satisfiable but not valid.

a. $p[x] \vee \neg p[x]$
b. $\exists x(p[x]) \supset \forall x(p[x])$
c. $\forall x(p[x]) \wedge \neg \exists x(p[x])$
d. $\exists x \forall y(p[x, y]) \supset \forall y \exists x(p[x, y])$
e. $x + y = y + x$
f. $\forall x(\neg p[x]) \equiv \neg \exists x(p[x])$

2. The sentence $\forall x \forall \varepsilon \exists \delta \forall y(|x-y| < \delta \supset |f(x) - f(y)| < \varepsilon)$ interpreted on the domain of real numbers asserts that f is a continuous function. Write a formula that asserts that f is a uniformly continuous function. Does one of these conditions imply the other logically?

3. Show that each pair of formulas is equivalent:

a. $\forall \xi(\alpha \wedge \beta)$ $\quad$ $\forall \xi(\alpha) \wedge \forall \xi(\beta)$
b. $\exists \xi(\alpha \vee \beta)$ $\quad$ $\exists \xi(\alpha) \vee \exists \xi(\beta)$
c. $\forall \xi(\alpha \vee \beta)$ $\quad$ $\forall \xi(\alpha) \vee \beta$ $\quad$ where β has no free ξ

d.	$\exists\xi(\alpha \supset \beta)$	$\alpha \supset \exists\xi(\beta)$	where α has no free ξ
e.	$\forall_\xi(\alpha \supset \beta)$	$\exists\xi(\alpha) \supset \beta$	where β has no free ξ
f.	$\neg\exists\xi(\alpha)$	$\forall\xi(\neg\alpha)$	

§6.3 Theories

Definition 6.9

A theory in a language L is a subset of L.

If $T_1 \subset T_2$, then we say that T_2 is an extension of T_1, and T_1 is a contraction of T_2. If $L_1 \subset L_2$, then we say that L_2 is an extension of L_1, and L_1 is a contraction of L_2.

If $M \models \alpha$ for all $\alpha \in T$, then $M \models T$.

If $T \subset L$, and if $M \models T$ implies $M \vdash \alpha$ for all models M in L, then $M \models \alpha$.

A theory is satisfiable if it has a model.

Definition 6.10

Two models M_1 and M_2 in the language L are said to be first order equivalent if $M_1 \models \alpha$ if and only if $M_2 \models \alpha$ for every α in L. We write $M_1 \sim M_2$ to denote first order equivalence.

Let M_1 be a model in the language L_1, and let L_2 be an extension of L_1. If M_2 is a model in L_2 which has the same domain as M_1 and the same interpretations for all the function and predicate names of L_1, then M_2 is an expansion of M_1, and M_1 is a contraction of M_2. (The word "extension" applied to models has a different meaning from "expansion" and is not used in this book.)

Problem Set 20

1. Prove that if $T \subset L$ is a theory such that if α is any formula of L, then either $T \models \alpha$, or $T \models \neg\alpha$, then all models for T in L are first order equivalent.

2. Prove that if $T_2 \subset L_2$ is an extension of $T_1 \subset L_1$, and M_2 is a model in L_2 such that $M_2 \models T_2$, then there is a model M_1 in L_1 such that $M_1 \models T_1$, and M_1 is a contraction of M_2.

As an example of a theory, consider the theory of partial ordering, which belongs to the language having only the binary predicate "<" meaning less than. The theory is:

$$\neg x < x$$
$$x < y \supset y < z \supset x < z$$

This theory can belong to any language containing the predicate "<". Any model that satisfies this theory must be a partial ordering in the usual sense because these are the axioms for a partial ordering.

Suppose we extend this theory by adding to it the formula $\exists y(x < y)$. This says that given any object, there is another object greater than it. Then there must be another object greater than that, and so forth. By applying the second axiom, which is a transitive law, we see that any object on this chain is < any object occurring further along the chain. The first axiom says that no object is < itself. So we can conclude that this theory having three formulas has only infinite models. It is satisfied rather easily, for example, by the real numbers, or the natural numbers, or the transfinite ordinal numbers, by letting < have its customary meaning in each case.

Another example of a theory is the theory of groups under addition, formalized in a language with the binary predicate "=", the binary function "+", the unary function "-", and the constant 0.

$$x = x$$
$$x = y \supset y = x$$
$$x = y \supset y = z \supset x = z$$
$$x = y \supset u = v \supset x+u = y+v$$
$$x = y \supset -x = -y$$
$$x+(y+z) = (x+y)+z$$
$$x+0 = x$$
$$x+(-x) = 0$$

Any model that satisfies this theory is a group. There are, of course, many different groups, and in some models the plus sign must be interpreted by an operation usually called multiplication, and "0" must be interpreted by "1" or "e". The axioms in the left column are the axioms for equality in the language $\{=, +, -, 0\}$. They are necessary to assure that we will be able to prove those things that we need to prove about equality.

Definition 6.11

Let L be any language containing the binary predicate "=". We call such a language a language with equality. Most languages that we study will be languages with equality. The theory E_L, or the theory of equality for the language L is the following set of axioms:

(i) $x = x$

(ii) $x = y \supset y = x$

(iii) $x = y \supset y = z \supset x = z$

(iv) For each n-ary function name φ in L, the axiom

$$x_1 = y_1 \supset \ldots \supset x_n = y_n \supset \varphi[x_1, \ldots, x_n] = \varphi[y_1, \ldots, y_n]$$

(v) For each m-ary predicate name ψ in L, the axiom

$$x_1 = y_1 \supset \ldots \supset x_m = y_m \supset \psi[x_1, \ldots, x_m] \supset \psi[y_1, \ldots, y_m]$$

The number of such axioms depends on the size of the language L, and might be infinite. The first three axioms are the theory of equivalence relations. The rest of them are necessary, as we shall prove later, to assure that we have axiomatized equality as well as is possible in first order logic.

Problem Set 21

1. What is the theory of linear orderings?

2. What is the theory of semi-groups? Of abelian groups?

3. Which of the following theories are satisfiable? Find a model for each satisfiable theory. Which theories have finite models?

a. $\neg x < x$

$x < y \supset y < z \supset x < z$

$x < y \supset \exists w(x < w \wedge \forall z(\neg x < z \vee \neg z < w))$

b. The formulas of (a) and $\exists y(x < y)$.

c. The formulas of (a) and

$\exists w \exists x(w < x \wedge \forall y(y < x \supset \exists z(y < z \wedge z < x)))$.

d. The formulas of (a) and
$\exists x(w < x \wedge \forall y(y < x \supset \exists z(y < z \wedge z < x)))$.

e. The formulas of (d) and $\exists x \forall y(\neg x < y)$.

CHAPTER SEVEN

FIRST ORDER LOGIC - DEDUCTION

Preview of Chapter Seven

We develop the mechanism for making formal deductions in first order logic. As with propositional logic, a deduction is a step-by-step process for obtaining a conclusion from given premises. It can be inspected for correctness by a proof-checker. Most of the theorems in this chapter are concerned with the existence of demonstrations, and have the practical effect of saving us time. They will also have theoretical applications in Chapter Eight. The mechanism of substitution, a necessary prerequisite, is discussed first.

§7.1 Substitution

In this book, we make a sharp distinction between the words "substitution" and "replacement" which is very useful, but has not won general acceptance at the present time. Our notation for substitution follows [Robinson]. We shall discuss replacement later in this chapter.

The LISP function subst (see §2.2) is a good example of a substitution operator. Subst[x, y, z] substitutes x for all occurrences of y in z. Sometimes, we wish to perform several substitutions simultaneously on the same object. For example, we may substitute Q for A, and R for B in the s-expression (A B C), in which case we get (Q R C). We can define a LISP function sublis that does this. The first argument is a list of pairs, and the second argument is the object of the substitution. The effect of each pair is to cause the first member of it to be substituted for all occurrences of the

second member of it. So sublis[((Q A) (R B)), (A B C)] = (Q R C).

```
sublis[x, s] ← [atom[s] → subl[x, s], T → cons[sublis[x, car[s]],
                                               sublis[x, cdr[x]]]]
subl[x, s] ← [null[x] → s, cadar[x] = s → caar[s], T → subl[cdr[x], s]]
```

Sublis performs what is known as a simultaneous substitution. It does not substitute on that which it has already substituted. For example, sublis[((A B) (B A)), (A X B Y)] = (B X A Y). The alternative to simultaneous substitution is sequential substitution. In this case, it makes a considerable difference what is done first. Thus subst[A, B, subst[B, A, (A X B Y)]] = (A X A Y), but subst[B, A, subst[A, B, (A X B Y)]] = (B X B Y).

For first order logic we shall need to substitute terms for variables occurring in formulas or terms. Furthermore, we only substitute for free occurrences of variables. An example of a substitution is to substitute the term g[y] for all free occurrences of the variable x in the formula p[x] ⊃ ∃x(q[x]). The result is p[g[y]] ⊃ ∃x(q[x]). Because the operation of substitution occurs frequently, we need a precise way of writing it, so that long explanations are not necessary. If α is any formula, τ is a term, and ξ is a variable, then by $\alpha(\tau/\xi)$ we mean the formula obtained by substituting τ for all free occurrences of ξ in α. We also allow α to be a term, in which case all occurrences of ξ in α are free. We can also specify a simultaneous substitution, where each pair separated by a "/" serves the same purpose as the pairs in SUBLIS. These are called substitution components. For example, if α is the formula p[x] ∧ q[y], then $\alpha(g[y]/x, h[x, y]/y)$ is the formula p[g[y]] ∧ q[h[x, y]].

In addition to not substituting for bound occurrences of a variable, there is another restriction in first order logic. Consider the result of substituting g[y] for all free occurrences of x in p[x] ∧ ∃y(q[y] ⊃ r[x]). The result is p[g[y]] ∧ ∃y(q[y] ⊃ r[g[y]]). We call this an improper substitution because the variable y in g[y] is captured by the quantifier in the second instance (from the left) where it is substituted. In order that the substitution $\alpha(\tau/\xi)$ be proper, it is necessary that wherever there is a free occurrence of ξ in α, it is not within the scope of any quantifier that binds a variable that occurs in τ. When $\alpha(\tau/\xi)$ is proper, we also say "τ is free for ξ in α".

Examples:

1. If we substitute y for x in $\forall x \exists y(r[x, y])$, this is a proper substitution because no substituting occurs. There is no free x in the formula, and it is permissible for bound occurrences of x to be within the scope of a quantifier on y.

2. If we substitute y for x in $p[x, y] \supset \exists x \exists y(q[x, y])$, the result is $p[y, y] \supset \exists x \exists y(q[x, y])$. This is a proper substitution because wherever x is free, it is not within the scope of a quantifier binding y, although a bound occurrence of x is within the scope of a quantifier on y.

Substitution plays an important part in the rules of deduction of first order logic, but in each case improper substitution is not allowed. We shall adopt the convention that substitution on formulas of first order logic is undefined if it is improper. In each case where a rule using substitution is given, the rule does not apply when the substitution is improper because no result is defined.

Formulas of first order logic are translated into LISP as follows. The idea should be obvious by now.

(i) If τ is a term, then τ^* is obtained by using the same rules as for forms in the language of recursive definitions. For example, g[x, A], where A is an object, is translated into (G X (QUOTE A)).

(ii) Atomic formulas are translated similarly.

(iii) Composite formulas are translated into (NOT α^*), (OR $\alpha_1^* \ldots \alpha_n^*$), (AND $\alpha_1^* \ldots \alpha_n^*$), (EQUIV $\alpha_1^* \alpha_2^*$), (EXISTS $\xi^* \alpha^*$), and (FORALL $\xi^* \alpha^*$), where the α's are formulas, and ξ is a variable.

Problem Set 22 --

1. Write a LISP function sub which is the equivalent of sublis for first order logic. If α is a formula, the τ_i terms, and the ξ_i variables, then sub[list[list[τ_1, ξ_1], ..., list[τ_n, ξ_n]], α] is the formula $\alpha(\tau_1/\xi_1, \ldots, \tau_n/\xi_n)$ if the substitution is proper, and NIL otherwise.

2. Write a LISP predicate inst of three arguments such that inst[α, ξ, β] is true if α is a formula, ξ is a variable, and there exists a term τ such that the substitution $\alpha(\tau/\xi)$ is proper, and the result is β.

§7.2 The Rules of Deduction

Definition 7.1

A deduction is a numbered sequence of formulas each having a valid justification. There are five types of justification:

(i) Given

(ii) Mp i, j

For this to be a valid justification of line n, it is necessary that $i < n$, $j < n$, and if line (i) is the formula α, and line (n) is the formula β, then line (j) must be the formula $\alpha \supset \beta$.

(iii) Taut

If a formula of propositional logic is a tautology, then the result of substituting formulas of first order logic for all its propositional variables is a tautology of first order logic. All occurrences of a particular propositional variable must be replaced by the same formula.

(iv) Q1 and Q2

Q1 and Q2 are axiom schemas for first order logic. Each schema represents an infinite set of formulas which are called the instances of the schema. If a formula is an instance of Q1, then Q1 is a valid justification for it, and similarly with Q2. The schemas are:

$$\text{Q1:} \quad \forall\xi(\alpha) \supset \alpha(\tau/\xi)$$

$$\text{Q2:} \quad \alpha(\tau/\xi) \supset \exists\xi(\alpha)$$

where α is any formula, ξ is any variable, and τ is any term, and $\alpha(\tau/\xi)$ is a proper substitution.

(v) Q3 i, and Q4 i

Q3 and Q4 are rules of inference for first order logic. The distinction between a rule of inference and an axiom schema

is that a rule of inference depends on previous lines of the deduction. Modus-ponens is also a rule of inference. "Q3 i" is a valid justification for line n if $i < n$, and there is an instance of the schema Q3 in which line (i) appears above the horizontal line, and line (n) appears below it. The case of Q4 is similar.

$$\text{Q3:}\quad \frac{\alpha \supset \beta}{\alpha \supset \forall \xi(\beta)}$$

$$\text{Q4:}\quad \frac{\beta \supset \alpha}{\exists \xi(\beta) \supset \alpha}$$

where β is any formula, ξ is any variable, and α is any formula which does not contain ξ free.

If T is any theory, and there is a deduction in which only formulas that are in T are justified as given, and if the conclusion of the deduction is the formula α, then we say that there is a deduction of α from T, and we write $T \vdash \alpha$. If there is a deduction of α in which no formula is justified as given, then we say that α is a theorem of logic, and we write $\vdash \alpha$.

The following sequence of seven steps is an example of a deduction in first order logic:

1.	$\forall y(p[x, y]) \supset p[x, y]$	Q1
2.	$p[x, y] \supset \exists x(p[x, y])$	Q2
3.	$(\forall y(p[x, y]) \supset p[x, y]) \supset (p[x, y] \supset \exists x(p[x, y])) \supset (\forall y(p[x, y]) \supset \exists x(p[x, y]))$	Taut
4.	$(p[x, y] \supset \exists x(p[x, y])) \supset (\forall y(p[x, y]) \supset \exists x(p[x, y]))$	Mp 1, 3
5.	$\forall y(p[x, y]) \supset \exists x(p[x, y])$	Mp 2, 4
6.	$\exists x \forall y(p[x, y]) \supset \exists x(p[x, y])$	Q4 5
7.	$\exists x \forall y(p[x, y]) \supset \forall y \exists x(p[x, y])$	Q3 6

Since this deduction has no given formulas, we may write $\vdash \exists x \forall y(p[x, y]) \supset \forall y \exists x(p[x, y])$.

The next example is a somewhat lengthy proof taken from the theory of formal arithmetic. It illustrates a great many points that will be made in the next few chapters, and you may wish to refer back to it. For the present, it

is simply an example of a formal deduction. We prove the sentence $0' + 0' = 0''$ from four axioms which are the first four lines of the demonstration. As an aid to comprehending the organization of the deduction, important subgoals are marked with an asterisk (*).

	1.	$\forall x \forall y \forall z(x = y \supset y = z \supset x = z)$	Given
	2.	$\forall x \forall y(x = y \supset x' = y')$	Given
	3.	$\forall m(m + 0 = m)$	Given
	4.	$\forall m \forall n(m + n' = (m + n)')$	Given
	5.	$\forall m \forall n(m + n' = (m + n)') \supset \forall n(0' + n' = (0' + n)')$	Q1
	6.	$\forall n(0' + n' = (0' + n)')$	Mp 4, 5
	7.	$\forall n(0' + n' = (0' + n)') \supset 0' + 0' = (0' + 0)'$	Q1
(*)	8.	$0' + 0' = (0' + 0)'$	Mp 6, 7
	9.	$\forall n(m + 0 = m) \supset 0' + 0 = 0'$	Q1
(*)	10.	$0' + 0 = 0'$	Mp 3, 9
	11.	$\forall x \forall y(x = y \supset x' = y') \supset \forall y(0' + 0 = y \supset (0' + 0)' = y')$	Q1
	12.	$\forall y(0' + 0 = y \supset (0' + 0)' = y')$	Mp 2, 11
	13.	$\forall y(0' + 0 = y \supset (0' + 0)' = y') \supset 0' + 0 = 0' \supset (0' + 0)' = 0''$	Q1
	14.	$0' + 0 = 0' \supset (0' + 0)' = 0''$	Mp 12, 13
(*)	15.	$(0' + 0)' = 0''$	Mp 10, 14
	16.	$\forall x \forall y \forall z(x = y \supset y = z \supset x = z) \supset \forall y \forall z(0' + 0' = y \supset y = z \supset 0' + 0' = z)$	Q1
	17.	$\forall y \forall z(0' + 0' = y \supset y = z \supset 0' + 0' = z)$	Mp 1, 16
	18.	$\forall y \forall z(0' + 0' = y \supset y = z \supset 0' + 0' = z) \supset \forall z(0' + 0' = (0' + 0)' \supset (0' + 0)' = z \supset 0' + 0' = z)$	Q1
	19.	$\forall z(0' + 0' = (0' + 0)' \supset (0' + 0)' = z \supset 0' + 0' = z)$	Mp 17, 18
	20.	$\forall z(0' + 0' = (0' + 0)' \supset (0' + 0)' = z \supset 0' + 0' = z) \supset -(0' + 0' = (0' + 0)' \supset (0' + 0)' = 0'' \supset 0' + 0' = 0'')$	Q1
	21.	$0' + 0' = (0' + 0)' \supset (0' + 0)' = 0'' \supset 0' + 0' = 0''$	Mp 19, 20
	22.	$(0' + 0)' = 0'' \supset 0' + 0' = 0''$	Mp 8, 21
	23.	$0' + 0' = 0''$	Mp 15, 22

We may safely conclude from this example that deduction is an extremely tedious process full of needless repetition of similar patterns, and that something must be done to speed it up. We shall consider this subject later.

Problem Set 23

1. Show (by writing a deduction) that each of the following formulas is a theorem of logic:

a. $\forall x(p[x] \equiv q[x]) \supset (\forall x(p[x]) \equiv \forall x(q[x]))$

b. $\forall x(p[x] \equiv q[x]) \supset (\exists x(p[x]) \equiv \exists x(q[x]))$

c. $\forall x \neg p[x] \equiv \neg \exists x(p[x])$

d. $\forall x(p[x] \wedge q[x]) \equiv (\forall x(p[x]) \wedge \forall x(q[x]))$

e. $\exists x(p[x] \vee q[x]) \equiv (\exists x(p[x]) \vee \exists x(q[x]))$

f. $\exists x(p[x] \wedge q[x]) \supset (\exists x(p[x]) \wedge \exists x(q[x]))$

g. $\forall x(p[x]) \supset \exists x(p[x])$

2. Which of the following formulas are instances of Q1 or Q2, and which are neither? Why?

a. $\forall x(p[x]) \supset p[x]$

b. $\forall x(p[x, y]) \supset p[y, y]$

c. $\forall x \exists y(p[x, y]) \supset \exists y(p[g[y], y])$

d. $p[g[y], y] \supset \exists x(p[x, y])$

3. Define the unary LISP predicates axq1 and axq2 which are true if their arguments are instances of Q1 or Q2 respectively.

4. Define the binary LISP predicates riq3 and riq4 which are true if the second argument is derived from the first argument by rules of inference Q3 or Q4 respectively.

5. Modify proofchk so that it is a proofchecker for first order logic. The only modifications to the format of a deduction are (i) TAUT must handle substitution instances efficiently, and there is then no longer a need for INST as a justification, (ii) the justifications Q1 and Q2 must be added, and (iii) the justifications (Q3 i) and (Q4 i) must be added.

§7.3 The Consistency Theorem

The statement of the consistency theorem for first order logic is the same as the consistency theorem for propositional logic (theorem 4.14), but the meaning behind it is considerably more subtle.

Theorem 7.3 (Consistency Theorem)

If $T \subset L$, $\alpha \in L$ and $T \vdash \alpha$, then $T \models \alpha$.

Proof: The proof is by induction and follows the same lines as the proof of theorem 4.14. The induction hypothesis is that if $T \models \alpha_j$ for $j < i$, then $T \models \alpha_j$ where the deduction is the sequence $\alpha_1, \ldots, \alpha_n$. There are seven cases to consider, and the first three are the same as in the previous proof.

(iv) If α_i is an instance of Q1, then it has the form $\forall\xi(\beta) \supset \beta(\tau/\xi)$. Let M be any model, and I any interpretation for the variables in this formula. If $V(M, I, \forall\xi(\beta))$ is false, then $M, I \models \alpha_i$. If $V(M, I, \forall\xi(\beta))$ is true, then $V(M, J, \beta)$ is true for any J differing from I at most at the variable ξ. In particular, there is that J that assigns to ξ the value which is $V(M, I, \tau)$. Therefore $V(M, I, \beta(\tau/\xi))$ is true because no variable in τ is bound by quantifiers in β, and so $M, I \models \alpha_i$ in this case also. What we have shown, then, is that every instance of Q1 is valid.

(v) If α_i is an instance of Q2, this also is valid, and the proof is left to the reader.

(vi) If α_i is derived from α_j by the rule Q3, then since $j < i$, the induction hypothesis is that $T \models \alpha_j$. Let α_j be the formula $\beta \subset \gamma$, where β has no free ξ. Then α_i is the formula $\beta \supset \forall\xi(\gamma)$. Let M be any model that satisfies T. Then $M, I \models \beta \supset \gamma$ for all interpretations I. Choose one such I. If M, I does not satisfy β, then $M, I \models \alpha_i$. If $M, I \models \beta$, then $M, I \models \gamma$ also. But then M, J also satisfies β where J is any interpretation differing from I at most on ξ, because β has no free ξ. So M, J also satisfies γ for all such J, and therefore $M, I \models \forall\xi(\gamma)$. So $M, I \models \alpha_i$ in either case, and the conclusion is that $T \models \alpha_i$.

(vii) If α_i is derived from α_j by the rule Q4, then since $j < i$, the induction hypothesis is that $T \models \alpha_j$. Let α_j be the formula $\beta \supset \gamma$ where γ has no free ξ. Then α_i is the formula $\exists\xi(\beta) \supset \gamma$. Let M be any model that satisfies T. Then $M, I \models \beta \supset \gamma$ for all interpretations I. Choose one such I. If $M, I \models \gamma$, then $M, I \models \alpha_i$. If M, I does not satisfy γ, then M, I does not satisfy β. Let J be any interpretation differing from I at most on ξ. M, J does not satisfy γ because γ has no free ξ. So M, J does not satisfy β, and since this is true for all such J, M, I does not satisfy $\exists\xi(\beta)$. So $M, I \models \alpha_i$ in either case,

and the conclusion is that $T \vdash \alpha_i$.

Corollary 7.4

If $\vdash \alpha$, then α is a valid formula.

Corollary 7.5

If the theory T is satisfiable (has a model), then it is consistent.

§7.4 Existence of Deductions, Replacement

Since deduction is a very tedious process, we would like to speed it up by introducing additional axioms and rules of inference. But, in fact, no matter how many additional rules we introduce, there will always be more that we would like to have. If we were to introduce a great many rules right from the start, then the proof of the consistency theorem would be very long because we would have to consider each rule separately and show that it is a valid form of reasoning. Now that we have proved the consistency theorem, we can deal with new axioms, and new rules of inference in a different way. What we can hope to show for each one is that it is eliminable in the sense that if we have a deduction using such an axiom or rule, then there is an effective way of obtaining a deduction that does not use it, but which proves the same conclusion from the same premises.

As a very brief example of this, consider the rule:

$$\text{R1:}\quad \frac{\alpha \supset \beta, \beta \supset \gamma}{\alpha \supset \gamma}$$

This is a derivative of the rule of modus-ponens which stated in this style is:

$$\frac{\alpha, \alpha \supset \beta}{\beta}$$

Now suppose we have a deduction that uses the rule R1:

1.	$\alpha \supset \beta$	Given
2.	$\beta \supset \gamma$	Given
3.	$\alpha \supset \gamma$	R1

We know that this proof can be expanded to:

1.	$\alpha \supset \beta$	Given
2.	$\beta \supset \gamma$	Given
3.	$(\alpha \supset \beta) \supset (\beta \supset \gamma) \supset (\alpha \supset \gamma)$	Taut
4.	$(\beta \supset \gamma) \supset (\alpha \supset \gamma)$	Mp 1, 3
5.	$\alpha \supset \gamma$	Mp 2, 4

This can be done in every situation in which R1 is used, so we can say that R1 is constructively eliminable. The consistency theorem then guarantees the correctness of the rule as a method of reasoning. This not only shows that it is correct reasoning, it shows that the introduction of the rule does not alter any of the properties of first order logic that we may prove in the future, because the rule itself is not essential to any deduction in which it is used.

