Mathematical Logic and Computability

J. Keisler, K. Kunen, T. Millar, A. Miller, J. Robbin

February 10, 2006

This version is from Spring 1987

Contents

List of Figures

1 Propositional Logic

In this book we shall study certain *formal languages* each of which abstracts from ordinary mathematical language (and to a lesser extent, everyday English) some aspects of its logical structure. Formal languages differ from natural languages such as English in that the syntax of a formal language is precisely given. This is not the case with English: authorities often disagree as to whether a given English sentence is grammatically correct. *Mathematical logic* may be defined as that branch of mathematics which studies formal languages.

In this chapter we study a formal language called *propositional logic*. This language abstracts from ordinary language the properties of the *propositional connectives* (commonly called conjunctions by grammarians). These are "not", "and", "or", "if", and "if and only if". These connectives are *extensional* in the sense that the truth value of a compound sentence built up from these connectives depends only the truth values of the component simple sentences. (The conjunction "because" is not extensional in this sense: one can easily give examples of English sentences p, q, and r such that p, q, r are true, but *'p because q'* is true and *'p because r'* is false.) Furthermore, we are only concerned with the meanings that common mathematical usage accord these connectives; this is sometimes slightly different from their meanings in everyday English. We now explain these meanings.

NEGATION. A sentence of form *'not p'* is true exactly when p is false. The symbol used in mathematical logic for "not" is $\neg$ (but in older books the symbol $\sim$ was used). Thus of the two sentences

$$\neg 2 + 2 = 4$$

$$\neg 2 + 2 = 5$$

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

CONJUNCTION. A sentence of form *'p and q'* is true exactly when both p and q are true. The mathematical symbol for "and" is $\wedge$ (or & in some older books). Thus of the four sentences

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

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

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

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

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

For the mathematician, the words "and" and "but" have the same meaning. In everyday English these words cannot be used interchangeably, but the difference is psychological rather than logical.

DISJUNCTION. A sentence of form *'p or q'* is true exactly when either p is true or q is true (or both). The symbol use in mathematical logic for "or" is $\vee$. Thus of the four sentences

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

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

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

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

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

Occasionally, the sentence *p or q* has a different meaning in everyday life from that just given. For example, the phrase "soup or salad included" in a restaurant menu means that the customer can have either soup or salad with his/her dinner at no extra cost but not both. This usage of the word "or" is called *exclusive* (because it excludes the case where both components are true). Mathematicians generally use the *inclusive* meaning explained above; when they intend the exclusive meaning they say so explicitly as in *p or q but not both.*

IMPLICATION. The forms *'if p, then q'*, *'q, if p'*, *'p implies q'*, *'p only if q'*, and *'q whenever p'* all having the same meaning for the mathematician: *p 'implies q'* is false exactly when p is true but q is false. The mathematical symbol for "implies" is $\Rightarrow$ (or $\supset$ in older books). Thus, of the four sentences

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

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

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

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

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

This usage is in sharp contrast to the usage in everyday language. In common discourse a sentence of form *if p then q* or *p implies q* suggests a kind of causality that is that q "follows" from p. Consider for example the sentence

"If Columbus discovered America, then Aristotle was a Greek."

Since Aristotle was indeed a Greek this sentence either has form *If true then true* or *If false then true* and is thus true according to the meaning of "implies" we have adopted. However, common usage would judge this sentence either false or nonsensical because there is no causal relation between Columbus's voyage and Aristotle's nationality. To distinguish the meaning of "implies" which we have adopted (viz. that *p implies q* is false precisely when p is true and q is false) from other possible meanings logicians call it *material implication*. This is the only meaning used in mathematics. Note that material implication is extensional in the sense that the truth value of *p materially implies q* depends only on the truth values of p and q and not on subtler aspects of their meanings.

EQUIVALENCE. The forms *'p if and only if q'*, *'p is equivalent to q'*; and *'p exactly when q'* all have the same meaning for the mathematician: they are true when p and q have the same truth value and false in the contrary case. Some authors use "iff" is an abbreviation for "if and only if". The mathematical symbol for *if and only if* is $\Leftrightarrow$ ($\equiv$ in older books). Thus of the four sentences

$$2+2=4 \Leftrightarrow 2+3=5$$

$$2+2=4 \Leftrightarrow 2+3=7$$

$$2+2=6 \Leftrightarrow 2+3=5$$

$$2+2=6 \Leftrightarrow 2+3=7$$

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

Evidently, *p if and only if q* has the same meaning as *if p then q and if q then p*. The meaning just given is called *material equivalence* by logicians. It is the only meaning used in mathematics. Material equivalence is the "equality" of propositional logic.

1.1 Syntax of Propositional Logic.

In this section we begin our study of a formal language (or more precisely a class of formal languages) called *propositional logic*.

A *vocabulary for propositional logic* is a non-empty set $\mathcal{P}_0$ of symbols; the elements of the set $\mathcal{P}_0$ are called *proposition symbols* and denoted by lower case letters $p, q, r, s, p_1, q_1, \ldots$. In the standard semantics[1] of propositional logic the proposition symbols will denote propositions such as $2+2 = 4$ or $2+2 = 5$. Propositional logic is not concerned with any internal structure these propositions may have; indeed, for us the only meaning a proposition symbol may take is a truth value – either *true* or *false* in the standard semantics.

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

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

Any finite sequence of these symbols is called a *string*. Or first task is to specify the *syntax* of propositional logic; i.e. which strings are grammatically correct. These strings are called *well-formed formulas*. The phrase

[1] In studying formal languages it is often useful to employ other semantics besides the "standard" one.)

well-formed formula is often abbreviated to *wff* (or a **wff built using the vocabulary** $\mathcal{P}_0$ if we wish to be specific about exactly which proposition symbols may appear in the formula). The set of *wffs of propositional logic* is then inductively defined [2] by the following rules:

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

(W:$\neg$) If $\mathbf{A}$ is a wff, then $\neg\mathbf{A}$ is a wff;

(W:$\wedge, \vee, \Rightarrow, \Leftrightarrow$) If $\mathbf{A}$ and $\mathbf{B}$ are wffs, then $[\mathbf{A} \wedge \mathbf{B}]$, $[\mathbf{A} \vee \mathbf{B}]$, $[\mathbf{A} \Rightarrow \mathbf{B}]$, and $[\mathbf{A} \Leftrightarrow \mathbf{B}]$ are wffs.

To show that a particular string is a wff we construct a sequence of strings using this definition. This is called *parsing* the wff and the sequence is called a *parsing sequence*. Although it is never difficult to tell if a short string is a wff, the parsing sequence is important for theoretical reasons. For example we parse the wff $[\neg p \Rightarrow [q \wedge p]]$.

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

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

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

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

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

In order to make our formulas more readable, we shall introduce certain abbreviations and conventions.

- The outermost brackets will not be written. For example, we write $p \Leftrightarrow [q \vee r]$ instead of $[p \Leftrightarrow [q \vee r]]$.
- For the operators $\wedge$ and $\vee$ association to the left is assumed. For example, we write $p \vee q \vee r$ instead of $[p \vee q] \vee r$.
- The rules give the operator $\neg$ the highest precedence: (so that $\neg p \wedge q$ abbreviates $[\neg p \wedge q]$ and not $\neg[p \wedge q]$ but we may insert extra brackets to remind the reader of this fact. For example, we might write $[\neg p] \wedge q$ instead of $\neg p \wedge q$.

[2] i.e. the set of wffs is the smallest set of strings satisfying the conditions (W:*).

- The list: $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$ exhibits the binary connectives in order of decreasing precedence. In other words, $p \wedge q \vee r$ means $[p \wedge q] \vee r$ and not $p \wedge [q \vee r]$ since $\wedge$ has a higher precedence than $\vee$.

- In addition to omitting the outermost brackets, we may omit the next level as well replacing them with a dot on either side of the principle connective. Thus we may write $p \Rightarrow q. \Rightarrow .\neg q \Rightarrow p$ instead of $[p \Rightarrow q] \Rightarrow [\neg q \Rightarrow p]$

1.2 Semantics of Propositional Logic.

Recall that $\mathcal{P}_0$ is the vocabulary of all proposition symbols of propositional logic; let $WFF(\mathcal{P}_0)$ denote the set of all wffs of propositional logic built using this vocabulary. We let $\top$ denote the truth value *true* and $\bot$ denote the truth value *false*.

A *model* $\mathcal{M}$ **for propositional logic of type** $\mathcal{P}_0$ is simply a function

$$\mathcal{M} : \mathcal{P}_0 \longrightarrow \{\top, \bot\} \ : \ p \mapsto p_{\mathcal{M}}$$

i.e. a specification of a truth value for each proposition symbol. This function maps p to $p_{\mathcal{M}}$. Or we say $\mathcal{M}$ interprets p to be $p_{\mathcal{M}}$.

We shall extend this function to a function

$$\mathcal{M} : WFF(\mathcal{P}_0) \longrightarrow \{\top, \bot\} : .$$

And write $\mathbf{A}_{\mathcal{M}}$ for the value $\mathcal{M}(\mathbf{A})$.

We write

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

(which is read "$\mathcal{M}$ **models** $\mathbf{A}$" or "$\mathbf{A}$ is true in $\mathcal{M}$") as an abbreviation for the equation $\mathbf{A}_{\mathcal{M}} = \top$; i.e. $\mathcal{M} \models \mathbf{A}$ means that $\mathbf{A}$ is true in the model $\mathcal{M}$. We write

$$\mathcal{M} \not\models \mathbf{A}$$

(which is read " $\mathbf{A}$ is false in $\mathcal{M}$") in the contrary case $\mathbf{A}_{\mathcal{M}} = \bot$; i.e. $\mathcal{M} \not\models \mathbf{A}$ means that $\mathbf{A}$ is false in the model $\mathcal{M}$. Then the assertion $\mathcal{M} \models \mathbf{A}$ is defined inductively by

(M:$\mathcal{P}_0$)	$\mathcal{M} \models \mathbf{p}$	if $\mathbf{p} \in \mathcal{P}_0$ and $\mathbf{p}_{\mathcal{M}} = \top$;
	$\mathcal{M} \not\models \mathbf{p}$	if $\mathbf{p} \in \mathcal{P}_0$ and $\mathbf{p}_{\mathcal{M}} = \bot$.
(M:$\neg$)	$\mathcal{M} \models \neg\mathbf{A}$	if $\mathcal{M} \not\models \mathbf{A}$;
	$\mathcal{M} \not\models \neg\mathbf{A}$	if $\mathcal{M} \models \mathbf{A}$.
(M:$\wedge$)	$\mathcal{M} \models \mathbf{A} \wedge \mathbf{B}$	if $\mathcal{M} \models \mathbf{A}$ and $\mathcal{M} \models \mathbf{B}$;
	$\mathcal{M} \not\models \mathbf{A} \wedge \mathbf{B}$	otherwise.
(M:$\vee$)	$\mathcal{M} \not\models \mathbf{A} \vee \mathbf{B}$	if $\mathcal{M} \not\models \mathbf{A}$ and $\mathcal{M} \not\models \mathbf{B}$;
	$\mathcal{M} \models \mathbf{A} \vee \mathbf{B}$	otherwise.
(M:$\Rightarrow$)	$\mathcal{M} \not\models \mathbf{A} \Rightarrow \mathbf{B}$	if $\mathcal{M} \models \mathbf{A}$ and $\mathcal{M} \not\models \mathbf{B}$;
	$\mathcal{M} \models \mathbf{A} \Rightarrow \mathbf{B}$	otherwise.
(M:$\Leftrightarrow$)	$\mathcal{M} \models \mathbf{A} \Leftrightarrow \mathbf{B}$	if either $\mathcal{M} \models \mathbf{A} \wedge \mathbf{B}$
		or else $\mathcal{M} \models \neg\mathbf{A} \wedge \neg\mathbf{B}$;
	$\mathcal{M} \not\models \mathbf{A} \Leftrightarrow \mathbf{B}$	otherwise.

Notice how the semantical rules (M:*) parallel the syntactical rules (W:*). To find the value of $\mathbf{A}$ in a model $\mathcal{M}$ we must apply the inductive definition of $\mathcal{M} \models \mathbf{A}$ to the parsing sequence which constructs the wff $\mathbf{A}$. For example, let us compute the value of $[p \Rightarrow \neg q] \Rightarrow [q \vee p]$ for a model $\mathcal{M}$ satisfying $p_{\mathcal{M}} = \top$ and $q_{\mathcal{M}} = \bot$. We first parse the wff.

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

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

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

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

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

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

Now we apply the definition of $\mathcal{M} \models \mathbf{A}$ to this construction:

(1) $\mathcal{M} \models p$ by (M:$\mathcal{P}_0$) and $p_{\mathcal{M}} = \top$.

(2) $\mathcal{M} \not\models q$ by (M:$\mathcal{P}_0$) and $q_{\mathcal{M}} = \bot$.

(3) $\mathcal{M} \models \neg q$ by (2) and (M:$\neg$).

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

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

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

The following easy proposition will motivate the definition of propositional tableaus.

Proposition 1.2.1 *Let $\mathcal{M}$ be a model for propositional logic and* $\mathbf{A}$ *and* $\mathbf{B}$ *be wffs. Then:*

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

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

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

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

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

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

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

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

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

1.3 Truth Tables

The evaluation of $\mathcal{M} \models \mathbf{A}$ is so routine that we can arrange the work in a table. First let us summarize our semantical rules in tabular form:

$\mathbf{A}$	$\neg\mathbf{A}$
$\top$	$\bot$
$\bot$	$\top$

and

$\mathbf{A}$	$\mathbf{B}$	$\mathbf{A} \wedge \mathbf{B}$	$\mathbf{A} \vee \mathbf{B}$	$\mathbf{A} \Rightarrow \mathbf{B}$	$\mathbf{A} \Leftrightarrow \mathbf{B}$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$
$\bot$	$\top$	$\bot$	$\top$	$\top$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\top$

Now we can evaluate $\mathcal{M} \models \mathbf{A}$ by the following strategy. We first write the wff down and underneath each occurrence of a proposition symbol write its value thus:

$$\begin{array}{ccccccccc} [p & \Rightarrow & \neg & q] & \Rightarrow & [q & \vee & p] \\ \top & & & \bot & & \bot & & \top \end{array}$$

and then we fill in the value of each formula on the parsing sequence under its corresponding principal connective thus:

$$\begin{array}{ccccccccc} [p & \Rightarrow & \neg & q] & \Rightarrow & [q & \vee & p] \\ \top & \top & \top & \bot & \top & \bot & \top & \top \end{array}$$

A wff $\mathbf{A}$ is called a *tautology* if it is true in *every* model: $\mathcal{M} \models \mathbf{A}$ for every model $\mathcal{M}$. To check if $\mathbf{A}$ is a tautology, we can make a *truth table* which computes the value of $\mathbf{A}$ in every possible model (each row of the truth table corresponds to a different model). The truth table will have 2^n rows if $\mathbf{A}$ contains exactly n distinct proposition symbols. For example,

$$\begin{array}{ccccccccc} [p & \Rightarrow & \neg & q] & \Rightarrow & [q & \vee & p] \\ \top & \bot & \bot & \top & \top & \top & \top & \top \\ \top & \top & \top & \bot & \top & \bot & \top & \top \\ \bot & \top & \bot & \top & \top & \top & \top & \bot \\ \bot & \top & \top & \bot & \bot & \bot & \bot & \bot \end{array}$$

The entries in the fifth column give the values for the whole wff and as the last of these is $\bot$ the wff is *not* a tautology.

Here is a tautology:

$\neg$	p	$\Rightarrow$	$[p$	$\Rightarrow$	$q]$
$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$
$\bot$	$\top$	$\top$	$\top$	$\bot$	$\bot$
$\top$	$\bot$	$\top$	$\bot$	$\top$	$\top$
$\top$	$\bot$	$\top$	$\bot$	$\top$	$\bot$

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

$\neg$	$\mathbf{A}$	$\Rightarrow$	$[\mathbf{A}$	$\Rightarrow$	$\mathbf{B}]$
$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$
$\bot$	$\top$	$\top$	$\top$	$\bot$	$\bot$
$\top$	$\bot$	$\top$	$\bot$	$\top$	$\top$
$\top$	$\bot$	$\top$	$\bot$	$\top$	$\bot$

This is because the wffs $\mathbf{A}$ and $\mathbf{B}$ can only take the values $\top$ and $\bot$ just like the proposition symbols p and q.

1.4 Induction on formulas and unique readability

In this section (unless we mention otherwise) by formula we mean unsimplified fully-bracketed well-formed formula.

PRINCIPLE OF INDUCTION ON FORMULAS. To prove that all formulas have a given property PROP:

- Show that every $p \in \mathcal{P}$ has the property PROP.
- Show that if $\mathbf{A}$is formula with property PROP then $\neg\mathbf{A}$ has PROP.
- Show that if $\mathbf{A}$and $\mathbf{B}$are formula with property PROP and * is one of $\{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$ then $[\mathbf{A} * \mathbf{B}]$ has property PROP.

This principle can be proved by using ordinary induction on the natural numbers (see section 5.8).

Lemma 1.4.1 *Every formula has the same number of [’s as]’s.*

This is clear for every $p \in \mathcal{P}$ since they contain no ['s or]'s. The inductive step in the case of $\neg$ does not add any brackets and in the case of $[\mathbf{A} * \mathbf{B}]$ one of each is added to the total number in **A**and **B**.

Lemma 1.4.2 *In every formula, every binary connective is preceded by more ['s than]'s.*

Here the basis step is true since atomic formulas contain no binary connectives. The inductive step $[\mathbf{A} * \mathbf{B}]$ since any binary connective occurring in **A**has one more [occurring before it in $[\mathbf{A} * \mathbf{B}]$ than it does in **A**. The main connective * is ok since **A**and **B**have a balanced set of brackets by the preceding lemma. And the binary connectives of **B**are ok (with respect to $[\mathbf{A} * \mathbf{B}]$) since **A**has a balanced set of brackets.

Theorem 1.4.3 *UNIQUE READABILITY. Each propositional formula C is either a*

*propositional symbol, is of the form $\neg\mathbf{A}$, or can be uniquely written in the form $[\mathbf{A} * \mathbf{B}]$ where **A**and **B**are formulas and * is a binary connective.*

The $\neg$ or * in this theorem is called the main connective of C. The cases that C is a proposition symbol or starts with $\neg$ is easy. To prove the theorem suppose that C is $[\mathbf{A} * \mathbf{B}]$, and there is another way to read C say $[\mathbf{A}' *' \mathbf{B}']$. Then either **A**is a proper initial segment of $\mathbf{A}'$ or vice-versa. Without loss of generality assume that **A**is a proper initial segment of **A**'. But then * is a binary connective which occurs somewhere in **A**' and so must be preceded by unbalanced brackets, but this contradicts the fact that the brackets must be balanced in **A**.

> The proof shows that the main connective of a formula can be picked out simply by finding the unique binary connective such that brackets are balanced on either side of it.

This result shows that the brackets do their job. Its need can be explained by seeing that the definition of truth value is ambiguous without brackets. For example, $p \vee q \wedge r$ is not well-defined since if the model M is defined by $\mathcal{M}(p) = \top$, $\mathcal{M}(q) = \top$, and $\mathcal{M}(r) = \bot$:

$$\mathcal{M} \models [p \vee [q \wedge r]]$$

but

$$\mathcal{M} \models \neg[[p \vee q] \wedge r].$$

In the simplified form obtained by dropping brackets the corresponding rule for picking out the main connective of **A**is the following:

> The main connective of **A**is the rightmost connective of lowest precedence which is preceded by an equal number of [' s as]'s in the simplified form of **A**. (This is again proved by induction, of course).

The definition of truth value is an example of definition by induction.

PRINCIPLE OF DEFINITION BY INDUCTION. A function with domain WFF can be defined uniquely by giving:

- A value f(p) for each $p \in \mathcal{P}_0$;
- A rule for computing f($\neg$**A**) from f(**A**);
- For each binary connective *, a rule for computing f([**A*** **B**]) from f(**A**) and f(**B**).