Problem Set 24

1. Show that the following are constructively eliminable rules of inference:

QX1: $\dfrac{\forall \xi(\alpha)}{\alpha(\tau/\xi)}$ QX2: $\dfrac{\neg \exists \xi(\alpha)}{\neg \alpha(\tau/\xi)}$

QX3: $\dfrac{\alpha}{\forall \xi(\alpha)}$ QX4: $\dfrac{\neg \alpha}{\neg \exists \xi(\alpha)}$

QX5: $\dfrac{\alpha(\tau/\xi)}{\exists \xi(\alpha)}$

2. Rules Q3 and Q4 are necessarily stated as rules of inference, and cannot be treated as axioms. Show that the following schemas are not valid by describing counter-model for an instance of each schema.

a. $(\alpha \supset \beta) \supset (\alpha \supset \forall \xi(\beta))$ where α has no free ξ.

b. $(\beta \supset \alpha) \supset (\exists \xi(\beta) \supset \alpha)$ where α has no free ξ.

The formulas α and $\alpha(\zeta/\xi)$ are said to be similar if the variable ζ does not occur free in α, and if ζ is free for ξ in α. When this is true, it will also

be the case that ξ does not occur free in $\alpha(\zeta/\xi)$, and ξ is free for ζ in $\alpha(\zeta/\xi)$, and $\alpha(\zeta/\xi)(\xi/\zeta)$ is the formula α.

Theorem 7.6 (Change of bound variables)

If α and $\alpha(\zeta/\xi)$ are similar, then $\vdash \forall\xi(\alpha) \equiv \forall\zeta(\alpha(\zeta/\xi))$, and $\vdash \exists\xi(\alpha) \equiv \exists\zeta(\alpha(\zeta/\xi))$.

Proof:

1.	$\forall\xi(\alpha) \supset \alpha(\zeta/\xi)$	Q1; Why is this substitution proper?
2.	$\forall\xi(\alpha) \supset \forall\zeta(\alpha(\zeta/\xi))$	Q3 1; Can $\forall\xi(\alpha)$ have free ζ?
3.	$\forall\zeta(\alpha(\zeta/\xi)) \supset \alpha$	Q1
4.	$\forall\zeta(\alpha(\zeta/\xi)) \supset \forall\xi(\alpha)$	Q3 3
5.	$\forall\xi(\alpha) \equiv \forall\zeta(\alpha(\zeta/\xi))$	Prop 2, 4

The $\exists$ case is symmetrical in form.

You will notice that as we acquire more techniques, deductions will become more and more condensed. At this point, there is no longer any reason to write out in full any sequence of steps that depends merely on propositional logic. We just write "Prop" and list the antecedents.

The distinction between replacement and substitution is that (i) replacement refers to replacing of an entire structure of some sort by another, whereas in substitution we always substitute in place of something atomic such as an atom or a variable, and (ii) it is not necessary to replace all occurrences of a given structure, but only as many as we wish. The semantic justification for replacement is that something may be replaced by something else that is in some sense its equal or equivalent. The semantic justification for substitution, on the other hand, is that we are obtaining a particular instance of a general statement.

An example of replacement is to take the formula $0'+0' = (\underline{0'+0})'$ and to replace the underlined term with the term $0'$. Our justification for doing this is that we have already concluded that these two terms are equal, i.e., $0'+0 = 0'$. The result of the replacement is $0'+0' = (0')'$ or dropping parentheses, $0'+0' = 0''$. Another example is to replace the first occurrence of $2+2$ in $(2+2)+2 = (2+2)+2$ with 4 because we already have $2+2 = 4$. This

gives $4+2 = (2+2)+2$. This is valid, even though we have not replaced all occurrences of $2+2$.

Now consider the substitution of 2 for x and 3 for y in $x+y = y+x$. This gives us $2+3 = 3+2$, which is an instance of the general rule. $2+3 = y+x$ has no useful meaning because partial substitution does not accomplish the purpose of substitution. Notice, also, that it is not meaningful to substitute for a constant.

It is important that entities that get replaced are proper sub-expressions in whatever context they appear, and not accidental pseudo-expressions caused by juxtaposition. For instance, if we start with the equation $2+3\times4 = 14$, and then replace "$2+3$" with "5", we get $5\times4 = 14$, which is incorrect. "$2+3$" is not a sub-expression of "$2+3\times4$" because the conventional association is "$2+(3\times4)$".

Theorem 7.7 (Replacement of Equivalent Formulas)

Let α and β be two formulas such that $T\vdash\alpha \equiv \beta$. Let γ be any formula, and δ be a formula that is obtained by replacing some (but not necessarily all) occurrences of α in γ by β. Then $T\vdash\gamma \equiv \delta$.

Proof: We begin the proof by identifying certain sub-formulas of γ and δ as "corresponding components". If an α occurring in γ is replaced by a β in δ, then the α and the β are corresponding components. Any sub-formula of γ which contains no occurrences of α that get replaced, and is not contained in a larger such formula is also a corresponding component to the sub-formula of δ which is identical to it both in content and in position. The formulas γ and δ are thus built up identically, starting with corresponding components using the propositional connectives and quantifiers. Also, corresponding components are either identical, or else one is α and the other is β. In either case they can be proven equivalent from T. We proceed by induction on the number of propositional connectives and the quantifiers to show that this equivalence extends up to the formulas γ and δ: (i) If γ_1 is $\neg\gamma_2$, and δ_1 is $\neg\delta_2$, and $T\vdash\gamma_2 \equiv \delta_2$, then $T\vdash\gamma_1 \equiv \delta_1$ because $(\gamma_1 \equiv \delta_1) \supset (\neg\gamma_1 \equiv \neg\delta_1)$ is a tautology. (ii-vi) The cases for the other propositional connectives and the universal quantifier are left to the reader. (vii) If γ_1 is $\exists\xi(\gamma_2)$, and δ_1 is $\exists\xi(\delta_2)$, and $T\vdash\gamma_2 \equiv \delta_2$, then $T\vdash\gamma_1 \equiv \delta_1$ because $\forall\xi(\gamma_2 \equiv \delta_2)$ follows by rule QX3,

and because $\vdash \forall \xi(\gamma_2 \equiv \delta_2) \supset (\exists \xi(\gamma_2) \equiv \exists \xi(\delta_2))$. (See problem set 23, No. 1b.)

Theorem 7.8 (Replacement of Equal Terms)

Let T be a theory with equality (i.e., $E_L \subset T$). Let τ_1 and τ_2 be terms such that $T \vdash \tau_1 = \tau_2$, let α be any formula, and let β be a formula resulting from replacing some occurrences of τ_1 in α by τ_2. Then $T \vdash \alpha \equiv \beta$.
Proof: By induction on the depth of the terms in α. Let the corresponding components be terms, either τ_1 and the τ_2 that replaces it, or identical terms that are in indentical positions in α and β, and are the largest possible such terms. If σ and θ are corresponding terms, then they can be proven equal from T. Larger terms are built from these by function composition. Let $\varphi[\sigma_1, \ldots, \sigma_n]$ and $\varphi[\theta_1, \ldots, \theta_n]$ be in corresponding positions, and by the induction hypothesis $T \vdash \sigma_i = \theta_i$. Then these terms can be proven to be equal because there is an axiom in E_L which is $x_1 = y_1 \supset \ldots \supset x_n = y_n \supset \varphi[x_1, \ldots, x_n] = \varphi[y_1, \ldots, y_n]$. Similarly, once all terms in corresponding positions are equal, the atomic formulas can be proven equivalent from the reflexive axiom of equality ($x = y \supset y = x$) and the double application of the axiom E_L which is $x_1 = y_1 \supset \ldots \supset x_m = y_m \supset \psi[x_1, \ldots, x_m] \supset \psi[y_1, \ldots, y_m]$. Once corresponding atomic formulas are proven equivalent, the induction proceeds as in theorem 7.7.

We introduce one more derived rule of inference obtained from Q3 and Q1:

$$\text{Inst:} \quad \frac{\alpha}{\alpha(\tau_1/\xi_1, \ldots, \tau_n/\xi_n)}$$

where the ξ_i are distinct variables, and the substitution is proper.

A shorter demonstration for $0' + 0' = 0''$ can now be given:

1.	$m + 0 = m$	Given
2.	$m + n' = (m + n)'$	Given
3.	$0' + 0 = 0'$	Inst 1
4.	$0' + 0' = (0' + 0)'$	Inst 2
5.	$0' + 0' = 0''$	Replacement 4, 3

§7.5 The Deduction Theorem

The deduction theorem for first order logic is rather subtle, and takes some effort to understand, but a correct perception of it will yield a lot of insight into the nature of rules Q1 thru Q4. The most naive statement of the theorem is actually false. It is not the case that if $T \cup \{\alpha\} \vdash \beta$, then $T \vdash \alpha \supset \beta$ for any formulas α and β. If this were true, then since $p[x] \vdash \forall x(p[x])$ by QX3, it would follow that $\vdash p[x] \supset \forall x(p[x])$, and then by another application of QX3, $\vdash \forall x(p[x] \supset \forall x(p[x]))$. But this sentence is not valid; it is not satisfied by the model on the domain $\{0, 1\}$ where $p[0]$ is true, and $p[1]$ is false.

We have chosen to interpret a formula standing as a line in a deduction as being equivalent to its universal closure. In fact, the rules QX1 and QX3 allow universal quantifiers to be added or stripped from the beginning of a formula at will as long as they take the whole formula as their scope.

The trouble seems to be that when such an open formula is incorporated into the left side of an implication, it is negated because $\alpha \supset \beta$ is the same as $\neg\alpha \vee \beta$. But its implicit universal quantifier gets left outside the negation and causes the error.

Theorem 7.9 (First Deduction Theorem)

If $T \cup \{\alpha\} \vdash \beta$, and α is a sentence, then $T \vdash \alpha \supset \beta$.

Proof: By induction on the demonstration $\beta_1, \ldots, \beta_n = \beta$. (Please review theorem 4.7.)

(i) If β_i is a tautology, then so is $\alpha \supset \beta_i$.

(ii) If β_i is in T, then $\alpha \supset \beta_i$ is derivable from β_i.

(iii) If β_i is α, then $\alpha \supset \alpha$ is a tautology.

(iv) If β_i follows from two antecedents by modus-ponens, then by the induction hypothesis $\alpha \supset \beta_j$ and $\alpha \supset (\beta_j \supset \beta_i)$ are provable from T. Then $\alpha \supset \beta_i$ is provable from these by propositional logic.

(v) If β_i is an instance of Q1 or Q2 then $\alpha \supset \beta_i$ is derivable from β_i.

(vi) If β_i follows from β_j by an application of Q3, then β_j is $\gamma \supset \delta$, and β_i is $\gamma \supset \forall\xi(\delta)$ where γ is a formula that has no free ξ. By the induction hypothesis, $T \vdash \alpha \supset \beta_j$ or $T \vdash \alpha \supset (\gamma \supset \delta)$. From this we derive $(\alpha \wedge \gamma) \supset \delta$, and then apply Q3 to get $(\alpha \wedge \gamma) \supset \forall\xi(\delta)$ which is valid because α is a sentence,

and so has no free ξ. From this we can get $\alpha \supset (\gamma \supset \forall\xi(\delta))$.

(vii) If β_i follows from β_j by Q4, then β_j is $\gamma \supset \delta$, and β_i is $\exists\xi(\gamma) \supset \delta$ where δ has no free ξ. By the induction hypothesis, $T \vdash \alpha \supset (\gamma \supset \delta)$, from which we deduce $\gamma \supset (\alpha \supset \delta)$. Since α has no free ξ, we can apply Q4 to this, getting $\exists\xi(\gamma) \supset (\alpha \supset \delta)$ and then rearrange to get $\alpha \supset (\exists\xi(\gamma) \supset \delta)$.

A formula is said to <u>depend</u> on a preceding formula in a deduction if there is a chain of antecedents working back to the preceding formula. If the conclusion of a deduction does not depend on one of the given formulas, then we could omit that formula and all its dependents without sacrificing the conclusion.

A variable is <u>varied</u> in a deduction any time Q3 or Q4 is used with that variable being the ξ mentioned in the deduction rule.

The fact that of the original rules of inference only Q3 and Q4 can vary a variable is quite significant. Suppose that the formula p[x] is given in a deduction. Without using Q3 or Q4 it is quite impossible to derive from it p[5] or $\forall x$(p[x]) or p[y]. Only these two rules have the power to interpret a free variable universally.

We stated earlier that the intended interpretation of the fact that β is deducible from α is that the universal closure of α semantically implies the universal closure of β. Let us consider a different interpretation. What if the interpretation of the deduction $\alpha \vdash \beta$ was that for any M and I if $M, I \models \alpha$ then $M, I \models \beta$? A study of the deduction rules and axioms of logic shows that all of propositional logic, including modus-ponens as well as Q1 and Q2, preserves this interpretation. But Q3 and Q4 do not.

So if $p[x] \vdash q[x]$, then we can certainly conclude that $\forall x(p[x]) \supset \forall x(q[x])$ using the standard interpretation of closure. If, in addition, the variable x is not varied in any formula that depends on p[x], then x has remained constant, so we can conclude using the deduction theorem that $p[x] \supset q[x]$.

<u>Theorem 7.10 (Final Deduction Theorem)</u>

If $T \cup \{\alpha\} \vdash \beta$, and no variable occurring free in α is varied in any formula depending on α, then $T \vdash \alpha \supset \beta$.

Proof: We shall reconsider case (vi) of the proof of theorem 7.9, and let the

reader do the same for case (vii). β_j is $\gamma \supset \delta$, where γ has no free ξ, and β_i is $\gamma \supset \exists\xi(\delta)$ and is derived by Q3. If α does not contain free ξ, then the construction described in the previous proof still works. On the other hand, if β_j is not dependent on α, then $T \vdash \beta_j$, and so $T \vdash \beta_i$. Then by propositional logic, $T \vdash \alpha \supset \beta_i$.

The deduction theorem makes many deductions shorter to write and easier to organize conceptually. As a brief example, we demonstrate $0 + m = m$. The third line is what is known as an induction axiom, and is part of the theory that this proof is taken from.

	1.	$m + 0 = 0$	Given
	2.	$m + n' = (m + n)'$	Given
	3.	$(0 + 0 = 0) \supset \forall m(0 + m = m \supset 0 + m' = m') \supset \forall m(0 + m = m)$	Given
	4.	$0 + 0 = 0$	Inst 1
	5.	$\forall m(0 + m = m \supset 0 + m' = m') \supset \forall m(0 + m = m)$	Mp 4, 3
(6)	6.	$0 + m = m$	Assume
	7.	$0 + m' = (0 + m)'$	Inst 2
(6)	8.	$0 + m' = m'$	Replacement 7, 6
	9.	$0 + m = m \supset 0 + m' + m'$	Discharge 8, 6
	10.	$\forall m(0 + m = m \supset 0 + m' = m')$	QX3 9
	11.	$\forall m(0 + m = m)$	Mp 10, 5
	12.	$0 + m = m$	QX1

The rules for incorporating the use of the deduction theorem into formal deductions are:

(i) There is a column for noting dependencies (we locate it to the left of the line number).

(ii) When a line is justified by "Assume", its own line number goes in the dependency column. Several such lines may appear in a deduction.

(iii) When a line has one or more antecedents under some rule of deduction, the dependencies of the antecedents are inherited. This means that a line that is dependent on several assumed lines will have the line numbers of all these assumed lines in its dependency column. (If a line is dependent on an assumed line through several different paths, the line number

of the assumed line still appears only once in the dependency column of the dependent line.)

(iv) A variable in a dependent line may not be varied if it appears free in any of the assumed lines that the dependent line depends on. This must be explicitly checked out when using Q3, Q4, or any rule of inference derived from Q3 or Q4.

(v) A dependency is removed by the process of discharge in which the assumed line is introduced as the premise of a "⊃". The dependencies may be removed in any order. (Refer to the transition from lines 8 to 9 in the preceding example.)

(vi) Only an independent line (having nothing in its dependencies column) is a valid conclusion of a deduction.

§7.6 The Choice Rule

When reasoning informally, we sometimes prove that there exists an x having a certain property, and then say, "Let k be such an x." If the constant k has not been used before in this chain of reasoning, its interpretation has not yet been restricted in any way, so no problem is created by doing this. The choice of the name k is arbitrary, so if we succeed in proving some result that does not involve k, then we should be able to prove the same result without mentioning k. It is important to realize that inventing the name k does not introduce a new object into the model of one's subject matter. It is only a new name that is being created, and it could turn out that the new name really describes an object already familiar under a different name.

In first order logic, a constant is a function of no arguments. Properly, it should have a set of brackets following it. So k[] is a constant. But often we omit the brackets for convenience. (In s-expression notation, which is more strict, a constant is enclosed by parentheses. For example, k[] translates into (K). This will always serve to distinguish a constant from a variable which would not have the parentheses, or an object which would be translated as (QUOTE *).)

Rule C

Within a deduction, if we have obtained a line which is the formula $\exists\xi(\alpha)$, then we may derive from this the line $\alpha(\varphi[]/\xi)$ where φ is a new 0-ary function name. The justification for the derived line is "Rule C j", where j is the line number of the first formula. If α has free variables other than ξ, then it is necessary not to vary any of these in any formula that contains the constant $\varphi[]$.

The conclusion of the proof must be a formula that does not have any of the new constants $\varphi[]$.

A deduction may have any number of both assumed lines and applications of Rule C.

Example:

(1)	1.	$\forall x\exists y(p[x, y] \wedge q[x, y])$	Assume
(1)	2.	$\exists y(p[x, y] \wedge q[x, y])$	QX1 1
(1)	3.	$p[x, k] \wedge q[x, k]$	Rule C 2
(1)	4.	$p[x, k]$	Prop 3
(1)	5.	$\exists y(p[x, y])$	QX5
(1)	6.	$\forall x\exists y(p[x, y])$	QX3
(1)	7.	$\forall x\exists y(q[x, y])$	Similarly
(1)	8.	$\forall x\exists y(p[x, y]) \wedge \forall x\exists y(q[x, y])$	Prop 6, 7
	9.	$\forall x\exists y(p[x, y] \wedge q[x, y]) \supset (\forall x\exists y(p[x, y]) \wedge \forall x\exists y(q[x, y]))$	Discharge 8, 1

Notice that the application of QX3 in line 6 varies x which occurs free in line 3. This is valid because line 5 does not have the constant k. If steps 5 and 6 are done in reverse order, i. e., $\forall x(p[x, k])$ and then $\exists y\forall x(p[x, y])$, the result is not valid.

The validity of Rule C depends on the fact that any conclusion not containing the new constant names can also be derived from a demonstration not using Rule C, as the following theorem shows.

Theorem 7.11 (Elimination of Rule C)

If $T \vdash \alpha$ using several applications of Rule C, and α does not contain any occurrences of the constants introduced by Rule C, then $T \vdash \alpha$ without using Rule C.

Proof: We shall prove the theorem for the case that only one application of Rule C is made in the deduction, and let the reader extend the proof.

Let line (i) be obtained from line (j) in the deduction by Rule C, where line (j) is $\exists\xi(\beta)$, and line (i) is $\beta(\varphi[]/\xi)$, and φ is a new constant. To show that $T \vdash \alpha$ without the use of Rule C, we shall show that this is true for each line in the deduction which is dependent on line (i), does not contain the constant φ, and is the first line in its dependency path going back to line (i) not to contain φ. Let these lines be the formulas γ_1 thru γ_n. If $T \vdash \gamma$ without Rule C for each such γ, the conclusion follows.

It is obvious that $T \cup \{\beta(\varphi[]/\xi)\} \vdash \gamma$ without use of Rule C. We can apply the deduction theorem here because we have explicitly stated that no variable occurring free in $\beta(\varphi[]/\xi)$ may be varied in any line containing occurrences of φ. Therefore, $T \vdash \beta(\varphi[]/\xi) \supset \gamma$. Now take any such deduction, and replace every occurrence of $\varphi[]$ in it with a variable ζ not occurring in either T or this deduction. The deduction is still valid, and its conclusion is $\beta(\zeta/\xi) \supset \gamma$. By Q4 we get $\exists\zeta(\beta(\zeta/\xi)) \supset \gamma$. But $\exists\xi(\beta)$ is already provable in T, and so by some changing of variables and modus-ponens, we get $T \vdash \gamma$.

Note: We did not consider the possibility that γ depends on $\beta(\varphi[]/\xi)$ by two different paths, and that it has two immediate antecedents, and is the first formula in one path not to contain φ, but the other path has been free of φ for some time and may have varied some of the variables of $\beta(\varphi[]/\xi)$. But the only rule of inference to have two antecedents is modus-ponens and if the conclusion of modus-ponens has no φ, then either both or neither of the antecedents have φ, and so the situation does not arise.

Theorem 7.12 (Constant Extensions)

If $T \subset L$ is a consistent theory, α is a formula in L containing only ξ free, and φ is a 0-ary function name not in the vocabulary of L, then $T \cup \{\exists\xi(\alpha) \supset \alpha(\varphi[]/\xi)\}$ is a consistent theory, and if $\beta \in L$ is provable in this theory, then β is also provable in T.

Problem Set 25

1. Prove theorem 7.12. (Hint: You will need as a lemma $\vdash \exists\xi(\alpha \supset \beta) \supset (\alpha \supset \exists\xi(\beta))$ when α has no free ξ. The method of proof is similar to the proof of theorem 7.11.)

2. Why isn't theorem 7.12 valid if $\exists\xi(\alpha)$ is not a sentence?

3. Theorems 7.7 and 7.8 state that:

$$\text{(i)} \quad \alpha \equiv \beta \vdash \gamma \equiv \delta$$

$$\text{and (ii)} \quad \tau_1 = \tau_2 \vdash \alpha \equiv \beta$$

where (i) δ derives from γ by replacing some occurrences of α with β, and (ii) β derives from α by replacing some occurrences of τ_1 with τ_2. If these theorems are applied to dependent lines in a proof making use of the deduction theorem, then it is important to know which variables are varied in the deductions symbolized by "$\vdash$" in lines (i) and (ii) above. This is so that no conditions of the deduction theorem are violated. Precisely which variables are varied in these deductions? Why is line 8 of the deduction following theorem 7.10 valid?

CHAPTER EIGHT

FIRST ORDER LOGIC - COMPLETENESS

§8.1 Completeness

The completeness theorem is simple to state, but lengthy to prove. We want to show that if $T \models \alpha$, then $T \vdash \alpha$. It turns out that if we can show that every consistent theory has a model, then the completeness theorem follows almost immediately. So given a consistent theorem T, we want to obtain a model for it. Since we have to do this in the abstract, i.e., for any theory, the only stuff we have available for the purpose of building a model is the vocabulary of the theory itself. To further complicate matters, there is no unique or canonical model for most theories, so the choice must be somewhat arbitrary. The program is roughly as follows:

(i) We show that more constants can be added to the language of the theory so that there is a name for every object that the theory asserts must exist.

(ii) We next extend the theory arbitrarily until it is complete.

(iii) We then show that there is a set of terms in the enlarged language that serves as the domain for a model in a fairly natural way. This model, with the extra names thrown away, is a model for the original theory.

The completeness theorem was first proved by Gödel. The present proof is derived by a method first used by Henkin.

Lemma 8.1 (Lindenbaum's Lemma)

Every consistent theory has a consistent complete extension.

Proof: Given $T \subset L$, a consistent theory. Let $\alpha_1, \alpha_2, \ldots$ be all the sentences

of L. Let T_0 be T, and T_{i+1} be $T_i \cup \{\alpha_{i+1}\}$ if α_i is independent of T_i; otherwise let T_{i+1} be the same as T_i. Let T* be the union of all the T_i. We show that each T_i is consistent by induction on i. T_0 is consistent because it is T. Assume that T_i is consistent. If T_{i+1} is the same as T_i, then it is consistent. If it is not the same, then T_{i+1} is $T_i \cup \{\alpha_{i+1}\}$, where α_{i+1} is independent of T_i. If T_{i+1} were inconsistent, then anything could be deduced from it, and in particular $T_{i+1} \vdash \neg\alpha_{i+1}$, so by the deduction theorem, $T_i \vdash \alpha_{i+1} \supset \neg\alpha_{i+1}$ or $T_i \vdash \neg\alpha_{i+1}$ which contradicts the fact that α_{i+1} is independent of T_i. So all the T are consistent, and therefore T* is consistent because any contradiction in T* would also be contained in some sufficiently large T_i. To show That T* is complete, let β be any formula in L. Then its universal closure is one of the α_i. If α_i is independent of T_{i-1}, then $\alpha_i \in T_i$, so in any case, α_i is provable or refutable in T_i and hence in T*, and so is β.

Definition 8.2

A ground term is a term with no variables in it. A ground formula is a formula with no variables in it. (A ground formula is always a sentence, but not all sentences are ground formulas.)

Definition 8.3

A theory $T \subset L$ is a Henkin theory if there is at least one ground term in L, and if whenever $\exists\xi(\alpha)$ is a sentence that is provable from T, then there is a ground term τ in L such that $T \vdash \alpha(\tau/\xi)$.

Lemma 8.4

If $T \subset L$ is a consistent theory, then there is an extension T* of T in an enlarged language L* which is a consistent complete Henkin theory.
Proof: Let T_0 be T, and L_0 be L. Let $k_{i,j}$ for $i \geq 1$ and $j \geq 1$ be a set of constants not in L. Given the language L_i, we define the language L_{i+1} by adding the constants $k_{i+1,1}$, $k_{i+1,2}$, ... to it. Given the theory $T_i \subset L_i$, we define the theory $T_{i+1} \subset L_{i+1}$ by enumerating all the sentences of L_i having only the variable x free (let this enumeration be $\alpha_{i,1}$, $\alpha_{i,2}$ ...) and adding to T_i all the sentences of the form $\exists x(\alpha_{i,j}) \supset \alpha_{i,j}(k_{i+1,j}/x)$ for j = 1, 2, 3 ...

Let L* be the union of the L_i, and T_∞ be the union of the T_i. T_∞ is consistent because it is derived from T by adding a great many new formulas, each one of which is a consistent extension by theorem 7.12. Let T* be a consistent complete extension of T_∞ by lemma 8.1. To show that T* is a Henkin theory, let $\exists\xi(\beta)$ be any sentence of L*. Let L_i be the least language of which it is a member. Let $\exists x(\gamma)$ be an equivalent formula by change of bound variables. Then γ is $\alpha_{i,j}$ for some j, and $\exists x(\gamma) \supset \gamma(k_{i+1,j}/x)$ is a member of T_{i+1}, and therefore T*. So if $\exists\xi(\beta)$ is provable in T*, then so is $\beta(k_{i+1,j}/\xi)$ via several operations on bound variables.

Lemma 8.5

A consistent, complete Henkin theory has a model.

Proof: Given the theory $T \subset L$, let D be the set of ground terms in L, but underlined. (If g[h[]] is a ground term in L, then $\underline{g[h[]]} \in D$.) D is non-empty because a Henkin theory always has at least one ground term. Let φ be an n-ary function name in L. We define the function $\tilde{\varphi}$ to interpret φ as follows. If $\underline{\tau 1}, \ldots, \underline{\tau n}$ are objects of D, then $\tilde{\varphi}(\underline{\tau 1}, \ldots, \underline{\tau n})$ is the object $\underline{\varphi[\tau 1, \ldots, \tau n]}$. Let ψ be an m-ary predicate name in L; then we define $\tilde{\psi}(\underline{\tau 1}, \ldots, \underline{\tau m})$ to be true if and only if $T \vdash \psi[\tau_1, \ldots, \tau_m]$. This defines a model in L. Call it M.

To show that $M \models T$, we shall prove that if α is any sentence in L, then $T \vdash \alpha$ if and only if $M \models \alpha$. The proof is by induction on the total number of logical connectives and quantifiers in α. Induction basis: If there are no quantifiers or logical connectives in α, then α must be a ground atomic formula. Then $T \vdash \alpha$ if and only if $M \models \alpha$ from the definition of function and predicate interpretations in M. Induction step: (i) If α is $\neg\beta$, then if $T \vdash \alpha$, then β is not provable in T because T is consistent, and hence not satisfied by M by the induction hypothesis. So $M \models \alpha$. If $M \models \alpha$, then M does not satisfy β, and β cannot be proven from T. Since T is complete, $T \vdash \alpha$. (ii) The rest of the logical connectives are left as an exercise. (iii) If α is the sentence $\exists\xi(\beta)$, then if $T \vdash \alpha$, there is a ground term τ such that $T \vdash \beta(\tau/\xi)$ because T is a Henkin theory. The sentence $\beta(\tau/\xi)$ has one less quantifier than α, and so by the induction hypothesis $M \models \beta(\tau/\xi)$. Therefore, $M \models \exists\xi(\beta)$. Now suppose that $M \models \exists\xi(\beta)$. Then M, $I \models \beta$ for some I interpreting ξ as an object in the

domain D of M. But this object $\underline{\tau}$ is an underlined ground term, and given the special way M was defined, $M \models \beta(\tau/\xi)$. By the induction hypothesis, $T \vdash \beta(\tau/\xi)$, and so $T \vdash \exists\xi(\beta)$. (iv) The $\forall$ case is left as an exercise.

Theorem 8.6

Every consistent theory has a model.

Proof: It has a consistent, complete Henkin extension in an enlarged language by lemma 8.4. By lemma 8.5, this theory has a model. Then removing the interpretations for the new constants from this model gives a model for the original theory. (See problem set 20, No. 2.)

Theorem 8.7 (Gödel's Completeness Theorem)

If $T \models \alpha$, then $T \vdash \alpha$.

Proof: (If α is not a sentence, consider its closure.) If $T \models \alpha$, then $T \cup \{\neg\alpha\}$ has no model. Therefore it is inconsistent by theorem 8.6. So anything can be proven from it. In particular, $T \cup \{\neg\alpha\} \vdash \alpha$, and so by the deduction theorem $T \vdash \neg\alpha \supset \alpha$, or $T \vdash \alpha$.

Theorem 8.8 (Compactness Theorem)

If T is not satisfiable, then there is a finite subset of T that is not satisfiable.

Proof: By theorem 8.6, if T is not satisfiable, then it is inconsistent. The demonstration of inconsistency must come from finitely many formulas of T. This finite inconsistent sub-theory has no model by corollary 7.5.

Theorem 8.9 (Skolem-Löwenheim Theorem)

If a theory has a model, it has a countable model.

Proof: If the theory has a model, then it is consistent by corollary 7.5. If it is consistent, then it has a model (theorem 8.6) which is countable by the method of proof of lemma 8.5.

The reason for producing these results in such rapid succession is to

demonstrate how many of the significant properties of first order logic follow from one central argument.

The completeness theorem has several useful interpretations. One of these is that first order deduction is strong enough to derive any conclusion which is valid. When we put completeness and consistency together, we have $T \vdash \alpha$ if and only if $T \models \alpha$. Therefore, the limitations of first order logic are linguistic. If a certain formula α cannot be derived from the theory T, it is because there are models for T in which α is false. If T is supposed to describe some model M in which α is true, then it evidently is not a complete description of M.

The completeness theorem allows us to assert many facts about provability without producing constructive proofs. Instead, we argue the case that something semantically follows from some theory, and then assert its provability from that theory by using the completeness theorem.

On another level, the completeness theorem in the form of theorem 8.6 provides a criterion for the "existence" of mathematical entities. If we invent some set of postulates, when is there a mathematical entity to which they apply? If the postulates can be formalized as a first order theory, then it is sufficient that they be consistent in order for there to be a model for them. Lemmas 8.4 and 8.5 show that consistent language, suitably extended, provides its own model or subject matter.

§8.2 Equality

We return now to the problem of equality. In §6.3, a set of axioms E_L for the equality predicate was proposed. In §7.3, it was proven that E_L is sufficient to prove the equivalence of formulas containing equal terms. In this section, we consider the model theoretic aspect of equality. From now on, we shall assume that any theory in a language containing the predicate "=" is an equality theory (has E_L as a subset) unless we state otherwise.

In §6.3, we discussed a theory that had only infinite models. Is there a theory that has only finite models? Consider the theory:

$$E_L$$

$$\exists x \exists y((x \neq y) \wedge \forall z(x = z \vee y = z))$$

Intuitively, this theory seems to say that there are exactly two different things that exist. It is satisfied by a model containing 0 and 1, with $0 \neq 1$. But it is also satisfied by the model whose domain is the integers if we interpret "=" to mean "congruent mod 2". There is nothing in the theory that prevents such an interpretation, although this is not the standard interpretation of "=". Furthermore, there are no axioms that can be added to the theory that would eliminate such interpretations.

Definition 8.10

If L is a language with equality, then a normal model in L is a model in which the interpretation of "=" is that any member of the domain of the model is "=" to itself and not "=" to any other object in the domain.

Clearly, any normal model for the two object theory must have cardinality 2. So there is an advantage to considering only normal models, since these are the ones we want anyway. The validity of this approach is confirmed by the following theorem.

Theorem 8.11

If T is a theory with equality, and M is any model for T, then there is a normal model M* which is first order equivalent to M.

Proof: In the model M, there is an interpretation for the predicate name "=" which we shall denote by the symbol "~". Since "~" satisfies the first three axioms of E_L, it is an equivalence relation on the domain D of M, and partitions D into co-sets. If $d \in D$, then we denote the co-set of all elements of D which are "~" to d as [d]. The set of all such co-sets will be called D* and is the domain of the normal model M* that we seek. We define function interpretations in M* by the following equation, where $\tilde{\varphi}$ is the interpretation of φ in M, and $\tilde{\tilde{\varphi}}$ is the new interpretation being defined on D*.

$$\tilde{\tilde{\varphi}}([d_1], \ldots, [d_n]) \text{ is } [\tilde{\varphi}(d_1, \ldots, d_n)]$$

That this is a consistent definition independent of the particular elements chosen to represent the co-sets follows from the fact that in the model M, the interpretation $\tilde{\varphi}$ of φ, and the interpretation "~" of "=" must satisfy axiom schema (iv) of E_L and therefore if $d_i \sim e_i$ for $1 \leq i \leq n$, then $\tilde{\varphi}(d_1, \ldots, d_n) \sim$

$\tilde{\varphi}(e_i, \ldots, e_n)$ and so $[\tilde{\varphi}(d_1, \ldots, d_n)]$ is the same co-set as $[\tilde{\varphi}(e_1, \ldots, e_n)]$. The interpretation $\tilde{\Psi}$ of a predicate name ψ, is derived from the interpretation $\tilde{\psi}$ similarly, and the consistency of this definition follows from axiom schema (v) of E_L.

To show that $M \sim M^*$, let α be any formula of L, and I an interpretation of the variables of α into D. Define the interpretation I* by $I^*(\xi) = [I(\xi)]$. Then show by induction on the logical connectives and quantifiers of α that if β is a subformula of α, then $M, I \models \beta$ if and only if $M^*, I^* \models \beta$.

From now on, when we speak of a model in a language with equality, we shall mean a normal model unless we explicitly state otherwise.

Problem Set 26

(All languages and theories have equality, and all models are normal.)

1. Specify a theory having infinite models, and finite models of cardinality $3 \times n$ for every $n \geq 1$, and having no finite models whose cardinality is not a multiple of three.

2. Specify a theory having models of cardinality p for every prime number p, and no other finite models.

3. Prove that if a theory has arbitrarily large finite models, that it must have infinite models. (Hint: Use the compactness theorem.)

4. Prove that if a consistent theory is complete, all models for it either have the same finite cardinality, or else they are all infinite.

§8.3 The Skolem-Löwenheim Theorem

This theorem was known early in this century before the completeness theorem was proven. It then had to have a proof that did not depend on deduction at all, but was entirely model-theoretic in nature, although the term "model" was not used until somewhat later.

If we consider a logic with equality, then the Skolem-Löwenheim theorem states that every satisfiable theory has a finite or countable (normal) model. This is rather puzzling because we can formalize the theory of real numbers in first order logic. This theory at first sight seems to require a model containing at least all the real numbers. When we study the axioms,

we find that they require the existence of all real roots of polynomials, and all real numbers defined by limits or integrals such as π or e or logarithms or Bessel functions. The theory even asserts topological closure properties such as that every non-empty set of real numbers bounded above must have a least upper bound. How then can this theory have a countable model?

This is known as Skolem's paradox, and the problem seems to lie either in our naive assumption of the absolute notion of "uncountable", or in the limitations of symbolic language to discuss what really exists. (You can take your choice.) The fact is that if we take a "description" to be a piece of writing of finite length composed of discreet symbols from a finite alphabet, then the set of all potential descriptions is countable. So regardless of what we consider to be acceptable or well-defined descriptions, we can only describe countably many real numbers. We then find that every number that we describe and look for really is in such a countable model, including, for example, the values of definite integrals which we know exist but cannot even compute.

If we believe that there really are "many" more real numbers than rational numbers or integers (and most mathematicians since Cantor act as if they believe this) then we must accept the situation that "most" real numbers are inaccessible to description in any manner. However, Skolem suggested that perhaps the notion of uncountable is relative to one's language, and that there are uncountably many real numbers in real number theory because there is no one-to-one correspondence possible between the real numbers and the natural numbers <u>within</u> the theory. But viewed from <u>outside</u> the theory, such a correspondence is possible as his countable model shows. Viewed this way, "uncountable" refers to our inability to "count" or specify an enumeration, rather than to the large size of a set.

This situation is further dramatized by the fact that it is possible to axiomatize set theory in first order logic. The Von Neumann-Bernays-Gödel (NBG) set theory has a finite number of axioms (see [Mendelson, Chapter 4]) and purports to be about sets of arbitrarily high cardinality and "classes" which are even bigger than sets, such as "the class of all sets". If NBG is consistent, then it has a countable model.[1] If it is not consistent, then

[1] That is, if one is willing to accept the fairly conservative portion of classical mathematical reasoning used in the proofs of 8.1 thru 8.5.

methods of reasoning used as a matter of course by mathematicians in all different fields are called into question.

Philosophically, one may believe that all the entities of mathematics are given a priori, but that our language has difficulty dealing with them, or, if like the intuitionists one restricts one's belief to those things that could at least potentially be written, then one may take all the higher infinities to be mere semantic constructs. There is current research [Yessenin-Volpin] which attempts to prove that axiomatic set theory is consistent from an "ultra-intuitionist" viewpoint that believes in nothing it cannot see. It is too early at this time to evaluate this work.

CHAPTER NINE

FIRST ORDER LOGIC - ADDITIONAL TOPICS

Preview of Chapter Nine

This chapter is a collection of several topics not all of which are sequentially related. The only one of these that is a necessary prerequisite for subsequent parts of this book is §9.1 which is the study of formal definitions.

The system we have been studying so far is known as Hilbert-type deduction. It is characterized by straight line proofs. Within the past decade, research in automated theorem proving has been dominated by a radically different approach known as resolution. §9.2 thru §9.4 are about resolution and its prerequisite topics. §9.5 is about still another form of deduction known as a Gentzen-type system.

In §9.6, we return to the Hilbert-type system which we shall use for the rest of this book, and discuss the question of decidability of theories.

§9.1 Definitions

When a formal theory is presented as a set of axioms T in a language L, it is usually necessary to make definitions as we proceed to develop the theory, for if we have to describe advanced concepts in primitive terms, the length of the formulas we must use to do this becomes explosively long. We shall have some examples to illustrate this later.

The main questions that we want to consider in this section are: How do we make definitions that do not add anything to the basic assumptions of the theory? How do we know that the theory is still consistent after we add

definitions to it? If the theory was designed to fit some model, how do we know that the definitions don't alter this?

Definition 9.1

Let $T \subset L$ be a theory, and $T_1 \subset L_1$ be an extension to T. We say that T_1 is a conservative extension of T if whenever $T_1 \vdash \alpha$ and $\alpha \in L$, then $T \vdash \alpha$.

A theory is consistent if and only if every conservative extension of it is consistent.

The easiest sort of definition that we can make is to replace some commonly occurring term by a new function name, or some commonly occurring formula by a new predicate name.

Rule X (Explicit Definitions):

An explicit definition is a line in a proof having the form:

$$\varphi[\xi_1, \ldots, \xi_n] = \tau$$

$$\text{or} \quad \psi[\xi_1, \ldots, \xi_m] \equiv \alpha$$

where φ is a new function name and τ is a term having no variables other than the ξ_i, or ψ is a new predicate name and α is a formula having no free variables other than the ξ_i.

The restriction on the free variables occurring in τ or α is necessary to avoid definitions that are ambiguous and have contradictory instantiations. For example, if we define $f[x] = x+y$, then two instances of this are $f[0] = 0+0$, and $f[0] = 0+1$, from which we can deduce $0 = 1$. Or if we define $p[x] \equiv (x > y)$, then we have $p[2] \equiv (2 > 1) \equiv (2 > 3)$ or $T \equiv F$.

Theorem 9.2

If $T_1 \subset L_1$ is an extension of $T \subset L$ by Rule X, then it is a conservative extension. Furthermore, if M is any model in L that satisfies T, then there is a unique expansion of M in L_1 that satisfies T_1.

Proof: Let M be a model in L that satisfies T. If φ is a new function name

in L_1 introduced by Rule X, then we define $\tilde{\varphi}(d_1, \ldots, d_n)$ for any elements d_i in the domain of M by letting this value be $V(M, I, \tau)$ where τ is the defining term in Rule X, and I is an interpretation of the variables ξ_i into the d_i respectively. Any other way of defining $\tilde{\varphi}$ would not satisfy the defining equation, and so the expansion of M is unique. If the extension is by way of a predicate name ψ, then we define $\tilde{\psi}(d_1, \ldots, d_m)$ to be true if and only if $M, I \vdash \alpha$, where I interprets the ξ_i which are the only free variables in α into the d_i respectively, and this expansion is also unique.

Suppose $T_1 \vdash \alpha$ and $\alpha \in L$. Let M be any model for T. M has an expansion that satisfies T_1 and therefore satisfies α. Since $\alpha \in L$, the contraction of M_1 to M also satisfies α. Since this is true for all M that satisfy T, we have $T \models \alpha$, and by completeness, $T \vdash \alpha$. So T_1 is a conservative extension of T.

The uniqueness quantifier $\exists_1 \xi$ means: "There exists exactly one ξ such that ...." It is not a new logical concept, but merely an abbreviation. The formula $\exists_1 \xi(\alpha)$ is an abbreviation for $\exists \xi(\alpha \wedge \forall \zeta(\alpha(\zeta/\xi) \supset \zeta = \xi))$, where ζ is a variable not occurring in α. This notation is used only in languages with equality.

If the formula $\exists_1 \xi(\alpha)$ has only the variables ζ_1 thru ζ_n free, and the (normal) model M satisfies it, then for every choice of d_1 thru d_n in the domain of M, there must be exactly one d_{n+1} such that if I interprets $\zeta_1, \ldots, \zeta_n, \xi$ into d_1 thru d_{n+1} respectively, then $M, I \vdash \alpha$. This defines a n-ary function on the domain of M.

Rule F (Function Definitions):

> In a deduction in a theory with equality, if line (j) is $\exists_1 \xi(\alpha)$ and has only ζ_1 thru ζ_n free, then we may derive as line (i) $\alpha(\varphi[\zeta_1, \ldots, \zeta_n]/\xi)$ where φ is an n-ary function name, and the justification for line (i) is "Rule F j", where $j < i$.

Theorem 9.1

If $T \subset L$, $T \vdash \exists_1 \xi(\alpha)$, α has only the variables ξ and the ζ_i free, φ is a new name, and T_1 is $T \cup \{\alpha(\varphi[\zeta_1, \ldots, \zeta_n]/\xi)\}$, then T_1 is a conservative

extension of T, and if M is any model that satisfies T, then there is a unique expansion of M that satisfies T_1.

The usefulness of definitions comes from the uniqueness of their model expansions, which is a much stronger condition than that the definitions are conservative extensions. It not only guarantees consistency, but means that the theory being developed is still applicable to the original model.

A strong proof-theoretic property of definitions is that they are eliminable. This means that every formula in the extension can be effectively mapped onto a formula of the original language in a manner that preserves provability. So anything that can be said in the extended language can be said in the original language, although it may be of prohibitive length and therefore not a practical thing to do. Proving the effective elimination of Rule X definitions is easy. Proving the effective elimination of Rule F definitions using proof-theoretic techniques is quite complicated combinatorially. It is done in [Kleene §74].

The following examples show how rapidly the process of definition can proceed. The theory N is the classical theory of natural numbers whose axioms we do not specify here. The theory is stated in the language $\{=, 0, ', +, \times\}$.

1.	$(m < n) \equiv \exists p(m + p' = n)$	Rule X
2.	$(m \geq n) \equiv \neg(m < n)$	Rule X
3.	$\text{prime}[m] \equiv \neg\exists p\exists q(0' < p \wedge p < m \wedge p \times q = m) \wedge 0' < m$	Rule X
4.	$\exists_1 p((n = 0 \supset p = 0) \wedge (0' < n \supset (n \times p < m' \wedge n \times p' \geq m')))$	(This is now provable.)
5.	$(n = 0 \supset m \div n = 0) \wedge (0 < n \supset (n \times (m \div n) < m' \wedge n \times (m \div n)' \geq m'))$	Rule F 4

This definition of division is peculiar. The reason is that Rule F only allows us to define total functions. In order to make division total, we have to arbitrarily define division by 0, it doesn't matter how. The second part of line 5 is the useful part, and it cannot be used to prove any properties of division by 0.

This brings up an interesting point, which is that the models of first

order logic always have total functions. This does not mean that we cannot model a subject that has partial functions. It does mean that if we provide axioms that do not completely specify a function, then we may expect the theory to be incomplete, and to be satisfied by all models that complete the partial functions in all possible ways.

To put this differently, suppose we had introduced as an axiom of N the formula $0 < n \supset (n \times (m \div n) < m' \wedge n \times (m \div n)' \geq m')$. This defines division except by 0. It allows us to prove all the ordinary results about division that we would like to prove, but formulas such as $m \div 0 = 0$ or $m \div 0 = 1$ will be independent of this theory. We may choose to use this approach because it is distasteful to make arbitrary choices that are not necessary.

We now introduce additional definition schemas to define functions and predicates by cases, and to define partial functions and predicates. It is important to know whether a given function or predicate has been introduced as total or partial. The rules X and F already specified, and the rule K that we give next define total functions. the rules PK and PF define partial functions.

Rule K (Definition by Cases):

The definition schemas:

$$\gamma_1 \supset \psi[\xi_1, \ldots, \xi_m] = \alpha_1$$
$$\ldots$$
$$\gamma_k \supset \psi[\xi_1, \ldots, \xi_m] = \alpha_k$$

and

$$\gamma_1 \supset \varphi[\xi_1, \ldots, \xi_n] = \tau_1$$
$$\ldots$$
$$\gamma_k \supset \varphi[\xi_1, \ldots, \xi_n] = \tau_k$$

are justified when (i) ψ or φ is a new name, (ii) the τ_i have no variables other than the ξ_i, and the α_i and γ_i have no free variables other than the ξ_i, (iii) $T \vdash \neg(\gamma_i \wedge \gamma_j)$ for $i < j \leq k$, and (iv) $T \vdash (\gamma_1 \vee \ldots \vee \gamma_k)$. If all previously defined function and predicate names used in such a definition are total, then the new function or predicate name is total.

Rule PK (Partial Definition by Cases):

Same as Rule K except that condition (iv) is not required.

Rule PF (Partial Function Definition):

If $T \vdash \gamma \supset \exists_1 \xi(\alpha)$ where the only free variables in this formula are ζ_1 thru ζ_n, and φ is a new function name, then:

$$\gamma \supset \alpha(\varphi[\zeta_1, \ldots, \zeta_n]/\xi)$$

is justified, and defines a partial function.

Problem Set 27

1. Prove theorem 9.3.

2. Prove that all total definition schemas imply unique model extensions, and that all partial definition schemas imply the existence of model extensions, and that all these extensions are conservative.

3. Critique the following proposal for an "ambiguous function" definition schema: If $T \vdash \exists \xi(\alpha)$, and the only free variables in this formula are ζ_1 thru ζ_n, and φ is a new function name, then define φ by $\alpha(\varphi[\zeta_1, \ldots, \zeta_n]/\xi)$.

§9.2 Herbrand's Theorem

Definition 9.4

A sentence is called a prenex normal form sentence if it is $Q_1 \xi_1 \ldots Q_n \xi_n(\alpha)$ where each Q_i is either $\forall$ or $\exists$, the ξ_i are distinct variables, and α has no quantifiers.

Theorem 9.5

Every sentence is equivalent to a sentence in prenex normal form having the same function and predicate names.

If T is a theory, then $Th(T) = Th(T_1)$ where T_1 is a set of prenex normal forms equivalent to the closures of the formulas in T.

Proof Sketch: To put a sentence in prenex normal form (i) eliminate "$\equiv$" by $(\alpha \equiv \beta) \rightarrow (\alpha \supset \beta) \wedge (\beta \supset \alpha)$, (ii) eliminate "$\supset$" by $(\alpha \supset \beta) \rightarrow (\neg\alpha \vee \beta)$, (iii) change

variables so that every quantifier has a distinct variable, (iv) move the quantifiers outward using transformations such as $\neg\forall\xi(\alpha) \rightarrow \exists\xi(\neg\alpha)$ and $\alpha \vee \forall\xi(\beta) \rightarrow \forall\xi(\alpha \vee \beta)$. These are all equivalences. (Note that in the last formula, α has no free ξ. Why?)

Herbrand was trying to solve the fundamental problem of first order logic, which is to determine when a formula α is a member of Th(T), by purely proof-theoretic techniques. As part of this program, he showed how a theory could be expanded into a form in which there were no quantifiers.

Given the theory T, we have the equivalent theory T_1 in prenex normal form. Let $Q_1\xi_1 \ldots Q_n\xi_n(\alpha)$ be a sentence of this theory. If Q_1 is universal, then it can be dropped by rule QX3. If it is existential, then we can drop the quantifier and make the substitution $\varphi[\,]/\xi_1$ in the manner of problem set 27, number 3. In either case, we have gotten rid of the first quantifier. This process can be repeated for each quantifier in turn, merely dropping the universal quantifiers, and substituting ambiguous function names for the existentially quantified variables. If ξ_i is existentially quantified, then it will be replaced by $\varphi_i[\xi_{i_1}, \ldots, \xi_{i_k}]$ where $Q_{i_1} \ldots Q_{i_k}$ are the universal quantifiers to the left of Q_i in the original formula. For example:

$$\forall x \exists y \forall z \exists w(p[x, f[y, w], z, g[y, z]])$$

becomes

$$p[x, f[h1[x], h2[x, z]], z, g[h1[x], z]]$$

where h1 and h2 are new function names. They are called <u>Herbrand function names</u>.

This process can be done for an entire theory T_1 in prenex normal form producing the <u>open theory</u> T_2. From the previous discussion it should be clear that $T_1 \cup T_2$ is a conservative extension of T_1, and that if M is a model for T_1, then there is an expansion of M that satisfies T_2. This expansion is not necessarily unique. Conversely, any model for T_2 can be contracted to a model for T_1; therefore T_1 is satisfiable if and only if T_2 is satisfiable.