The abbreviated form of **A**(dropping parentheses) is another example. Another example is in the first exercise set. Other good examples are the tree height of a formula, and the result of substituting a formula **A**for a predicate symbol p in a formula. The height(**A**) is defined by

- height(p)=1 for any $p \in \mathcal{P}_0$
- height($\neg$**A**)=height(**A**)+1
- height([**A** $*$ **B**])=max(height(**A**),height(**B**))+1

The following principle is more general and corresponds to the strong form of induction on natural numbers:

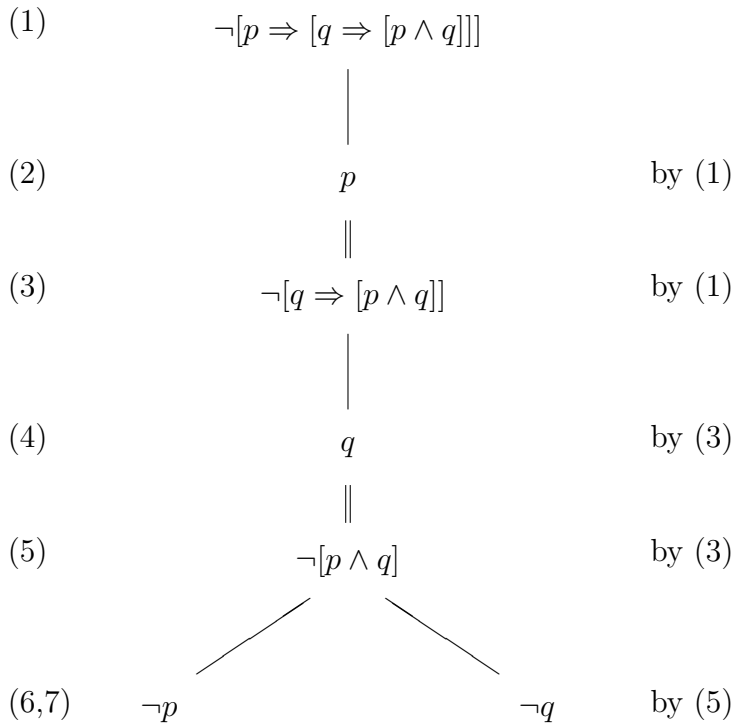
PRINCIPLE STRONG OF INDUCTION ON FORMULAS. To prove that all formulas have a given property PROP:

- Show that every $p \in \mathcal{P}_0$ has the property PROP.
- Show that for any formula **A**if all formulas of length less than **A**have property PROP; then **A**has property PROP.

1.5 Tableaus.

Often one can see very quickly (without computing the full truth table) whether some particular wff is a tautology by using an indirect argument. As an example we show that the wff $p \Rightarrow [q \Rightarrow [p \wedge q]]$ is a tautology. If not, there is a model $\mathcal{M}$ for its negation, i.e. (1) $\mathcal{M} \models \neg[p \Rightarrow [q \Rightarrow [p \wedge q]]$. From (1) we obtain (2) $\mathcal{M} \models p$ and (3) $\mathcal{M} \models \neg[q \Rightarrow [p \wedge q]$. From (3) we obtain (4) $\mathcal{M} \models q$ and (5) $\mathcal{M} \models \neg[p \wedge q]$. From (5) we conclude that either (6) $\mathcal{M} \models \neg p$ or else (7) $\mathcal{M} \models \neg q$. But (6) contradicts (2) and (7) contradicts (4). Thus no such model $\mathcal{M}$ exists; i.e. the wff $p \Rightarrow [q \Rightarrow [p \wedge q]]$ is a tautology as claimed.

We can arrange this argument in a diagram (called a *tableau*):



The steps in the original argument appear at nodes of the tableau. The number to the left of a formula is its step number in the argument; the number to the right is the number of the earlier step which justified the

given step. The two branches in the tree at node (5) correspond to the two possibilities in the case analysis. There are two ways to move from formula (1) down to the bottom of the diagram: viz. (1)-(2)-(3)-(4)-(5)-(6) and (1)-(2)-(3)-(4)-(5)-(7); along each of these two ways there occurs a formula and its negation: viz (2) and (6) for the former way and (4) and (7) for the latter.

Before explaining the method of tableaus precisely we generalize slightly our problem. If $\mathbf{H}$ is a set of wffs and $\mathcal{M}$ is a model we shall say $\mathcal{M}$ **models** $\mathbf{H}$ (or $\mathcal{M}$ is a model of $\mathbf{H}$) and write $\mathcal{M} \models \mathbf{H}$ iff $\mathcal{M}$ models every element $\mathbf{A}$ of $\mathbf{H}$:

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

Of course, when $\mathbf{H}$ is a finite set, say $\mathbf{H} = \{\mathbf{A}_1, \mathbf{A}_2, \ldots, \mathbf{A}_n\}$ then $\mathcal{M} \models \mathbf{H}$ if and only if $\mathcal{M}$ models the conjunction: $\mathcal{M} \models \mathbf{A}_1 \wedge \mathbf{A}_2 \wedge \ldots \wedge \mathbf{A}_n$ but the notation is handy (especially for infinite sets $\mathbf{H}$).

A set of sentences $\mathbf{H}$ is called *semantically consistent* iff it has a model; i.e. $\mathcal{M} \models \mathbf{H}$ for some $\mathcal{M}$. Now a wff $\mathbf{A}$ is a tautology if and only if the set $\{\neg\mathbf{A}\}$ consisting of the single wff $\neg\mathbf{A}$ is inconsistent so we shall study the problem of deciding if a given finite set $\mathbf{H}$ of wffs is consistent rather than the special case of deciding whether a given wff is a tautology.

A wff $\mathbf{A}$ is called a *semantic consequence* of the set of wffs $\mathbf{H}$ iff every model of $\mathbf{H}$ is a model of $\mathbf{A}$. Evidently, $\mathbf{A}$ is a semantic consequence of $\mathbf{H}$ if and only if the set $\mathbf{H} \cup \{\neg\mathbf{A}\}$ is semantically inconsistent.

A *tree* $\mathbf{T}$ is a system consisting of a set of points called the *nodes* of the tree, a distinguished node $r_{\mathbf{T}}$ called the *root* of the tree, and a function π, or $\pi_{\mathbf{T}}$, which assigns to each node t distinct from the root another node $\pi(t)$ called the *parent* of t; it is further required that for each node t the sequence of nodes

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

defined inductively by

$$\pi^0(t) = t$$

and

$$\pi^{k+1}(t) = \pi(\pi^k(t))$$

terminates for some n at the root:

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

The nodes $\pi^1(t), \pi^2(t), \pi^3(t), \ldots$ are called the *proper ancestors* of t; a node t' is an *ancestor* of t iff it is either t itself or is a proper ancestor of t. Thus the root is an ancestor of every node (including itself). Conversely, the nodes s whose parent is t are called the *children* of t; the set of children of t is denoted by $\pi^{-1}(t)$:

$$\pi^{-1}(t) = \{s : \pi(s) = t\}.$$

A node of the tree which is not the parent of any other node (i.e. not in the range of the function π) is called a *terminal node*. A node is terminal if and only if it has no children.

A sequence

$$\mathbf{\Gamma} = (t, \pi(t), \pi^2(t), \ldots, r_{\mathbf{T}})$$

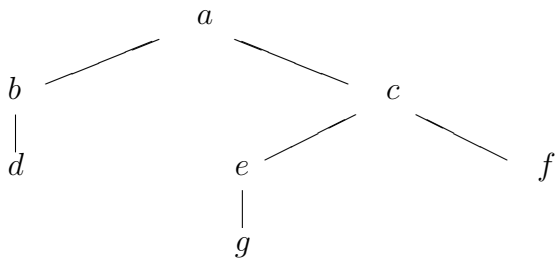
starting at a terminal node t is called a *branch* or *finite branch*. Thus finite branches and terminal nodes are in one-one correspondence.

In a tree with infinitely many nodes we may have an infinite branch. An infinite sequence

$$\mathbf{\Gamma} = (p_0, p_1, p_2, \ldots)$$

is an *infinite branch* is an infinite branch if p_0 is the root node and for every n p_{n+1} is a child of p_n.

It is customary to draw trees upside down with the root at the top and each node connected to its parent by a line. For example, in the tree



the root is a; the parent function is defined by $\pi(b) = \pi(c) = a$, $\pi(d) = b$, $\pi(e) = \pi(f) = c$, $\pi(g) = e$; the children of the various are given by $\pi^{-1}(a) = \{b, c\}$, $\pi^{-1}(b) = \{d\}$, $\pi^{-1}(c) = \{e, f\}$, $\pi^{-1}(e) = \{g\}$, $\pi^{-1}(d) = \pi^{-1}(e) = \pi^{-1}(g) = \emptyset$; the terminal nodes are d, f, g; and the branches are (d, b, a), (f, c, a), (g, e, c, a).

By a *labeled tree for propositional logic* we shall mean a system $(\mathbf{T}, \mathbf{H}, \Phi)$ consisting of a tree $\mathbf{T}$, a set of wffs $\mathbf{H}$ which is called the set of *hypotheses*,

and a function Φ which assigns to each nonroot node t a wff $\Phi(t)$. In order to avoid excessive mathematical notation we shall say "$\mathbf{A}$ occurs at t" or even "$\mathbf{A}$ is t" in place of the equation $\mathbf{A} = \Phi(t)$. We will also denote the labeled tree by $\mathbf{T}$ rather than the more cumbersome $(\mathbf{T}, \mathbf{H}, \Phi)$,

A wff which which occurs at a child of a node t will be called a *child wff* (or simply *child*) of t, and a wff at a child of a child of t will be called a *grandchild wff* of t. A wff which is either in the hypothesis set $\mathbf{H}$ or else occurs at some ancestor node of t is called an *ancestor wff* of t or simply an *ancestor* of t.

We call the labeled tree $\mathbf{T}$ a *propositional tableau* [3] iff at each nonterminal node t one of the following conditions holds:

$\boxed{\neg\neg}$ t has an ancestor $\neg\neg\mathbf{A}$ and a child $\mathbf{A}$.

$\boxed{\wedge}$ t or its parent has an ancestor $\mathbf{A} \wedge \mathbf{B}$ a child $\mathbf{A}$ and grandchild $\mathbf{B}$.

$\boxed{\neg\wedge}$ t has an ancestor $\neg[\mathbf{A} \wedge \mathbf{B}]$ and two children $\neg\mathbf{A}$ and $\neg\mathbf{B}$.

$\boxed{\vee}$ t has an ancestor $\mathbf{A} \vee \mathbf{B}$ and two children $\mathbf{A}$ and $\mathbf{B}$.

$\boxed{\neg\vee}$ t or its parent has an ancestor $\neg[\mathbf{A} \vee \mathbf{B}]$ a child $\neg\mathbf{A}$ and grandchild $\neg\mathbf{B}$.

$\boxed{\Rightarrow}$ t has an ancestor $\mathbf{A} \Rightarrow \mathbf{B}$ and two children $\neg\mathbf{A}$ and $\mathbf{B}$.

$\boxed{\neg\Rightarrow}$ t or its parent has an ancestor $\neg[\mathbf{A} \Rightarrow \mathbf{B}]$ a child $\mathbf{A}$ and grandchild $\neg\mathbf{B}$.

$\boxed{\Leftrightarrow}$ t has an ancestor $[\mathbf{A} \Leftrightarrow \mathbf{B}]$ and two children $\mathbf{A} \wedge \mathbf{B}$ and $\neg\mathbf{A} \wedge \neg\mathbf{B}$.

$\boxed{\neg\Leftrightarrow}$ t has an ancestor $\neg[\mathbf{A} \Leftrightarrow \mathbf{B}]$ and two children $\mathbf{A} \wedge \neg\mathbf{B}$ and $\neg\mathbf{A} \wedge \mathbf{B}$.

In each case, the ancestor wff is called the node *used* at t and the other wffs mentioned are called the nodes *added* at t. What this definition says is this. To build a propositional tableau start with a tree $\mathbf{T}_0$ consisting of a

[3] This definition is summarized in section D.

single node (its root) and a set **H** hypotheses. Then extend the tableau $\mathbf{T}_0$ to a tableau $\mathbf{T}_1$ and extend $\mathbf{T}_1$ to $\mathbf{T}_2$ and so on. Each extension uses one of a set of nine rules which apply to a propositional tableau **T** to extend it to a larger propositional tableau $\mathbf{T}'$. These rules involve choosing a terminal node t of **T** and a formula **C** which appears on the branch through t. Depending on the structure of the wff **C** we extend the tree **T** by adjoining either one child, one child and one grandchild, or two children of t. At the new node or nodes we introduce one or two wffs which are well formed subformulas of the wff **C**.

For reference we have summarized the nine extension rules in Figure 1. This figure shows the node t and a formula **C** above it; the vertical dots indicate the branch of the tableau through t so the figure shows **C** on this branch. (It is not precluded that **C** be at t itself.) Below t in the figure are the wffs at the children of t, and when appropriate the grandchild of t. When both child and grandchild are added together in a single rule, they are connected by a double line. Thus the $\boxed{\neg\neg}$ rule yields one child, the $\boxed{\wedge}$, $\boxed{\neg\vee}$, and $\boxed{\neg\Rightarrow}$ rules each yield one child and one grandchild, and the remaining five rules yield two children.

Any finite tableau within the memory limits of the computer can be built with the TABLEAU program. The Why command shows which formula was used at each node of the tableau.

1.6 Soundness and Confutation

We say that a formula **A** is along or on a branch $\mathbf{\Gamma}$ if **A** is either a hypothesis (hence attached to the root node) or is attached to a node of $\mathbf{\Gamma}$.

The definition of propositional tableau has been designed so that the following important principle is obvious.

Theorem 1.6.1 (The Soundness Principle.) *Let* **T** *be a propositional tableau with hypothesis set* **H**. *Let* $\mathcal{M}$ *be a propositional model of the hypothesis set* **H**: $\mathcal{M} \models \mathbf{H}$. *Then there is a branch* $\mathbf{\Gamma}$ *such that* $\mathcal{M} \models \mathbf{\Gamma}$, *that is,* $\mathcal{M} \models \mathbf{A}$ *for every wff* **A** *along* $\mathbf{\Gamma}$.

Now call a branch $\mathbf{\Gamma}$ of a tableau *contradictory* iff for some wff **A** both **A** and $\neg\mathbf{A}$ occur along the branch. When every branch of a propositional tableau **T** with hypothesis set **H** is contradictory we say that the tableau is a *confutation* of the hypothesis set **H**.

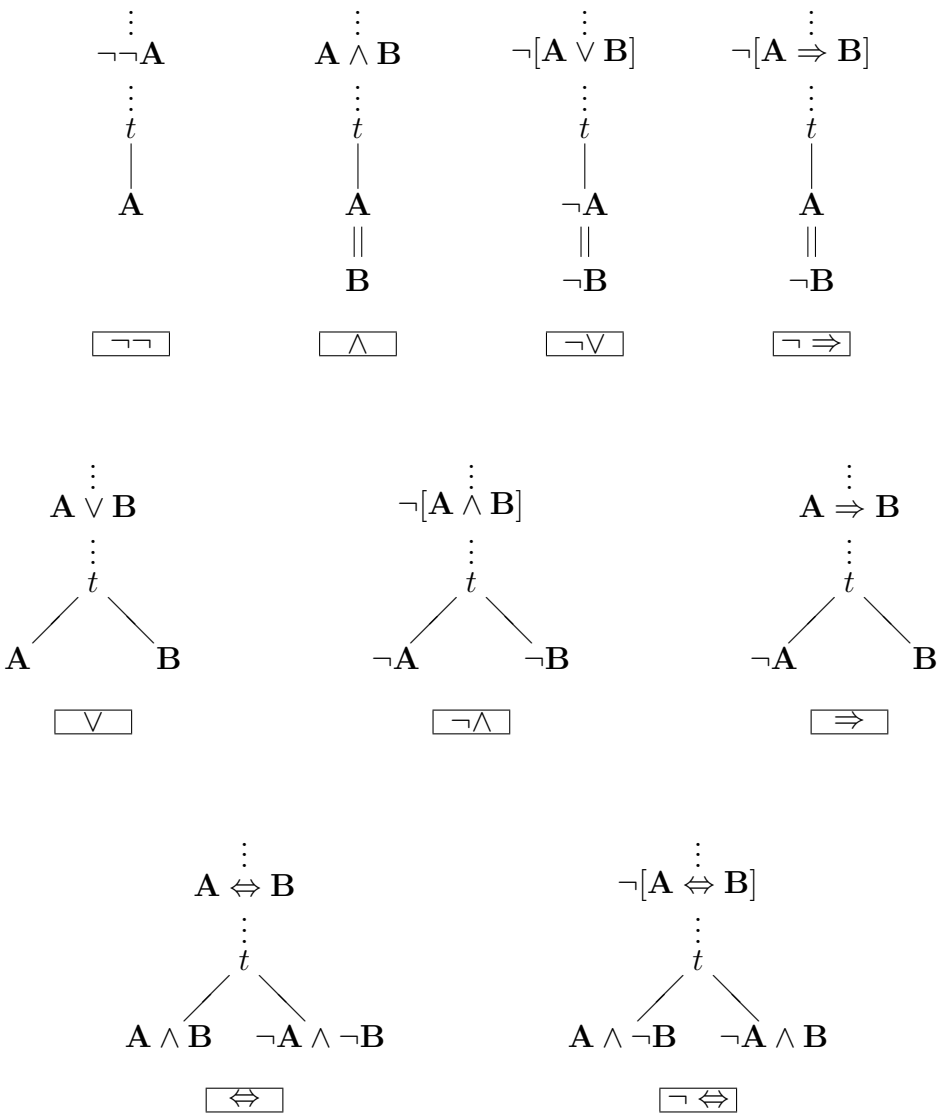


Figure 1: Propositional Extension Rules.

In the TABLEAU program, a node is colored red if every branch through the node is contradictory. In a confutation every node is colored red.

Since we cannot have both $\mathcal{M} \models \mathbf{A}$ and $\mathcal{M} \models \neg\mathbf{A}$ it follows that $\mathcal{M} \not\models \mathbf{\Gamma}$ for any contradictory branch $\mathbf{\Gamma}$. Hence

Corollary 1.6.2 *If a set* $\mathbf{H}$ *of propositional wffs has a tableau confutation, then* $\mathbf{H}$ *has no models, that is,* $\mathbf{H}$ *is semantically inconsistent.*

A tableau confutation can be used to show that a propositional wff is a tautology. Remember that a propositional wff $\mathbf{A}$ is a tautology iff it is true in every model, and also iff $\neg\mathbf{A}$ is false in every model. Thus if the one-element set $\{\neg\mathbf{A}\}$ has a confutation, then $\mathbf{A}$ is a tautology. By a *tableau proof* of $\mathbf{A}$ we shall mean a tableau confutation of $\neg\mathbf{A}$. More generally, by a **tableau proof of A from the set of hypotheses H** we shall mean a tableau confutation of the set $\mathbf{H} \cup \{\neg\mathbf{A}\}$. Thus to build a tableau proof of $\mathbf{A}$ from $\mathbf{H}$ we add the new hypothesis $\neg\mathbf{A}$ to $\mathbf{H}$ and build a tableau confutation of the new set $\mathbf{H} \cup \{\neg\mathbf{A}\}$.

Corollary 1.6.3 *If a propositional wff* $\mathbf{A}$ *has a tableau proof, then* $\mathbf{A}$ *is a tautology. If a propositional wff* $\mathbf{A}$ *has a tableau proof from a set of propositional wffs* $\mathbf{H}$*, then* $\mathbf{A}$ *is a semantic consequence of* $\mathbf{H}$*, that is,* $\mathbf{A}$ *is true in every model of* $\mathbf{H}$*.*

1.7 Completeness

We have seen that the tableau method of proof is *sound:* any set $\mathbf{H}$ of propositional wffs which has a tableau confutation is semantically inconsistent. In this section we prove the converse statement that the tableau method is *complete:* any semantically inconsistent set has a tableau confutation.

Theorem 1.7.1 *(Completeness Theorem.) Let* $\mathbf{H}$ *be a finite set of propositional wffs. If* $\mathbf{H}$ *is semantically inconsistent (that is,* $\mathbf{H}$ *has no models), then* $\mathbf{H}$ *has a tableau confutation.*

Theorem 1.7.2 *(Completeness Theorem, Second Form.) If a wff* $\mathbf{A}$ *is a semantic consequence of a set of wffs* $\mathbf{H}$*, then there is a tableau proof of* $\mathbf{A}$ *from* $\mathbf{H}$*.*

The completeness theorem can also be stated as follows: If $\mathbf{H}$ does not have a tableau confutation, then $\mathbf{H}$ has a model.

To prove the completeness theorem we need a method of constructing a model of a set of wffs $\mathbf{H}$. To do this, we shall introduce the concept of a finished branch and prove two lemmas. Recall that a branch of a tableau is called contradictory, iff it contains a pair of wffs of the form $\neg\mathbf{A}$ and $\mathbf{A}$. (In the TABLEAU program, a branch is contradictory if its terminal node is red.)

By a *basic wff* we shall mean a propositional symbol or a negation of a propositional symbol. The basic wffs are the ones which cannot be broken down into simpler wffs by the rules for extending tableaus.

A branch $\mathbf{\Gamma}$ of a tableau is said to be *finished* iff $\mathbf{\Gamma}$ is not contradictory and every nonbasic wff on $\mathbf{\Gamma}$ is used at some node of $\mathbf{\Gamma}$. (In the TABLEAU program, a branch $\mathbf{\Gamma}$ is finished iff its terminal node is yellow and each node of $\mathbf{\Gamma}$ is either a basic wff or is shown by the *Why* command to be invoked at some other node of $\mathbf{\Gamma}$.)

Lemma 1.7.3 *(Finished Branch Lemma.) Let $\mathbf{\Gamma}$ be a finished branch of a propositional tableau. Then any model of the set of basic wffs on $\mathbf{\Gamma}$ is a model of all the wffs on $\mathbf{\Gamma}$.*

Let $\mathcal{M}$ be a model of all basic wffs on $\mathbf{\Gamma}$. The proof is to show by strong induction on formulas that for any formula $\mathbf{C}$

$$\text{if } \mathbf{C} \text{ is on } \mathbf{\Gamma} \text{ then } \mathcal{M} \models \mathbf{C}$$

Let

$$\mathbf{\Delta} = \mathbf{\Gamma} \cup \mathbf{H}$$

be the set of formulas which occur along the branch. Then $\mathbf{\Delta}$ is not contradictory (i.e. contains no pair of wffs $\mathbf{A}$, $\neg\mathbf{A}$) and for each wff $\mathbf{C} \in \mathbf{\Delta}$ either $\mathbf{C}$ is basic or one of the following is true:

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

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

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

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

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

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

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

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

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

When $\mathbf{C}$ is basic, this is true by the hypothesis of the lemma. In any other case, one of the nine cases $[\neg\neg] \ldots [\neg \Leftrightarrow]$ applies and by the hypothesis of induction $\mathcal{M} \models \mathbf{A}$ and/or $\mathcal{M} \models \mathbf{B}$ provided $\mathbf{A} \in \mathbf{\Delta}$ and/or $\mathbf{B} \in \mathbf{\Delta}$. But then (applying the appropriate case) it follows that $\mathcal{M} \models \mathbf{C}$ by the definition of $\models$. This proves the finished branch lemma.

It follows from the finished branch lemma that every finished branch has a model. Since the set of hypotheses is contained in any branch of the tableau, any set $\mathbf{H}$ of wffs which has a tableau with at least one finished branch has a model.

Define a propositional tableau $\mathbf{T}$ to be *finished* iff every branch of $\mathbf{T}$ is either contradictory or finished. Thus a finished tableau for $\mathbf{H}$ is either a confutation of $\mathbf{H}$ (i.e. all its branches are contradictory) or else (one of its branches is not contradictory) has a finished branch whose basic wffs determine a model of $\mathbf{H}$.

Lemma 1.7.4 *(Extension Lemma.) For every finite hypothesis set* $\mathbf{H}$ *there exists a finite finished tableau with* $\mathbf{H}$ *attached to its root node.*

First let us assume that our hypothesis set consists of a single formula, say $\mathbf{H} = \{\mathbf{A}\}$. We prove the lemma in this case by using the principle of strong induction on formulas. If $\mathbf{A}$ is a propositional symbol or the negation of a propositional symbol (i.e. basic wff), then the tableau with a single node , the root node, is already finished.

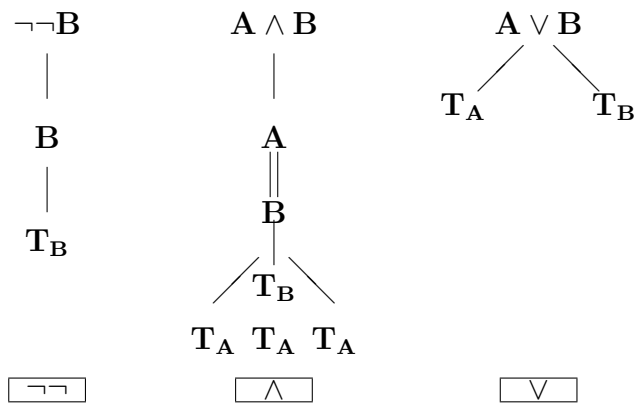


Figure 2: Proof of Extension Lemma

The inductive case corresponds to the nine cases illustrated in figure 1. For example if $\mathbf{A} = \neg\neg\mathbf{B}$ then by induction we have a finished tableau for $\mathbf{B}$, say $\mathbf{T_B}$. Then the finished tableau for $\mathbf{A}$ would be as in figure 2 $\boxed{\neg\neg}$.

If $\mathbf{H} = \{\mathbf{A} \wedge \mathbf{B}\}$, then by induction we have a finished tableau for $\mathbf{A}$, say $\mathbf{T_A}$, and also one for $\mathbf{B}$, say $\mathbf{T_B}$. Then to build the finished tableau for $\{\mathbf{A} \wedge \mathbf{B}\}$ simply hang $\mathbf{T_A}$ on every terminal node of $\mathbf{T_B}$ (see figure 2 $\boxed{\wedge}$).

If $\mathbf{H} = \{\mathbf{A} \vee \mathbf{B}\}$, then we take the finished tableaus $\mathbf{T_A}$ and $\mathbf{T_B}$ (which have at their root nodes the formulas $\mathbf{A}$ and $\mathbf{B}$) and hang them beneath $\{\mathbf{A} \vee \mathbf{B}\}$ (see figure 2 $\boxed{\vee}$).

The proofs of the other cases are similar to this one. The proof for the case that $\mathbf{H}$ is not a single formula is similar the $\wedge$ case[4] above.

The next two results will be used to prove the compactness theorem.

Theorem 1.7.5 *(Koenig Tree Lemma.) If a tree has infinitely many nodes and each node has finitely many children, then the tree has an infinite branch.*

To prove this inductively construct an infinite set of nodes $p_0, p_1, p_2, \ldots$ with properties

1. p_0 is the root node;

[4]Actually there is a glitch in the case of $\boxed{\Leftrightarrow}$. This is because one of the formulas which replaces $\mathbf{A} \Leftrightarrow \mathbf{B}$ is $\neg\mathbf{A} \wedge \neg\mathbf{B}$ which is two symbols longer. One kludge to fix this is to regard the symbol $\Leftrightarrow$ as having length four instead of one.

2. p_{n+1} is a child of p_n; and

3. each p_n has infinitely many nodes beneath it;

Given p_n with infinitely many nodes beneath it, note that one of its children must also have infinitely many nodes beneath it (since an infinite set cannot be the union of finitely many finite sets). Let p_{n+1} be any of these.

Lemma 1.7.6 *Every finite propositional tableau can be extended to a finite finished tableau.*

An algorithm for doing this is illustrated by the computer program COMPLETE. Construct a sequence of tableaus by starting with the given tableau (in which every node has been colored unused). At each step of the algorithm select any unused node color it used and extend the tableau on every branch below the selected node which is not contradictory (and color all basic formulas used). The algorithm must terminate for the following reasons. We show that the "limit" tableau must have finite height. Suppose not and apply Koenig's lemma to conclude that there must be an infinite branch $\mathbf{\Gamma}$. Put another tree ordering on $\mathbf{\Gamma}$ by letting node p be a child of node q if q was used to get the node p. Note that every node has at most two children (for example a node with $[\mathbf{A} \wedge \mathbf{B}]$ attached to it may have two children nodes one with $\mathbf{A}$ attached to it and one with $\mathbf{B}$ attached to it). Since $\mathbf{\Gamma}$ is infinite it follows from Koenig's lemma that there is an infinite subsequence of nodes from $\mathbf{\Gamma}$

$$p_0, p_1, p_2, p_3, \ldots$$

which is a branch thru this second tree. But note that the lengths of formula attached to this sequence must be strictly decreasing.

Another proof of this lemma is given in the exercises.

Theorem 1.7.7 *(Compactness Theorem.) Let* $\mathbf{H}$ *be a set of propositional wffs. If every finite subset of* $\mathbf{H}$ *has a model, then* $\mathbf{H}$ *has a model.*

Proof: We give the proof only in the case that $\mathbf{H}$ is countable. Let $\mathbf{H} = \{\mathbf{A}_1, \mathbf{A}_2, \ldots\}$ be a countably infinite set of wffs and suppose that each finite subset of $\mathbf{H}$ has a model. Form an increasing chain $\mathbf{T}_n$ of finite finished tableaus for $\mathbf{H}_n = \{\mathbf{A}_1, \mathbf{A}_2, \ldots, \mathbf{A}_n\}$ by extending terminal nodes of finished

branches of $\mathbf{T}_n$ to get $\mathbf{T}_{n+1}$. Let $\mathbf{T}$ be the union. By Koenig's lemma, $\mathbf{T}$ has an infinite branch $\mathbf{\Gamma}$. Each branch $\mathbf{\Gamma}$ of $\mathbf{T}$ intersects $\mathbf{T}_n$ in a finished branch of $\mathbf{T}_n$, so any model of the basic wffs of $\mathbf{\Gamma}$ is a model of $\mathbf{H}$.

We now give some applications of the propositional compactness theorem.

One example is that the four color theorem for finite maps implies the four color theorem for infinite maps. That is:

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

Suppose that C is a set of countries on some given map. Let $\mathcal{P}_0$ be defined by

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

The idea is that the proposition letter p_n^i is to express the fact that the color of country n is i. So let $\mathbf{H}$ be the set of all sentences of the following forms:

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

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

Another example: Given a set of students and a set of classes, suppose each student wants one of a finite set of classes, and each class has a finite enrollment limit. If each finite set of students can be accommodated, then the whole set can. The proof of this is an exercise. Hint: let your basic propositional letters consist of p_{sc} where s is a student and c is a class. And p_{sc} is intended to mean "student s will take class c".

1.8 Computer Problem

In this assignment you will construct tableau proofs in propositional logic. Use the TABLEAU program commands to LOAD the problem, do your work, and then FILE your answer on your diskette. The file name of your answer should be the letter A followed by the name of the problem. (For example, your answer to the CYCLE problem should be called ACYCLE).

This diskette contains an assignment of seven problems, called CASES, CONTR, CYCLE, EQUIV, PIGEON, PENT, and SQUARE. It also has the extra files SAMPLE, ASAMPLE, and RAMSEY. SAMPLE is a problem which was done in the class lecture and ASAMPLE is the solution. RAMSEY[5] is included as an illustration of a problem which would require a much larger tableau.

Be sure your name is on your diskette label.

Here are the problems with comments, listed in order of difficulty.

CASES

The rule of proof by cases. (Can be done in 8 nodes).

Hypotheses: $a->c, b->c$

To prove: $[a \vee b]->c$

CONTR

The law of contraposition. (Can be done in 14 nodes).

Hypotheses: none

To prove: $[p->q] <-> [notq->notp]$

CYCLE

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

Hypotheses: $p->q, q->r, r->s, s->p, p \vee q \vee r \vee s$

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

EQUIV

[5] I have a wonderful proof of this theorem, but unfortunately there isn't room in the margin of memory of my computer for the solution.

Two wffs which are equivalent to a third wff are equivalent to each other. (30 nodes)

Hypotheses: $p <-> q, q <-> r$

To prove: $p <-> r$

PIGEON

The pigeonhole principle: Among any three propositions there must be a pair with the same truth value. (34 nodes)

To prove: $[p <-> q] \vee [p <-> r] \vee [q <-> r]$

PENT

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

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

$not[b1\&b2], not[b2\&b3], not[b3\&b4], not[b4\&b5], not[b5\&b1], not[r1\&r2], not[r2\&r3], not[r3\&r$

Find a tableau confutation.

SQUARE

There are nine propositional symbols which can be arranged in a square:

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

Assume that there is a letter such that for every number the proposition is true (that is, there is a row of true propositions). Prove that for every number there is a letter for which the proposition is true (that is, each column contains a true proposition). (58 nodes)

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

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

RAMSEY

The simplest case of Ramsey's Theorem can be stated as follows. Out of any six people, there are either three people who all know each other or three people none of whom know each other. This problem has 15 proposition symbols ab, ac, ..., ef which may be interpreted as meaning "a knows b", etc.

The problem has a list of hypotheses which state that for any three people among a,b,c,d,e,f, there is at least one pair who know each other and one pair who do not know each other. There is tableau confutation of these hypotheses, and I would guess that it requires between 200 and 400 nodes. I do not expect anyone to do this problem, but you are welcome to try it out and see what happens.

1.9 Exercises

1. Think about the fact that the forms 'P only if Q' and 'if P then Q' have the same meaning.

2. Brown, Jones, and Smith are suspects in a murder. The testify under oath as follows.

Brown: Jones is guilty and Smith is innocent.

Jones: If Brown is guilty, then so is Smith.

Smith: I am innocent, but at least one of the others is guilty.

Let B, J, and S be the statements "Brown is guilty", "Jones is guilty", and "Smith is guilty", respectively.

- Express the testimony of each suspect as a wff built up from the proposition symbols B, J, and S.
- Write out the truth tables for each of these three wffs in parallel columns.
- Assume that everyone is innocent. Who committed perjury?
- Assume that everyone told the truth. Who is innocent and who is guilty?
- Assume that every innocent suspect told the truth and every guilty suspect lied under oath. Who is innocent and who is guilty?

3. Show that the following are tautologies (for any wffs **A**, **B**, and **C**). Use truth tables.

(1) $\neg\neg\mathbf{A} \Leftrightarrow \mathbf{A}$

(2) $\mathbf{A} \Rightarrow [\mathbf{B} \Rightarrow \mathbf{A}]$

(3) $[\mathbf{A} \Rightarrow \mathbf{B}]. \Rightarrow .[\mathbf{B} \Rightarrow \mathbf{C}] \Rightarrow [\mathbf{A} \Rightarrow \mathbf{C}]$

(4) $[\mathbf{A} \wedge \mathbf{B}] \wedge \mathbf{C} \Leftrightarrow \mathbf{A} \wedge [\mathbf{B} \wedge \mathbf{C}]$

(5) $[\mathbf{A} \vee \mathbf{B}] \vee \mathbf{C} \Leftrightarrow \mathbf{A} \vee [\mathbf{B} \vee \mathbf{C}]$

(6) $\mathbf{A} \wedge \mathbf{B} \Leftrightarrow \mathbf{B} \wedge \mathbf{A}$

(7) $\mathbf{A} \vee \mathbf{B} \Leftrightarrow \mathbf{B} \vee \mathbf{A}$

(8) $\mathbf{A} \wedge [\mathbf{B} \vee \mathbf{C}] \Leftrightarrow [\mathbf{A} \wedge \mathbf{B}] \vee [\mathbf{A} \wedge \mathbf{C}]$

(9) $\mathbf{A} \vee [\mathbf{B} \wedge \mathbf{C}] \Leftrightarrow [\mathbf{A} \vee \mathbf{B}] \wedge [\mathbf{A} \vee \mathbf{C}]$

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

(11) $\neg[\mathbf{A} \vee \mathbf{B}] \Leftrightarrow \neg\mathbf{A} \wedge \neg\mathbf{B}$

(12) $\neg[\mathbf{A} \wedge \mathbf{B}] \Leftrightarrow \neg\mathbf{A} \vee \neg\mathbf{B}$

(13) $\neg\mathbf{A} \Rightarrow \neg\mathbf{B} \Leftrightarrow \mathbf{B} \Rightarrow \mathbf{A}$

(14) $[[\mathbf{A} \Rightarrow \mathbf{B}] \Rightarrow \mathbf{A}] \Rightarrow \mathbf{A}$

These laws have names. (1) is the *law of double negation*; (2) is the *law of affirmation of the consequent*; (3) is the *transitive law for implication* (4) is the *associative law for conjunction*; (5) is the *associative law for disjunction*; (6) is the *commutative law for conjunction*; (7) is the *commutative law for disjunction*; (8) is the *distributive law for conjunction over disjunction*; (9) is the *distributive law for disjunction over conjunction*; (10) is the *self-distributive law for implication*; (11) and (12) are *DeMorgan's laws*; (13) is the *law of contraposition*; (14) is *Pierce's law*.

4. Show that $[\mathbf{A} \Rightarrow \mathbf{B}] \Rightarrow \mathbf{A}$ is a tautology if **A** is $p \Rightarrow p$ and **B** is q but is *not* a tautology if **A** and **B** are both $p \Rightarrow q$. (The aim of this exercise is to

make sure you distinguish between a proposition symbol p and a variable $\mathbf{A}$ used to stand for a wff which may have more complicated structure.)

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

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

6. a) Make a finished tableau for the hypothesis

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

b) Choose one of the finished branches, $\mathbf{\Gamma}$, and circle the terminal node of $\mathbf{\Gamma}$.

c) Using the Finished Branch Lemma, find a wff A such that:

(i) A has exactly the same models as the set of wffs on the branch $\mathbf{\Gamma}$ which you chose, and

(ii) The only connectives occurring in A are $\wedge$ and $\neg$.

7. Let M be the model for propositional logic such that $p_M = T$ for every propositional symbol p. Prove by induction that for every wff A:

Either the $\neg$ symbol occurs in A, or A is true in M.

8. Using the Soundness and Completeness Theorems for propositional tableaus, prove that if

A has a tableau proof from H and

B has a tableau proof from $H \cup \{A\}$

then

B has a tableau proof from H.

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

have a tableau before invoking the Kill command, then you have a tableau after using the Kill command.

10. If p is a propositional symbol and C is a propositional wff, then for each propositional wff A the wff A(p//C) formed by substituting C for p in A is defined inductively by:

- p(p//C) = C;
- If q is different from p, then q(p//C) = q.
- $(\neg A)(p//C) = \neg(A(p//C))$.
- $[A * B](p//C) = [A(p//C) * B(p//C)]$, for each binary connective *.

For example,

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

Prove: If A has a tableau proof then A[p//C] has a tableau proof with the same number of nodes (In fact, the same tree but different wffs assigned to the nodes).

11. Prove that for any propositional symbol p and wffs A, B, and C,

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

is a tautology. (Show by induction on the wff A that every model of

$$B \Leftrightarrow C$$

is a model of

$$A(p//B) \Leftrightarrow A(p//C)).$$

12. Let $\mathbf{T}_0$ be any finite propositional tableau and suppose that $\mathbf{T}_1$ is any extension produced by applying the program COMPLETE. Let $\mathbf{\Gamma}_0$ be any branch in $\mathbf{T}_0$ and let $\mathbf{\Gamma}_1$ be a subbranch of $\mathbf{T}_1$ which starts at the terminal node of $\mathbf{\Gamma}_0$ and ends at a terminal node of $\mathbf{T}_1$. Show that the length of $\mathbf{\Gamma}_1$ can be at most $2nm$ where n is the number of formulas on $\mathbf{\Gamma}_0$ and m is their maximum length. Note that this gives an alternative proof of Lemma 1.7.6.

13. Given a countable set of students and a countable set of classes, suppose each student wants one of a finite set of classes, and each class has a finite enrollment limit. Prove that:

> If each finite set of students can be accommodated, then the whole set can.

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

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

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

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

15. Construct a parsing sequence for the wffpn

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

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

16. State the principle of induction as it should apply to wffpn. Prove using induction that for any wffpn $\mathbf{A}$ that the number of logical symbols of the kind $\{\wedge, \vee, \Leftrightarrow, \Rightarrow\}$ in $\mathbf{A}$ is always exactly one less than the number of nonlogical symbols.

17. Prove using induction that for any wffpn $\mathbf{A}$ and any occurence of a nonlogical symbol p in $\mathbf{A}$ except the last that the number of logical symbols of the kind $\{\wedge, \vee, \Leftrightarrow, \Rightarrow\}$ to the left of p is strictly greater than the number of nonlogical symbols to the left of p.

18. State and prove unique readability of formulas in Polish notation.

2 Pure Predicate Logic

The predicate logic developed here will be called **pure predicate logic** to distinguish it from the **full predicate logic** of the next chapter.