Let L_2 be the language of T_2. It is the language of T (and T_1) together with all the Herbrand function names. Let H be the set of all

ground terms in L_2. We add one constant to L_2 if necessary to make sure that H is not empty.

Let T_3 be the set of all ground instances of T_2. (If α is an open formula in T_2 having distinct variables ξ_1 thru ξ_n, and h_1 thru h_n are in H, then $\alpha(h_1/\xi_1, \ldots, h_n/\xi_n)$ is a ground instance of α.) If T_2 is satisfiable, then obviously T_3 is satisfiable. The converse is also true, but needs a proof, which we supply presently.

When we look at the formulas of T_3, we see that not only are there no quantifiers, but there are no variables either. A formula in T_3 is simply a logical compounding of ground atomic formulas. If we view each distinct ground atomic formula as a distinct propositional variable, then we can regard T_3 as a theory of propositional logic. If T_3 is satisfiable as a first order theory, then it is also satisfiable as a propositional theory by allowing a first order model to supply truth values for each ground atomic formula.

Conversely, if T_3 is satisfiable as a propositional theory, then it is satisfiable as a first order theory. To show this, let M be a propositional model for T_3. We define the model M' on the domain H of ground terms by defining function interpretations in the same manner as in lemma 8.5, i.e., $\tilde{\varphi}(h_1, \ldots, h_n)$ is the term $\varphi[h_1, \ldots, h_n]$. We define $\tilde{\psi}(h_1, \ldots, h_m)$ to be true if and only if $M \models \psi[h_1, \ldots, h_m]$. This defines M', and $M' \models T$ because it produces the same valuations on ground atomic formulas as does M.

M' also satisfies T_2 because if $\alpha \in T_2$, then α is an open formula, and if I is any interpretation of the variables of α into H, then $M, I \models \alpha$, because the corresponding ground instance in T_3 is also satisfied by M'. (This sort of argument can only be used when we already know that the language has a ground term to express every object in the domain of the model. The situation is similar in some ways to lemma 8.5.) This proves that T_2 is satisfiable if and only if T_3 is satisfiable.

Theorem 9.6 (Herbrand's Theorem)

Suppose that T is an inconsistent theory. This fact can be demonstrated in the following way. Let T_1 be the prenex normal form for T. Let T_2 be the open theory obtained from T_1 by dropping quantifiers and introducing Herbrand function names. Let T_3 be the set of all ground instances

of T_2 (making sure that H is not empty). Then there is some finite set of formulas in T_3 whose conjunction is propositionally inconsistent.
Proof: If T is inconsistent, then it is unsatisfiable by the consistency theorem. If T is unsatisfiable, then T_1, T_2 and T_3 are unsatisfiable as noted in the preceding discussion. Then T_3 is propositionally unsatisfiable. By the compactness theorem for propositional logic, some finite part of T_3 is unsatisfiable, and by the completeness theorem for propositional logic, the conjunction of this finite set of formulas is inconsistent propositionally.

This proof would not have been satisfactory to Herbrand. The statement of the theorem makes no reference to models, and can be proven using only finitary proof-theoretic methods. Such a proof is given in [Herbrand, p. 168]. The proof is complicated and has error which has been corrected by subsequent logicians. (Herbrand's paper was presented as a thesis at the Sorbonne in 1930. In 1931 Herbrand was killed in an alpine climbing accident when a piton came out. He was 23 years old.)

If we can demonstrate inconsistency, then we can also demonstrate provability because $T \cup \{\neg\alpha\}$ is inconsistent if and only if $T \vdash \alpha$. The insight of Herbrand's theorem is that in all cases only a finite amount of model construction effort is necessary to show that no model can be built for a theory. This suggests an entirely new approach to creating demonstrations than the Hilbert-type system, and Herbrand's theorem is the "completeness" theorem for this new type of demonstration. This idea will be expanded in §9.4.

§9.3 Substitution and Unification

The theory of substitution and unification is part of the theory of resolution developed by [Robinson]. It is interesting enough in its own right to be presented as a separate topic. It is perhaps part of the answer to the question: What is the equivalent in the theory of symbolic processing to the number theoretician's interest in factoring, least common multiples and so forth?

Before we can perform the operation of substitution, we need something on which to do the substituting. We could develop the theory of substitution on s-expression but, instead, we shall do it the way Robinson does it

so that nothing needs to be altered for §9.4.

Definition 9.7

A literal is either an atomic formula or else it is a negated atomic formula (i.e., an atomic formula preceded by "¬"). A clause is a finite set of literals. Literals and clauses that do not have variables are called ground literals and ground clauses.

The interpretation of a clause that we shall use in §9.4 is the disjunction or "or" of its literals. The idea of a set of literals rather than a sequence is that a set does not specify an order for its components, nor is it meaningful for an element of a set to be a member several times over. This is a useful condensation of the associative, commutative and idempotent properties of "∨". A clause can be represented by the usual finite set notation which is a list of elements enclosed by braces and separated by commas.

Examples of Literals:

p[x, y]
¬p[k[], j[]]
¬q[x, f[y, g[x, y]]]
r[x, (A B C)]

Examples of Clauses:

{¬p[x, y], r[x, (A B C)], ¬q[x, f[y, g[x, y]]]}
{x+y = 3, 1+2 ≠ 3}

Definition 9.8

A substitution component is an expression of the form "τ/ξ" where τ is a term and ξ is a variable, and $\tau \neq \xi$. Its meaning is "substitute τ for all occurrences of ξ." A substitution is a finite set of substitution components such that each ξ_i is distinct. Its meaning is "substitute each τ_i for all occurrences of its ξ_i." This is a simultaneous substitution.

If C is a clause, and θ is a substitution, then $C\theta$ is the clause resulting from performing θ on C. For example, if C is {p[x, y], ¬q[f[y]]}, and θ is {g[z]/x, f[x]/y}, then $C\theta$ is {p[g[z], f[x]], ¬q[f[f[x]]]}. The notation $C\theta\lambda$ means

the clause resulting from first performing θ on C, and then performing λ on the result of this. This is a "postfix" operator style of notation which has the advantage that the operations get performed from left to right.

Definition 9.9

If θ is the substitution $\{\tau_1/\xi_1, \ldots, \tau_n/\xi_n\}$ and λ is the substitution $\{\sigma_1/\zeta_1, \ldots, \sigma_m/\zeta_m\}$, then the composition of θ and λ, written $\theta\lambda$, is the substitution defined as follows: Let λ' be the set of all components of λ except those for which ζ_i is one of the ξ's. Let θ' be the set of all components of the form $\tau_i\lambda/\xi_i$ where τ_i/ξ_i is in θ, and $\tau_i\lambda$ is the result of performing λ on τ_i, except those cases where $\tau_i\lambda$ is ξ_i in which case $\tau_i\lambda/\xi_i$ is not a substitution component. Then $\theta\lambda$ is defined to be the union of the sets θ' and λ'.

This definition of composition of substitutions is not commutative because it is intended to produce the substitution which is "first do θ, then do λ". If the τ's replace all occurrences of the ξ's and then λ is performed, they will get changed into $\tau\lambda$'s. The σ_i/ζ_i components can act on the original text only when ζ_i is not one of the ξ's. However, even if they get thrown out they still have an effect in defining the $\tau_i\lambda/\xi$ components. For example, the composition of {f[x]/x} with itself is {f[f[x]]/x}.

Corollary 9.10

For any clause C, and any substitutions θ and λ, $(C\theta)\lambda = C(\theta\lambda)$.

For any substitutions θ, λ and μ, $(\theta\lambda)\mu = \theta(\lambda\mu)$. (Substitution is associative.)

The set of all substitutions form a semi-group, with the empty substitution as identity.

Examples of composition of substitutions:

{x/y} {x/y, y/x} = {y/x}

{x/y, y/x} {x/y} = {x/y}

{g[x, y]/x, h[y, z]/y} {f1[y]/x, f2[z]/y, f3[x]/z} = {g[f1[y], f2[z]]/x, h[f2[z], f3[x]]/y, f3[x]/z}

$\{n^2 + 2/m, 3 \times m/n\}\ \{n^2 - 3/m\} = \{n^2 + 2/m, 3 \times (n^2 - 3)/n\}$

Definition 9.11

A finite set of literals is called a singleton if it has exactly one element. If C is a finite set of literals, and $C\theta$ is a singleton, then θ is said to be a unifier of C. θ is a most general unifier of C if it is a unifier of C, and if for every λ which is a unifier of C, $\lambda = \theta\mu$ for some μ.

Not every set can be unified. A necessary but not sufficient condition for a set to be unifiable is that either all literals begin with "¬" or else none of them do, and that they all have the same predicate name. At the opposite extreme, if a set is already a singleton, then every substitution is a unifier of it, and the empty substitution is its most general unifier.

Examples:

{p[3], p[5]} cannot be unified.

{p[3], p[x]} has most general unifier {3/x}.

{p[x], p[f[y]]} has most general unifier {f[y]/x}.

{p[x], p[f[x]]} cannot be unified.

{q[f[y], x], q[x, f[z]]} has most general unifier {f[y]/x, y/z} or {f[z]/x, z/y}.

The unification algorithm is an effective process for finding the most general unifier of a set of literals if it exists. The algorithm as given does not work for clauses containing infix or postfix operators or other relaxations of grammar, and we do not attempt to change this.

Let C be a finite set of literals. The disagreement set D of C is the set of all well-formed terms or formulas that begin at the first symbol position at which not all of the literals of C agree. We can think of a cursor moving character by character from left to right on all the literals in C and stopping as soon as there is any discrepancy between any two literals. We then copy the smallest well-formed term or formula that starts at each cursor position, and this is the disagreement set. For example, the disagreement set of {p[x, h[x, y], y], p[x, g[y], y], p[x, a, b]} is {h[x, y], g[y], a}. If C has at least two literals, then the disagreement set of C has at least two elements. The disagreement set is obviously computable.

The unification alogrithm is stated as a program, with program variables C, D, θ, ξ and τ. C is initialized to the set to be unified, and σ is initialized to the empty substitution.

Loop: C:= Cθ; (Performing θ on C is specified here.)
If C is a singleton then terminate with most general unifier θ;
D:= disagreement set of C arranged in a sequence with variables ahead of other elements;
ξ:= first element of D;
τ:= second element of D;
If ξ is not a variable then fail;
If τ contains occurrences of ξ then fail;
θ:= $\theta\{\tau/\xi\}$; (Composition of substitutions is specified here.)
Go to loop;

Theorem 9.12 (Unification Theorem)

If C is a finite set of literals, then if it has a unifier, it has a most general unifier, and the unification algorithm will compute one. Otherwise, the algorithm will terminate with a fail. The algorithm always terminates. (Proof in [Robinson].)

Problem 27

The LISP function sublis[x, y] performs a substitution on the s-expression y when x is a list of pairs, each of which is a substitution component. (See §8.1.) Let us call x a substitution if it is a list of pairs, and the cadr's of the pairs are all different atoms, and car and cadr of each pair are distinct. Define a LISP function compose[x, y] such that if x and y are substitutions, then compose[x, y] is a substitution, and if z is any s-expression, then sublis[y, sublis[x, z]] = sublis[compose[x, y], z].

§9.4 Resolution

We continue from the concluding remark of §9.2. Starting with a theory T that we wish to demonstrate inconsistent, we generate T_1 in prenex normal form, and T_2 which is an open theory. The next transformation in

this process of preparation is to put the formulas of T_2 into what is known as conjunctive normal form.

Definition 9.13

If L_1 thru L_n are literals, then $L_1 \vee \ldots \vee L_n$ is called a disjunct. If D_1 thru D_m are disjuncts, then $D_1 \wedge \ldots \wedge D_m$ is called a conjunctive normal form.

It follows from DeMorgan's Laws, and the distributive laws for logical connectives that every open formula is equivalent to a conjunctive normal form. Having put a formula in conjunctive normal form, we can then turn each disjunct into a clause simply by eliminating any redundancies and making a set of the literals. Now if we have a theory in such form, each formula is a conjunction of clauses. Since a theory is semantically the conjunction of its formulas, we can further collapse the whole structure and regard the theory as simply a (possibly infinite) set of clauses in conjunction. The boundaries of formulas are no longer important. If T_2 is an open theory, we call the equivalent set of clauses T_3.

If T_3 is unsatisfiable, then there is some finite set of ground instances of T_3 which is inconsistent. Call this T_4. Ground resolution is an essentially propositional rule of inference on ground clauses that is used to demonstrate the inconsistency of T_4.

Definition 9.14 (Ground Resolution)

If α is an atomic formula, then α and $\neg\alpha$ are called complementary literals. A ground resolvent of a pair of clauses having complementary literals is the clause consisting of all the other literals of both clauses, as is indicated by the following schema, where α and $\neg\alpha$ are complementary, and the β_i and γ_i are any literals and $i \geq 0$, and $j \geq 0$.

$$\frac{\{\alpha, \beta_1, \ldots, \beta_n\} \quad \{\neg\alpha, \gamma_1, \ldots, \gamma_m\}}{\{\beta_1, \ldots, \beta_n, \gamma_1, \ldots, \gamma_m\}}$$

This rule is not only propositionally valid, but is complete in the

following sense: The empty clause has the value "false" in all interpretations. Robinson denotes the empty clause by the symbol □. If a set of ground clauses is inconsistent, then it is possible to deduce □ by a finite number of applications of ground resolution, and no other rules of inference or axioms.

So far, there is no great efficiency in this schema. It is not any faster than earlier decision procedures such as [Davis and Putnam]. The major advantage of resolution compared to ground resolution is that it is not necessary to generate the theory T_4 at all. Resolution is a combination of ground resolution and instantiation. But instead of generating ground clauses, it does no more instantiation than is necessary. In resolution all substitutions are as general as possible.

Resolution is defined as a deduction rule that has two clauses (not generally ground clauses) as its antecedents, and another clause as its consequent. A pair of clauses may have no resolvents, or one resolvent, or more than one resolvent. The completeness theorem for resolution is that if T_3 is unsatisfiable, then there is some finite sequence of resolutions on T_3 that generates □. The completeness theorem follows from Herbrand's Theorem and is in Robinson's paper.

Definition 9.15

Let C and D be two clauses. Let C' be obtained from C by substituting the variables x1, x2 ... for the variables occurring in C, and D' be obtained similarly from D using the variables y1, y2 ... This is to guarantee that C' and D' have distinct variables without their being substantially different from C and D.

Suppose that there are sets L, M and N such that: (i) $L \subset C'$, (ii) $M \subset D'$, (iii) L and M are non-empty, (iv) N is the set of all atomic formulas that are either in L or M, or whose complements are in L or M, (v) N is unifiable, and θ is a most general unifier of N, and (vi) $L\theta$ and $M\theta$ are complementary singletons. Then $(C' - L)\theta \cup (D' - M)\theta$ is a resolvent of C and D.

As an example of resolution, we prove the validity of the sentence $\forall x(p[x] \equiv q[x]) \supset (\exists x(p[x]) \equiv \exists x(q[x]))$. (See problem set 21, No. 16.) First,

the entire sentence is negated, to obtain the one sentence theory T, then it is put in prenex normal form, T_1, and the quantifiers are dropped introducing the constants k_1 and k_2 in T_2. Then it is put in conjunctive normal form, giving the theory T_3 which is the first six lines of the proof presented below. This is not at all obvious, and it will probably take some effort to obtain this result, and also to verify that T_3 really is the denial of the original formula. It is worthwhile doing this. Note that it is essential to the meaning of line 6 that it have two distinct variables.

Since each line in the demonstration is a clause, we do not bother with the braces. The renaming of variables is also relaxed in a manner that does not affect the demonstration. Lines 3 and 5 are superfluous.

1.	$\neg p[x]\ \ q[x]$	
2.	$\neg q[x]\ \ p[x]$	
3.	$p[k_1]\ \ \neg p[x]$	
4.	$p[k_1]\ \ q[k_2]$	
5.	$\neg q[x]\ \ q[k_2]$	
6.	$\neg p[x]\ \ \neg q[y]$	
7.	$q[k_1]\ \ q[k_2]$	Res 1, 4
8.	$\neg p[x]\ \ q[k_1]$	Res 6, 7
9.	$\neg p[x]$	Res 6, 8
10.	$\neg q[x]$	Res 2, 9
11.	$q[k_2]$	Res 7, 10
12.	$\square$	Res 10, 11

Lest we give the impression that resolution is obscure, we offer a proof of $0' + 0' = 0''$ from the same assumptions as the long demonstration in §7.2. In doing the preparatory work for this problem, we come across an interesting property of resolution. Suppose we wish to prove α from a set of formulas β_1 thru β_n which can be axioms, definitions, or previously proven theorems. We do this by demonstrating the inconsistency of $\neg(\beta_1 \supset \ldots \supset \beta_n \supset \alpha)$. In conjunctive normal form, this becomes $\beta_1 \wedge \ldots \wedge \beta_n \wedge \neg\alpha$. This means that the premises of the demonstration do not have to be negated, and that each one can be prepared independently. Only α needs to be negated. In the following demonstration, lines 1 thru 4 are given, and line 5

is the negation of what we are trying to prove.

1. $x \neq y \quad y \neq z \quad x = z$
2. $x \neq y \quad x' = y'$
3. $m + 0 = m$
4. $m + n' = (m + n)'$
5. $0' + 0' \neq 0''$
6. $(m + 0)' = m'$ Res 2, 3
7. $(m + n)' \neq z \quad m + n' = z$ Res 1, 4
8. $m + 0' = m'$ Res 6, 7
9. $\square$ Res 5, 8

It is very characteristic of resolution that although we can prove $m + 0' = 0'$ directly with no negations of desired results, we cannot prove $0' + 0' = 0''$ this way. The reason for this is that the latter is an instance of the former, and resolution always keeps things in their most general form. The preceding demonstration is about as efficient as one could hope for. Each line represents a bit of reasoning leading directly to the desired result.

Since the invention of resolution, a great deal of effort has gone into making it even more efficient. Resolution fits in well with many different heuristic devices used by artificial intelligence programs. It has been shown that resolution is complete under severe restrictions as to the order in which different clauses get introduced. The effect of such restrictions is to cut through the combinatorial explosiveness of having to resolve all clauses in all possible ways. When there is a model of the subject matter available, it becomes possible to use it to drive the resolution into fruitful lines of attack. There is now an entire book about resolution and the many techniques that have been invented to increase its efficiency. [Chang and Lee]

In comparing a Hilbert-type proof system with resolution, let us start with some of the differences. A Hilbert system is a linear method of deduction following precise rules and therefore subject to mechanical verification which we call proofchecking. It has more symbols than are actually needed, and at every point offers many different options. There are always different ways of expressing the same thing. Most of the design effort, including the various kinds of definitions, has gone into making it possible for a person who is inventing a proof to formalize it in a manner which approxi-

mates his own use of language.

Resolution, on the other hand, has been designed for the purpose of mechanical theorem proving. Rather than allowing for flexibility in expression, just the opposite tactic is used. The input data is reduced to a canonical form as soon as possible, even at the cost of making it humanly unintelligible. The combinatorial complexity is reduced by having a single rule of inference, by keeping all assertions in their most general form, and by heuristics, all of which provide restrictions rather than introduce additional options. The result is the most powerful in-depth mechanical theorem prover available today.

We might ask what use is it? Even if further improvements resulted in a speed-up by a factor of 10^{10}, this would not be enough to give a theorem prover the appearance of "intelligence". The idea of a theorem prover as a sort of universal intelligence has been largely abandoned by people working in artificial intelligence. The usefulness of a theorem prover seems to be in filling in the gaps left by some more intuitive process, whether that process is human or machine.

§9.5 Gentzen-Type Systems

Gentzen developed a system of deduction quite different in appearance from Hilbert-type systems, for the purpose of studying the properties of deductions. An exposition of Gentzen's system can be found in [Kleene §77]. We do not describe the system here, but simply comment that rather than being linear like a Hilbert deduction, a deduction in Gentzen's system has the shape of a tree with the resultant theorem at the base of the tree, and a branching structure going up from this. The tip of every branch is a certain type of trivial tautology.

An interesting aspect of a Gentzen-type system, which has a certain appeal for artificial intelligence programming, is that it is highly suitable for working backwards from the goal to the given data, creating a structure of subgoals on the way. A list of subgoals may be conjunctive or disjunctive, that is, either it is necessary to solve all of them, or only one of them. This sort of alternating tree is similar to a move tree in a two-person game such as chess. A Gentzen-type system would have been at least as suitable as a

Hilbert-type system for the purposes of this book, and probably more so for anyone building a real proofchecker. We have used a Hilbert-type system only because it is more familiar and easier to explain initially.

Extensive research has been done on modified Gentzen-type systems. [Yonezawa] has designed a theorem prover containing many fewer and simpler rules than Gentzen originally had. He also has restrictions on generating substitution instances that make for efficiency. Yonezawa proves that this restricted system is nevertheless complete. When one looks at this program, one gets the feeling of seeing the basic principle of resolution (substitutions kept most general) in a different form. This suggests an interesting field of study which might be called comparative proof theory.

§9.6 Decidability

A theory T is called effective if T is a recursively enumerable set. If T is an effective theory, then Th(T) is a recursively enumerable set since it is theoretically possible to enumerate all deductions in T.

The theory T is called decidable if Th(T) is a recursive set. This does not follow in any way from T being a recursive or even a finite set.

Theorem 9.16

If $T \subset L$ is decidable and $\alpha \in L$, then $T \cup \{\alpha\}$ is decidable.

Proof: If $T \vdash \alpha$, then $Th(T \cup \{\alpha\}) = Th(T)$. If $T \vdash \neg\alpha$, then $T \cup \{\alpha\}$ is inconsistent, and $Th(T \cup \{\alpha\}) = L$. The interesting case is where α is independent of T. We can assume that α is a sentence. Then by the deduction theorem $T \cup \{\alpha\} \vdash \beta$ if and only if $T \vdash \alpha \supset \beta$, and this is decidable because T is decidable.

Corollary 9.17

Every consistent decidable theory can be extended to a complete consistent decidable theory.

First order logic is called decidable if the set of all valid sentences is a recursive set.

Corollary 9.18

If there is at least one undecidable theory having a finite axiomatization, then first order logic is undecidable. (In Chapter Twelve we provide such a theory.)

Problem Set 29

1. Prove corollary 9.17. (See lemma 8.1.)
2. Prove corollary 9.18.

CHAPTER TEN

INFORMAL ARITHMETIC

Preview of Chapter Ten

The study of the natural numbers is known as number theory. When we say "arithmetic", we mean the more generalized study of s-expressions including natural numbers, or possibly the study of discrete data structures in general, which we comment on briefly. The study is "informal" in the sense of being a mathematical discussion in English as distinct from a formal theory expressed in first order logic (which we study beginning in Chapter Eleven).

§10.1 The Postulates of Arithmetic

Peano's postulates for the natural numbers are:

1. Zero is a number.
2. The successor of a number is a number.
3. Zero is not the successor of any number.
4. No two numbers have the same successor.
5. Any property which is true for zero, and is such that if it is true for some number it is also true for the successor of that number, is true for all numbers.

These axioms are stated informally, and do not come with any instructions on how to reason logically from them. The notion of equality and its properties, as well as the notion of a function, and the fact that successor is a function, are also not explicitly given. In trying to reason

from such a set of axioms, it is not quite clear which assumptions that we bring to the problem are logical, set-theoretic, arithmetic, etc. That is why formal systems were developed.

The last postulate is known as the induction principle, and has always been the most controversial of them. We have already used induction in many of the proofs of theorems in this book. The notion of "property" in the induction postulate is a bit vague. In formal number theory, property is taken to mean "predicate".

The LISP postulates are a complete analogue to Peano's postulates. They even correspond in number. They are:

1. Atoms are s-expressions.
2. Cons of any two s-expressions is an s-expression.
3. Cons of two s-expressions is never an atom.
4. If α differs from β, or if γ differs from δ, then cons of α and γ differs from cons of β and δ.
5. Any property which is true for all atoms, and is such that if it is true for α and β it is also true for cons of α and β, is true for all s-expressions.

The induction principle can be used informally on s-expressions to discuss properties of tree-type structures. For example, consider the LISP function reverse defined by:

```
reverse[x, y] ← [atom[x] → x, T → cons[reverse[cdr[x]],
                                       reverse[car[x]]]]
```

This recursive definition can be stated in English without reference to car and cdr as follows:

(i) Reverse of an atom is itself.

(ii) Reverse of the cons of two s-expressions is reverse of the second consed with reverse of the first.

From (i) it follows that reverse of reverse of an atom is itself. Now suppose that reverse of reverse of α is itself, and the same for β. Then by (ii) reverse of reverse of cons α and β is reverse of (reverse of β consed with reverse of α) which is reverse of cons of β and α. Applying (ii) again we get

that this is reverse of α consed with reverse of β which is α consed with β. This supplies the induction step, and from the induction principle we conclude that reverse of reverse of any s-expression is itself.

There are only two differences between the LISP postulates and Peano's postulates. One is that cons is binary, while successor is unary. The other is that there are many atoms but only one zero. So in addition to the LISP postulates we need some atom postulates:

1. Every atom is either a name or a number but not both.
2. The names are in one-to-one correspondence with the numbers.

Another way of putting (2) is to say that the names can be effectively enumerated.

Neither predecessor, nor car and cdr are mentioned in these postulates. The reason for this is to avoid the fact that these are partial functions. However, there is no problem introducing them as either partially defined functions, or functions completed in an arbitrary way.

The functions plus and times are not mentioned in the theory either. If one tries to define these in the language already given by Peano's postulates, one finds that there is no way to do this that does not add something more to the theory. In fact, when we formalize this theory, it turns out that there is no way to make these definitions so that they are conservative.

There is no reasonable LISP analogue for plus and times. Therefore, starting from this point, the two theories diverge.

§10.2 Primitive Recursion

The reason why the definitions of plus and times are not conservative is because they are recursive. Recursive definitions do not always terminate, and, as we have seen in Chapter Five, there is no general way to decide which ones do and which ones do not. We have not considered so far what happens when a recursive definition is added to a first order theory. This topic is important, but needs a full and detailed treatment which we provide in Chapter Fourteen. For the moment, let us note that it is "safe" to add a recursive definition to a theory if we know that it defines a total function, but

that such definitions are not necessarily conservative. (Note that none of the definition schemas of §9.1 allow any recursion at all.)

Because the problem of deciding which procedures compute total recursive functions is not generally decidable, it is useful to define a subset of the recursive procedures which are easily recognized by their restricted syntax, always define total recursive functions, and define a wide variety of important and useful functions. The set of primitive recursive procedures meets these criteria, and any function that can be computed by a primitive recursive procedure is called a primitive recursive function. They are discussed informally here, and formally in Chapter Twelve.

The basic idea of primitive recursion is to recur in a manner which counts down, and terminates at zero. In an explicit definition of f(n), f would not appear in the definition because this is "circular" or recursive. In a primitive recursive definition, f(n') is defined in terms of f(n), and f(0) is defined explicitly. If f has more than one argument, then it is necessary to count down on only one argument. For example:

(i) The sum of m and 0 is m.

(ii) The sum of m and the successor of n is the successor of the sum of m and n.

Here the primitive recursion is on the second argument or n. If n is 0, the definition is explicit and does not refer to the sum of anything. Otherwise, the sum of some number and the successor of n is defined in terms of the sum of that number and n.

The fact that primitive recursive definition always defines a total function is derived from the fact that counting downward always arrives at zero after finitely many operations.

The definitions of plus and times given in §2.2 are examples of primitive recursive definition. After these, we can make the definitions

$$m^n \leftarrow [n = 0 \rightarrow 1, T \rightarrow m \times m^{n^-}]$$

$$\text{hyperexpt}[m, n] \leftarrow [n = 0 \rightarrow 1, T \rightarrow m^{\text{hyperexpt}[m, n^-]}]$$

Hyperexpt[5, 3], for example, is 5^{5^5}.

An example of an arithmetic function that is not primitive recursive is Ackerman's function. It grows faster than any primitive recursive function.

Ackerman's function is a function of three arguments, p, m and n. If p is 0, then it adds m and n; if p is 1, then it multiplies them; if p is 2, then it computes m^n; if p is 3, then it hyperexponentiates, etc.

```
ack[p, m, n] ← [p = 0 → m+n, n = 0 → [p = 1 → 0, T → 1],
                T → ack[p⁻, m, ack[p, m, n⁻]]]
```

Ackerman's function belongs to the class of double recursive functions. There is a transcendental hierarchy of recursion schemas of which primitive and double recursions are merely the first two steps.

The concept of primitive recursion can be applied to definitions of s-expressions as well as numbers. The idea here is to count downward by taking car and cdr. In a primitive recursive definition on s-expressions, the function must be defined explicitly for atomic arguments, and otherwise defined in terms of the function applied to car and/or cdr of its argument. A function of more than one argument must follow this scheme for one only argument.

The function subst is a typical example of primitive recursion. Almost every LISP function we have defined so far except for apply and its subsidiaries is also primitive recursive. Even proofcheck and propeval are primitive recursive, although it may take some rearranging of the definitions to realize this.

§10.3 Other Arithmetics

We use the term "arithmetic" to mean a formal mathematical system consisting of expressions that can be written in some finite alphabet, and subject to a grammatical description. This is somewhat related to what a programmer would call a "data type". S-expressions, integers, arrays, and even floating point numbers can be considered arithmetics, but real numbers, or set theory cannot, because the theory is not about entities each of which has a standard description in some notation. Arithmetics always have countable domains.

The following question are important to an examination of any arithmetic:

1. Is there a syntactic description of the domain of objects?
2. Is there a set of basic functions and predicates such that all

computable functions and predicates on the domain are recursive in terms of the basic ones?

3. Is there an induction principle which applies to the domain?

4. Is there a primitive recursion schema on the domain?

5. Is there an axiomatization of the domain?

As an example, we examine the integers using this syllabus. We assume that the natural numbers have already been examined.

1. An integer is either a natural number, or a natural number other than zero preceded by a minus sign.

2. All computable functions can be defined using the language of recursive functions starting only from successor, predecessor and equality. (Equality may be considered as given prior to any particular arithmetic because it is a "logical" notion.) The predecessor is essential here, and cannot be defined from successor as it can be for the natural numbers. At this point, you might try to define addition, subtraction, multiplication, the ordering relations, the predicate positive[n], and the absolute value of n.

3. There are several useful induction principles, all of which are equivalent. (i) If a property is true of 0 and inherited under successor and predecessor, then it is true for all integers. (ii) If a property is true for 0, and inherited under successor and negation, then it is true for all integers. Any combination of a basis step and an induction step that covers all integers is a valid induction principle.

4. The most obvious primitive recursion schema is to define a function explicitly for zero, and then to define it for positive cases in terms of the function of the predecessor of the argument, and for negative cases in terms of the function of the successor of the argument. This means counting up or down, but always toward zero.

5. The equivalent of Peano's postulates seems to be: (i) Zero is an integer. (ii) The successor and predecessor of an integer are integers. (iii) The successor of the predecessor of an integer and the predecessor of the successor of an integer are both equal to that integer. (iv) Zero is not positive. (v) The successor of zero is positive. (vi) The successor of a positive number is positive. (vii) An induction principle such as 3(i) above.

Without (iv) thru (vi), we could be describing a finite set of objects

arranged in a circular chain. But these axioms specify that 0 is not positive, while $0'$, $0''$, etc., are. So 0 cannot belong to this sequence.

[McCarthy] considers methods of defining arithmetics from given base sets using as basic operations "disjoint union" and "cross product" on sets. He shows how the defining equation for an arithmetic answer questions 1 and 2 of our syllabus. This method could easily be extended to provide answers for the rest of the questions also.

One's not half of two, it's two are halves of one:

e. e. cummings

CHAPTER ELEVEN

FORMAL ARITHMETIC

Preview of Chapter Eleven

The arithmetic of numbers and s-expressions discussed in Chapter Ten is formalized into a system which consists of a theory, plus a set of rules for extending the theory by means of definitions and primitive recursive schemas. A sample of the development of the theory then follows.

§11.1 Multi-Type Logic

The use of types in first order logic is a convenient abbreviation, and not a new theoretical concept. Formal arithmetic is a theory about s-expressions, and about numbers which are a special type of s-expression. We adopt the convention that variables beginning with the letters m, n, p and q are to range over numbers, while variables beginning with the letters r thru z are to range over s-expressions. We have already been using these conventions throughout this book. Variables beginning with the letters a thru k are reserved for future use.

When writing formal schemas, we shall let the Greek letters ξ(xi) and ζ(zeta) stand for s-expression variables, and η(eta) and ν(nu) stand for numeric variables.

A formula having the from $\forall\eta(\alpha)$ is an abbreviation for $\forall\xi(\text{num}[\xi] \supset \alpha(\xi/\eta))$, and a formula having the from $\exists\eta(\alpha)$ is an abbreviation for $\exists\xi(\text{num}[\xi] \wedge \alpha(\xi/\eta))$, where ξ is a new variable. An open formula having numeric variables is equivalent to its closure. Everything we need to know about the use of typed variables follows from these facts. If we simply keep in mind

the intended interpretation, we shall not go wrong.

Having assigned types to variables, it then becomes reasonable to assign types to function names, predicate names, and terms in some cases. If a function has n arguments, then it has n argument types and a value type. If a predicate has m arguments, then it has m argument types. (Its value type is "truth value" or π.)

For the purpose of first order theory of arithmetic, we consider the following functions and predicates to be basic, and assign types to them as follows:

equal:	$S \times S \to \pi$	successor:	$N \to N$
atom:	$S \to \pi$	cons:	$S \times S \to S$
name:	$S \to \pi$	enum:	$N \to S$
num:	$S \to \pi$		

We have now created a very precise situation in which each of these is a total function or predicate on its intended domain. This will be quite useful in presenting the theory that follows.

We now proceed to assign types to terms. If a term has a type according to these rules, it will be called a well-typed term. But not all terms will be well typed, and we do not intend to exclude terms that are not well typed from consideration. The type of a variable has been given. Variables that do not begin with the letter m, n, p, q or r thru z are not typed for the present. The type of a number is numeric, and the type of any other object is s-expression. If $\varphi[\tau_1, \ldots, \tau_n]$ is a term such that for each i, if the i-th argument type of φ is numeric, then τ_i is numeric, and if the i-th argument type of φ is s-expression, then the type of τ_i is either s-expression or numeric, then the entire term is well typed, and its type is the value type of φ. Otherwise, the term is not well typed. We can also define atomic formulas to be well typed in the same manner.

If we were working with more than these two types, the same principle would apply. Some types are sub-types of others in the sense that all numbers are s-expressions. That being the case, the i-th argument term of such a term should be either the i-th argument type of the main function of the term, or a sub-type of that type.

These conventions allow us to say that a term such as length[x]+1 is a numeric term because length is numeric valued. A term such as car[x]+cadr[x] is not well typed, but it might still be meaningful, depending on whether or not x is a list of numbers.

The conventions on typed variables affect the idea of substitution. Q1 and Q2 need to be modified. The following rules are valid for substitution on a numeric variable:

Q1a: $\forall\eta(\alpha) \supset \text{num}[\tau] \supset \alpha(\tau/\eta)$

Q1b: $\forall\eta(\alpha) \supset \alpha(\tau/\eta)$ where τ is numeric

Q2a: $\alpha(\tau/\eta) \supset \text{num}[\tau] \supset \exists\eta(\alpha)$

Q2b: $\alpha(\tau/\eta) \supset \exists\eta(\alpha)$ where τ is numeric

Examples:

Q1a: $\forall n(n' > 0) \supset \text{num}[\text{car}[x]] \supset \text{car}[x]' > 0$

Q1b: $\forall n(n' > 0) \supset 3' > 0$

The definition schemas X, F, K, PF and PK of §9.1 get modified appropriately. We shall examine the situation for Rule F; the rest are similar.

Suppose that we have deduced the formula $\exists_1\eta(\alpha)$. There are two abbreviations in use here, and just as a reminder, we write this formula in its expanded form.

$$\exists\xi(\text{num}[\xi] \wedge \alpha(\xi/\eta) \wedge \forall\zeta((\text{num}[\zeta] \wedge \alpha(\zeta/\eta)) \supset \xi = \zeta))$$

Let the formula α have only η, ξ_1 thru ξ_n, and ν_1 thru ν_m free. Let φ be a new name. Then we can write $\alpha(\varphi[\xi_1, \ldots, \xi_n, \nu_1, \ldots, \nu_m]/\eta)$. The function φ will have a numeric value type because η is a numeric variable, and will have n s-expression arguments followed by m numeric arguments. There is, of course, no reason to list them in this order, but whatever order is used in the term $\varphi[\ldots]$ will determine the argument type description of φ once and for all.

§11.2 Axioms for the Theory of Arithmetic

The axioms are listed in groups with some discussion when necessary.

Group A: The theory of equality, E_L.

This group includes the three equivalence axioms for "=", and an axiom for every function and predicate name that will ever be introduced into the theory. (See definition 6.11.)

Group B: Peano Arithmetic

B1 $\quad num[n']$

B2 $\quad n' \neq 0$

B3 $\quad m' = n' \supset m = n$

B4 $\quad \alpha(0/\eta) \supset \forall\eta(\alpha \supset \alpha(\eta'/\eta)) \supset \forall\eta(\alpha)$

These axioms correspond to Peano's postulates 2 thru 5. For postulate 1, see the computation schema, Group G.

Schema C: Primitive Recursion on the Natural Numbers

$$\left.\begin{aligned} \varphi[\xi_1, \ldots, \xi_n, 0] &= \tau_1 \\ \varphi[\xi_1, \ldots, \xi_n, \eta'] &= \tau_2 \end{aligned}\right\}$$

where (i) φ is a new name, (ii) τ_1 has no occurrences of φ, and no variables other than the ξ_i, (iii) every occurrence of φ in τ_2 is of the form $\varphi[\ldots, \eta]$, and τ_2 has no variables other than the ξ_i and η, and (iv) τ_1 and τ_2 are well typed. Some of the ξ_i may be of numeric type, and the argument η does not have to be placed last.

The function φ defined in this schema will be of numeric value type if both τ_1 and τ_2 are of numeric type, and will have s-expression value type if one or both of the τ_i are s-expression typed. The argument types of φ come from the types of the ξ_i, and the type of η which is numeric.

The primitive recursion schemas for "+" and "x" are part of the basic theory. They are:

$$\left.\begin{aligned} m + 0 &= m \\ m + n' &= (m + n)' \end{aligned}\right\}$$

and

$$\left.\begin{array}{l} m \times 0 = 0 \\ m \times n' = m + (m \times n) \end{array}\right\}$$

Group D: S-Expression Arithmetic

D1: $\neg atom[cons[x, y]]$

D2: $cons[w, x] = cons[y, z] \supset (w = y \wedge x = z)$

D3: $\forall \xi(atom[\xi] \supset \alpha) \supset \forall \xi \forall \zeta(\alpha \supset \alpha(\zeta/\xi) \supset \alpha(cons[\xi, \zeta]/\xi)) \supset \forall \xi(\alpha)$

Schema E: Primitive Recursion on the S-Expressions

$$\left.\begin{array}{l} atom[\zeta] \supset \varphi[\xi_1, \ldots, \xi_n, \zeta] = \tau_1 \\ \varphi[\xi_1, \ldots, \xi_n, cons[\zeta_1, \zeta_2]] = \tau_2 \end{array}\right\}$$

where (i) φ is a new name, (ii) τ_1 has no occurrences of φ, and no variables other than the ξ_i and ζ, (iii) every occurrence of φ in τ_2 is either $\varphi[\ldots, \zeta_1]$ or $\varphi[\ldots, \zeta_2]$, and τ_2 has no variables other than the ξ_i, ζ_1 and ζ_2, and (iv) τ_1 and τ_2 are well typed.

The comment about the type description of φ made for Schema C holds for Schema E, except that the recursion variable here is always of s-expression type.

Group F: Atoms

F1 $name[x] \supset atom[x]$

F2 $num[x] \supset atom[x]$

F3 $atom[x] \supset (name[x] \equiv \neg num[x])$

F4 $name[enum[n]]$

F5 $name[x] \supset \exists_1 n(enum[n] = x)$

Group G: Computation Schema

All true ground literals formed from the basic functions listed in §11.1, and the functions predecessor, plus, times, car, cdr, and their compositions.

This schema is for the purpose of saving us time, and to enable us to make free use of numbers and s-expressions. Without this schema, the theory would be about the numerals 0, $0'$, $0''$, etc., but not about 1, 2, 3, etc. All such literals can be evaluated rapidly by a computer program.

Examples:

$2 + 2 = 4$	cons[A, (B C)] = (A B C)
$2 + 2 \neq 5$	num[cadr[(2 3)]]
¬atom[(A B C)]	¬num[A]

Group H: Embedding

If the theory of s-expressions is embedded in a larger theory in which there are things that are not s-expressions, then we need a predicate sexpr[a] having universal scope and true for s-expressions only. (The variable "a" is not of s-expression type.) We need to add sexpr to the computation schema, and we need two other axioms, namely: sexpr[cons[x, y]], where x and y are s-expression variables, and atom[a] ⊃ sexpr[a]. This situation presents itself when we consider a second order theory in which there are sets of s-expressions.

In addition to these axioms, we need definition schemas. In §9.1, we defined schemas X, F, K, PF and PK. Schema X is really a special case of schema K in which k = 1, and γ_1 is T (true). These form a part of the theory of arithmetic, with suitable allowances being made for types.

Definitions made with quantifiers do not, in general, define functions that are computable. To define functions by explicit schemas that always result in computable functions, we must introduce as special cases of F, K, PF and PK the rules CF, CK, CPF and CPK. These have the same schemas as F, K, PF and PK, except that no quantifiers are permitted in any of the formulas of these "computable" schemas. For example, CF is the rule that permits $\alpha(\varphi[\xi_1, \ldots, \xi_n]/\zeta)$ after having deduced $\exists_1 \zeta(\alpha)$ where α has no quantifiers.

We can now say something about each function and predicate name

defined in the theory of arithmetic by examining its history of antecedent definitions. When we do this, we find that some functions have been totally defined, while others have been partially defined. Some are computable from the definitions, and some are not. These two combine in all four ways. For example, a function may be only partially defined, but the definition gives an effective method for deciding whether it is defined and computing it for those cases in which it is defined.

We have defined eight definition schemas, not counting schema X which is a special case of schema K. The way in which these schemas preserve computability and totality is summarized as follows:

Schema:	F	K	PF	PK	CF	CK	CPF	CPK
Preserves totality:	yes	yes	no	no	yes	yes	no	no
Preserves computability:	no	no	no	no	yes	yes	yes	yes

Definition 11.1

A basic function (for the first order theory of arithmetic, not for computability) is equal(=), successor($'$), cons, atom, num, name or enum.

A primitive recursive function is a function that may have the primitive recursion schemas, and CK in its history of definition, but no other definition schemas.

A total function has only the primitive recursion schemas, and the definition schemas F and K in its history. (CF and CK are special cases of these.)

A computable partial function has only the primitive recursion schemas, and CPF and CPK in its history. (CF and CK are special cases of these.) The special quality of these functions is that it is possible to compute the domains of definition, and to compute the values for specific arguments within these domains.

A total computable function has only the primitive recursion schemas, CF and CK in its history of definition.

It is evident that the primitive recursive functions are total computable functions by this classification schema.

The language used in definition 11.1 is a bit sloppy. When we used the word "function", what we really meant was "function name or predicate name". What we have just done is to introduce a classification schema for the names introduced into the theory of arithmetic by the various definitional schemas. The fact that a given name is classified as "computable" does indeed mean that it corresponds to a computable function, but a function name not classified as "computable" may also correspond to a computable function, although the method used to define it does not of itself provide a computational procedure.

A name classified as "computable" but not "total" has the peculiarity that there is an effective means of deciding whether or not it is defined for a given set of arguments, and then there is an effective means of computing the value when it is defined. This is more than can be said for partial recursive functions in general. This special cateogry is useful for predecessor, subtraction, division, car and cdr, functions defined only on lists, functions defined only on lists of numbers, etc.

We now have a developing system with many built-in conveniences for making definitions. We have been calling it a "theory", but it is not strictly speaking a theory, but rather a theory, and a set of rules for creating extensions. Once a certain extension is created, it restricts the use of a certain name which then cannot be used to create some other extension.

The system we have just described has a model which is the domain of s-expressions, with the basic functions having their standard interpretation. Each extension has a corresponding enlargement of the model. If the extension is total, then a uniquely defined function or predicate is added to the model. If the definition is not total, then there may not be a unique enlargement of the model, but there will be at least one enlargement.

As was already mentioned, the total definition schemas are conservative, and in fact eliminable, but the primitive recursion schemas are not so. This raises the question as to whether there is some language with a finite vocabulary that is adequate to describe the theory. If we restrict ourselves to the numeric part of the theory, then Gödel answered this question by showing that the only instances of the primitive recursion schema needed are those for "+" and "x", and that once these formulas have been given, all

other primitive recursive functions can be defined using only rules F and K. So formal number theory is presented in the language $\{=, 0, ', +, \times\}$. The proof of this fact involves coding finite sequences of numbers into single numbers, and then showing that there is a function definable from + and × that can extract the i-th component of such a sequence.

§11.3 Development of the Theory

The purpose of this section is to provide some concrete examples of the system specified in §11.2. The first part is about number theory, and the second part is about s-expression theory.

In the development that follows, many shortcuts will be used to make the formal deductions less tedious. We shall assume various properties of propositional logic, quantifiers, variables, and equality, including symmetry, transitivity and replacement. However, every detail involving the properties of arithmetic will be written out in full, i.e., all references to the axiom system we have just presented will be completely explicit. The distinction between properties of logic and equality on the one hand, and properties of arithmetic on the other can be made very precise.

We start out by repeating the following definitions:

D1: $+: N \times N \to N$ $\quad \left.\begin{array}{l} m + 0 = m \\ m + n' = (m + n)' \end{array}\right\}$ Schema C

D2: $\times: N \times N \to N$ $\quad \left.\begin{array}{l} m \times 0 = 0 \\ m \times n' = m + (m \times n) \end{array}\right\}$ Schema C

Th1: $0 + m = m$

The proof from almost identical axioms has already been given.

Th2: $m' + n = (m + n)'$

	1.	$m' + 0 = m'$	Instance of D1
	2.	$m + 0 = m$	D1
	3.	$m' + 0 = (m + 0)'$	Replacement 1, 2
(4)	4.	$m' + n = (m + n)'$	Assume
	5.	$m' + n' = (m' + n)'$	Instance of D1
(4)	6.	$m' + n' = (m + n)''$	Replacement 4, 5

(4)	7.	$m' + n' = (m + n')'$	Replacement 6, D1
	8.	$m' + n = (m' + n)' \supset m' + n' = (m + n')'$	Discharge 4, 7
	9.	$(m' + 0 = m') \supset \forall n(m' + n = (m+n)' \supset m' + n' = (m + n')) \supset \forall n(m' + n = (m + n)')$	Instance of B4
	10.	$m' + n = (m + n)'$	From 1, 8, 9

Problem 30

1. Th3: $m + n = n + m$
2. Th4: $0 \times m = m$
3. Demonstrate $m \neq 0 \supset \exists_1 n(n' = m)$. Then predecessor can be defined by $m \neq 0 \supset m^{-'} = m$.

From here, one may proceed to prove the commutivity of multiplication, the associativity of addition and multiplication, the distributive laws, and then move into the area of primes and factoring.

Because this is a first order theory, one cannot talk about sets of numbers, but only individual numbers. For example, one cannot state directly, let alone prove, that every number can be factored uniquely except for the order of the factors, into prime factors. However, one can state this indirectly because the set of factors of any number is always a finite set. It is possible to state, and to prove, that for every number there is a list of primes, unique except for order, whose product is that number.

Every non-empty set of numbers has a least member, but this cannot even be stated indirectly so as to apply to all infinite sets of numbers. A related concept is to say that any predicate satisfied by at least one number has a least number that satisfies it. If ψ is any numeric predicate, then we can prove as a theorem $\exists n(\psi[n]) \supset \exists n(\psi[n] \wedge \forall m(\psi[m] \supset m \geq n))$. However, the statement "This theorem schema is true for any ψ," lies outside the scope of first order logic because it informally quantifies on a predicate, whereas first order logic quantifies on variables only.

Second order logic quantifies over first order predicates. However, there is no effective method of deduction for second order logic which is semantically complete in the sense that if $T \models \alpha$, then $T \vdash \alpha$. An alternative to

second order logic is to stay with first order logic, and to develop a second order theory whose intended model has a domain of two types, numbers and sets of numbers. (We do this in Chapter Fifteen, only for s-expressions and sets of s-expressions.) But there is no escaping the essential incompleteness, which in the latter case presents itself as an incomplete theory rather than as an incomplete logic. Still, second order number theory is more powerful than first order number theory. In fact, second, third and even fourth order theories are in constant use by mathematicians, and their formalization is a necessity that must be faced. For example, one may speak of real numbers, functions of real numbers, and families of functions of real numbers, the latter being a third order concept. Such investigations lead us to the study of axiomatic set theory.

In the development of first order s-expression theory, we find it convenient to introduce the infix "*" to represent cons. We shall have it associate from right to left, so that A*B*NIL = A*(B*NIL) = (A B). The function append which is familiar to LISP programs will be represented by a colon(:). Its primitive recursive definition is:

D5: $(:)\colon S \times S \to S$ $\left.\begin{array}{l} \text{atom}[x] \supset x:z = z \\ (x*y):z = x*(y:z) \end{array}\right\}$ Schema E

Th6: $\text{atom}[x] \supset x:[y:z] = [x:y]:z$

(1)	1.	$\text{atom}[x]$	Assume
	2.	$\text{atom}[x] \supset x:[y:z] = y:z$	Instance of D5
(1)	3.	$x:[y:z] = y:z$	Modus ponens 1, 3
(1)	4.	$x:y = y$	Modus ponens 1, D5
(1)	5.	$x:[y:z] = [x:y]:z$	Replacement 4, 3
	6.	$\text{atom}[x] \supset x:[y:z] = [x:y]:z$	Discharge 1, 5

Problem 31

1. Th7: $x:[y:z] = [x:y]:z$. Hint: It is important to choose the correct induction instance. If we induct on x, then Th6 is the basis step. Show that if $u:[y:z] = [u:y]:z$ and $v:[y:z] = [v:y]:z$, then $[u*v]:[y:z] = [[u*v]:y]:z$.

2. Define the partial computable functions car and cdr.

The theory of s-expressions has no standard curriculum, unlike number theory. At this point, one might formalize notions of permutation, combination, rotation, etc., or one might define sublis, and develop formally the theory of substitution presented in §9.3.

CHAPTER TWELVE

RECURSION AND DEDUCTION

Preview of Chapter Twelve

Starting with this chapter, we unite two subjects which have been developed more or less independently until now. In Chapters Two and Five we developed the language of recursive functions, which is a language for describing formal computations on s-expressions. The notion of recursion is shown to be absolute, and completely independent of this method of defining it, because, by Turing's and Church's theses, it is identified with effectively computable.