In this chapter we study simultaneously the family of languages known as *first-order languages* or *predicate logic.* These languages abstract from ordinary language the properties of phrases like "for all" and "there exists". We also study the logical properties of certain linguistic entities called *predicates.*

A *predicate* is a group of words like "is a man", "is less than", "belongs to", or even "is" which can be combined with one or more names of *individuals* to yield meaningful sentences. For example, "Socrates is a man", "Two is less than four", "This hat belongs to me", "He is her partner". The names of the individuals are called *individual constants.* The number of individual constants with which a given predicate is combined is called the *arity* of the predicate. For instance, "is a man" is a 1-ary or *unary* predicate and "is less than" is a 2-ary or *binary* predicate. In formal languages predicates are denoted by symbols like P and Q; the sentence *'x satisfies P'* is written *P(x).*

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

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

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

same as the sentence $P(a_1) \vee P(a_2) \vee \ldots \vee P(a_n)$. In other words, if

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

then

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

and

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

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

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

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

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

have the following "infinite" versions

$$\neg\exists x P(x) \Leftrightarrow \forall x \neg P(x)$$

and

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

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

$$\forall x P(x) \Leftrightarrow \forall y P(y)$$

and

$$\exists x P(x) \Leftrightarrow \exists y P(y).$$

For example, the sentence *"there is an x satisfying $x + 7 = 5$"* has exactly the same meaning as the sentence *"there is a y satisfying $y + 7 = 5$"*. We say that the second sentence arises from the first by *alphabetic change of a bound variable*.

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

$$x + y = y + x$$

we are supposed to understand that the author means

$$\forall x \forall y : \qquad x + y = y + x.$$

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

A precise notation for predicate logic is important because natural language is ambiguous in certain situations. Particularly troublesome in English is the word "any" which sometimes seems to mean "for all" and sometimes 'there exists". For example, the sentence "$Q(x)$ is true for any x" means " $Q(x)$ is true for all x" and should be formalized as $\forall x Q(x)$. But what about the sentence " if $Q(x)$ is true for any x, then p"? Perhaps (as before) "any" means "for all" and so it should be formalized $[\forall x Q(x)] \Rightarrow p$. But consider the following text which might appear in a mathematics book:

> This shows $Q(x)$ for $x = 83$. But earlier we saw that if $Q(x)$ holds for *any* x *whatsoever* then p follows. This establishes p and completes the proof.

In this fragment what "we saw earlier" was $[\exists x Q(x)] \Rightarrow p$ i.e. "any" means "there exists". The formal sentences $[\forall x Q(x)] \Rightarrow p$ and $[\exists x Q(x)] \Rightarrow p$ obviously have different meanings since the antecedents $[\forall x Q(x)]$ and $[\exists x Q(x)]$ differ. Since $[\exists x Q(x)] \Rightarrow p$ is equivalent [6] to $\forall x[Q(x) \Rightarrow p]$ the ambiguity in English is probably caused by the fact that the rules of grammar in English are not specific about parsing; parsing is handled unambiguously in formal languages using brackets.

[6] See blah blah below

2.1 Syntax of Predicate Logic.

A **vocabulary for pure predicate logic** is a non-empty set $\mathcal{P}$ of symbols which is decomposed as a disjoint union

$$\mathcal{P} = \bigcup_{n=0}^{\infty} \mathcal{P}_n.$$

The symbols in the set $\mathcal{P}_n$ are called n-ary **predicate symbols** (with the symbols in $\mathcal{P}_0$ called **proposition symbols** as before). The words *unary, binary, ternary* mean respectively 1-ary, 2-ary, 3-ary.

In the intended interpretation of predicate logic the predicate symbols denote relations such as $x < y$ or $x + y = z$.

The **primitive symbols of the pure predicate logic** are the following:

- the **predicate symbols** from $\mathcal{P}$;
- an infinite set $VAR = \{x, y, z, \ldots\}$ of symbols which are called **individual variables**;
- the **negation sign** $\neg$
- the **conjunction sign** $\wedge$
- the **disjunction sign** $\vee$
- the **implication sign** $\Rightarrow$
- the **equivalence sign** $\Leftrightarrow$
- the **left bracket** [
- the **right bracket**]
- the **left parenthesis** (
- the **right parenthesis**)
- the **comma** ,
- the **universal quantifier** $\forall$

- the **existential quantifier** $\exists$

Any finite sequence of these symbols is called a *string*. Our first task is to specify the *syntax* of pure predicate logic; i.e. which formulas are grammatically correct. These strings are called **well-formed formulas** or sometimes simply **formulas**. The phrase *well-formed formula* is often abbreviated to **wff**. The set of **wffs of pure predicate logic** is then inductively defined by the following rules:

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

(W:$\mathcal{P}_n$) If $\mathbf{p} \in \mathcal{P}_n$ is an n-ary predicate symbol, and $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n \in \mathit{VAR}$ are individual variables, then $\mathbf{p}(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)$ is a wff;

(W:$\neg$) If $\mathbf{A}$ is a wff, the $\neg\mathbf{A}$ is a wff;

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

(W:$\forall, \exists$) If $\mathbf{A}$ is a wff and $\mathbf{x} \in \mathit{VAR}$ is an individual variable, then the formulas $\forall\mathbf{x}\mathbf{A}$ and $\exists\mathbf{x}\mathbf{A}$ are wffs.

If we wish to emphasize that the proposition symbols appearing in a wff $\mathbf{A}$ come from a specific vocabulary $\mathcal{P}$ we may call it a **wff built using the vocabulary $\mathcal{P}$**.

To show that a particular formula is a wff we construct a sequence of formulas using this definition. This is called **parsing** the wff and the sequence is called a **parsing sequence**. Although it is never difficult to tell if a short formula is a wff, the parsing sequence is important for theoretical reasons. For example, let us assume that $\mathcal{P}_0$ contains a propositional symbol q, $\mathcal{P}_1$ contains a unary predicate symbol P, and that VAR contains an individual variable x. We parse the wff $\forall x\,[P(x) \Rightarrow q]$.

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

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

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

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

Now we parse the wff $\forall x P(x) \Rightarrow q$.

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

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

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

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

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

- Certain well-known predicates like $=$ and $<$ are traditionally written in infix rather than prefix notation and we continue this practice. Thus when our set $\mathcal{P}_2$ of binary predicate symbols contains $=$ we write $\mathbf{x} = \mathbf{y}$ rather than $=(\mathbf{x}, \mathbf{y})$. We do the same thing in other cases such as for the usual order relations ($\leq$, $<$, etc.).

- According to the syntactical rules just given, the quantifiers have the highest precedence. Thus $\forall x P(x) \Rightarrow q$ means $[\forall x P(x) \Rightarrow q]$ and not $\forall x[P(x) \Rightarrow q]$. Since students sometimes confuse these two we may sometimes insert extra brackets and write $[\forall x P(x)] \Rightarrow q$ for $\forall x P(x) \Rightarrow q$

2.2 Free and Bound Variables.

Let $\mathbf{x}$ be an individual variable and $\mathbf{Q}$ be a quantifier i.e. $\mathbf{Q}$ is either $\forall$ or $\exists$. Suppose the wff $\mathbf{Qx\,B}$ is a well-formed part of the wff $\mathbf{A}$. The well-formed part $\mathbf{B}$ is called the **scope** of the particular occurrence of the quantifier $\mathbf{Qx}$ in $\mathbf{A}$. Every occurrence of $\mathbf{x}$ in $\mathbf{QxB}$ (including the occurrence immediately after the $\mathbf{Q}$) is called a **bound occurrence** of $\mathbf{x}$ (in $\mathbf{A}$). Any occurrence of $\mathbf{x}$ in $\mathbf{C}$ which is not a bound occurrence is called a **free occurrence** of $\mathbf{x}$ (in $\mathbf{A}$). For example in the wff

$$P(x, y) \Rightarrow \forall x[\exists y R(x, y) \Rightarrow Q(x, y)] \tag{1}$$

the first occurrence of x is free, the three remaining occurrences of x are bound, the first and last occurrences of y are free, the second and third occurrences of y are bound, the well-formed part $[\exists y R(x, y) \Rightarrow Q(x, y)]$ is

the scope of the quantifier $\forall x$ and the well-formed part $R(x, y)$ is the scope of the quantifier $\exists y$. If we make an change of bound variable (say replacing all bound occurrences of x by u and all bound occurrences of y by v) we obtain

$$P(x, y) \Rightarrow \forall u[\exists v R(u, v) \Rightarrow Q(u, y)] \tag{2}$$

which has exactly the same meaning as the original wff.

Now we shall tentatively denote by

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

the result of replacing all free occurences of $\mathbf{x}$ in $\mathbf{C}$ by $\mathbf{y}$. For example, if $\mathbf{C}$ is the wff $R(x) \vee [Q(x) \Rightarrow \exists x P(x, z)]$ then $\mathbf{C}(x//u)$ is the wff $R(u) \vee [Q(u) \Rightarrow \exists x P(x, z)]$.

There is a problem with this notation. We certainly want any wff of the form

$$\forall \mathbf{x}\mathbf{C} \Rightarrow \mathbf{C}(\mathbf{x}//\mathbf{y}) \tag{3}$$

to be valid (i.e. true in any interpretation) for it says that if $\mathbf{C}$ is true for all $\mathbf{x}$ it is in particular true for $\mathbf{x} = \mathbf{y}$. But consider the case where $\mathbf{C}$ is $\exists y Q(x, y)$; then we would obtain

$$\forall x \exists y Q(x, y) \Rightarrow \exists y Q(y, y) \tag{4}$$

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

More precisely, we say that the individual variable $\mathbf{y}$ is **free for** the individual variable $\mathbf{x}$ in the wff $\mathbf{C}$ if no free occurence of $\mathbf{x}$ in $\mathbf{C}$ occurs in a well-formed part of $\mathbf{C}$ which is of the form $\forall \mathbf{y}\mathbf{B}$ or $\exists \mathbf{y}\mathbf{B}$. Henceforth we will use the notation $\mathbf{C}(\mathbf{x}//\mathbf{y})$ only in the case that $\mathbf{y}$ is free for $\mathbf{x}$ in $\mathbf{C}$.

A wff with no free variables is called a *sentence.* A sentence has a meaning (truth value) once we specify (1) the meanings of all the propositional symbols and predicate symbols which appear in it and (2) the range of values which the bound individual variables assume. For example, the sentence $\exists x \forall y \ x \leq y$ is true if $\leq$ has its usual meaning and the variables x and y range over the natural numbers (since $\forall y \ 0 \leq y$) but is false if the variables x and y range over the integers. By contrast the truth value of a wff which

has one or more free variables depends on more: viz. on the values of the free variables. For example, the wff $x = y$ is true if $x = 2$ and $y = 2$ but is false if $x = 2$ and $y = 3$.

2.3 Semantics of Predicate Logic.

For any set X and any natural number $n > 0$ let $REL_n(X)$ denote the set of all n-ary relations on X. A n-ary relation on X is a subset of the set X^n of all length n sequences $(x_1, x_2, \ldots, x_n)$ with elements from X so that

$$R \in REL_n(X) \text{ iff } R \subset X^n.$$

A *model for pure predicate logic of type* $\mathcal{P}$ is a system $\mathcal{M}$ consisting of a non-empty set $U_{\mathcal{M}}$ called the *universe of the model* $\mathcal{M}$, a function

$$\mathcal{P}_0 \longrightarrow \{\top, \bot\} : \mathbf{p} \mapsto \mathbf{p}_{\mathcal{M}}$$

which assigns a truth value $\mathbf{p}_{\mathcal{M}}$ to each proposition symbol, and for each n a function

$$\mathcal{P}_n \longrightarrow REL_n(U_{\mathcal{M}}) : \mathbf{p} \mapsto \mathbf{p}_{\mathcal{M}}$$

which assigns an n-ary relation to each n-ary predicate symbol.

Another notation for the universe of $\mathcal{M}$ is $|\mathcal{M}|$. Also M is used to denote the universe of $\mathcal{M}$.

A *valuation* in $\mathcal{M}$ is a function

$$v : VAR \longrightarrow U_{\mathcal{M}}$$

which assigns to each variable $\mathbf{x} \in VAR$ a value $v(\mathbf{x}) \in U_{\mathcal{M}}$ in the universe of the model $\mathcal{M}$. We denote the set of all valuations in $\mathcal{M}$ by $VAL(\mathcal{M})$:

$$v \in VAL(\mathcal{M}) \text{ iff } v : VAR \longrightarrow U_{\mathcal{M}}.$$

Given a valuation $v \in VAL(\mathcal{M})$ and a variable $\mathbf{x} \in VAR$ the semantics explained below require consideration of the set of all valuations $w \in VAL(\mathcal{M})$ which agree with v except possibly on $\mathbf{x}$; we denote this set of valuations by $VAL(\mathbf{x}, v)$. Thus for $v \in VAL(\mathcal{M})$ and $\mathbf{x} \in VAR$:

$$w \in VAL(\mathbf{x}, v) \text{ iff } \begin{cases} w \in VAL(\mathcal{M}) \text{ and} \\ v(\mathbf{y}) = w(\mathbf{y}) \text{ for all } \mathbf{y} \in VAR \text{ distinct from } \mathbf{x}. \end{cases}$$

Note that in particular $v \in VAL(\mathbf{x}, v)$ for certainly $v(\mathbf{y}) = v(\mathbf{y})$ for all $\mathbf{y} \in VAR$ distinct from $\mathbf{x}$.

Let $\mathcal{M}$ be a model for predicate logic, v be a valuation in $\mathcal{M}$, and $\mathbf{A}$ be a wff. We will define the notation

$$\mathcal{M}, v \models \mathbf{A}$$

which is read "$\mathcal{M}$ *models* $\mathbf{A}$ in the *valuation* v" or "$\mathbf{A}$ is true in the model $\mathcal{M}$ with the valuation v". and its opposite

$$\mathcal{M}, v \not\models \mathbf{A}$$

which is read " $\mathbf{A}$ is false in the model $\mathcal{M}$ with the valuation v". Roughly speaking, the notation $\mathcal{M}, v \models \mathbf{A}$ means that $\mathbf{A}$ is true in the model $\mathcal{M}$ when the free variables $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m$ which appear in $\mathbf{A}$ are given the values $v(\mathbf{x}_1), v(\mathbf{x}_2), \ldots, v(\mathbf{x}_m)$ respectively. More precisely, we define $\mathcal{M}, v \models \mathbf{A}$ is inductively by

(M:$\mathcal{P}_0$) $\mathcal{M}, v \models \mathbf{p}$ if $\mathbf{p}_{\mathcal{M}} = \top$;
$\mathcal{M}, v \not\models \mathbf{p}$ if $\mathbf{p}_{\mathcal{M}} = \bot$.

(M:$\mathcal{P}_n$) $\mathcal{M}, v \models \mathbf{p}(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)$ if $(v(\mathbf{x}_1), v(\mathbf{x}_2), \ldots, v(\mathbf{x}_n)) \in \mathbf{p}_{\mathcal{M}}$;
$\mathcal{M}, v \not\models \mathbf{p}(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)$ if $(v(\mathbf{x}_1), v(\mathbf{x}_2), \ldots, v(\mathbf{x}_n)) \notin \mathbf{p}_{\mathcal{M}}$.

(M:$\neg$) $\mathcal{M}, v \models \neg\mathbf{A}$ if $\mathcal{M}, v \not\models \mathbf{A}$;
$\mathcal{M}, v \not\models \neg\mathbf{A}$ if $\mathcal{M}, v \models \mathbf{A}$.

(M:$\wedge$) $\mathcal{M}, v \models \mathbf{A} \wedge \mathbf{B}$ if $\mathcal{M}, v \models \mathbf{A}$ and $\mathcal{M}, v \models \mathbf{B}$;
$\mathcal{M}, v \not\models \mathbf{A} \wedge \mathbf{B}$ otherwise.

(M:$\vee$) $\mathcal{M}, v \not\models \mathbf{A} \vee \mathbf{B}$ if $\mathcal{M}, v \not\models \mathbf{A}$ and $\mathcal{M}, v \not\models \mathbf{B}$;
$\mathcal{M}, v \models \mathbf{A} \vee \mathbf{B}$ otherwise.

(M:$\Rightarrow$) $\mathcal{M}, v \not\models \mathbf{A} \Rightarrow \mathbf{B}$ if $\mathcal{M}, v \models \mathbf{A}$ and $\mathcal{M}, v \not\models \mathbf{B}$;
$\mathcal{M}, v \models \mathbf{A} \Rightarrow \mathbf{B}$ otherwise.

(M:$\Leftrightarrow$) $\mathcal{M}, v \models \mathbf{A} \Leftrightarrow \mathbf{B}$ if either $\mathcal{M}, v \models \mathbf{A} \wedge \mathbf{B}$
or else $\mathcal{M}, v \models \neg\mathbf{A} \wedge \neg\mathbf{B}$;

$\mathcal{M}, v \not\models \mathbf{A} \Leftrightarrow \mathbf{B}$ otherwise.

(M:$\forall$) $\mathcal{M}, v \models \forall \mathbf{x}\mathbf{A}$ if $\mathcal{M}, w \models \mathbf{A}$ for every $w \in VAL(v, \mathbf{x})$;
$\mathcal{M}, v \not\models \forall \mathbf{x}\mathbf{A}$ otherwise.

(M:$\exists$) $\mathcal{M}, v \models \exists \mathbf{x}\mathbf{A}$ if $\mathcal{M}, w \models \mathbf{A}$ for some $w \in VAL(v, \mathbf{x})$;
$\mathcal{M}, v \not\models \exists \mathbf{x}\mathbf{A}$ otherwise.

Let $\mathcal{M}$ be a model, v and w be valuations, and $\mathbf{A}$ be a wff such that $v(\mathbf{x}) = w(\mathbf{x})$ for every variable which has a free occurence in $\mathbf{A}$. Then [7]

$$\mathcal{M}, v \models \mathbf{A} \text{ if and only if } \mathcal{M}, w \models \mathbf{A}.$$

In particular, if $\mathbf{A}$ is a sentence (i.e. if $\mathbf{A}$ has no free variables) then the condition $\mathcal{M}, v \models \mathbf{A}$ is independent of the choice of the valuation v; i.e. $\mathcal{M}, v \models \mathbf{A}$ for some valuation v if and only if $\mathcal{M}, v \models \mathbf{A}$ for every valuation v. Hence for sentences $\mathbf{A}$ we define

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

(read "$\mathcal{M}$ **models** $\mathbf{A}$" or " $\mathbf{A}$ is true in $\mathcal{M}$") to mean that $\mathcal{M}, v \models \mathbf{A}$ for some (and hence every) valuation v and we define

$$\mathcal{M} \not\models \mathbf{A}$$

(read "$\mathbf{A}$ is false in $\mathcal{M}$") to mean that $\mathcal{M}, v \not\models \mathbf{A}$ for some (and hence every) valuation v.

To find the value of $\mathbf{A}$ in a model $\mathcal{M}$ we parse $\mathbf{A}$ according to the inductive definition of wff and then apply the definition of $\mathcal{M} \models \mathbf{A}$ to the *parsing sequence* of $\mathbf{A}$.

Example 1. We compute the value of sentence $\forall x P(x) \Rightarrow q$ in a model $\mathcal{M}$ satisfying $U_{\mathcal{M}} = \{0, 1\}$ a two element set, $q_{\mathcal{M}} = \bot$, and $P_{\mathcal{M}} \in REL_1(U_{\mathcal{M}})$ given by

$$P_{\mathcal{M}} = \{0\}.$$

We first parse the sentence.

[7] See section B.

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

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

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

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

Now we apply the definition.

(1) $\mathcal{M}, v \not\models P(x)$ if $v(x) = 1$ by (M:$\mathcal{P}_1$) as $1 \notin U_{\mathcal{M}}$.

(2) $\mathcal{M} \not\models \forall x P(x)$ by (1) and (M:$\forall$).

(3) $\mathcal{M} \not\models q$ by (M:$\mathcal{P}_0$) as $q_{\mathcal{M}} = \bot$.

(4) $\mathcal{M} \models \forall x P(x) \Rightarrow q$ by (2), (3), and (M:$\Rightarrow$).

Example 2. We compute the value of sentence $\forall x[P(x) \Rightarrow q]$. in the model $\mathcal{M}$ of previous example 1. We first parse the sentence.

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

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

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

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

Now we apply the definition. Let w be any valuation such that $w(x) = 0$.

(1) $\mathcal{M}, w \models P(x)$ by (M:$\mathcal{P}_0$) as $w(x) = 0$.

(2) $\mathcal{M}, w \not\models q$ by (M:$\mathcal{P}_0$) as $q_{\mathcal{M}} = \bot$.

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

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

Example 3. We compute the value of $\forall y \exists x\ x \leq y \Rightarrow \exists x \forall y\ x \leq y$ for a model $\mathcal{M}$ satisfying $U_{\mathcal{M}} = \mathbf{N}$ the natural numbers and $\leq_{\mathcal{M}}$ is the usual order relation on $\mathbf{N}$:

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

We first parse the wff.

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

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

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

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

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

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

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

(1) $\mathcal{M}, v \models\ x \leq y$ iff $v(x) \leq v(y)$.

(2) $\mathcal{M}, v \models \exists x\ x \leq y$ for every v since $\mathcal{M}, w \models\ x \leq y$ if $w(y) = v(y)$ and $w(x) = 0$.

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

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

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

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

Example 4. We compute the value of the wff of example 3 for a slightly different model. Take $U_{\mathcal{M}} = \mathbf{Z}$ the set of integers with $\leq_{\mathcal{M}}$ the usual order relation on $\mathbf{Z}$:

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

(1) $\mathcal{M}, v \models\ x \leq y$ iff $v(x) \leq v(y)$.

(2) $\mathcal{M}, v \models \exists x\ x \leq y$ for every v since $\mathcal{M}, w \models\ x \leq y$ if $w(y) = v(y)$ and $w(x) = v(y)$.

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

(4) $\mathcal{M}, v \not\models \forall y\ x \leq y$ for every v since $\mathcal{M}, w \not\models\ x \leq y$ if $w(x) = v(x)$ and $w(y) = v(x) - 1$.

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

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

2.4 A simpler notation.

The notation $\mathcal{M}v \models \mathbf{A}$ is very precise and therefore useful for theoretical discussions but for working examples it is somewhat cumbersome. Hence we shall adopt a notation which supresses explicit mention of the valuation v. To do this we simply substitute the values $v(x_1), v(x_2), \ldots, v(x_n)$ in for the variables $x_1, x_2, \ldots, x_n$ in the wff $\mathbf{A}$ to produce a new "wff" $\mathbf{A}^*$ and write $\mathcal{M} \models \mathbf{A}^*$ instead of $\mathcal{M}, v \models \mathbf{A}$. For example we will write

$$\mathcal{M} \models 3 < 7$$

in place of the more cumbersome [8]

$$\mathcal{M}, v \models x < y \text{ where } v(x) = 3 \text{ and } v(y) = 7.$$

Let's redo the examples in the new notation.

Example 1. Consider the model $\mathcal{M}$ given by

$$U_{\mathcal{M}} = \{0, 1\}, \qquad q_{\mathcal{M}} = \bot, \qquad P_{\mathcal{M}} = \{0\}.$$

(1) $\mathcal{M} \not\models P(1)$.

(2) $\mathcal{M} \not\models \forall x P(x)$.

(3) $\mathcal{M} \not\models q$.

(4) $\mathcal{M} \models \forall x P(x) \Rightarrow q$.

Example 2. In the model $\mathcal{M}$ of previous example 1:

(1) $\mathcal{M} \models P(0)$.

(2) $\mathcal{M}, w \not\models q$.

(3) $\mathcal{M}, w \not\models P(0) \Rightarrow q$.

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

[8] See the relevant remarks in section B.

Example 3. Consider the model

$$U_{\mathcal{M}} = \mathbf{N}$$

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

(1) $\mathcal{M} \models\ 0 \leq b$ for every $b \in U_{\mathcal{M}} = \mathbf{N}$ and hence

(2) $\mathcal{M} \models \exists x\ x \leq b$. As b is arbitrary

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

(4) $\mathcal{M} \models \forall y\ 0 \leq y$ so

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

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

Example 4. Now take

$$U_{\mathcal{M}} = \mathbf{Z}$$

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

(1) $\mathcal{M}, v \models\ a \leq b$ iff $a \leq b$. Thus

(2) $\mathcal{M}, v \models \exists x\ x \leq b$ for every b since $\mathcal{M} \models\ a \leq a$. Thus

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

(4) $\mathcal{M} \not\models \forall y\ a \leq y$ for every a since $\mathcal{M}, w \not\models\ a \leq a - 1$. Hence

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

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

2.5 Tableaus.

The analog of a tautology in propositional logic is a valid sentence in predicate logic. All of mathematics can be formulated as a list of theorems showing that certain inportant sentences of predicate logic are valid.

A sentence **A** of predicate logic is said to be *valid* iff **A** is true in every model.

In propositional logic it is possible to test whether a wff is valid in a finite number of steps by constructing a truth table. This cannot be done in predicate logic. In predicate logic there are infinitely many universes to consider. Moreover, for each infinite universe set there are infinitely many possible models, even when the vocabulary of predicate symbols is finite. Since one cannot physically make a table of all models, another method is needed for showing that a sentence is valid. To this end, we shall generalize the notion of tableau proof from propositional logic to predicate logic. As before, a formal proof of a sentence **A** will be represented as a tableau confutation of the negation of **A**.

There is one important difference between the method of truth tables in propositional logic and the method of tableaus in predicate logic, to which we shall return later in this book. In propositional logic, the method of truth tables works for all sentences; if a sentence is valid the method will show that it is valid in a finite number of steps, and if a sentence is *not* valid the method will show that it that it is not valid in a finite number of steps. However, in predicate logic the method of tableau proofs only works for valid sentences. If a sentence *is* valid, there is a tableau proof which shows that it is valid in finitely many steps. But tableau proofs do not give a method of showing in finitely many steps that a sentence is *not* valid.

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

branch of the tableau and at least one valuation v in $\mathcal{M}$ such that every wff on the branch is true for $\mathcal{M}, v$.

A *labeled tree for predicate logic* is a system $(\mathbf{T}, \mathbf{H}, \Phi)$ where $\mathbf{T}$ is a tree, $\mathbf{H}$ is a set of sentences of predicate logic, and Φ is a function which assigns to each nonroot node t of $\mathbf{T}$ a wff $\Phi(t)$ of predicate logic. (The definition is exactly the same as for propositional logic, except that the wffs are now wffs of predicate logic.) We shall use the same lingo (ancestor, child, parent etc.) as we did for propositional logic and also confuse the node t with the formula $\Phi(t)$ as we did there.

We call the labeled tree $(\mathbf{T}, \mathbf{H}, \Phi)$ a *tableau for predicate logic* [9] iff at each non-terminal node t either one of the tableau rules for propositional logic (see section 1.5) holds or else one of the following conditions holds:

$\boxed{\forall}$ t has an ancestor $\forall\mathbf{xA}$ and a child $\mathbf{A}(\mathbf{x}//\mathbf{y})$ where $\mathbf{y}$ is free for $\mathbf{x}$.

$\boxed{\neg\forall}$ t has an ancestor $\neg\forall\mathbf{xA}$ and a child $\neg\mathbf{A}(\mathbf{x}//\mathbf{z})$ where $\mathbf{z}$ is a variable which does not occur in any ancestor of t;

$\boxed{\exists}$ t has an ancestor $\exists\mathbf{xA}$ and a child $\mathbf{A}(\mathbf{x}//\mathbf{z})$ where $\mathbf{z}$ is a variable which does not occur in any ancestor of t;

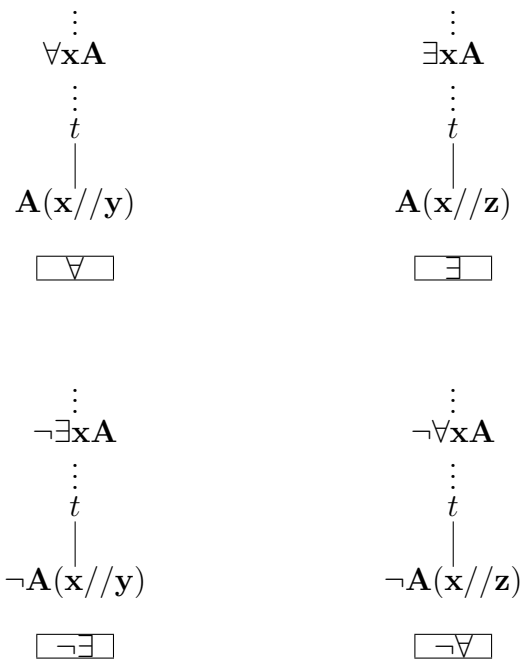
$\boxed{\neg\exists}$ t has an ancestor $\neg\exists\mathbf{xA}$ and a child $\neg\mathbf{A}(\mathbf{x}//\mathbf{y})$ where $\mathbf{y}$ is free for $\mathbf{x}$.

The for new rules are summarized in figure 3 which should be viewed as an extension of figure 1 of section 1.5.

Notice that the $\boxed{\forall}$ and $\boxed{\neg\exists}$ rules are similar to each other, and the $\boxed{\exists}$ and $\boxed{\neg\forall}$ rules are similar to each other. The $\boxed{\forall}$ and $\boxed{\neg\exists}$ rules allow any substitution at all as long as there is no confusion of free and bound variables. These rules are motivated by the fact that if $\mathcal{M}, v \models \forall\mathbf{xA}$ then $\mathcal{M}, v \models \mathbf{A}(\mathbf{x}//\mathbf{y})$ for any variable whatsoever (and if $\mathcal{M}, v \models \neg\exists\mathbf{xA}$ then $\mathcal{M}, v \models \neg\mathbf{A}(\mathbf{x}//\mathbf{y})$).

On the other hand, the $\boxed{\exists}$ and $\boxed{\neg\forall}$ rules are very restricted, and only allow us to substitute a completely new variable $\mathbf{z}$ for $\mathbf{x}$. In an informal mathematical proof, if we know that $\exists\mathbf{xA}$ is true we may introduce a new symbol $\mathbf{z}$ to name the element for which $\mathbf{A}(\mathbf{x}//\mathbf{z})$ is true. It would be incorrect to use a symbol which has already been used for something else. This

[9] This definition is summarized in section D.



y is free for **x** **z** is new

Figure 3: Quantifier Rules for Pure Predicate Logic.

informal step corresponds to the $\exists$ rule for extending a tableau. A similar remark applies to the $\neg\forall$ rule.

As for propositional tableaus, a *tableau confutation* of a set $\mathbf{H}$ of sentences in predicate logic is a finite tableau $(\mathbf{T}, \mathbf{H}, \Phi)$ such that each branch is **contradictory**, that is, each branch has a pair of wffs $\mathbf{A}$ and $\neg\mathbf{A}$. A *tableau proof* of a sentence $\mathbf{A}$ is a tableau confutation of the set $\{\neg\mathbf{A}\}$, and a *tableau proof of* $\mathbf{A}$ **from the hypotheses** $\mathbf{H}$ is a tableau confutation of the set $\mathbf{H} \cup \{\neg\mathbf{A}\}$.

It is usually much more difficult to find formal proofs in predicate logic than in propositional logic, because if you are careless your tableau will keep growing forever. One useful rule of thumb is to try to use the $\exists$ and $\neg\forall$ rules, which introduce new variables, as early as possible. Quite often, these new variables will appear in substitutions in the $\forall$ or $\neg\exists$ rules later on. The rule of thumb can be illustrated in its simplest form in the following examples.

Example 1. A tableau proof of $\exists y\ P(y)$ from $\exists x\ P(x)$.

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

Example 2. A tableau proof of $\forall y \exists x\ P(x, y)$ from $\exists x \forall y\ P(x, y)$.

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

2.6 Soundness

The proof of the soundness theorem for predicate logic is similar to the proof of the soundness theorem for propositional logic, with extra steps for the quantifiers. It comes from the following basic lemma.

Lemma 2.6.1 *If* $\mathbf{T}$ *a finite tableau in predicate logic and hypothesis set* $\mathbf{H}$ *and* $\mathcal{M}$ *is a model of the set of sentences of* $\mathbf{H}$*, then there is a branch* $\mathbf{\Gamma}$ *and a valuation* v *in* $\mathcal{M}$ *such that* $\mathcal{M}, v \models \mathbf{A}$ *for every wff* $\mathbf{A}$ *on* $\mathbf{\Gamma}$*, that is,* $\mathcal{M}, v \models \mathbf{\Gamma}$

This is proved by induction on the cardinality of $\mathbf{T}$.

Theorem 2.6.2 *(Soundness Theorem.) If a sentence* $\mathbf{A}$ *has a tableau proof, then* $\mathbf{A}$ *is valid. If* $\mathbf{A}$ *has a tableau proof from a set* $\mathbf{H}$ *of sentences, then* $\mathbf{A}$ *is a semantic consequence of* $\mathbf{H}$*, that is every model of* $\mathbf{H}$ *is a model of* $\mathbf{A}$*.*

2.7 Completeness

The completeness theorem for predicate logic uses many of the ideas introduced in the completenes theorem for propositional logic. One inportant difference is that it is necessary to consider infinite tableaus.

Theorem 2.7.1 *(Completeness Theorem.) Let* $\mathbf{A}$ *be any sentence in pure predicate logic. If* $\mathbf{A}$ *has no models, then* $\mathbf{A}$ *has a tableau confutation. In other words, If* $\mathbf{A}$ *does not have a tableau confutation, then* $\mathbf{A}$ *has a model.*

To prove the completeness theorem we shall extend the concepts of a basic wff and a finished branch to predicate logic.

By an *atomic wff* we mean either a propositional symbol alone or a wff of form $\mathbf{p}(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)$ where $\mathbf{p}$ is an n-ary predicate symbol and $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ are variables. By a *basic wff* in predicate logic we mean a wff which is either an atomic wff or the negation of an atomic wff.

Let $\mathbf{\Delta}$ be a set of wffs of pure predicate logic. Let $U_{\mathbf{\Delta}}$ be the set of variables which occur freely in some wff $\mathbf{C} \in \mathbf{\Delta}$.

We call a set $\mathbf{\Delta}$ of wffs *contradictory* iff it contains some pair of wffs of form $\mathbf{A}$, $\neg\mathbf{A}$. We call $\mathbf{\Delta}$ *finished* iff $\mathbf{\Delta}$ is not contradictory, and for each wff $\mathbf{C} \in \mathbf{\Delta}$ either $\mathbf{C}$ is a basic wff or else one of the following is true:

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

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

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

$[\forall]$ $\quad$ $\mathbf{C}$ has form $\forall\mathbf{x}\mathbf{A}$ where $\mathbf{A}(\mathbf{x}//\mathbf{z}) \in \mathbf{\Delta}$ for every $\mathbf{z} \in U_{\mathbf{\Delta}}$;

$[\neg\forall]$ $\quad$ $\mathbf{C}$ has form $\neg\forall\mathbf{x}\mathbf{A}$ where $\neg\mathbf{A}(\mathbf{x}//\mathbf{z}) \in \mathbf{\Delta}$ for some $\mathbf{z} \in U_{\mathbf{\Delta}}$;

$[\exists]$ $\quad$ $\mathbf{C}$ has form $\exists\mathbf{x}\mathbf{A}$ where $\mathbf{A}(\mathbf{x}//\mathbf{z}) \in \mathbf{\Delta}$ for some $\mathbf{z} \in U_{\mathbf{\Delta}}$;

$[\neg\exists]$ $\quad$ $\mathbf{C}$ has form $\neg\exists\mathbf{x}\mathbf{A}$ where $\neg\mathbf{A}(\mathbf{x}//\mathbf{z}) \in \mathbf{\Delta}$ for every $\mathbf{z} \in U_{\mathbf{\Delta}}$.

The wffs in $\mathbf{\Delta}$ required by the definition are said to be *witnesses* for $\mathbf{C}$. For instance in case $[\vee]$ where $\mathbf{C}$ is $\mathbf{A} \vee \mathbf{B}$ we say that $\mathbf{A}$ is a witness for $\mathbf{C}$ if $\mathbf{A} \in \mathbf{\Delta}$ and that $\mathbf{B}$ is a witness for $\mathbf{C}$ if $\mathbf{B} \in \mathbf{\Delta}$.

The definition of finished set parallels the definition of tableau. It should be noted, however, that the key point in the definition of finished set is that *every* possible application of the tableau rules has been made. This remark is especially relevant in comparing the $\boxed{\forall}$ and $\boxed{\neg\exists}$ tableau extension rules with the $[\forall]$ and$[\neg\exists]$ clauses in the definition of finished set. The latter two rules say that *every* possible substitution instance must lie in the finished set, whereas the former two rules say that the tableau may be extended by an arbitrary substitution.

To simplify our exposition, we will impose a further condition on finished sets in the arguments given below. We call a set $\mathbf{\Delta}$ of wffs *unconfused* iff no variable having a free occurrence in some wff of $\mathbf{\Delta}$ has a bound occurrence in any other wff of $\mathbf{\Delta}$. The idea of this definition is that in an unconfused set the variables in the set $U_{\mathbf{\Delta}}$ can be considered as constants since they are never used as bound variables.

Lemma 2.7.2 *(Finished Branch Lemma.) Suppose $\mathbf{\Delta}$ is an unconfused, finished set of wffs with $U_{\mathbf{\Delta}}$ non-empty. Let $\mathcal{M}$ be any model for pure predicate logic whose universe $U_{\mathcal{M}}$ is the set $U_{\mathbf{\Delta}}$, and let v be the valuation in $\mathcal{M}$ which assigns each variable in $U_{\mathbf{\Delta}}$ to itself. If $\mathcal{M}, v \models \mathbf{A}$ for every basic wff $\mathbf{A} \in \mathbf{\Delta}$, then $\mathcal{M}, v \models \mathbf{A}$ for every wff $\mathbf{A} \in \mathbf{\Delta}$, that is, $\mathcal{M}, v \models \mathbf{\Delta}$.*

The proof is by induction on the logical complexity of formulas, that is the number of logical symbols appearing in the formula. The basis step is

for atomic formula and the negation of an atomic formula. This follows from the definition of $\mathcal{M}$ and the fact that the branch is not contradictory. The inductive case breaks down into the thirteen subcases corresponding to the items in the definition of finished set. Here we give a few example proofs:

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

By induction if $\neg\mathbf{A} \in \mathbf{\Delta}$, then $\mathcal{M}, v \models \neg\mathbf{A}$; and if $\mathbf{B} \in \mathbf{\Delta}$ then $\mathcal{M}, v \models \mathbf{B}$. In either case $\mathcal{M}, v \models \mathbf{C}$.

$[\forall]$ where $\mathbf{C} \in \mathbf{\Delta}$ and $\mathbf{C}$ has form $\forall\mathbf{x}\mathbf{A}$ where $\mathbf{A}(\mathbf{x}//\mathbf{z}) \in \mathbf{\Delta}$ for every $\mathbf{z} \in U_{\mathbf{\Delta}}$;

By induction $\mathcal{M}, v \models \mathbf{A}(\mathbf{x}//\mathbf{z})$ for every $\mathbf{z} \in U_{\mathbf{\Delta}}$, hence by the definition of $\models$, $\mathcal{M}, v \models \mathbf{C}$.

$[\exists]$ where $\mathbf{C} \in \mathbf{\Delta}$ and $\mathbf{C}$ has form $\exists\mathbf{x}\mathbf{A}$ where $\mathbf{A}(\mathbf{x}//\mathbf{z}) \in \mathbf{\Delta}$ for some $\mathbf{z} \in U_{\mathbf{\Delta}}$;

By induction $\mathcal{M}, v \models \mathbf{A}(\mathbf{x}//\mathbf{z})$ for some $\mathbf{z} \in U_{\mathbf{\Delta}}$. So if $w \in VAL(x, v)$ is such that $w(\mathbf{x}) = \mathbf{z}$ then $\mathcal{M}, w \models \mathbf{A}$ and so by the definition of $\models$ we have $\mathcal{M}, v \models \exists\mathbf{x}\mathbf{A}$.

The other subcases are similar.

Define a tableau $\mathbf{T}$ to be *finished* if $\mathbf{T}$ is unconfused and every branch is either finished or else both finite and contradictory. In a finished tableau, the finished branches, if any, may be either finite or infinite. If all the branches of a tableau are finite and contradictory, then the tableau will have finitely many nodes and will be a confutation.

Lemma 2.7.3 *(Tableau Extension Lemma.) For every sentence there exists a finished tableau with the sentence attached to the root node.*

We construct by induction a sequence of finite tableaus

$$\mathbf{T}_0 \subset \mathbf{T}_1 \subset \mathbf{T}_2 \ldots$$

which are nested as shown. The finished tableau will be the union of these tableaus (i.e. $\bigcup_{n=0,1,2,\ldots} \mathbf{T}_n$). The tableau $\mathbf{T}_0$ is just the trivial tree with only the root node and the given sentence attached to it. Let $\{a_0, a_1, a_2, \ldots\}$ be an infinite sequence of variables which do not occur in the given sentence. We will construct an unconfused tableau by only using these variables in the quantifier rules.

Given the tableau $\mathbf{T}_n$ we construct the tableau $\mathbf{T}_{n+1}$ to have the following properties. For any noncontradictory branch $\mathbf{\Gamma}$ of $\mathbf{T}_{n+1}$ and formula $\mathbf{A}$ on $\mathbf{\Gamma} \cap \mathbf{T}_n$:

1. If $\mathbf{A}$ is of the form $\forall \mathbf{x}\mathbf{B}$ then for every $i = 0, 1, 2, \ldots, n$ the formula $\mathbf{B}(\mathbf{x}//a_i)$ is on $\mathbf{\Gamma}$.