In Chapters Six thru Eleven, we have developed the subject of first order logic as a language for making assertions, and proving consequences of these assertions, and then particularized this to the theory of s-expressions. The only relations between deduction and recursion that we have established so far are that deduction is subject to mechanical verification, i. e., "proofcheck" is recursive, and that certain types of definition within first order arithmetic provide recursive descriptions.

There are two important questions about the relation between deduction and recursion that we consider in the rest of this book. The first is the problem of representing, and discussing recursive functions or effective procedures within first order logic. The second is the problem of reducing deduction to computation in routine cases. In this chapter, we begin with the first of these questions by "representing" recursive functions in arithmetic.

§12.1 Expressibility and Representability

In this chapter, let us consider the theory of arithmetic as consisting

only of those function and predicate names that we classified as "total" in Chapter Eleven. These are the names that necessarily lead to unique model extensions because of their definitional history. Since the standard model for formulas in this language is unique, we can speak of a formula as being either "true" or "false" according to whether or not this model satisfies it. There is no middle ground. (We are not claiming that there is an effective procedure for deciding which formulas are true and false, only that each one must be either true or false.)

Definition 12.1

If $\tilde{\psi}$ is a predicate in the sense of being a mapping from S^m into π, then it is an arithmetic predicate if there is an m-ary predicate name ψ that can be defined in the theory of arithmetic such that for any s-expressions σ_1 thru σ_m, $\tilde{\psi}(\sigma_1, \ldots, \sigma_m)$ is true if and only if $\psi[\sigma_1, \ldots, \sigma_m]$ is true.

For any formula α, we write $A \vdash \alpha$ to mean that there is a deduction of α from the theory of arithmetic. A is understood to mean the theory consisting of all the axioms and axiom schemas discussed in Chapter Eleven, and the definitions and primitive recursion schemas necessary to define all the function and predicate names in α. This is not the most satisfactory notation, because it does not fully specify A. But it will not lead us into error if we are aware of this.

Definition 12.2

The predicate $\tilde{\psi}$ is expressible if it is possible to define a predicate name ψ in arithmetic such that for any s-expression σ_1 thru σ_m, if $\tilde{\psi}(\sigma_1, \ldots, \sigma_m)$ is true than $A \vdash \psi[\sigma_1, \ldots, \sigma_m]$, and if $\tilde{\psi}(\sigma_1, \ldots, \sigma_m)$ is false, then $A \vdash \neg\psi[\sigma_1, \ldots, \sigma_m]$.

The n-ary function $\tilde{\varphi}$ is representable if it is possible to define a function name φ in arithmetic such that for any s-expressions σ_1 thru σ_{n+1}, if $\tilde{\varphi}(\sigma_1, \ldots, \sigma_n) = \sigma_{n+1}$, then $A \vdash \varphi[\sigma_1, \ldots, \sigma_n] = \sigma_{n+1}$.

The notions of arithmetic, expressible and representable, may also be relativized to functions and predicates having numeric arguments or values.

Corollary 12.3

All expressible predicates are arithmetic.

All expressible predicates and representable functions are recursive.

The effectiveness of the theory A, and the fact that the theorems of an effective theory are recursively enumerable are a sufficient proof of effectiveness. Keep in mind that for a procedure to define a total function is not the same thing as our being able to prove that it defines a total function.

§12.2 Primitive Recursion

Corresponding to the definition of primitive recursive functions in the system A, there is a subset of the language of recursive function definitions that leads to primitive recursion. We list the corresponding schemas side by side:

Schema C:

$$\left.\begin{aligned}&\varphi[\xi_1, \ldots, \xi_n, 0] = \tau_1\\&\varphi[\xi_1, \ldots, \xi_n, \eta'] = \tau_2\end{aligned}\right\} \qquad \varphi[\xi_1, \ldots, \xi_n, \eta] \leftarrow [\eta = 0 \rightarrow \tau_1, T \rightarrow \tau_2(\eta^- / \eta)]$$

Schema E:

$$\left.\begin{aligned}&\text{atom}[\zeta] \supset \varphi[\xi_1, \ldots, \xi, \zeta] = \tau_1\\&\varphi[\xi_1, \ldots, \xi_n, \zeta_1 * \xi_2] = \tau_2\end{aligned}\right\} \qquad \begin{aligned}\varphi[\xi_1, \ldots, \xi_n, \zeta] \leftarrow [&\text{atom}[\zeta] \rightarrow \tau_1,\\ &T \rightarrow \tau_2(\text{car}[\zeta]/\zeta_1, \text{cdr}[\zeta]/\zeta_2)]\end{aligned}$$

Rule CK:

$$\left.\begin{aligned}&\gamma_1 \supset \psi[\xi_1, \ldots, \xi_m] \equiv \alpha_1\\&\ldots\\&\gamma_k \supset \psi[\xi_1, \ldots, \xi_m] \equiv \alpha_k\end{aligned}\right\} \qquad \psi[\xi_1, \ldots, \xi_m] \leftarrow [\gamma_1 \rightarrow \alpha_1, \ldots, \gamma_k \rightarrow \alpha_k]$$

and

$$\left.\begin{aligned}&\gamma_1 \supset \varphi[\xi_1, \ldots, \xi_n] = \tau_1\\&\ldots\\&\gamma_k \supset \varphi[\xi_1, \ldots, \xi_n] = \tau_k\end{aligned}\right\} \qquad \varphi[\xi_1, \ldots, \xi_n] \leftarrow [\gamma_1 \rightarrow \tau_1, \ldots, \gamma_k \rightarrow \tau_k]$$

The restrictions for Schemas C and E, and Rule CK are given in §11.2 and §9.1, respectively. In particular, there are no quantifiers anywhere in these schemas, and it must be provable that exactly one of the γ_i must hold. The definitions in the right column define the subclass of the recursive functions that are the primitive recursive functions.

Theorem 12.4

All primitive recursive predicates are expressible, and all primitive recursive functions are representable.

Proof: By nested induction. The outer induction is on the length of the definition history. The basis of the induction is that the basic functions are representable. This follows from the computation schema, Group G. The induction step is to show that for each schema, if all the preceding definitions are representable, then it is representable also.

For Rule CK, the fact that any ground instance of the schema can be proven or refuted follows from the induction hypothesis, and the replacement of equal terms and formulas, since there are no variables or quantifiers to deal with. For the schemas C and E, there is also an inner induction needed. The preceding method will work only for the case that $\eta = 0$ in Schema C, or atom[ζ] in the case of Schema E. But this is the basis for an induction on the natural numbers or the s-expressions whereby if $\varphi[\ldots,\eta]$ can be represented, then $\varphi[\ldots,\eta']$ can be represented, or if $\varphi[\ldots,\zeta_1]$ and $\varphi[\ldots,\zeta_2]$ can be represented, then $\varphi[\ldots,\zeta_1{}^*\zeta_2]$ can be represented.

§12.3 The Incompleteness of Arithmetic

We are now able to demonstrate that the theory of arithmetic is incomplete. This is not in itself surprising, because we have not investigated the axioms presented in §11.2 very seriously, and there is no reason to believe that they are sufficient to prove everything that we would like to be able to prove about arithmetic. However, the incompleteness theorems will apply to any attempt to strengthen these axioms also. We prove incompleteness in three different ways.

Lemma 12.6

There is a primitive recursive predicate aproof[x, y] which is true if and only if x is the s-expression translation of a formula α in the system A, and y is the s-expression translation of a deduction that proves α in A.

We do not offer a formal proof of lemma 12.6 which would have to begin by writing out such a proofchecker. By now, you should be aware that if the amount of "work" that is involved in evaluating a recursive function is bounded exponentially by the size of its argument, then it will be primitive recursive.

In order to be able to assert meaningfully that $A \vdash \alpha$, we must make sure that the names used in α have the meaning that we intend. We shall say that a sequence of lines as in a deduction determines α if every name appearing in α except for the basic names is totally defined in this sequence. Suppose λ is the s-expression translation of a sequence that determines α. Then if there is some s-expression μ such that aproof[λ:μ, α*] is true, then we can reasonably assert that α has been proven. (The symbol ":" means append, and μ is the continuation of a deduction that begins with λ. α* is the s-expression translation of α.)

The predicate aproof, or something similar to it, is what Gödel called "the arithmetization of metamathematics", meaning that we can interpret an arithmetic fact, namely that the predicate aproof is true for certain arguments, as an assertion about the provability of some formula.

The key to Gödel's incompleteness theorem is that the arithmetization of metamathematics allows us to create a sentence which asserts "I am not provable in arithmetic." If this formula is provable in arithmetic, then it is not true, and so arithmetic is capable of proving things that are false. If the formula is refutable in arithmetic, then if arithmetic is true, it is provable, and so again we have deduced something false. So if arithmetic is true in the sense that the standard model satisfies it, then it is incomplete, and this sentence is true but neither provable nor refutable.

Theorem 12.7 (Gödel's Incompleteness Theorem)

The system A is incomplete, in the sense that there is a formula β

such that neither $A \vdash \beta$, nor $A \vdash \neg\beta$.

Proof: The primitive recursive function subquote is defined as follows:

$$\text{subquote}[x, y, z] \leftarrow [\text{atom}[z] \rightarrow [y = z \rightarrow \text{list}[\text{QUOTE}, x], T \rightarrow z], \text{car}[z] = \text{QUOTE} \rightarrow z, T \rightarrow \text{subquote}[x, y, \text{car}[z]]^{*}\text{subquote}[x, y, \text{cdr}[z]]]$$

Let λ be an s-expression translation of a determining sequence for aproof, append(:), and subquote. Consider the formula:

$$\alpha: \quad \neg\exists x(\text{aproof}[\lambda{:}x, \text{subquote}[y, Y, y]])$$

Its translation is the s-expression:

$$\alpha^{*}: \quad \text{(NOT (EXISTS X (APROOF (APPEND (QUOTE } \lambda\text{) X) (SUBQUOTE Y (QUOTE Y) Y))))}$$

α^{*} is a genuine s-expression, and the only thing that prevents our writing it out in full is that we have not written a program for aproof, and then converted it into a sequence of primitive recursive definitions in A. This would make the s-expression λ perhaps two or three written pages in length.

Now consider the formula:

$$\beta: \quad \neg\exists x(\text{aproof}[\lambda{:}x, \text{subquote}[\alpha^{*}, Y, \alpha^{*}])$$

Its translation is the s-expression:

$$\beta^{*}: \quad \text{(NOT (EXISTS X (APROOF (APPEND (QUOTE } \lambda\text{) X) (SUBQUOTE (QUOTE } \alpha^{*}\text{) (QUOTE Y) (QUOTE } \alpha^{*}\text{)))))}$$

β is a sentence containing the ground term $\text{subquote}[\alpha^{*}, Y, \alpha^{*}]$. This term can be evaluated using the definition of subquote, and the value turns out to be the s-expression β^{*}. Since subquote is primitive recursive, it is representable, and therefore $A \vdash \text{subquote}[\alpha^{*}, \lambda, \alpha^{*}] = \beta^{*}$. Then by replacement of equal terms, we have:

$$(*) \quad A \vdash \beta \equiv \neg\exists x(\text{aproof}[\lambda{:}x, \beta^{*}])$$

Now suppose that β as determined by λ were provable in A. Then we could write out such a deduction, and code this deduction into an s-expression beginning with λ. Call the tail of this deduction μ. Then since it is a valid

deduction, aproof[λ:μ, β*] would be true. Since aproof is primitive recursive and therefore expressible, this formula would also be provable, and from it and the formula (*) we could deduce $\neg\beta$. Therefore arithmetic would be inconsistent.

Suppose on the other hand that $\neg\beta$ could be proven in A. Then we could prove $\exists x$(aproof[λ:x, β*] from this and (*). Assuming that the standard model satisfies arithmetic, there must be some s-expression μ such that aproof[λ:μ, β*] is true. Translating λ:μ back gives us a deduction of β, so once again A would be inconsistent.

Assuming that A is a consistent system, and that the standard model for the s-expressions satisfies A, then we must conclude that β is neither provable nor refutable from A.

This proof mirrors accurately the construction used by Gödel in his proof which was for the theory of natural numbers in the language {=, 0, ′, +, x}. However, his reasoning about this construction was quite different because he did not assume that arithmetic was necessarily consistent, and since he was restricting himself to finitary mathematics, the concept of a standard model could not be used. What he proved was that either arithmetic is incomplete or else it is either inconsistent or at least ω-inconsistent, which means that there is some formula α such that $A \vdash \exists x(\alpha)$, yet $A \vdash \neg\alpha(0/x)$, $\neg\alpha(0'/x)$, $\neg\alpha(0''/x)$, etc.

At first, one might think that this incompleteness theorem indicates that the theory A is too weak and should have some stronger axioms. For example, we might add β as an additional axiom, since it is true but unprovable from A. It turns out, however, that the incompleteness of arithmetic has nothing to do with this particular choice of a set of axioms. Any true, effective extension of A will also be incomplete.

To show this, let B be any true, effective extension of A. The effectiveness of B means that its axioms must at least be recursively enumerable. From this, it follows that there is a primitive recursive predicate bproof[x, y] which is true if and only if x is a proof of y in the theory B. Bproof is expressible (in A) because it is primitive recursive. It is expressible in B because B is a consistent extension of A, and so the incompleteness proof can be repeated in B, generating a formula undecidable in B.

It is not even necessary to arithmetize deduction in order to show that arithmetic is incomplete. It is sufficient to arithmetize computation. Starting from the definition of apply, we define the function applyk[x, y, p] which has the property that if apply[x, y] = z, then there is a number p_0 such that if $p \geq p_0$, then applyk[x, y, p] = list[z], but if $p < p_0$, then applyk[x, y, p] = NIL. If apply[x, y] is undefined, then for all p, applyk[x, y, p] = NIL. One way to define applyk is to add an extra argument to every subsidiary function of apply. Each time a function is called, this argument gets decremented. If it ever gets down to zero, then the computation is interrupted, and the value is NIL. It is also necessary to modify every function so that all explicitly undefined conditions get checked out, and so that a value of NIL gets referred to the top level of the computation promptly.

Lemma 12.8

Applyk is a primitive recursive function.

Alternate Proof that Arithmetic is Incomplete: If arithmetic were complete, then every arithmetic predicate would be expressible, and hence recursive. We know that the predicates halt and total, defined in Chapter Five, are not recursive. They are, however, arithmetic because they can be defined by:

$$\text{halt}[x, y] \equiv \exists z \exists p(\text{applyk}[x, y, p] = \text{list}[z]) \qquad \text{Rule X}$$

$$\text{total}[x] \equiv \forall y \exists z \exists p(\text{applyk}[x, \text{list}[y], p] = \text{list}[z]) \qquad \text{Rule X}$$

Therefore arithmetic is incomplete.

Problem Set 32

1. Let A be the theory of arithmetic including the definitions of applyk and halt. Show that there is a finite set of axioms T in A such that if τ_1 and τ_2 are ground terms containing no constants other than 0, and no functions other than successor, enum and cons, then if halt[τ_1, τ_2] is true, then $T \vdash \text{halt}[\tau_1, \tau_2]$.

2. Show that first order logic is undecidable. (See corollary 9.18.)

There are still further ways of demonstrating that arithmetic is incomplete, and each one illuminates a different aspect of the problem. Gödel's proof and the proof of Tarski's theorem in Chapter Thirteen are related to Epimenides' paradox. Epimenides was a Cretan philosopher of the fifth century B.C. who pondered the truth of the assertion: "This very sentence that I am now speaking is a lie." Epimenides was dimly remembered by the Apostle Paul who wrote the famous slander: "One of their very number, a prophet of their own said, 'Cretans are always liars, hurtful beasts, idle and lazy gluttons.' " (Epistle to Titus, I, 12)

Another approach to incompleteness has been developed by [Chaitin] starting from what is known as Berry's paradox, which goes something like this: "Consider the smallest number that takes at least one hundred words to describe." If we ignore for the moment the problem of what is a valid "description" of a number, it is evident that some very large numbers can be described in very few words; for example "one billion hyperexponentiated one billion times". Among all the possible descriptions for any number, there must be one or more having the least number of words. So associated with each number is a number which is the word count of its shortest description(s). The smallest number for which this count is at least one hundred is the number that is referred to in the quoted sentence above. Yet that sentence which has less than a hundred words "describes" the number in question. This is the paradox.

Chaitin replaces the ambiguous concept of "shortest description length in English" with the precise notion of "information theoretic complexity". The information theoretic complexity of an expression is the shortest instruction that can be given to a computer that will cause the computer to print out the expression in question. Obviously, the number one billion hyperexponentiated to the one billion is not very complex because a program to generate it is quite trivial. Information theoretic complexity does not consider the amount of time taken by the computer, or the amount of intermediate storage required, unlike the "complexity" of current complexity theory research. One may argue that the definition of information theoretic complexity is arbitrary because it depends on the choice of computer. This is true, but since any universal computer can simulate any other one, the

difference in complexity as measured by one computer and another cannot differ by more than a fixed constant, and this can be kept manageably small if the computers in question are of fairly simple description themselves.

Let us fix the definition of complexity more precisely. The function size[x], the number of characters required to print the s-expression x, is a recursive function. We now define the complexity of an s-expression x as being the least size of any s-expression y such that apply[y, NIL] = x. The complexity of the s-expression which is a list of the first billion prime numbers is evidently quite moderate, because we can easily write a program to generate it, and cast this program in the form of a recursive function of no arguments. The complexity of a list of one billion random numbers would be large, however, somewhat the same order as the size of the list itself. The complexity of any s-expression cannot be more than slightly larger than its own size, because x can be generated by the function (FN () (QUOTE x)).

The function complexity[x] can be defined in arithmetic using Rule F because the following formula is provable.

$$\exists_1 n(\exists y(\text{size}[y] = n \wedge \exists p(\text{applyk}[y, \text{NIL}, p] = \text{list}[x])) \wedge \forall z(\text{size}[z] \geq n \vee \neg\exists p(\text{applyk}[z, \text{NIL}, p] = \text{list}[x])))$$

We are now in a position to formalize Berry's paradox. Let g[n] be a recursive function that enumerates all theorems of arithmetic with applyk, size and complexity defined. G is not all that complex in itself. It must contain the deduction rules for first order logic, the axioms of arithmetic, the definition of applyk, and some enumeration machinery. Consider the first formula in the enumeration g[0], g[1] ... that is of the form complexity[α] > 1,000,000,000 for some s-expression α. If arithmetic is true, α cannot be generated by any program of moderate length, yet we have just described such a method which consists in enumerating the function g until we come to such a formula. This process can easily be formalized into a function of no arguments. The only way out of the contradiction is to assume that no formula of the form complexity[α] > 1,000,000,000 will ever be generated in the sequence g[0], g[1], ... But this sequence contains all the provable formulas of arithmetic, and so the conclusion is that only finitely many formulas of the form complexity[α] > n are provable, and that n is not

much larger than the complexity of the enumeration process. This is a startlingly different way for arithmetic to be incomplete.

Chaitin's article is highly readable, and relates information theoretic complexity to the notion of "random sequence" as well as computability and incompleteness.

It is sometimes claimed that the various incompleteness and undecidability results are not useful to the computer programmer concerned with artificial intelligence or mechanical inference, because all these theorems are based on weird techniques that border on paradox and always involve self-application or diagonalization. One never wants to do those particular things, anyway, in any practical situation. I would argue that, on the contrary, self-application is precisely what one wants to do, because a system of deduction that can examine its own behavior is that much more powerful. Chapter Thirteen is an examination of this very question. By proving incompleteness in three different ways, I hope I have made the point that incompleteness is a result of the richness of logic, rather than indicating its impoverishment.

§12.4 Representability of Recursive Functions

Let φ be an n-ary total recursive function. Let φ^* be the s-expression translation of a sequence of recursive definitions that computes φ. The following formula contains exactly the variables x_1 thru x_n and y free:

$$\begin{aligned}&\exists p(\exists z(\text{applyk}[\varphi^*, \text{list}[x_1, \ldots, x_n], p] = y^* z) \wedge \\ &\qquad \forall m(m < p \supset \text{atom}[\text{applyk}[\varphi^*, \text{list}[x_1, \ldots, x_n], p]])) \vee \\ &\qquad \forall m(\text{atom}[\text{applyk}[\varphi^*, \text{list}[x_1, \ldots, x_n], m]] \wedge y = \text{NIL})\end{aligned}$$

Calling this formula α for the moment, it is possible to prove $\exists_1 y(\alpha)$ within arithmetic. In fact, such a proof is completely independent of the definition of applyk and the s-expression φ^*, and depends only on the principle of any non-empty set of numbers having a least member. Either there is a least p such that $\text{applyk}[\varphi^*, \text{list}[x_1, \ldots, x_n], p]$ is non-atomic, in which case y is car of that value, or else the second part of the disjunct holds and y is NIL. Therefore, we can define the function φ by Rule F, getting $\alpha(\varphi[x_1, \ldots, x_n]/y)$. This happens to be true for any s-expression φ^*. If φ^* defines only a partial

function computationally, then the function φ defined in first order arithmetic is completed by having the value NIL wherever the computation does not produce a value. If φ^* is not a procedure at all, φ will still be a constantly NIL function. But φ will not necessarily be computable.

While there is no process that always tells us whether φ^* computes a total function, in each case where it does, φ will be representable in arithmetic, for if σ_1 thru σ_n are any s-expressions, then for some number p and some s-expression σ_{n+1}, $A \vdash \text{applyk}[\varphi^*, \text{list}[\sigma_1, \ldots, \sigma_n], p] = \text{list}[\sigma_{n+1}]$, and for $m < p$, $A \vdash \text{applyk}[\varphi^*, \text{list}[\sigma_1, \ldots, \sigma_n], m] = \text{NIL}$. Therefore:

Theorem 12.9

All total recursive functions are representable.

CHAPTER THIRTEEN

METAMATHEMATICS

Preview of Chapter Thirteen

Given a formal system of deduction, a metamathematical statement is an assertion about the system, very often about the system as a whole. For example: (i) Arithmetic is consistent. (ii) Arithmetic is incomplete. (iii) The formula β cannot be deduced within the system. (iv) The formula γ cannot be deduced within the system except by a deduction whose length is astronomically long. (v) The name LENGTH defines a function having an s-expression argument and a numerical value. (vi) Every formula of the type $\forall\xi(\alpha) \supset \exists\xi(\alpha)$ is provable. (vii) Replacement of equal terms is a derived rule of inference.

Metamathematical reasoning is the method by which we arrive at statements such as these. It is impossible in any practical sense to do without metamathematical reasoning, and in fact we have used it throughout the book. If we want a practical system of logical inference, it will be necessary to formalize at least part of metamathematical reasoning, and that is the purpose of this chapter. Much of it has to do with formalizing the semantic notion of "truth", just as in Chapter Twelve we formalized the syntactic property of "provability".

§13.1 Truth and Tarski's Theorem

We first define truth as a semantic or model theoretic concept, and then later in the chapter we shall make use of some axioms concerning truth. It is important to proceed in this order because it is only by having a clear

model-theoretic concept that we shall know that our axioms are reasonable.

If α is a formula of arithmetic, all of whose function and predicate names are defined and total, then we shall define truth[α^*] to be true if and only if α is true. Defining neg[x] to be list[NOT, x], either truth[α^*] or truth[neg[α^*]], but not both, must hold for any totally defined arithmetic formula α, because either $S \vdash \alpha$ or $S \vdash \neg\alpha$, where S is the standard model of the s-expressions.

We now make the important point that the predicate "truth" is not an arithmetic predicate. It lies outside of the system A, and if the formula α contains "truth" then α is not an arithmetic formula, and the above discussion does not apply to α at all. If "truth" were an arithmetic predicate, then it would be possible to establish Epimenides' paradox within arithmetic. This is known as Tarski's theorem.

Theorem 13.1 (Tarski's Theorem)

Arithmetic truth is not arithmetic.

Proof: Suppose to the contrary that it were possible to define truth[x] within arithmetic such that if α is any arithmetic formula, then $S \vdash \alpha$ if and only if $S \vdash$ truth[α^*]. Let β be the formula $\neg$truth[subquote[y, Y, y]]. Let γ be the formula $\neg$truth[subquote[β^*, Y, β^*]]. Then $\gamma \equiv \neg$truth[γ^*], so $S \vdash \gamma$ if and only if $S \vdash \neg$truth[γ^*] if and only if $S \vdash$ truth[neg[γ^*]] if and only if $S \vdash \neg\gamma$.

Because $S \vdash \alpha$ if and only if $S \vdash$ truth[α^*] is true only for arithmetic formulas, it becomes necessary to express the predicate "α^* is arithmetic" itself within arithmetic. If we did not have definitions, the problem would be easy. An arithmetic formula would be one whose function and predicate names are only the basic ones. But since we do allow definitions, the problem is administratively more complicated, although not conceptually so.

An administrative function is a function that makes certain system information available within the system. These functions are not charged with the semantics of "truth", and so we may consider them to be ordinary arithmetic functions. They tell us what has been written down in the system so far. The only administrative function that we need now is defn[x]. If x is a name that has been defined by any of the definition rules or primitive

recursion schemas, then defn[x] is a list of the name of the rule, and the lines of the definition itself. For example, defn[PLUS] might be ((SCHEMA C) (EQUAL (PLUS M 0) M) (EQUAL (PLUS M (ADD1 N)) (ADD1 (PLUS M N)))). It is evident that starting with a certain amount of initial knowledge, and the information obtained from defn, the history of any function name can be investigated, and various determinations made, such as that it is total, primitive recursive, etc. In particular, if the history of definition does not include TRUTH, then it is arithmetic. From this, we can define the predicate arith[x] which is true if and only if x is α^* for some well-formed arithmetic formula α. Arith itself is total, arithmetic, and computable.