2. If $\mathbf{A}$ is of the form $\neg\exists \mathbf{x}\mathbf{B}$ then for every $i = 0, 1, 2, \ldots, n$ the formula $\neg\mathbf{B}(\mathbf{x}//a_i)$ is on $\mathbf{\Gamma}$.

3. If $\mathbf{A}$ is of any other form, then we require that it be used at least once on $\mathbf{\Gamma}$. For example, If $\mathbf{A}$ is of the form $\exists \mathbf{x}\mathbf{B}$ then for some integer k (possibly much bigger than n) the formula $\mathbf{B}(\mathbf{x}//a_k)$ is on $\mathbf{\Gamma}$.

$\mathbf{T}_{n+1}$ is constructed in finitely many stages by taking care of all formulas occuring in $\mathbf{T}_n$ one at a time. Now we claim that $\mathbf{T} = \bigcup_{n=0,1,2,\ldots} \mathbf{T}_n$ is a finished tableau. Let $\mathbf{\Gamma}$ be any branch of $\mathbf{T}$. If $\mathbf{\Gamma}$ is not contradictory we must show that $\mathbf{\Delta}$ which is the set of all formula on $\mathbf{\Gamma}$ is a finished set. Note that $U_{\mathbf{\Delta}} = \{a_0, a_1, a_2, \ldots\}$. Suppose that $\mathbf{A} \in \mathbf{\Delta}$, then for some n $\mathbf{A}$ is in $\mathbf{T}_n$. Since $\mathbf{\Gamma} \cap \mathbf{T}_{n+1}$ is a branch of $\mathbf{T}_{n+1}$, by the construction $\mathbf{A}$ has been used on $\mathbf{\Gamma} \cap \mathbf{T}_{n+1}$ and hence on $\mathbf{\Gamma}$. Now suppose that $\mathbf{A}$ has the form $\forall \mathbf{x}\mathbf{B}$, then for every $m > n$ and $i \leq m$ the formula $\mathbf{B}(\mathbf{x}//a_i)$ is on $\mathbf{\Gamma} \cap \mathbf{T}_m$. Hence for every $i = 0, 1, 2, \ldots$ the formula $\mathbf{B}(\mathbf{x}//a_i)$ is on $\mathbf{\Gamma}$. Similarly if $\mathbf{A}$ has the form $\neg\exists \mathbf{x}\mathbf{B}$, then for every $i = 0, 1, 2, \ldots$ the formula $\neg\mathbf{B}(\mathbf{x}//a_i)$ is on $\mathbf{\Gamma}$.

The completeness theorem can now be deduced as follows. Let $\mathbf{A}$ be any sentence. Let $\mathbf{T}$ be a finished tableau with $\neg\mathbf{A}$ attached to the root node. If all branches of $\mathbf{T}$ are finite and contradictory, then by the Koenig tree lemma $\mathbf{T}$ is a finite tree and hence a confutation of $\mathbf{A}$. Otherwise $\mathbf{T}$ must have a finished branch and therefore $\neg\mathbf{A}$ has a model.

Note that this proof shows that any sentence of pure predicate logic that has a model has an infinite model, i.e. one with an infinite universe. This will not be the case for the full predicate logic.

As in the case of propositional logic, there is a second form of the completeness theorem and a compactness theorem.

Recall that a sentence $\mathbf{A}$ is *valid* iff it is true in every model, and is a *semantic consequence* of a set of sentences $\mathbf{H}$ iff it is true in every model of $\mathbf{H}$.

Theorem 2.7.4 *(Completeness Theorem, Second Form.) If a sentence* **A** *is a semantic consequence of a set of sentences* **H** *then there is a tableau proof of* **A** *from* **H**. *In particular, a valid sentence has a tableau proof.*

This can be proved by constructing a finished tableau for any finite set of sentences. In fact, it is possible to construct a finished tableau for any countably infinite set of sentences. This stronger result and the Koenig tree lemma give a proof of the compactness theorem for this special case.

Theorem 2.7.5 *(Compactness Theorem.) Let* **H** *be any set of sentences of pure predicate logic. If every finite subset of* **H** *has a model, then* **H** *has a model.*

A finished set which has only finitely many terms will have a finite model. The program BUILD is a modification of the TABLEAU program which helps you build a finished branch with constants from a given universe, and displays the corresponding in graphical form.

2.8 Computer problem using PREDCALC

This set of problems uses the PREDCALC program. Its purpose is to make the student more familiar with the behavior of truth values of formulas of predicate logic in a model. There are twelve problems, named ONE.PRD through TWELVE.PRD. Problems ONE and TWO are text problems and the others are graphics problems. In each problem, a goal formula or graph will appear on the screen and your task is to use the "calculator" keys to get an exact copy of the goal in position one of the stack. PREVIEW.PRD is the sample described in section 6.3.

Suggestions: To solve a graphics problem, think of a formula which has the required graph and write it down, then make a parsing sequence for the formula and build it up step by step. You always start out with atomic formulas involving $=, <, +$, or $*$. To see the graph of the goal formula in detail, invoke the View command and then press 5. If you want to keep part of what you did and change the rest, invoke the Mem command and replay as far as you want by pressing the Enter key, then make your changes.

The problems are listed in order of difficulty. As in the previous problem set, you should give your solution the name of the problem preceded by the

letter A. The number of steps needed for a solution and other comments are given below.

ONE. 11 steps.
TWO. 14 steps.
THREE. 6 steps. (A single point)
FOUR. 4 steps.
FIVE. 5 steps.
SIX. 5 steps.
SEVEN. 6 steps. (The graph of the relation "x + z is even")
EIGHT. 7 steps.
NINE. 6 steps.
TEN. 8 steps. (A 3 dimensional checkerboard pattern)
ELEVEN. 16 steps. (The four main diagonals of the cube)
TWELVE. 28 steps. (The graph of the relation "$z < x + y$")

2.9 Computer problem

In this assignment you will construct tableau proofs in predicate logic. Use the TABLEAU program commands to LOAD the problem, do your work, and then FILE your answer on your diskette. The file name of your answer should be the letter A followed by the name of the problem.

There are three groups of problems on your diskette which use the TABLEAU program, called

- SHORT1 - SHORT8,
- SET1 - SET6,
- and ORDER1 -ORDER6.

The first group of problems, called

SHORT1.TBU through SHORT8.TBU,

develop some of the basic properties of quantifiers. You should do these problems first by hand on a piece of paper, and then do them on the computer to check your work. This will help you discover any misunderstandings you may have.

The problems below are also in files on your problem diskette. In each problem, you must give a tableau proof.

The notation $H \vdash A$ means that A has a tableau proof from H.

SHORT1.	(3 nodes)	$\exists x\ p(x,x) \vdash \exists x \exists y\ p(x,y)$
SHORT2.	(3 nodes)	$\exists y\ p(y) \vdash \exists y \forall x\ p(y)$
SHORT3.	(4 nodes)	$\forall y \forall x\ p(x,y) \vdash \forall x \forall y\ \ p(x,y)$
SHORT4.	(6 nodes)	$p \wedge \exists x\ \ q(x) \vdash \exists x\ \ [p \wedge q(x)]$
SHORT5.	(7 nodes)	$\exists x\ \ [p(x) \wedge q(x)] \vdash \exists x\ \ p(x) \wedge \exists x\ \ q(x)$
SHORT6.	(11 nodes)	$\vdash \exists x\ \ p(x) \Leftrightarrow \neg \forall x\ \ \neg p(x)$
SHORT7.	(9 nodes)	$\forall x \exists y\ \ F(x,y) \vdash \forall x \exists y \exists z\ \ [F(x,y) \wedge F(y,z)]$
SHORT8.	(23 nodes)	$\vdash \forall x\ \ p(x) \wedge \forall x\ \ q(x) \Leftrightarrow \forall x\ \ [p(x) \wedge q(x)]$

There are two more groups of problems. The problems called

SET1.TBU through SET6.TBU

are about sets, while the problems

ORDER1.TBU through ORDER6.TBU

concern partial orders. These problems are more difficult, and you need an overall picture of your proof so that you will be able to choose useful substitutions for the quantifiers. Before doing the formal proof on the computer, you should make a sketch of the main steps of the proof with pencil and paper.

Here are the problems with comments and the approximate number of nodes required for the tableau proof. You should try problems with fewer nodes first.

SET1. (12 nodes). The predicate in(x,y) means that x is an element of y, and the predicate subset(x,y) means that x is a subset of y. The hypothesis defines subset(x,y) in terms of in(x,y). The conclusion states that every set is a subset of itself.

SET2. (23 nodes) The predicate empty(x) means that x is an empty set. The first hypothesis is the same as before. The second hypothesis defines empty(x) in terms of in(x,y). The conclusion states that the empty set is a subset of every set.

SET3. (34 nodes) The predicate union(x,y,z) means that z is the union of x and y. The hypotheses define subset(x,y) and union(x,y) in terms of in(x,y). The conclusion states that x is a subset of the union of x and y.

SET4. (39 nodes) The hypothesis is again the definition of subset(x,y) in terms of in(x,y). The conclusion is the transitivity law for subsets, that if x in a subset of y and y is a subset of z then x is a subset of z.

SET5. (50 nodes) The predicate eq(x,y) means that x and y are elements of the same sets. The predicate single(x) means that x has exactly one element. The hypotheses define the predicates subset(x,y), eq(x,y), and single(x). The conclusion states that if single(x) and y contains some element of x then x is a subset of y.

SET6. (61 nodes) The hypotheses define the predicates subset(x,y) and union(x,y,z). The conclusion states that if both x and y are subsets of u then the union of x and y is a subset of u.

ORDER1. (21 nodes) The hypotheses state that $\leq$ is a partial ordering. The conclusion states that if w $\leq$ x $\leq$ y $\leq$ z then w $\leq$ z.

ORDER2. (27 nodes) The predicate glb(x,y,z) means that z is the greatest lower bound of x and y in the partial ordering $\leq$, that is, z is the greatest element which is $\leq$ both x and y . The conclusion states that if x $\leq$ y then x is the greatest lower bound of x and y.

ORDER3. (33 nodes) The hypotheses are the same as for ORDER2. The conclusion states that if z and t are both greatest lower bounds of x and y, then z $\leq$ t. (Since the same reasoning gives t $\leq$ z, this shows that any two greatest lower bounds of x and y are equal).

ORDER4. (32 nodes) The hypotheses state that $\leq$ is a partial ordering, and that for any two elements x,y there is an element t such that x $\leq$ t and y $\leq$ t. The conclusion states that for any three elements x,y,z there is an element t such that x $\leq$ t, y $\leq$ t, and z $\leq$ t.

ORDER5. (44 nodes) The predicate x $<$ y means that x $\leq$ y but not y $\leq$ x. The hypotheses state that $\leq$ is a partial ordering and define the predicate x $<$ y in terms of x $\leq$ y. The conclusion states that if x $<$ y $\leq$ z then x $<$ z.

ORDER6. (119 nodes) The predicate eq(x,y) means that x $\leq$ y and y $\leq$ x. The hypotheses state that $\leq$ is a partial ordering and define the predicates glb(x,y,z) and eq(x,y) in terms of x $\leq$ y. The conclusion is an associative law for greatest lower bounds. If we write $x * y$ for the greatest bound of x and

y, and x = y for eq(x,y), the conclusion states that

$$(a * b) * c = a * (b * c).$$

2.10 Exercises

1. In the following, **N** denotes the set of **natural numbers**:

$$\mathbf{N} = \{0, 1, 2, 3, \ldots\}$$

so that the phrase $\forall x \in \mathbf{N}$ means *'for all natural numbers x'* and the phrase $\exists x \in \mathbf{N}$ means *'there is a natural number x such that'*. The notations $+$, $=$, $\leq$, and $<$ all have their usual meanings. Which of the following are true?

(1) $\forall x \in \mathbf{N}\, \forall y \in \mathbf{N}\, \forall z \in \mathbf{N}\, [x + y = z \Rightarrow y + x = z]$.

(2) $\forall x \in \mathbf{N}\, \exists y \in \mathbf{N}\, [x + y = x]$.

(3) $\exists y \in \mathbf{N}\, \forall x \in \mathbf{N}\, [x + y = x]$.

(4) $\forall x \in \mathbf{N}\, \exists y \in \mathbf{N}\, [x < y]$.

(5) $\forall x \in \mathbf{N}\, \exists y \in \mathbf{N}\, [y < x]$.

(6) $\exists y \in \mathbf{N}\, \forall x \in \mathbf{N}\, [x < y]$.

(7) $\forall x \in \mathbf{N}\, [[\forall y \in \mathbf{N}\ x \leq y] \Rightarrow x = 0]$.

(8) $\forall x \in \mathbf{N}\, \forall y \in \mathbf{N}\, [[x \leq y \wedge y \leq x] \Rightarrow x = y]$.

(9) $\forall x \in \mathbf{N}\, \forall y \in \mathbf{N}\, [[x < y \wedge y < x] \Rightarrow x \neq y]$.

(10) $[\forall x \in \mathbf{N}\, \exists y \in \mathbf{N}\ x < y] \Rightarrow [\exists y \in \mathbf{N}, 8 < y]$.

(11) $[\forall x \in \mathbf{N}\, \exists y \in \mathbf{N}\ x < y] \Rightarrow [\exists y \in \mathbf{N}\, y < y]$.

2. Let **Z** denote the set of **integers**:

$$\mathbf{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}.$$

so that the phrase $\forall x \in \mathbf{Z}$ means *'for all integers x'* and the phrase $\exists x \in \mathbf{Z}$ means *'there is an integer x such that'*. Which of the sentences of exercise 1. remain true when $\mathbf{N}$ is replaced by $\mathbf{Z}$?

3. a) The string

$$\exists x\ [\forall y\ p(x,y) \Rightarrow \neg q(x) \vee r(y)]$$

is an abbreviation for a wff in predicate logic. Change the string into the wff which it abbreviates by inserting brackets in the correct places.

b) Write down a parsing sequence for the wff.

c) For each wff of your parsing sequence, circle every occurrence of a variable which is bound in that wff.

4. In this problem you are to find a model $\mathcal{M}$ for predicate logic with one binary predicate symbol p. The universe of $\mathcal{M}$ is the set $\{0,1,2\}$ and the relation $p_{\mathcal{M}}$ is a subset of the set of pairs (i,j) with i,j from $\{0,1,2\}$. Your answer will be counted as correct iff the wff

$$\forall x \exists y\ p(x,y) \wedge \exists x \forall y\ p(x,y) \wedge \neg \exists y \forall x\ p(x,y)$$

is true in your model $\mathcal{M}$.

5. This exercise is to give a formal proof of the completeness theorem.

1. For all H and for all T, T is a finished tableau for H iff T is a tableau for H and for every $\mathbf{\Gamma}$ a branch of T either $\mathbf{\Gamma}$ is finished or $\mathbf{\Gamma}$ is finite and contradictory.

2. For all H and for all T, T is a confutation of H iff T is a finite tableau for H and every branch of T is contradictory.

3. For all H ,T and $\mathbf{\Gamma}$ if T is a tableau for H and $\mathbf{\Gamma}$ is a finished branch of T, then there exists $\mathcal{M}$ such that $\mathcal{M}$ models H.

4. For every H there exists T such that T is a finished tableau for H.

5. For all H and for every T if T is an infinite tableau for H, then there exists a $\mathbf{\Gamma}$ such that $\mathbf{\Gamma}$ is an infinite branch of T.

6. For all H if there does not exists $\mathcal{M}$ such that $\mathcal{M}$ models H, then there exists T such that T is a confutation of H.

The hypotheses (1-5) are the three lemmas and the definitions of finished tableau for H and contradictory branch. Item 6 is the statement to be proved, the completeness theorem.

Consider the following language of pure predicate logic:

$$\mathcal{P}_1 = \{F, Cn, I\} \;\; \mathcal{P}_2 = \{Ft, B, S, T, Cf\}$$

Let F(x) be interpreted as "x is a finished branch", Cn(x) as "x is a contradictory branch", and I(x) as "x is infinite". Let Ft(x,y) be interpreted as "x is a finished tableau for hypothesis set y", B(x,y) as "x a branch thru tableau y", S(x,y) as " x is a model of hypothesis set y", T(x,y) as " x is a tableau for hypothesis set y", and Cf(x,y) as "x is a confutation of hypothesis set y".

Write out the above hypothesis set and sentence to prove in this language of pure predicate logic. (It is understood that H is always a finite set of sentences and infinite means not finite).

Give a tableau proof.

We say that a model $(P, \leq)$ is a *linear order* (where $\leq$ is a binary relation on the universe P) iff

1. $(P, \leq) \models \forall x \; x \leq x$
2. $(P, \leq) \models \forall x \forall y \forall z \; [x \leq y \wedge y \leq z \Rightarrow x \leq z]$
3. $(P, \leq) \models \forall x \forall y \; [x \leq y \wedge y \leq x \Rightarrow x = y]$
4. $(P, \leq) \models \forall x \forall y \; [x \leq y \vee y \leq x]$

If $(P, \leq)$ satisfies only the first three it is called a *partial order*.

6. Find a linear order $(P, \leq)$ which satisfies all of the following:

$$(P, \leq) \models \forall x \exists y \; \neg x \leq y$$

$$(P, \leq) \models \forall x \exists y \neg y \leq x$$
$$(P, \leq) \models \forall x \forall y \, [\neg y \leq x \Rightarrow \exists z [\neg z \leq x \wedge \neg y \leq z]$$

7. Show by induction that for any finite partial order $(P, \leq)$ (i.e. P is finite) there is a linear order $(P, \leq^*)$ which extendes $\leq$, i.e. for every $a, b \in P$ if $a \leq b$, then $a \leq^* b$.

8. Use the Compactness Theorem for **Propositional Logic** and the last problem to show that every partial order (finite or infinite) can be extended to a linear order.
Hint: Let $(P, \leq)$ be any partial order. Let $\mathcal{P}_0 = \{R_{ab} \, : \, a, b \in P\}$. Consider interpreting the symbol R_{ab} as "$a \leq^* b$". Keep in mind that you are not given $\leq^*$; you must show that it exists.

3 Full Predicate Logic

In this chapter we enrich predicate logic by adding operation symbols and a special symbol for equality. We shall call this enriched language *full predicate logic* to distinguish it from the simpler **pure predicate logic** developed in the last[10] chapter. Full predicate logic is closer to the usual language of mathematics. Although it is in principle possible to express everything in the pure predicate logic of the previous chapter, in practice it is usually more convenient to develop mathematics in full predicate logic.

The set $\mathcal{P}_2$ of binary predicate symbols for full predicate logic always contains the equality symbol, $=$. Of course, we always write $\tau = \sigma$ in place of the more cumbersome $= (\sigma, \tau)$. The equality symbol should really be thought of as a logical symbol (like the propositional connectives and quantifiers) rather than part of the vocabulary because it will be interpreted in a fixed way in all models.

A *vocabulary for full predicate logic* consists of a set $\mathcal{P}$ of predicate symbols, and a set $\mathcal{F}$ of operation symbols, composed of disjoint unions

$$\mathcal{P} = \bigcup_{n=0}^{\infty} \mathcal{P}_n, \qquad \mathcal{F} = \bigcup_{n=0}^{\infty} \mathcal{F}_n.$$

The symbols in the set $\mathcal{P}_n$ are the n-ary predicate symbols, and the symbols in the set $\mathcal{F}_n$ are the n-ary operation symbols. These sets may or may not be empty (but $\mathcal{P}_2$ always contains the equality symbol). The symbols in $\mathcal{P}_0$ are called **proposition symbols** and the symbols in $\mathcal{F}_0$ are called *constant symbols*. The *primitive symbols of full predicate logic* are those of pure predicate logic together with the operation symbols from $\mathcal{F}$.

Variables, constant symbols, and operation symbols may be combined to form terms. The set of all *terms* is defined inductively by the following rules:

(T: *VAR*) Any variable is a term.

(T:$\mathcal{F}_0$) Any constant symbol is a term.

(T:$\mathcal{F}_n$) If $\mathbf{f} \in \mathcal{F}_n$ is a operation symbol, where $n > 0$, and $\tau_\mathbf{1}, \tau_\mathbf{2}, \ldots, \tau_\mathbf{n}$ are terms, then $\mathbf{f}(\tau_\mathbf{1}, \tau_\mathbf{2}, \ldots, \tau_\mathbf{n})$ is a term.

[10] One important difference in the chapter is that we build models from constant symbols and not variables as in the pure predicate logic. Well its hard to get five authors to agree to anything.