We now postulate the following formal metamathematical axioms which are justified because they are true, that is, they are satisfied by the model which is the standard model for the s-expressions enlarged (non-conservatively) by interpreting the predicate truth[x] to be true if x is α^* where α is a true arithmetic formula, and false if x is α^* and α is a false arithmetic formula, and leaving truth[x] unspecified for all other x. Notice that none of these axioms make any assertion about truth[x] unless arith[x] is true.

M1: Semantic Completeness and Consistency of Arithmetic

$$arith[x] \supset (truth[x] \equiv \neg truth[neg[x]])$$

M2: Validity of the Axioms of Logic

$$arith[x] \supset taut[x] \supset truth[x]$$
$$arith[x] \supset q1[x] \supset truth[x]$$
$$arith[x] \supset q2[x] \supset truth[x]$$

M3: Validity of the Rules of Inference of Logic

$$arith[y] \supset mp[x, y, z] \supset truth[x] \supset truth[y] \supset truth[z]$$
$$arith[x] \supset q3[x, y] \supset truth[x] \supset truth[y]$$
$$arith[x] \supset q4[x, y] \supset truth[x] \supset truth[y]$$

M4: Truth of the Axioms of Arithmetic

$$arith[x] \supset ax[x] \supset truth[x]$$

where ax[x] is true if x is α^* for some formula α which is an axiom or instance of an axiom schema in Group A, B, D, F or G.

$\text{arith}[x] \supset \text{ninduct}[x] \supset \text{truth}[x]$

$\text{arith}[x] \supset \text{sinduct}[x] \supset \text{truth}[x]$

where those predicates assert that x is an instance of B4 or D3, respectively.

M5: Truth of Formulas Introduced as Definitions

$\text{arith}[x] \supset \exists y(\text{member}[x, \text{cdr}[\text{defn}[y]]]) \supset \text{truth}[x]$

If x is α^* for a formula α introduced by some definition or primitive recursion schema, then it is a member of defn of the name that was defined. (Car of this list is the name of the schema.)

M6: Truth of the Predicate Truth

$\text{arith}[\alpha^*] \supset (\alpha \equiv \text{truth}[\alpha^*])$

This is an axiom schema which cannot be represented in the present system as a single axiom.

Schema M6 is at the very center of the notion of formal metamathematics. It is bidirectional. First it allows that if we can assert some formula α then we can assert that α is true. In the other direction, it allows us to pass from the assertion that α is true to α itself.

§13.2 Metamathematical Deduction

Let us modify the primitive recursive function aproof[x, y] slightly by requiring that any definitions occurring in x be consistent with the system A. We can now do this by using defn. This allows us to dispense with the nuisance of the determining sequence λ used in Chapter Twelve. It is now possible to prove by induction on the length of the deduction y:

$$(**) \quad \text{arith}[x] \supset \exists y(\text{aproof}[y, x]) \supset \text{truth}[x]$$

The formula β of theorem 12.7 cannot be deduced within the system A, but at the time that we proved this, we argued metamathematically that β was true. We can formalize this argument as follows:

(1)	1.	$\neg\beta$	Assume
	2.	$\beta \equiv \neg\exists x(\text{aproof}[x, \beta^*])$	This is provable in A.

	3.	arith[β*]	Also provable in A.
(1)	4.	$\exists$x(aproof[x, β*])	Prop 1, 2
(1)	5.	truth[β*]	Instance of **, 3 and 4.
(1)	6.	β	M6 5
	7.	$\neg\beta \supset \beta$	Discharge 6, 1
	8.	β	Prop 7

This shows that formal metamathematics allows us to prove some formulas of A that are not provable in A. However, it does not allow us to complete A. Halt and total are still not recursive, since the notion of recursiveness is absolute and therefore not dependent on one's choice of an axiom system. No formula of the sort complexity[α] > 1,000,000,000 can be proven in formal metamathematics, or in any truthful system whose axioms can be enumerated by a function of feasible complexity.

The following extreme case shows that there are formulas having proofs of unfeasible length in arithmetic that have feasible proofs in metamathematics. Consider the formulas:

α: $\neg\exists x(\text{size}[x] < 10^{10} \wedge \text{aproof}[x, \text{subquote}[y, Y, y]])$

β: $\neg\exists x(\text{size}[x] < 10^{10} \wedge \text{aproof}[x, \text{subquote}[\alpha^*, Y, \alpha^*]])$

β asserts that there is no proof of β (in arithmetic) of feasible length. If there were, arithmetic would be untrue, and so we may assume that there is no such proof. β is therefore true. Unlike the formula β of theorem 12.7, however, this one is provable in A. Let $\sigma_1, \ldots, \sigma_n$ be an enumeration of the finitely many s-expressions whose size is less than 10^{10}. For each i, $A \vdash \neg\text{aproof}[\sigma_i, \beta^*]$. From all of these results, and the assertion that this list is complete, it is possible to prove β because the existential quantifier is bounded. Of course such a proof is much larger than 10^{10} in size.

The metamathematical proof of β is so similar to the preceding proof that we do not even need to write it down.

Theorem schemas are metamathematical assertions that occur very commonly. We do not want to have to write out a deduction for each instance of a schema that occurs frequently. Consider the least number schema which is:

$$\exists\eta(\alpha) \supset \exists\eta(\alpha \wedge \forall\nu(\alpha(\nu/\eta) \supset \nu \geq \eta))$$

This is true for any well-formed formula α, any numeric variable η, and any numeric variable ν which is free for η in α. The assertion that all instances of this schema are provable is not even metamathematical. It is:

$$\exists y \exists z \exists w(\text{wff}[y] \wedge \text{numvar}[z] \wedge \text{numvar}[w] \wedge \neg\text{null}[\text{sub}[\text{list}[w, z], y]] \wedge x = \text{list}[\text{IMPLIES}, \text{list}[\text{EXIST}, z, y], \text{list}[\text{EXISTS}, z, \text{list}[\text{AND}, y, \text{list}[\text{FORALL}, w, \text{list}[\text{IMPLIES}, \text{sub}[\text{list}[w, z], y], \text{list}[\text{GTQ}, w, z]]]]]]) \supset \exists u(\text{aproof}[u, x])$$

where sub is defined in problem set 22, No. 1. Let us abbreviate this to $\text{lnp}[x] \supset \exists y(\text{aproof}[y, x])$, where lnp stands for least number principle. This formula is provable by formalizing a deduction schema for this theorem schema. It is tedious work, and one first has to deal with some properties of substitutivity. But having done this, we can then deduce from (**) the formula:

$$\text{arith}[x] \supset \text{lnp}[x] \supset \text{truth}[x]$$

The advantage of having this formula is that given any arithmetic formula α such that $\text{lnp}[\alpha^*]$ is provable, we can derive α itself from M6. Lnp is a simple primitive recursive formula that merely tests its argument to see if it has a certain format. Lnp is called a theorem schema. In general, a theorem schema is any unary predicate ϕ such that:

$$\text{arith}[x] \supset \phi[x] \supset \text{truth}[x]$$

has been proven, and an inference schema is any n+1-ary predicate ψ such that:

$$\text{arith}[x_1] \supset \ldots \supset \text{arith}[x_{n+1}] \supset \text{truth}[x_1] \supset \ldots \supset \text{truth}[x_n] \supset \psi[x_1, \ldots, x_{n+1}] \supset \text{truth}[x_{n+1}]$$

has been proven.

Metamathematics allows us to demonstrate that a predicate defined in arithmetic is a theorem schema or inference schema. This solves half of the problem of reducing deduction to computation in routine cases. The other half of the problem is to prove that the predicate defined in the logical theory is the same as the predicate computed by some procedure in a programming language. When this has been established, we can then compile

the procedure, knowing that it is a valid addition to our collection of proof techniques. Theoretical results relevant to this problem are presented in Chapter Fourteen.

Problem Set 33

1. Show that $\alpha \equiv \text{truth}[\alpha^*]$ where α is any formula is an inconsistent schema.

2. Using (**) and M1 thru M6, prove an airthmetic formula asserting that arithmetic is consistent.

3. Why is M4 necessary to the proof of (**) even though each ground instance of M4 can be deduced from M6?

§13.3 The Hierarchy of Truth

The notion of truth in the system we have just described can be formalized by means of a predicate truth1[x] which is outside that system. This leads to a hierarchy of truth functions, each of which can reason meta-mathematically on the systems below it. It is possible to define a predicate truth[x, r] where r is a rank number, and to axiomatize truth so that at each rank the truth of formulas of lesser rank can be discussed. An arithmetic formula is of rank 0, and any formula in which all occurrences of truth are of the form truth[..., n] where n is a number is of rank n+1. If a formula contains truth[x, y], where y is anything other than a number, then the formula is outside the rank system, and cannot be discussed on any level. It is natural at this point to extend this idea even further by letting the second argument of truth be any ordinal number. This creates a whole new situation. It is not clear how much of this hierarchy is actually useful, but it would seem that having at least several levels of it are.

CHAPTER FOURTEEN

THE RECURSION THEOREM

Preview of Chapter Fourteen

The purpose of this chapter is to relate functions described by procedures, which are, in general, partial recursive functions, with descriptions of functions in first order arithmetic. We need to do this in order to prove theorems about procedures, and in order to find procedures for computing functions that have been defined logically. One example of the latter is the problem of computing a function which has been proven to be a theorem schema or inference schema by the methods outlined in Chapter Thirteen.

The recursion theorem is a basic result in recursive function theory. Its relevance to these problems has been recognized by researchers in the semantics of programming languages, a complex subject which we do not even approach except for the very trivial "language of recursive functions" as specified in Chapter Two. Research in this area, ranging from abstract topology to detailed semantic descriptions of ALGOL-60, is being done by [Scott], [Strachey] and others in the Oxford Programming Research Group, and [Milner], [Newry], [Igorashi] and others at the Stanford Artificial Intelligence Laboratory.

§14.1 The Nature of the Problem

In Chapter Twelve we describe a correspondence between procedures and formulas of first order arithmetic for the special case of primitive recursive functions. It is easy to generalize this syntactic correspondence, but not immediately useful because of problems of consistency.

Consider a recursive definition having the general form:

$$\varphi[\xi_1, \ldots, \xi_n] \leftarrow \epsilon$$

It can be converted into a set of formulas of first order arithmetic by replacing the "$\leftarrow$" with "=", and then applying the distributive rule of conditional forms, and the conversion of conditional forms into logical formulas, described below, until there are no more conditional forms. Since every part of the language of recursive functions except for the conditional is part of the language of logic, the result must be a set of formulas of logic.

Distributive Rule for Conditional Forms

Transform $\varphi[\ldots, [\pi_1 \rightarrow \epsilon_1, \ldots, \pi_n \rightarrow \epsilon_n], \ldots]$ into $[\pi_1 \rightarrow \varphi[\ldots, \epsilon_1, \ldots], \ldots, \pi_n \rightarrow \varphi[\ldots, \epsilon_n, \ldots]]$ where φ is any function or predicate name, including "=".

Conversion of Conditionals into Logical Formulas

When the conditional form $[\pi_1 \rightarrow \epsilon_1, \ldots, \pi_n \rightarrow \epsilon_n]$ is not a sub-form (i.e., when it is on the outside), transform it into the sequence of formulas:

$$\pi_1 \supset \epsilon_1$$
$$\neg\pi_1 \supset \pi_2 \supset \epsilon_2$$
$$\ldots$$
$$\neg\pi_1 \supset \ldots \supset \neg\pi_{n-1} \supset \pi_n \supset \epsilon_n$$

When the conditional form is on the outside of everything except for logical connectives, transform it into the conjunction of the formulas of this schema.

Example

subst[x, y, z] $\leftarrow$ [atom[z] $\rightarrow$ [y = z $\rightarrow$ x, T $\rightarrow$ z], T $\rightarrow$ subst[x, y, car[z]]*subst[x, y, cdr[z]]]

becomes

atom[z] $\supset$ ((y = z $\supset$ subst[x, y, z] = x) $\wedge$ (y $\neq$ z $\supset$ subst[x, y, z] = z))

$\neg$atom[z] $\supset$ subst[x, y, z] = subst[x, y, car[z]]*subst[x, y, cdr[z]]

Unfortunately, there is no justification for changing "←" into "=". A procedure cannot be inconsistent; it can at worst not produce a value, or produce a value not anticipated. The assertion that the left half is equal to the right half may be logically inconsistent. There are three situations that may arise from a recursive definition; it may over-define, under-define, or exactly define a function.

Case I: If a recursive definition defines a total function, then the transformation into logic produces a set of formulas that represents the function. Subst is an example of this. The function subst computed by the recursive definition is the same as the function represented by the two formulas.

Case II: The recursive definition is under-defined. In this case, the function computed by the recursive definition is partial, and there are more than one completions of the function that are model enlargements satisfying the formulas. Consider:

$$f[n] \leftarrow f[n + 1]$$
$$f[n, m] \leftarrow f[m, n]$$

Both of these definitions compute totally undefined functions. The first is satisfied by any constant function; the second is satisfied by any commutative function (on the natural numbers).

Case III: The recursive definition is over-defined. In this case, the function computed by the recursive definition is partial, and there are no completions of it that satisfy the formulas. There are no model enlargements, and the system is inconsistent. An example is the definition:

$$f[n] \leftarrow f[n] + 1$$

As a procedure, it does not converge. As an assertion, $f[n] = f[n] + 1$ is inconsistent.

Combinations of Cases I and III also occur.

The last example is extreme, but there is no general method for deciding which recursive definitions are over-determined. Nor can we regard them as undesirable. The definition of apply given in §2.4 is over-defined, and there is no way to avoid this.

In §12.4, we proved that all total recursive functions are represent-

able, although we cannot always decide what is a total recursive function. Suppose we call a partial function $\tilde{\varphi}$ partially representable if it is possible to define a function name φ in arithmetic such that if $\tilde{\varphi}(\sigma_1, \ldots, \sigma_n)$ is defined and has the value σ_{n+1}, then $A \vdash \varphi[\sigma_1, \ldots, \sigma_n] = \sigma_{n+1}$.

Theorem 14.1

All partial recursive functions are partially representable.

Proof: This is implicit in the proof of theorem 12.9. The representing function described in that proof has the value NIL for those arguments for which the partial recursive function is undefined, but it may not be possible to compute this NIL.

This method of representation is indirect, depending on the definition of an interpreter function applyk which itself is fairly complicated. The recursion theorem which follows is relevant to obtaining a direct transformation of a recursive definition into arithmetic without the danger of inconsistency, and in a manner that allows us to prove logical assertions about the procedure itself.

Problem 34

Show that apply is over-determined.

§14.2 The Recursion Theorem

The notation that we use here folows [Scott] in his work on lattice theory and programming languages, although we do not actually define a lattice.

We introduce an object "$\perp$" called "bottom" or "undefined". Letting S be the set of s-expressions, S_1 is the set $S \cup \{\perp\}$. The symbol "$\sqsubseteq$" meaning "is less than or equally defined than" is a binary operator on S_1 defined by: $\perp \sqsubseteq \perp$, $\perp \sqsubseteq \alpha$, and $\alpha \sqsubseteq \alpha$ where α is any s-expression. "$\sqsubseteq$" is a partial ordering.

The notion of equality on the domain S_1 will be represented by the symbol "$\cong$". The symbol "=" will mean computational equality. "$\cong$" is not

a computable predicate. The two equalities can be compared in the following table, where α and β are distinct s-expressions:

c	d	c = d	c ≅ d
α	α	T	T
α	β	F	F
α	$\perp$	$\perp$	F
$\perp$	α	$\perp$	F
$\perp$	$\perp$	$\perp$	T

The predicates atom, name and num, and the functions cons are extended to the domain S_1 in the same manner as "=", by defining the value to be $\perp$ if any argument is $\perp$. The functions enum, successor, predecessor, car and cdr are also extended to S_1 by defining the value to be $\perp$ if the argument is $\perp$ or if the value is not defined, e.g., enum[A] = $\perp$, and car[A] = $\perp$.

A function φ is called monotonic if $a_i \sqsubseteq b_i$ for $1 \le i \le n$ implies that $\varphi(a_1, \ldots, a_n) \sqsubseteq \varphi(b_1, \ldots, b_n)$, for all a_i and b_i in S_1. The basic functions mentioned in the preceding paragraph are all monotonic.

The ordering "$\sqsubseteq$" extends to functions by defining $\varphi_1 \sqsubseteq \varphi_2$ if for all a_1 thru a_n in S_1, $\varphi_1(a_1, \ldots, a_n) \sqsubseteq \varphi_2(a_1, \ldots, a_n)$.

Let $\{a_i\}$ be an infinite sequence of elements of S_1. It is a monotonic sequence if $a_i \sqsubseteq a_j$ for $i \le j$. A sequence of functions $\{\varphi_i\}$ (all having the same number of arguments) is a monotonic sequence if $\varphi_i \sqsubseteq \varphi_j$ for $i \le j$. An upper bound for a sequence is an object such that any member of the sequence is "$\sqsubseteq$" to it. A least upper bound for a sequence is an upper bound that is "$\sqsubseteq$" to any other upper bound.

Corollary 14.2

Every monotonic sequence has a least upper bound. If each function in the monotonic sequence $\{\varphi_i\}$ is itself a monotonic function, then the lub of the sequence is also a monotonic function.

A functional is a function that takes functions as arguments, i.e., it has one or more domains that are themselves function spaces. The notation for functionals is a bit cumbersome. When we write $\Phi: [S_1^n \to S_1], S_1^n \to S_1$, we

mean that Φ is a functional whose first argument is an n-ary function on S_1, whose 2nd thru n+1-th arguments are members of S_1, and whose value is in S_1. More complex possibilities exist, but this particular type of functional will be the only kind that we need to discuss here. Functionals can be monotonic in the same way functions are, since all their argument and value domains are partially ordered.

If Φ is a monotonic functional of the type mentioned above, and $\{\varphi_i\}$ is a monotonic sequence of n-ary functions, and a_1 thru a_n are a fixed set of elements of S_1, then $\{\Phi(\varphi_i, a_1, \ldots, a_n)\}$ is a monotonic sequence in S_1, and therefore has a least upper bound. The functional Φ is said to be continuous if it is monotonic, and if for every monotonic sequence $\{\varphi_i\}$, and every choice of a_i in S_1:

$$\text{lub}\{\Phi(\varphi_i, a_1, \ldots, a_n)\} \cong \Phi(\text{lub}\{\varphi_i\}, a_1, \ldots, a_n)$$

A fixpoint for the functional Φ is a function φ such that for every choice of a_1 thru a_n in S_1, $\Phi(\varphi, a_1, \ldots, a_n) \cong \varphi(a_1, \ldots, a_n)$. A least fixpoint for Φ is a fixpoint which is "$\sqsubseteq$" to any other fixpoint.

Theorem 14.3 (Fixpoint Theorem)

If Φ is a continuous functional having one n-ary functional argument, and n ordinary arguments, then it has a least fixpoint which is monotonic.
Proof: Define φ_0 by letting $\varphi_0(a_1, \ldots, a_n) \cong \perp$ for all a_i. Define φ_{n+1} by letting $\varphi_{n+1}(a_1, \ldots, a_n) \cong \Phi(\varphi_n, a_1, \ldots, a_n)$. We can show by induction that the sequence $\{\varphi_i\}$ is monotonic because $\varphi_0 \sqsubseteq \varphi_1$, and if $\varphi_n \sqsubseteq \varphi_{n+1}$, then $\varphi_{n+1}(a_1, \ldots, a_n) \cong \Phi(\varphi_n, a_1, \ldots, a_n) \sqsubseteq \Phi(\varphi_{n+1}, a_1, \ldots, a_n) \cong \varphi_{n+2}(a_1, \ldots, a_n)$, so $\varphi_{n+1} \sqsubseteq \varphi_{n+2}$. Let φ be the lub of the sequence $\{\varphi_i\}$. Because Φ is continuous, $\Phi(\varphi, a_1, \ldots, a_n) \cong \text{lub}\{\Phi(\varphi_i, a_1, \ldots, a_n)\} \cong \text{lub}\{\varphi_i(a_1, \ldots, a_n)\} \cong \varphi(a_1, \ldots, a_n)$. So φ is a fixpoint for Φ. Now let ψ be any other fixpoint of Φ. $\varphi_0 \sqsubseteq \psi$, and if $\varphi_n \sqsubseteq \psi$, then $\varphi_{n+1}(a_1, \ldots, a_n) \cong \Phi(\varphi_n, a_1, \ldots, a_n) \sqsubseteq \Phi(\psi, a_1, \ldots, a_n) \cong \psi(a_1, \ldots, a_n)$, or $\varphi_{n+1} \sqsubseteq \psi$. By induction, $\varphi_i \sqsubseteq \psi$ for all i, and so ψ is an upper bound for $\{\varphi_i\}$. Since φ is the lub of $\{\varphi_i\}$, $\varphi \sqsubseteq \psi$, and so φ is the least fixpoint of Φ. φ is a monotonic function because if $a_i \sqsubseteq b_i$, then $\varphi(a_1, \ldots, a_n) \cong \Phi(\varphi, a_1, \ldots, a_n) \sqsubseteq \Phi(\varphi, b_1, \ldots, b_n) \cong \varphi(b_1, \ldots b_n)$.

This proof of the fixpoint theorem has been an exercise in abstract algebra; it uses no properties of S_1 other than that S_1 is a partially ordered set having a least element, and such that every monotonic sequence has a lub. But the notions of monotonic and continuous are surprisingly useful in the theory of computation. We have already noted that the basic functions of computation are monotonic. In fact, any partial recursive function is monotonic because that simply means that supplying more information about the arguments of the function does not decrease the possibility that the function has a value.

Let φ be a partial recursive function on S. We extend it to be a total function on S_1 by letting the value be $\perp$ wherever the value was previously undefined. φ may also have $\perp$ as an argument, in which case the value will be $\perp$ unless the argument is not needed in the computational process, and a value is obtained without it. By Church's thesis, there is an effective procedure that computes the value of φ whenever it is an s-expression, but may never terminate if the value is $\perp$. Let φ_i be the function such that $\varphi_i(a_1, \ldots, a_n)$ is defined by doing i amount of work on the computation of $\varphi(a_1, \ldots, a_n)$, and returning the value if one is obtained, and being undefined otherwise. The sequence $\{\varphi_i\}$ is not uniquely determined unless we fix a particular procedure for computing φ, and specify an exact definition of work. But by merely postulating that every computation requires some finite amount of work, we see that every such sequence has φ as its lub.

Let $\Phi(\varphi, a_1, \ldots, a_n)$ be a functional. We would like to call Φ partial recursive if there is an effective procedure for computing it. But this requires that we specify how this procedure is to be given functions as arguments. If φ is a partial recursive function, then the problem is simplified. We simply give to the procedure Φ a procedure that computes φ, and require that the value be independent of which procedure for φ is used. But we do not wish to restrict the argument φ of Φ to partial recursive functions only. So we invent the notion of an oracle which is like a black box, or an on-line intervention in a computational process.

The purpose of an oracle is to simulate the effect of a partial recursive function even when it is not. A black box that gives the value of $\varphi(a_1, \ldots, a_n)$ when it is defined, and replies "$\perp$" when it is not defined does too

much, because when a recursive process does not terminate we are not generally told that; we simply wait for ever. On the other hand, a black box which gives the value of $\varphi(a_1, \ldots, a_n)$ when it is defined, and hangs up forever if it is not defined, is insufficient, because we can run any process for a certain amount of time to see if it produces a value within that time. A workable idea is to make use of the notion of a function as a limit. So, given the function φ, let $\{\varphi_i\}$ be any sequence of functions whose lub is φ. Then an oracle for φ is a black box that when interrogated about $\varphi_i(a_1, \ldots, a_n)$ for particular i, either produces a value or replies "$\perp$".

We now define a partial recursive functional $\Phi(\varphi, a_1, \ldots, a_n)$ as a functional for which there is an effective procedure which computes its value when given the arguments a_i, and an oracle for φ. If the value of Φ is "$\perp$", then the procedure is permitted not to terminate. Implicit in the idea that Φ is a function of φ, and not of the particular oracle chosen to represent φ, is the requirement that the value of the computation is independent of the choice of oracle for φ.

Lemma 14.4

All partial recursive functionals are continuous.

Proof: The symbol "$\perp$" never enters into an effective procedure. It is used in discussions about effective procedures to mean that information is not available. A procedure can never contain "if $x = \perp$, then ...". This is sufficient to make all effective procedures monotonic.[1] Now let a_1 thru a_n be a particular choice of objects in S_1, and let $\{\varphi_i\}$ be any monotonic sequence. In the following discussion $\Phi(\varphi, a_1, \ldots, a_n)$ is abbreviated to $\Phi(\varphi)$.

Let φ be the lub of the sequence $\{\varphi_i\}$. If $\Phi(\varphi) \cong \perp$, then $\Phi(\varphi_i) \cong \perp$ for each i, since Φ is monotonic. So $\text{lub}\{\Phi(\varphi_i)\} \cong \Phi(\varphi)$. If $\Phi(\varphi) \cong \alpha$ where α is an s-expression, then since this computation is independent of the oracle used

[1] One way to be sure that the procedure does not act on the information "this argument is undefined" is to replace each individual argument with an oracle for a constant function $\psi[]$ based on a sequence $\{\psi_i\}$. This sequence either produces the argument for some i, or it never does, and the argument is "$\perp$". In other words, the procedure has to work to obtain each argument, and it can never know if or when it will get the argument until it gets it.

for φ, we can let the oracle be based on the sequence $\{\varphi_i\}$. Because α was computed by an effective procedure, it can only have interrogated the oracle a finite number of times. Let φ_k be the highest function in the sequence $\{\varphi_i\}$ that was used. Consider the computation of $\Phi(\varphi_k)$ where φ_k is represented by an oracle using the sequence beginning with φ_1 thru φ_k, and then being φ_k from there on. This computation must proceed exactly like the previous one, because no function with an index greater than k will ever be interrogated. So $\Phi(\varphi) \cong \Phi(\varphi_k) \sqsubseteq \text{lub}\{\Phi(\varphi_i)\} \sqsubseteq \Phi(\varphi)$ and so Φ is continuous.