These rules are used repeatedly. For example, if y is a variable, c is a constant, f is binary, and g is unary, then $g(f(c, g(y)))$ is a term. Terms, like wffs, have parsing sequences. The above example is parsed as follows:

(1) c is a term by (T:$\mathcal{F}_0$).

(2) y is a term by (T: VAR).

(3) $g(y)$ is a term by (2) and (T:$\mathcal{F}_1$).

(4) $f(c, g(y))$ is a term, by (1), (3), and (T:$\mathcal{F}_2$).

(5) $g(f(c, g(y)))$ is a term by (4) and (T:$\mathcal{F}_1$).

We continue using the abbreviations and notational conventions introduced earlier in addition add the the usual mathematical conventions regarding infix notation and parentheses.

- The familiar binary operation sumbols $+$, $-$, and $*$ are written in infix notation so that $(x + y)$ is written instead of $+(x, y)$.
- The outer parantheses may be suppressed so that $x + y$ means $(x + y)$.
- Multiplication has a higher precedence than addition or subtraction so that $x + y * z$ means $x + (y * z)$ and not $(x + y) * z$.
- Operations of equal precedence associate to the left in the absence of explicit parantheses so that $x-y-z$ means $(x-y)-z$ and not $x-(y-z)$.

The set of wffs is defined inductively as before except that the argument places in the predicate symbols may be filled by terms. here are the rules.

(W:$\mathcal{P}_0$) Any propositional symbol is a wff.

(W:$\mathcal{P}_n$) If $\mathbf{p} \in \mathcal{P}_n$ ia a predicate symbol and $\tau_\mathbf{1}, \tau_\mathbf{2}, \ldots, \tau_\mathbf{n}$ are terms, then $\mathbf{p}(\tau_\mathbf{1}, \tau_\mathbf{2}, \ldots, \tau_\mathbf{n})$ is a wff.

(W:$\neg$) If $\mathbf{A}$ is a wff, then $\neg\mathbf{A}$ is a wff.

(W:$\wedge, \vee, \Rightarrow, \Leftrightarrow$) If $\mathbf{A}$ and $\mathbf{B}$ are wffs, then $[\mathbf{A} \wedge \mathbf{B}]$, $[\mathbf{A} \vee \mathbf{B}]$, $[\mathbf{A} \Rightarrow \mathbf{B}]$, and $[\mathbf{A} \Leftrightarrow \mathbf{B}]$ are wffs.

(W:$\forall, \exists$) If **A** is a wff, and **x** is a variable, then $\forall$**xA** and $\exists$**xA** are wffs.

(If it is necessary to explicitly specify the vocabulary $\mathcal{P} \cup \mathcal{F}$ used in the definition of wff, we shall refer to the wffs defined here as **built using the vocabulary** $\mathcal{P} \cup \mathcal{F}$.)

An atomic wff and basic wff are defined as before except that now abritrary terms way occupy the argument positions. Thus **atomic wffs** are those constructed by rules (W:$\mathcal{P}_0$) and (W:$\mathcal{P}_n$) above, while a **basic wff** is a wff which is either an atomic wff of the negation of an atomic wff.

Free and bound occurrences of variables in wffs are defined as before: an occurrence of a variable **x** in a wff **A** is a *bound occurrence* iff it is in a well-formed part of form **QxB** where (**Q** is either $\forall$ or $\exists$); all other occurrences are called *free*. A term is said to be **free for** the variable **x** in a wff **A** if no free occurrence of **x** occurs within a well-formed part of **A** of form **QyB** where the variable **y** occurs in τ and **Q** is either $\forall$ or $\exists$. Given a wff **A**, a variable **x**, and a term τ which is free for **x** in **A**, $\mathbf{A}(\mathbf{x}//\tau)$ is the wff obtained by replacing each free occurrence of **x** in **A** by τ. A *sentence* is a wff with no free variables.

3.1 Semantics.

For any set X and any natural number $n > 0$ recall that $REL_n(X)$ denote the set of all n-ary relations on X. A n-ary relation on X is a subset of the set X^n of all length n sequences $(x_1, x_2, \ldots, x_n)$ with elements from X so that

$$R \in REL_n(X) \text{ iff } R \subset X^n.$$

Also denote by $FUN_n(X)$ the set of all functions from X^n to X. Thus

$$f \in FUN_n(X) \text{ iff } f : X^n \longrightarrow X.$$

We also define $REL_0(X)$ to be the two element set of truth values,

$$REL_0(X) = \{\top, \bot\}$$

and $FUN_0(X)$ to be X itself,

$$FUN_0(X) = X.$$

A **model for full predicate logic of type** $\mathcal{P} \cup \mathcal{F}$ is a system $\mathcal{M}$ consisting of a non-empty set $U_{\mathcal{M}}$ called the **universe of the model** $\mathcal{M}$, and for each $n \geq 0$ a pair of functions

$$\mathcal{P}_n \longrightarrow REL_n(U_{\mathcal{M}}) : \mathbf{p} \mapsto \mathbf{p}_{\mathcal{M}}$$

and

$$\mathcal{F}_n \longrightarrow FUN_n(U_{\mathcal{M}}) : \mathbf{f} \mapsto \mathbf{f}_{\mathcal{M}}$$

which assign to each predicate symbol $\mathbf{p}$ a relation $\mathbf{p}_{\mathcal{M}}$ and to each function symbol $\mathbf{f}$ a function $\mathbf{f}_{\mathcal{M}}$. Unless otherwise stated, we shall assume that the model **respects equality** meaning that

$$=_{\mathcal{M}} \quad = \quad \{(a, b) \in U_{\mathcal{M}} : a = b\}.$$

We shall define the notion that $\mathcal{M} \models \mathbf{A}$ where $\mathbf{A}$ is a sentence, this time without recourse to valuations as in the previous chapter. The notation we use here is favored by many logicians over the notation using valuations, but according to section B the two notations are entirely equivalent. The one drawback of the new notation is that we must extend the notion of wff. To this end we define a **wff with constants from the model** $\mathcal{M}$ to be a wff built up using the vocabulary $\mathcal{P} \cup \mathcal{F}'$ where $\mathcal{F}'$ is obtained from $\mathcal{F}$ by replacing $\mathcal{F}_0$ by $\mathcal{F}_0 \cup U_{\mathcal{M}}$. In other words a wff with constants from $\mathcal{M}$ may have elements of the model in the argument places of the function and predicate symbols.

Now we can extend the function $\mathcal{F}_0 \longrightarrow U_{\mathcal{M}}$ to a function which assigns to each term τ a value $\tau_{\mathcal{M}} \in \mathbf{U}_{\mathcal{M}}$ inductively as follows.

(M:$\mathcal{F}_0$) If $u \in U_{\mathcal{M}}$ then $u_{\mathcal{M}} = u$.

(M:$\mathcal{F}_n$) If $\tau_{\mathbf{1}}, \tau_{\mathbf{2}}, \ldots, \tau_{\mathbf{n}}$ are terms and $\mathbf{f} \in \mathcal{F}_n$ is a function symbol, then

$$\mathbf{f}(\tau_{\mathbf{1}}, \tau_{\mathbf{2}}, \ldots, \tau_{\mathbf{n}})_{\mathcal{M}} = \mathbf{f}_{\mathcal{M}}(\tau_{\mathbf{1}\mathcal{M}}, \tau_{\mathbf{2}\mathcal{M}}, \ldots, \tau_{\mathbf{n}\mathcal{M}})$$

Now we define $\mathcal{M} \models \mathbf{A}$ where $\mathbf{A}$ is a sentence with constants from $\mathcal{M}$ much as before. The inductive definition is as follows.

(M:$\mathcal{P}_0$) $\quad \mathcal{M} \models \mathbf{p} \qquad$ iff $\mathbf{p}_{\mathcal{M}} = \top$;

(M:$\mathcal{P}_n$) $\mathcal{M} \models \mathbf{p}(\tau_\mathbf{1}, \tau_\mathbf{2}, \ldots, \tau_\mathbf{n})$ iff $(\tau_{\mathbf{1}\mathcal{M}}, \tau_{\mathbf{2}\mathcal{M}}, \ldots, \tau_{\mathbf{n}\mathcal{M}}) \in \mathbf{p}_\mathcal{M}$;

(M:$\neg$) $\mathcal{M} \models \neg\mathbf{A}$ iff not - $\mathcal{M} \models \mathbf{A}$;

(M:$\wedge$) $\mathcal{M} \models \mathbf{A} \wedge \mathbf{B}$ iff $\mathcal{M} \models \mathbf{A}$ and $\mathcal{M} \models \mathbf{B}$;

(M:$\vee$) $\mathcal{M} \models \mathbf{A} \vee \mathbf{B}$ iff $\mathcal{M} \models \mathbf{A}$ or $\mathcal{M} \models \mathbf{B}$;

(M:$\Rightarrow$) $\mathcal{M} \models \mathbf{A} \Rightarrow \mathbf{B}$ iff $\mathcal{M} \models \mathbf{A}$ implies $\mathcal{M} \models \mathbf{B}$;

(M:$\Leftrightarrow$) $\mathcal{M} \models \mathbf{A} \Leftrightarrow \mathbf{B}$ iff ($\mathcal{M} \models \mathbf{A}$ if and only if $\mathcal{M} \models \mathbf{B}$)

(M:$\forall$) $\mathcal{M} \models \forall\mathbf{x}\mathbf{A}$ iff $\mathcal{M} \models \mathbf{A}(\mathbf{x}//u)$ for every $u \in U_\mathcal{M}$;

(M:$\exists$) $\mathcal{M} \models \exists\mathbf{x}\mathbf{A}$ iff $\mathcal{M} \models \mathbf{A}(\mathbf{x}//u)$ for some $u \in U_\mathcal{M}$.

The fact that the model respects equality may be succinctly expressed by the assertion

$$\mathcal{M} \models \tau = \sigma \textbf{ if and only if } \tau_\mathcal{M} = \sigma_\mathcal{M}.$$

3.2 Tableaus.

In full predicate logic, a tableau may be formed using all the rules for tableaus in proposition logic (see figure1) plus additional rules for handling variable free terms and the equality relation. A *labeled tree for full predicate logic* is as for propositional logic, except that now the wffs are sentences (i.e. wffs with no free variables) of full predicate logic. We assume for tableau proofs in full predicate logic that we have an inexhaustible supply of new constant symbols.

A tableau for full predicate logic is as before except we allow the following rules also:

We first generalize the rules for quantifiers to allow the substitution of terms:

$\boxed{\forall}$ t has an ancestor $\forall\mathbf{x}\mathbf{A}$ and a child $\mathbf{A}(\mathbf{x}//\tau)$ where τ is a term with no variables in it;

$\boxed{\neg\forall}$ t has an ancestor $\neg\forall\mathbf{x}\mathbf{A}$ and a child $\neg\mathbf{A}(\mathbf{x}//c)$ where c is a constant which does not occur in any ancestor of t;

$\boxed{\exists}$ t has an ancestor $\exists\mathbf{x}\mathbf{A}$ and a child $\mathbf{A}(\mathbf{x}//c)$ where c is a constant which does not occur in any ancestor of t;

$\boxed{\neg\exists}$ t has an ancestor $\neg\exists\mathbf{x}\mathbf{A}$ and a child $\neg\mathbf{A}(\mathbf{x}//\tau)$ where τ is a term with no variables in it.

In addition we allow the following equality substitution rules:

$\boxed{=1}$ t has an ancestor $\mathbf{B}(\ldots\tau\ldots)$, another ancestor of form $\tau = \sigma$, and one child $\mathbf{B}(\ldots\sigma\ldots)$.

$\boxed{=2}$ t has an ancestor $\mathbf{B}(\ldots\tau\ldots)$, another ancestor of form $\sigma = \tau$, and one child $\mathbf{B}(\ldots\sigma\ldots)$.

$\boxed{=3}$ t has one child $\sigma = \sigma$.

Here $\mathbf{B}(\ldots\tau\ldots)$ and $\mathbf{B}(\ldots\sigma\ldots)$ denote basic wffs (i.e. an atomic formula or the negation of an atomic formula) such that the latter results from the former by replacing one occurrence of the term τ by the term σ. It is not required that occurrence of τ be an entire argument of the proposition symbol $\mathbf{B}$, it may be a part of an argument; for example, we may take $a = b$ for $\tau = \sigma$, $p(f(a), a, b)$ for $\mathbf{p}(\ldots\tau\ldots)$, and $p(f(b), a, b)$ for $\mathbf{p}(\ldots\sigma\ldots)$. The rules $\boxed{=1}$ and $\boxed{=2}$ differ only in that in the former the equality ancestor is $\tau = \sigma$ while in the latter it is $\sigma = \tau$.

The rule is justified by the fact that if $\mathbf{A}$, $\mathbf{B}$, and $\tau = \sigma$ are as in the rule, then

$$\mathcal{M} \models \mathbf{A} \wedge [\tau = \sigma] \Rightarrow \mathbf{B}$$

for any model $\mathcal{M}$. And for any model and any term σ

$$\mathcal{M} \models \sigma = \sigma$$

(In the TABLEAU program, the equality rule is invoked by typing the G key at the node $\mathbf{A}$ to put $\mathbf{A}$ in the Get box, typing the S key ate the node

$\tau = \sigma$ to put either $\tau = \sigma$ or $\sigma = \tau$ into the Sub box (pressing the S key again toggles between these two), then going to the end of the branch and typing the E key to extend the tableau.) The third rule is invoked by using the = key.

The basic definitions are the same as before except for the addition of these new rules. A branch $\mathbf{\Gamma}$ of a tableau is said to be *contradictory* iff $\mathbf{\Gamma}$ contain a pair of wffs of the form $\mathbf{A}$ and $\neg\mathbf{A}$.

The notions of a tableau confutation and a tableau proof are defined as before. A tableau $\mathbf{T}$ is said to be a *confutation* of a set of sentences $\mathbf{H}$ if $\mathbf{T}$ is a tableau with hypothesis set $\mathbf{H}$ and every branch of $\mathbf{T}$ is contradictory. A *tableau proof* of $\mathbf{A}$ from $\mathbf{H}$ is a tableau confutation of $\mathbf{H} \cup \{\neg\mathbf{A}\}$.

Example 1. A tableau proof of $\forall x \forall y[x = y \Rightarrow f(x) = f(y)]$.

(1)	$\neg\forall x \forall y[x = y \Rightarrow f(x) = f(y)]$	$\neg$ to be proved
	$\vert$	
(2)	$\neg\forall y[a = y \Rightarrow f(a) = f(y)]$	by (1)
	$\vert$	
(3)	$\neg[a = b \Rightarrow f(a) = f(b)]$	by (2)
	$\vert$	
(4)	$a = b$	by (3)
	$\vert$	
(5)	$f(a) \neq f(b)$	by (3)
	$\vert$	
(6)	$f(b) \neq f(b)$	by (4) and (5)
	$\vert$	
(7)	$f(b) = f(b)$	by equality rule 3

Example 2. A tableau proof of $\forall x \forall y \forall z[x = y \wedge y = z \Rightarrow x = z]$.

(1)	$\neg\forall x \forall y \forall z[x = y \wedge y = z \Rightarrow x = z].$	$\neg$ to be proved
	$\vert$	
(2)	$\neg\forall y \forall z[a = y \wedge y = z \Rightarrow a = z].$	by (1)
	$\vert$	
(3)	$\neg\forall z[a = b \wedge b = z \Rightarrow a = z].$	by (2)
	$\vert$	
(4)	$\neg[a = b \wedge b = c \Rightarrow a = c].$	by (3)
	$\vert$	
(5)	$a = b \wedge b = c.$	by (4)
	$\vert$	
(6)	$a \neq c$	by (4)
	$\vert$	
(7)	$a = b$	by (5)
	$\vert$	
(8)	$b = c$	by (5)
	$\vert$	
(9)	$a = c$	by (7) and (8)

3.3 Soundness

The proof of the soundness theorem for full predicate logic much as before. We restate the theorem here, since we have now formulated our semantics using constants from the model rather than valuations.

Lemma 3.3.1 *Let* $\mathbf{H}$ *be a set of sentences of full predicate logic and* $\mathbf{T}$ *be a finite tableau in predicate logic with hypothesis set* $\mathbf{H}$*. If* $\mathcal{M}$ *is a model of the set of sentences of* $\mathbf{H}$*, then there is a branch* $\mathbf{\Gamma}$ *such that* $\mathcal{M} \models \mathbf{A}$ *for every wff* $\mathbf{A}$ *on* $\mathbf{\Gamma}$*, that is,* $\mathcal{M} \models \mathbf{\Gamma}$

Theorem 3.3.2 *(Soundness Theorem.) If a sentence* $\mathbf{A}$ *has a tableau proof, then* $\mathbf{A}$ *is valid. If* $\mathbf{A}$ *has a tableau proof from a a set* $\mathbf{H}$ *of sentences, then* $\mathbf{A}$ *is a semantic consequence of* $\mathbf{H}$*, i. e. every model of* $\mathbf{H}$ *is a model of* $\mathbf{A}$*.*

3.4 Completeness

The completeness theorem for full predicate logic is similar to the one for predicate logic with some additional twists.

Theorem 3.4.1 *(Completeness Theorem.) Let* $\mathbf{H}$ *be a set of sentences of full predicate logic. If* $\mathbf{H}$ *has no models, then* $\mathbf{H}$ *has a tableau confutation. If* $\mathbf{H}$ *does not have a tableau confutation, then* $\mathbf{H}$ *has a model.*

Let $\mathbf{\Delta}$ be any set of sentences and denote by $U_{\mathbf{\Delta}}$ the set of all variable free terms τ which have an occurrence in some wff of $\mathbf{\Delta}$.

We call a set $\mathbf{\Delta}$ of wffs *closed under the equality rules* iff any basic wff obtained from two wffs of $\mathbf{\Delta}$ by an equality substitution is again a element of $\mathbf{\Delta}$, in other words, iff for all terms τ and σ in $U_{\mathbf{\Delta}}$ and all basic wffs $\mathbf{B}(\ldots\tau\ldots)$ the following conditions hold:

$[=1]$ if $[\tau = \sigma] \in \mathbf{\Delta}$ and $\mathbf{B}(\ldots\tau\ldots) \in \mathbf{\Delta}$ then $\mathbf{B}(\ldots\sigma\ldots) \in \mathbf{\Delta}$.

$[=2]$ if $[\sigma = \tau] \in \mathbf{\Delta}$ and $\mathbf{B}(\ldots\tau\ldots) \in \mathbf{\Delta}$ then $\mathbf{B}(\ldots\sigma\ldots) \in \mathbf{\Delta}$.

$[=3]$ $[\tau = \tau] \in \mathbf{\Delta}$

A set $\mathbf{\Delta}$ of sentences is called **contradictory** iff it contains a pair of wffs of form $\mathbf{A}$, $\neg\mathbf{A}$.

The definition of a finished set for full predicate logic is verbatim the same as the definition of a finished set of wffs for pure predicate logic given before except that now

- the set $U_{\mathbf{\Delta}}$ has its new meaning (the set of all variable free terms τ which have an occurrence in some wff of $\mathbf{\Delta}$,
- the set $\mathbf{\Delta}$ must be closed under equality rules and the new quantifier rules.

The complete definition may be found in section E.

Lemma 3.4.2 *(Finished Branch Lemma.) Suppose* $\mathbf{\Delta}$ *is finished set of sentences. Let* $\mathcal{M}$ *be any model for full predicate logic such that each element of the universe set* $U_{\mathcal{M}}$ *of* $\mathcal{M}$ *is named by at least one term in* $U_{\mathbf{\Delta}}$*:*

$$U_{\mathcal{M}} = \{\tau_{\mathcal{M}} : \tau \in \mathbf{U_\Delta}\}.$$

If $\mathcal{M} \models \mathbf{B}$ *for every basic wff* $\mathbf{B} \in \mathbf{\Delta}$*, then* $\mathcal{M} \models \mathbf{A}$ *for every wff* $\mathbf{A} \in \mathbf{\Delta}$*, that is,* $\mathcal{M} \models \mathbf{\Delta}$*.*

The version of the finished branch lemma just given does not assert the existence of the required model $\mathcal{M}$. In the case of pure predicate logic this point was trivial: we simply took $U_{\mathcal{M}} = U_{\mathbf{\Delta}}$ and defined $\mathcal{M} \models \mathbf{A}$ for atomic $\mathbf{A}$ by $\mathcal{M} \models \mathbf{A}$ iff $\mathbf{A} \in \mathbf{\Delta}$. The present situation is a bit more complicated, for the set $\mathbf{\Delta}$ may contain a sentence of form $\tau = \sigma$ where τ and σ are distinct terms. The model just described will not do since it will not respect equality.