Theorem 14.5 (Kleene's Recursion Theorem)

Every partial recursive functional has a least fixpoint which is a partial recursive function.

Proof: By lemma 14.4, if Φ is partial recursive, it is continuous. By theorem 14.3, it then has a least fixpoint φ. To show that φ is partial recursive, consider the sequence $\{\varphi_i\}$ in the proof of theorem 14.3 of which φ is the lub. φ_0 is partial recursive because it is represented by the process that never produces a value. Suppose φ_n is partial recursive. Then φ_{n+1} is partial recursive because $\varphi_{n+1}(a_1, \ldots, a_n) \cong \Phi(\varphi_n, a_1, \ldots, a_n)$, and there are effective procedures for φ_n and Φ. By induction, all the φ_i are partial recursive, and so φ is partial recursive because it is computed by the procedure that tries all the φ_i.

§14.3 Application of the Recursion Theorem

Consider a recursive definition:

$$\varphi[\xi_1, \ldots, \xi_n] \leftarrow \epsilon$$

where ϵ has no free variables other than the ξ_i, and every function and predicate name in ϵ, except for φ, is already defined and partial recursive. Then ϵ is a partial recursive functional because it specifies a computation depending on the functional argument φ, and the s-expression arguments ξ_1 thru ξ_n. Furthermore, if "$\leftarrow$" is replaced by "$=$", then we have the fixpoint equation for this functional.

Unfortunately, the situation gets a bit messy here because there are various semantics that one can propose for the language of recursive

definitions. The choice of semantics will determine what functional $\Phi(\varphi, \xi_1, \ldots, \xi_n)$ is specified by the form ϵ. We shall briefly consider two of them here: Φ_L is the functional specified by the LISP semantics described in Chapter Two. Φ_C is the functional specified by the complete semantics, that is, the semantics that computes as much as is possible from the information available in ϵ. The two semantics never produce conflicting values, but Φ_C may produce a value where Φ_L fails to do so, that is, $\Phi_L \sqsubseteq \Phi_C$. There are two significant differences between the two:

I: LISP evaluation uses call by value. This sometimes gets hung up because all arguments for a function must be pre-evaluated, even if they are not needed for its computation. For example, the definition:

$$f[m, n] \leftarrow [m = 0 \rightarrow 1, T \rightarrow f[m - 1, f[m, n]]]$$

computes in LISP a function that is 1 if m is 0, and is otherwise undefined. But the complete semantics uses call by name, which does not attempt to evaluate the inner f[m, n], and so does not get into an endless cycle. It computes the function which is 1 for all numeric arguments. This problem is discussed thoroughly in [Vuillemin].

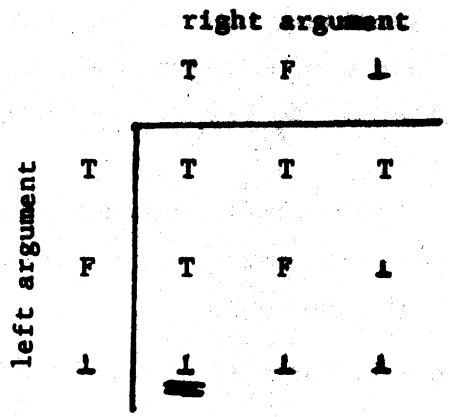
II: LISP semantics specifies a left-to-right order of evaluation for conditionals and logical operators. For example, the definition:

$$f[n] \leftarrow [f[n] = 0 \rightarrow 1, T \rightarrow 1]$$

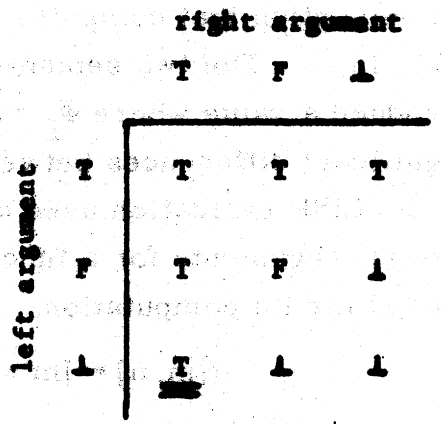
computes the totally undefined function in LISP, but the constant function f[n] = 1 in the complete semantics.

In LISP, the form $\epsilon_1 \vee \epsilon_2$ is evaluated by first evaluating ϵ_1. If ϵ_1 has no value, then the expression is undefined. In the complete semantics, the expression is true if either branch is true. This point can be stated by means of the three valued truth tables for the operation "$\vee$", keeping in mind that the interpretation of "$\perp$" is "information not available", or "value unknown". (See tables at top of next page.)

Both of these tables are monotonic, a necessity for them to be computable. We might call the first one "weak", and the second "strong" or "symmetric". We have chosen the word "complete" because of the property of semantic completeness which is the same as in logic. (From B we can deduce A $\vee$ B without having to prove that A is true or false.) The strong



LISP "∨"



Complete "∨"

truth tables are discussed in [Kleene, §64] in connection with partial recursive functions. His term for "monotonic" is "regular".

Let us examine some recursive functionals and their fixpoints taken from [Manna and Vuillemin]. Consider the functional:

$$\Phi(\varphi, a, b) \text{ is } [a = b \rightarrow b + 1, T \rightarrow \varphi[a, \varphi[a - 1, b + 1]]]$$

A fixpoint for Φ is a function φ such that $\varphi(a, b) \cong \Phi(\varphi, a, b)$ for every choice of a and b in S_1. "+" and "-" are functions that are undefined for non-numerical arguments, or if the result of subtraction is negative. The nature of "$\cong$" is such that for the equation to be true, both sides must be the same number, or both must be undefined. Notice, also, that "=" used in the conditional expression is undefined if either argument is undefined. We now specify three functions, each of which is a fixpoint of Φ:

φ_1: $a + 1$

φ_2: if $a \geq b$ then $a + 1$ else $b - 1$

φ_3: if $a \geq b$ and $a - b$ is even, then $a + 1$
else not defined

A certain amount of investigation will convince one that each of these is a fixpoint. It can also be shown that $\varphi_3 \quad \varphi_1$ and $\varphi_3 \quad \varphi_2$. φ_3 is in fact the least fixpoint of Φ.

We now extend the theory A of Chapter Eleven to be a theory A_1 about the model S_1 which is the s-expressions with the added object "$\perp$", and the definitions of the basic functions extended appropriately. However, we shall not use "$\perp$", "$\in$" or "$\cong$" anywhere in the language of the theory, because they are not computable.

The variables beginning with r thru z range over s-expressions, while variables beginning with a, b and c range over S_1. The axioms of group H are now needed because we admit to the possibility of their being something that is not an s-expression. In this theory, list[x] $\neq$ x is a theorem, but list[a] $\neq$ a is not, because of the counter-example cons[$\perp$, NIL] $\cong \perp$.

Specifically, we have axiomatized the theory of s-expressions so as to admit the possibility that there might be things that are not s-expressions. But we have not axiomatized S_1 particularly; we simply note that S_1 is one model that satisfies the theory A_1.

Consider the recursive definition f[a] $\leftarrow$ f[a] + 1. Its least fixpoint is the totally undefined function. Since this is a "total" function on the domain S_1, the equation f[a] = f[a] + 1 is satisfiable in S_1, and no inconsistency results from it. The instantiation f[3] = f[3] + 1 is satisfied because $\perp \cong \perp + 1$, and "=" approximates "$\cong$" to the extent that it is computable. One cannot derive 0 = 1 from this formula, because if we start from the theorem m = m + 1 $\supset$ 0 = 1 (which is provable), we find that replacing m with f[3] is not a valid substitution because f[3] is not a numeric typed term.

Partial Recursion Schema

> If φ is a new name, then the transformation of the recursive definition $\varphi[x_1, \ldots, x_n] \leftarrow \epsilon$ into a set of formulas of A_1 may be used as a definition for φ.

Not only is this rule consistent, but it makes all partial recursive functions partially representable, and all total recursive functions representable in a direct manner. We present the following theorem without proof because there is too much detail that we have not completed; it is not difficult conceptually.

Theorem 14.6 (Partial Representation)

Let the function name φ be defined in A_1 by the partial recursion

schema, where all the other function and predicate names in the schema are already defined and (partially) representable. Then $\tilde{\varphi}(\sigma_1, \ldots, \sigma_n) = \sigma_{n+1}$, where $\tilde{\varphi}$ is the least fixpoint of ϵ in the complete semantics, if and only if $A_1 \vdash \varphi[\sigma_1, \ldots, \sigma_n] = \sigma_{n+1}$.

This theorem is the justification for the "completeness" of this particular choice of semantics.

If a function φ has been defined by the partial recursion schema, we may be able to demonstrate that it is a total recursive function by proving the formula $\exists y(\varphi[x_1, \ldots, x_n] = y)$. This also allows us to assert that the function φ is a well-typed s-expression valued function. Other totality and type information may be developed similarly. One may be able to prove $\exists_n(\varphi[x, m] = n)$, which types φ as a total numeric-valued function having an s-expression argument and a numeric argument. If one can prove $\psi[x] \supset \exists y(\varphi[x] = y)$, then one has shown that φ is defined at least for those values where $\psi[x]$ is true.

It is not possible to prove the totality of all total recursive functions in this manner, since this would make "total" recursively enumerable. But it is possible in many cases. In particular, it is always possible to prove that primitive recursive definitions define total recursive functions. (The argument is by induction.)

One word of caution on this schema. The model S_1 introduces "$\perp$" into the domain, but not into the logic itself. The model is still a model of standard two-valued first order logic. So while the recursion schema permits replacement of "$\leftarrow$" with "=", it would be inconsistent to replace "$\leftarrow$" with "$\equiv$". $p[x] \equiv \neg p[x]$ is inconsistent in the present system, although one could develop a three-valued logic.

The recursion theorem can be stated in a multi-dimensional form which is that given the set of equations:

$$\Phi_1(\varphi_1, \ldots, \varphi_k, \xi_1, \ldots, \xi_{n_1}) \cong \varphi_1(\xi_1, \ldots, \xi_{n_1})$$
$$\ldots$$
$$\Phi_k(\varphi_1, \ldots, \varphi_k, \xi_1, \ldots, \xi_{n_k}) \cong \varphi_k(\xi_1, \ldots, \xi_{n_k})$$

where the Φ_i are partial recursive, there is a set of least fixpoints $\tilde{\varphi}_1$ thru $\tilde{\varphi}_k$

which are partial recursive. This conveniently corresponds to the programmer's habit of defining recursive functions in interdependent batches. The partial recursion schema may be extended to permit this.

Problem Set 35

1. Investigate the work of Vuillemin, and the Oxford Group, to see how the recursion theorem is used in the study of the semantics of programming languages. How do they deal with the problem of the computed function of LISP and ALGOL being less than the semantically complete fixpoint?

2. Extend the syntax of first order logic to allow conditionals used either as logical connectives, or choice functions within terms, so that conditionals can be nested inside each other. Add transformation rules that are consistent, and make this logic complete semantically. Theorem 14.6 is now trivial to prove.

CHAPTER FIFTEEN

SECOND ORDER ARITHMETIC AND SET THEORY

§15.1 Second Order Arithmetic

Starting with the system A presented in Chapter Twelve, we can develop a second order theory of s-expressions. The model for this theory has as its domain the set $S \cup \{S^2\}$, i.e., there will be both s-expressions and sets of s-expressions in the domain. Set variables will begin with a capital R, S or T, and be followed by at least one lower case letter.

The basic predicate of set theory is membership. In second order arithmetic, things that are members are s-expressions, and things that have members are sets. So:

$$a \in b \supset (\text{sexpr}[a] \wedge \text{set}[b])$$

The principle of extensionality is that two sets are equal if they have the same members:

$$\text{EXT:} \quad \forall x(x \in Sa \equiv x \in Sb) \supset Sa = Sb$$

The principle of comprehension is that there is a set to correspond to every property definable in the theory, or:

$$\text{COMP:} \quad \exists Sa \forall x(x \in Sa \equiv \alpha)$$

where α is any formula not having the variable Sa free.

From the extensionality axiom, one can prove that the existential quantifier in the comprehension axiom schema is unique, i.e., $\exists_1 Sa \forall x(x \in Sa \equiv \alpha)$.

The induction axiom schemas of first order arithmetic can be replaced by single formulas in second order arithmetic:

NIND: $0 \in Sa \supset \forall n(n \in Sa \supset n' \in Sa) \supset \forall n(n \in Sa)$

SIND: $\forall x(atom[x] \supset x \in Sa) \supset \forall x\forall y(x \in Sa \supset y \in Sa \supset x*y \in Sa) \supset \forall x(x \in Sa)$

Together with the axioms of A, these are the axioms for second order arithmetic, A^2.

We can define certain classes of sets, and even given them special variable types that are sub-types of the type "set". The most obvious one is the set of numbers. We define the type "nset" by:

$$nset[Sa] \equiv \forall x(x \in Sa \supset num[x])$$

Variables of type "nset" will start with a capital N, such as Na, Nb, etc.

The least number principle can be stated as a single axiom:

$$\exists n(n \in Na) \supset \exists_1 n(n \in Na \wedge \forall m(m \in Na \supset n \leq m))$$

First order functions and predicates can be represented as individual sets in second order arithmetic. If $\tilde{\psi}$ is an m-ary predicate on the s-expressions, then it is represented by the set containing only lists of length m, and such that $list[\sigma_1, \ldots, \sigma_n]$ is a member of the set if and only if $\tilde{\psi}(\sigma_1, \ldots, \sigma_n)$ is true. If $\tilde{\varphi}$ is an n-ary function on s-expressions, then it is represented by a set containing only lists whose length is n+1, and such that $list[\sigma_{n+1}, \sigma_1, \ldots, \sigma_n]$ is a member of the set if and only if $\varphi(\sigma_1, \ldots, \sigma_n) = \sigma_{n+1}$. Putting the value first is a matter of convenience. It is easy to make definitions such as:

$$Parfun3[Sa] \equiv (\forall x(x \in Sa \supset s4[x]) \wedge \forall x\forall y(x \in Sa \supset y \in Sa \supset cdr[x] = cdr[y] \supset x = y))$$

$$Totfun3[Sa] \equiv (Parfun3[Sa] \wedge \forall x(s3[x] \supset \exists y(y*x \in Sa)))$$

Parfun and Totfun are second order predicates. Obviously, one can continue to make specific definitions of functions and predicates having such and such numeric or symbolic arguments and values.

There are second order functions or functionals which process first order functions and might be called combinators of first order functions. These are abstract, rather than procedural operations and do not correspond to recursive processes necessarily. For example, given the unary partial functions $\tilde{\varphi}_1$, and $\tilde{\varphi}_2$, there is the partial function $\tilde{\varphi}_2(\tilde{\varphi}_1(s))$. The second order function Compose(Sa, Sb) has this composition function as its value. It is trivial to prove:

$$\text{Parfun1}[Sa] \supset \text{Parfun1}[Sb] \supset \exists_1 Sc(\text{parfun1}[Sc] \wedge \forall x \forall y(\text{list}[x, y] \in Sc \equiv \exists z(\text{list}[z, y] \in Sa \wedge \text{list}[x, z] \in Sb)))$$

Using Rule PF, we define this unique Sc to be Compose[Sa, Sb].

Corresponding to any procedure for computing a first order partial recursive function, is the set which is the function it computes. We can call this the extension of the procedure. Trivially:

$$\exists_1 Sa \forall y(y \in Sa \equiv \exists z \exists w \exists n(\text{applyk}[x, z, n] = \text{list}[w] \wedge y = w * z))$$

Using Rule F, we define this unique Sa to be Extension[x].

It is possible to define an ordinary first order recursive function pcompose such that if x and y are s-expression translations of procedures for unary partial recursive functions, pcompose[x, y] will be a procedure for computing the composition of the two functions. Then for any such x and y the following identity holds:

$$\text{Extension}[\text{pcompose}[x, y]] = \text{Compose}[\text{Extension}[x], \text{Extension}[y]]$$

It is even possible to define an abstract Apply by:

$$\exists_1 y(y * x \in Sa) \supset \text{Apply}[Sa, x] * x \in Sa \qquad \text{Rule PF}$$

This second order function applies any function (represented by a set) to its list of arguments, and produces a value (abstractly). The evaluation of a partial recursive function by an interpreter coincides with a special case of this in the sense that:

$$\exists n(\text{applyk}[x, y, n] = \text{list}[z]) \supset \text{Apply}[\text{Extension}[x], y] = z$$

The purpose of this discussion has been to show that a much larger number of situations can be discussed very precisely in second order arithmetic than in first order. This is done at the expense of making the discussion abstract, in that the entities being discussed are no longer constructable. It seems as though any mathematical discussion cannot realistically be kept at the first order level. When we want to go beyond the second level, we can either explicitly formulate third order and fourth order arithmetic, etc., or we can go into axiomatic set theory.

Problem Set 35

1. Show that there are formulas of first order logic that are not prov-

able in first order logic, but are provable in second order logic.

2. Prove that second order logic is incomplete.

3. Define the functions Union, Intersection and Complement (with respect to the set of s-expressions). Union, for example, is a function of two arguments which are sets, and the value is the set which is their union.

4. What is an impredicative definition? How does the axiom schema COMP avoid impredicative definitions?

§15.2 Axiomatic Set Theory

There are basically two styles of axiomatic set theory. Zermelo-Fraenkel (ZF) set theory is a theory about sets only, while von Neumann-Bernays-Gödel (NBG) set theory is a theory about sets and classes, which are universal objects that are too big to be called sets. ZF has axiom schemas giving rise to infinitely many individual axioms, while NBG is finitely axiomatized. For the reader wishing an introduction to set theory, [Shoenfield, Chapter 9] discusses ZF, and [Mendelson, Chapter 4] discusses NBG. Set theory is discussed informally, that is, without reference to an axiomatization in first order logic, in [Halmos].

Two of the important concepts developed in set theory are cardinality, and ordinality. We are using the concept of cardinality when we investigate second order arithmetic and mention higher arithmetic. One of the principles of set theory is that, given any set, there is the set of all subsets of that set (known as the power set) which is of higher cardinality than the original set. So when set theory axioms are added to arithmetic, we automatically get sets of s-expressions, sets of sets of s-expressions, etc. Axiomatic set theory, as it is commonly presented, is abstract in that the only basis for constructing sets is the empty set. But it is easy to merge the axioms of set theory with an existing theory such as first order arithmetic.

The other major concept of set theory is ordinality. We have hardly mentioned ordinal numbers in this book, yet the theory of ordinals enriches the study of recursive functions, and axiom systems at almost every level.

There is a whole hierarchy of ordinal numbers even when we restrict ourselves to countable ordinals - those having the lowest infinite cardinality. The smallest transfinite ordinal is called ω. There is the sequence ω, $\omega + 1$,

$\omega + 2$, ..., and an ordinal $\omega \times 2$ that is greater than any of these. There is the sequence $\omega \times 2$, $\omega \times 2 + 1$, ..., and the ordinal $\omega \times 3$ which is greater than these. The ordinal ω^2 is greater than any ordinal in the sequence ω, $\omega \times 2$, $\omega \times 3$, etc. All of these and many more are still countable.

Ordinals are the natural mathematical structure for representing the idea of transcendence. For example, Gödel's theorem allows us to find a formula independent of a certain axiom system. This can be repeated ad infinitum, but even after adding infinitely many axioms, we can still find an independent formula, and after adding sequences of sequences of new axioms, we still find that we can obtain an independent formula. The unsuccessful effort to finally complete the axiom system leads naturally to Kleene's concept of a constructive ordinal.

BIBLIOGRAPHY

Chaitin, Gregory, "Information Theoretic Complexity", IEEE Transactions on Information Theory, January 1974, pp. 10-15.

Chang, C. L. and Lee, R. C., Symbolic Logic and Mechanical Theorem Proving, Academic Press, 1973.

Davis, Martin, Computability and Unsolvability, McGraw-Hill, 1958.

Davis, Martin and Putnam, Hilary, "A Computing Procedure for Quantification Theory", Journal of the ACM, March 1960, pp. 201-215.

Halmos, Paul R., Naive Set Theory, Van Nostrand, 1960.

Hart, Timothy and Levin, Michael, "LISP - 240 Exercises and Solutions" in The Programming Language LISP, Its Operation and Applications, ed. Edmund Berkeley and D. G. Bobrow, MIT Press, 1964.

Igorashi, Shigeru, "Admissibility of Fixed-Point Induction in First-Order Logic of Typed Theories", Stanford Artificial Intelligence Project, Memo AIM-168, May 1972.

Jack, Alex, 221A Baker Street, The Adamantine Sherlock Holmes, The Kanthaka Press, 248 Tappan St., Brookline, Mass. 02146, price $2.95.

Kleene, Stephen Cole, Introduction to Metamathematics, Van Nostrand, 1952.

Manna, Zohar and Vuillemin, Jean, "Fixpoint Approach to the Theory of Computation", Communications of the ACM, July 1972, pp. 528-536.

Margaris, Angelo, First Order Mathematical Logic, Blaisdell, 1967.

McCarthy, John, "A Basis for a Mathematical Theory of Computation", in Computer Programs and Formal Systems, ed. Braffort and Hirschberg, North Holland, 1967.

McCarthy, John et al., LISP 1.5 Programmer's Manual, MIT Press, 1965.

Mendelson, Elliott, Introduction to Mathematical Logic, Van Nostrand, 1964.

Milarepa, The One Hundred Thousand Songs of Milarepa, trans. Garma C. C. Chang, Harper Colophon, 1970.

Milner, Robin, "Logic for Computable Functions (LCF), Description of a Machine Implementation", Stanford Artificial Intelligence Project, Memo AIM-169, May 1972.

Milner, Robin, "Models of LCF", Stanford Artificial Intelligence Project, Memo AIM-186, January 1973.

"There is a great wind coming whose electrical storm shall be felt for the duration of the century," Samdup mused raising his dark eyebrows and staring out meditatively at the ominous sky.

"Humanity shall become possessed by its technology," Martha agreed, handing Mycroft the gramophone recordings of the German Embassy, the keys to the one hundred horsepower Benz, and the telephone number of the real Von Herling who remained in his hotel room tied up with his telephone cord.

"Good grief," she continued, as her voice changed from a Suffolk to a New Jersey accent, "even the opera itself will vanish and people shall listen to recorded music at home." Smiling at Sherlock, she took off her old lady's grey wig and her beautiful long red hair streamed down her shoulders. "We are our own machines, and all of the powers of the universe are within us."

-The Adamantine Sherlock Holmes-

Nagel, Ernest and Newman, James R., Gödel's Proof, New York University Press, 1958.

Newry, Malcolm, "Axioms and Theorems for Integers, Lists, and Finite Sets in LCF", Stanford Artificial Intelligence Project, Memo AIM-184, January 1973.

Robinson, John A., "A Machine-Oriented Logic Based on the Resolution Principle", Journal of the ACM, January 1965, pp. 23-41.

Scott, Dana, "The Lattice of Flow Diagrams", Tech. Memo No. PRG-3, Programming Research Group, Oxford University Computing Laboratory, 45 Banbury Rd., Oxford, England.

Shoenfield, Joseph R., Mathematical Logic, Addison-Wesley, 1967.

Strachey, Christopher, "Varieties of Programming Languages", Programming Research Group, Oxford, Technical Memo PRG-10. (See [Scott].)

Vuillemin, Jean, "Correct and Optimal Implementations of Recursion in a Simple Programming Language", Rapport de Recherche No. 24, Institut de Recherche d'Informatique et d'Automatique, July, 1973.

Weissman, Clark, LISP 1.5 Primer, Dickenson, 1967.

Yessenin-Volpin, Alexander I., "The Main Problem in the Foundation of Mathematics", Boston Colloquium for the Philosophy of Science, March 1974.

Yonezawa, Akinori, "On a Proof Procedure of the First Order Predicate Calculus", Master's Thesis, Tokyo University, 1972.

This empty page was substituted for a blank page in the original document.

CS-TR Scanning Project
Document Control Form

Date : 3/7/96

Report # LCS-TR-131

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 198 (196-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☐ Single-sided or
☒ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☒ Unknown ☐ Other:_____

Check each if included with document:

☐ DOD Form ☐ Funding Agent Form ☐ Cover Page
☐ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages(by page number): _____

Photographs/Tonal Material (by page number): _____

Other (note description/page number):

Description : Page Number:

IMAGE MAP: (1-192) UNNUM'ED TITLE, ABSTRACT, ACKN, MISC, (QUOTE), PREFACE(6), PREREQUISITES, CONTENTS (2), BLANK PAGES, 1-174, UNNUM'ED BIBLIOGRAPHY (3), UNNUM'ED BLANK. 093-196) SCANCONTROL, TRGT'S(3)

Scanning Agent Signoff:
Date Received: 3/7/96 Date Scanned: 3/15/96 Date Returned: 3/14/96

Scanning Agent Signature: Michael W. Cook

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 3/15/1996

M.I.T. Libraries
Document Services