To overcome this difficulty we need three lemmas. In all three lemmas we assume that $\mathbf{\Delta}$ be a finished set of sentences. We also call terms τ and σ in $U_{\mathbf{\Delta}}$ *equivalent* iff $[\tau = \sigma] \in \mathbf{\Delta}$. We write $\tau \equiv \sigma$ as an abbreviation for "τ and σ are equivalent".

Lemma 3.4.3 *Suppose that* $\mathbf{\Delta}$ *be a finished set of sentences. Then* $\equiv$ *is an equivalence relation. That is,*

(reflexivity) $\tau \equiv \tau$*;*

(symmetry) *if* $\tau \equiv \sigma$ *then* $\sigma \equiv \tau$*;*

(transitivity) *if* $\tau_{\mathbf{1}} \equiv \tau_{\mathbf{2}}$ *and* $\tau_{\mathbf{2}} \equiv \tau_{\mathbf{3}}$ *then* $\tau_{\mathbf{1}} \equiv \tau_{\mathbf{3}}$*;*

for $\tau, \sigma, \tau_{\mathbf{1}}, \tau_{\mathbf{2}}, \tau_{\mathbf{3}} \in \mathbf{U_\Delta}$*.*

Lemma 3.4.4 *Let* $\mathbf{\Delta}$ *be a finished set of sentences. Suppose* $\tau_{\mathbf{1}}, \tau_{\mathbf{2}}, \ldots, \tau_{\mathbf{n}}, \sigma_{\mathbf{1}}, \sigma_{\mathbf{2}}, \ldots, \sigma_{\mathbf{n}} \in \mathbf{U_\Delta}$ *and* $\mathbf{f} \in \mathcal{F}_n$*. If*

$$\tau_{\mathbf{1}} \equiv \sigma_{\mathbf{1}},\ \tau_{\mathbf{2}} \equiv \sigma_{\mathbf{2}},\ \ldots,\ \tau_{\mathbf{n}} \equiv \sigma_{\mathbf{n}},$$

then

$$\mathbf{f}(\tau_{\mathbf{1}}, \tau_{\mathbf{2}}, \ldots, \tau_{\mathbf{n}}) \equiv \mathbf{f}(\sigma_{\mathbf{1}}, \sigma_{\mathbf{2}}, \ldots, \sigma_{\mathbf{n}}).$$

Lemma 3.4.5 *Let* $\mathbf{\Delta}$ *be a finished set of sentences. Suppose* $\tau_\mathbf{1}, \tau_\mathbf{2}, \ldots, \tau_\mathbf{n}, \sigma_\mathbf{1}, \sigma_\mathbf{2}, \ldots, \sigma_\mathbf{n} \in \mathbf{U_\Delta}$ *and* $\mathbf{p} \in \mathcal{P}_n$. *If*

$$\tau_\mathbf{1} \equiv \sigma_\mathbf{1},\ \tau_\mathbf{2} \equiv \sigma_\mathbf{2},\ \ldots,\ \tau_\mathbf{n} \equiv \sigma_\mathbf{n},$$

then

$$\mathbf{p}(\tau_\mathbf{1}, \tau_\mathbf{2}, \ldots, \tau_\mathbf{n}) \in \mathbf{\Delta} \Longleftrightarrow \mathbf{p}(\sigma_\mathbf{1}, \sigma_\mathbf{2}, \ldots, \sigma_\mathbf{n}) \in \mathbf{\Delta}.$$

Lemma 3.4.6 *Let* $\mathbf{\Delta}$ *be a finished set of sentences. Then there is a model* $\mathcal{M}$ *such that for every atomic sentence* $\mathbf{A}$ *we have*

$$\mathcal{M} \models \mathbf{A} \Longleftrightarrow \mathbf{A} \in \mathbf{\Delta}.$$

Proof: For each $\tau \in \mathbf{U_\Delta}$ let $[\tau]$ denote the equivalence class of τ:

$$[\tau] = \{\sigma \in \mathbf{U_\Delta} : \tau \equiv \sigma\}.$$

By lemma 3.4.3 we have

$$[\tau] = [\sigma] \Longleftrightarrow \tau \equiv \sigma.$$

We define the universe of our model $\mathcal{M}$ to be the set of equivalence classes:

$$U_\mathcal{M} = \{[\tau] : \tau \in \mathbf{U_\Delta}\}.$$

Now by lemma 3.4.4 each function symbol $\mathbf{f} \in \mathcal{F}_n$ determines a function $\mathbf{f}_\mathcal{M} \in FUN_n(U_\mathcal{M})$ by the condition

$$\mathbf{f}_\mathcal{M}([\tau_\mathbf{1}], [\tau_\mathbf{2}], \ldots, [\tau_\mathbf{n}]) = [\mathbf{f}(\tau_\mathbf{1}, \tau_\mathbf{2}, \ldots, \tau_\mathbf{n})]$$

and by lemma 3.4.5 each predicate symbol $\mathbf{p} \in \mathcal{P}_n$ determines a relation $\mathbf{p}_\mathcal{M} \in REL(U_\mathcal{M})$ by the condition

$$([\tau_\mathbf{1}], [\tau_\mathbf{2}], \ldots, [\tau_\mathbf{n}]) \in \mathbf{p}_\mathcal{M} \Longleftrightarrow \mathbf{p}(\tau_\mathbf{1}, \tau_\mathbf{2}, \ldots, \tau_\mathbf{n}) \in \mathbf{U_\Delta}.$$

This model $\mathcal{M}$ obviously satisfies the conclusion of the lemma.

As in pure predicate logic, a tableau in full predicate logic is *finished* iff every branch is either finished or else both finite and contradictory.

Lemma 3.4.7 *Every finite set of sentences is the hypothesis set of a finished tableau.*

Recall that a sentence **A** is *valid* iff it is true in every model, and is a *semantic consequence* of a set of sentences **H** iff it is true in every model of **H**.

Theorem 3.4.8 *(Completeness Theorem, Second Form.) If a sentence* **A** *is a semantic consequence of a set of sentences* **H** *then there is a tableau proof of* **A** *from* **H**. *In particular, a valid sentence has a tableau proof.*

Theorem 3.4.9 *(Compactness Theorem.) Let* **H** *be any set of sentences of full predicate logic. If every finite subset of* **H** *has a model, then* **H** *has a model.*

3.5 Peano Arithmetic

Peano arithmetic is a particular set of sentences in full predicate logic which codifies the properties of the natural numbers and has a central role in mathematics. These sentences are called Peano's axioms. The vocabulary consists of a constant symbol 0, a unary function symbol s called the successor function, and two infix binary function symbols + and * called the sum and product functions. The most important model of Peano arithmetic is the model **N**with universe 0,1,2,... and the usual interpretations of the symbols 0, s, +, and *. This model is called the *standard model of Peano arithmetic.* It will be seen in the last part of the course that there also exist other models of Peano arithmetic.

Peano arithmetic is an infinite set of sentences, consisting of finitely many basic axioms and an infinite collection of sentences called the induction principle. Proofs using the induction principle are called proofs by induction on the natural numbers. Throughout the course we have seen informal proofs by induction on the natural numbers and on other objects such as wffs and tableaus. Here are the Peano axioms.

BASIC AXIOMS

$\forall x \neg s(x) = 0$
$\forall x \forall y [s(x) = s(y) \Rightarrow x = y]$
$\forall x \; x + 0 = x$
$\forall x \forall y \; x + s(y) = s(x + y)$
$\forall x \; x * 0 = 0$
$\forall x \forall y \; x * s(y) = (x * y) + x$

INDUCTION PRINCIPLE

The infinite set consisting of all sentences of the form

$$\forall y_1 ... \forall y_n [B(0) \wedge \forall x [B(x) \Rightarrow B(s(x))] \Rightarrow \forall x B(x)]$$

where B is a wff with free variables $x, y_1, ..., y_n$.

This sentence is called the induction principle for the wff B in the variable x. To improve readability we wrote B(x) for B, B(0) for B(x//0), and B(s(x)) for B(x//s(x)). In a formal tableau proof, the cases of the induction principle which are needed for the proof are included in the hypothesis list.

3.6 Computer problem

There are seven problems, called

GROUP1.TBU, GROUP2.TBU,

CALC1.TBU, CALC2.TBU, CALC3.TBU,

ZPLUS.TBU, AND PLUS.TBU.

You should load the problem in with the TABLEAU program, then make a proof sketch on paper, and finally use your proof sketch as a guide to make a formal tableau proof with the TABLEAU program. In many cases your sketch will contain a string of equations. As usual, you should file your answer on your diskette using the TABLEAU program, with the name of the problem preceded by an A.

The problems use predicate logic with function symbols and equality substitutions. Here are some comments on the problems. You should try the problems with shorter solutions (fewer nodes) first.

GROUP1. (16 nodes). The hypotheses are axioms from group theory with a binary infix operation $*$ and a constant symbol e for the identity element. The first hypothesis is the associative law, the second hypothesis is that every element has a right inverse, and the other two hypotheses state that e is a two sided identity element (it will be shown as an example in class that the fourth hypotheses can be proved from the other three). The sentence to be proved is that every element has a left inverse.

GROUP2. (21 nodes). The hypotheses are the axioms for groups. The sentence to be proved is the cancellation law.

CALC1. (6 nodes). The hypotheses state that the function f is onto and that g is an inverse function of f. The sentence to be proved is that f is an inverse function of g.

CALC2. (35 nodes). This is the theorem from calculus which states that a bounded increasing real function f(x) approaches a limit as x approaches infinity. The vocabulary has a constant c, a unary function f, and a binary infix predicate $<$. The first hypothesis states that the function f is increasing, the second and third hypotheses state that c is the least upper bound of the range of f, and the last two hypotheses are needed facts about the order relation. The sentence to be proved states that c is the limit of f(x) as x approaches infinity.

CALC3. (87 nodes). This is the main part of the the Intermediate Value Theorem from calculus. The vocabulary has constants c and o, a unary function f, a binary infix predicate $\leq$, and a ternary predicate p. The first two hypotheses are facts about the order relation. The next two hypotheses state that c is the least upper bound of the set of all x such that $f(x) \leq 0$. The fifth hypothesis defines the relation p(x,y,z) to mean that y belongs to the open interval (x,z). The long sixth hypothesis uses the relation p to state that the function f(x) is continuous for all x. The sentence to be proved is that $f(c) \leq 0$.

(A similar proof will show that $0 \leq f(c)$. This leads to the theorem that if f is continuous and $f(a) < 0 < f(b)$ then there is a point c between a and b with $f(c) = 0$.)

The problems ZPLUS and PLUS are examples of proofs using the induction principle for the natural numbers. The vocabulary has a constant 0 for zero, a unary function s for successor, and a binary function + (written in infix notation x + y) for the sum. The hypotheses in each problem give the rules for computing the sum. The other hypotheses are cases of the induction principle for natural numbers.

ZPLUS. (12 nodes). The third hypothesis is the induction principle for the wff 0+x=x in the variable x. The sentence to be proved is that for all x, 0+x=x.

PLUS. (38 nodes). The third and fourth hypotheses are the induction principle for $\forall y \;\; x + y = y + x$ in the variable x, and the induction principle for $x + y = y + x$ in the variable y. The last hypothesis is the sentence proved in the preceding problem. The sentence to be proved is the commutative law for the sum.

If you get stuck, you may look at the hints below.
HINT FOR GROUP1: If $a * b = e$ and $b * c = e$ then

$$a = a * e = a * (b * c) = (a * b) * c = e * c = c,$$

so that $b * a = e$.
HINT FOR GROUP2: If $a * c = b * c$ and $c * d = e$ then

$$a = a * e = a * (c * d) = (a * c) * d = (b * c) * d = b * (c * d) = b * e = b.$$

HINT FOR PLUS: If $\forall y \ \ [a + y = y + a]$ and $s(a) + b = b + s(a)$, then

$$s(a) + s(b) = s(s(a) + b) = s(b + s(a)) = s(s(b + a)) = s(s(a + b)) =$$

$$= s(a + s(b)) = s(s(b) + a) = s(b) + s(a).$$

3.7 Exercises

1. Let B be the wff

$$y = s(x) \wedge \exists y \ x + y = z.$$

a) Write down the wff B(x//0)
b) Is the term s(x) free for x in B? If it is, write down the wff B(x//s(x))
c) Is the term $x * y$ free for x in B? If it is, write down the wff B(x//$x * y$)
d) Is the term $x * y$ free for y in B? If it is, write down the wff B(y//$x * y$)
e) Write down the sentence B(v) where v is a valuation such that v(x)=2, v(y)=4, v(z)=6.

2. Give a tableau proof of the sentence

$$\forall x \ [0 * x = 0 \Rightarrow 0 * s(x) = 0]$$

from the set of hypotheses

$$\forall x \ \ x + 0 = x$$

$$\forall x \forall y \ \ x * s(y) = x * y + x$$

3. Give a tableau proof of the sentence

$$\forall x \ \ 0 * x = 0$$

from Peano arithmetic. Here is a start (showing only the axioms of Peano arithmetic which are needed for your proof).

$$\neg\forall x \;\; 0 * x = 0$$

$$\forall x \;\; x + 0 = x$$

$$\forall x \;\; x * 0 = 0$$

$$\forall x \forall y \;\; x * s(y) = x * y + x$$

$$0 * 0 = 0 \wedge \forall x \;\; [0 * x = 0 \Rightarrow 0 * s(x) = 0] \Rightarrow \forall x \;\; 0 * x = 0$$

4. Suppose that:
   T is a finite tableau in predicate logic.
   H is the set of hypotheses of T.
   A is a wff whose only free variable is x.
   b is a constant symbol.
   Every branch of T is either contradictory or contains the sentence **A**(x//b).
   Describe a simple way to change T into a tableau proof of $\exists x$ **A** from H.

5. Suppose that:

   1. H is a finite set of sentences of predicate logic with no equality or function symbols.

   2. No constant symbols occur in H.

   3. H has at least one model.

   4. H has a finished tableau with fewer than 100 nodes.

   Prove that H has a model whose universe has fewer than 100 elements.

6. Give an example of a single wff A in predicate logic with variables x,y,and z such that x is free for y but not for z, y is free for z but not for x, z is not free for x, and z is not free for y.

7. Give a tableau proof of the sentence

$$\forall y \;\; [R(y) \Leftrightarrow \exists x \;\; [R(x) \wedge x = y]].$$

In the next two problems give informal direct proofs using the rules for tableaus and the following additional rules:

a) If C follows from a branch $\mathbf{\Gamma}$ by propositional logic, then C may be added to the end of $\mathbf{\Gamma}$.

b) (To prove $C \Rightarrow D$) Temporarily assume C and prove D, then add $C \Rightarrow D$ to the end of the branch $\mathbf{\Gamma}$.

c) (To prove $\forall x \;\; C$) Prove C(x//b) where b is new, then add $\forall x \;\; C$ to the end of the branch $\mathbf{\Gamma}$.

8.

$$\forall x \forall y \forall z \;\; [glb(x,y,z) \Leftrightarrow z \leq x \wedge z \leq y \wedge \forall t \;\; [t \leq x \wedge t \leq y \Rightarrow t \leq z]]$$

$$\vdash$$

$$\forall x \forall y \forall z \forall t \;\; [glb(x,y,z) \wedge glb(x,y,t) \Rightarrow z \leq t]$$

9.

$$\forall x \forall y \;\; [x \subset y \Leftrightarrow \forall z \;\; [z \in x \Rightarrow z \in y]]$$

$$\forall x \forall y \forall z \;\; [z = x \cup y \Leftrightarrow \forall t \;\; [t \in z \Leftrightarrow t \in x \vee t \in y]]$$

$$\vdash$$

$$\forall x \forall y \forall z \;\; [z = x \cup y \Rightarrow x \subset z]$$

10. Give a finished set for the set of sentences

$$\forall x \;\; p(x,x)$$

$$\exists x \forall y \;\; p(x,y)$$

$$\exists x \forall y \;\; p(y,x)$$

$$\exists x \exists y \;\; [\neg p(x,y) \wedge \neg p(y,x)]$$

with the four constants 0,1,2,3 and no free variables.

We say that a model $(P, \leq)$ is a *linear order* (where $\leq$ is a binary relation on the universe P) iff

1. $\forall x \in P\ x \leq x$
2. $\forall x \in P \forall y \in P \forall z \in P\ [x \leq y \wedge y \leq z \Rightarrow x \leq z]$
3. $\forall x \in P \forall y \in P\ [x \leq y \wedge y \leq x \Rightarrow x = y]$
4. $\forall x \in P \forall y \in P\ [x \leq y \vee y \leq x]$

If $(P, \leq)$ satisfies only the first three it is called a *partial order*.

11. Let P be the set of all subsets of $\{0, 1\}$ (there are eight). Let $\leq$ be the subset relation on P. Show the this is a partial order. Draw a diagram of it.

12. Let S be the set of all strings in the alphabet $\{a, b, c, ..., z\}$. Let $\leq$ be the binary relation on S defined by $s \leq t$ iff s is a substring of t. Show that $(S, \leq)$ is a partial order.

Given a partial order define a strict partial order by $x < y \Leftrightarrow x \leq y \wedge x \neq y$. 13. Which of the above sentences are true if $\leq$ is replaced by $<$? 14.

Show that in a partial order $x < y \Leftrightarrow x \leq y \wedge\ /y \leq x$. 15. Show that in a

linear order $x < y \Leftrightarrow /y \leq x$. And show by example that this is not true in a partial order.

16. Find a linear order $(P, \leq)$ which satisfies all of the following:

$$(P, \leq) \models \forall x \exists y\ x < y$$

$$(P, \leq) \models \forall x \exists y\ y < x$$

$$(P, \leq) \models \forall x \forall y\ [x < y \Rightarrow \exists z [x < z \wedge z < y]$$

17. Show by induction that for any finite partial order $(P, \leq)$ (i.e. P is finite) there is a linear order $(P, \leq^*)$ which extendes $\leq$, i.e. for every $a, b \in P$ if $a \leq b$, then $a \leq^* b$.

18. Use the Compactness Theorem for **Propositional Logic** and the last problem to show that every partial order (finite or infinite) can be extended to a linear order.
Hint: Let $(P, \leq)$ be any partial order. Let $\mathcal{P}_0 = \{R_{ab} \ : \ a, b \in P\}$. Consider interpreting the symbol R_{ab} as "$a \leq^* b$". Keep in mind that you are not given $\leq^*$; you must show that it exists.

19. Prove using the compactness theorem of propositional logic that for any set X and binary relation $R \subset X \times X$ if

1. for every finite $X' \subset X$ there exists a 1-1 function $f : X' \mapsto X$ such that $\forall x \in X' < x, f(x) > \in R$; and
2. for every $x \in X$
$$\{y \in X \ : < x, y > \in R\}$$
is finite;

then there exists a 1-1 function $f : X \mapsto X$ such that

$$\forall x \in X < x, f(x) > \in R.$$

Show that the second item above is necessary by giving a counterexample.

4 Computable Functions

In this chapter we explore what it means for a function to be "computable".

4.1 Numerical Functions.

A *numerical function* is a function f defined on a set of n-tuples of natural numbers:

$$Dom(f) \subset \mathbf{N}^n$$

and taking natural numbers as values:

$$Ran(f) \subset \mathbf{N}.$$

The positive integer n is called the *arity* of the numerical function; it is the number of inputs $x_1, x_2, \ldots, x_n$ required to produce an output $f(x_1, x_2, \ldots, x_n)$. A numerical function with arity n is also called an *n- ary numerical function.* (This usage arose from more traditional terminology where *unary* meant 1-ary, *binary* meant 2 -ary, *ternary* meant 3-ary, etc.)

The function f is called *totally defined* or *total* iff $f(x_1, x_2, \ldots, x_n)$ is defined for *all* inputs $(x_1, x_2, \ldots, x_n)$ i.e. iff

$$Dom(f) = \mathbf{N}^n.$$

When we wish to emphasize that a numerical function is not assumed to be total we call it *partially defined* or simply *partial.* (It is important to remember that the adjective *partial* is redundant: a partial numerical function and a numerical function are the same[11] thing. When we call a function partial we do not exclude the possibility that the function is total.)

4.2 Examples.

Some well known total numerical functions include

- The zero function ($n = 1$) defined by

  $$\mathbf{0}(x) = 0$$

  for $x \in \mathbf{N}$;

[11] At least one of the coauthor's prefers that the default of numerical function (recursive function, computable function, etc.) is total function. It is better in most cases always to be clear by saying "total" or "partial".

- The succesor function ($n = 1$) defined by

$$s(x) = x + 1$$

for $x \in \mathbf{N}$;

- The addition function ($n = 2$) defined by

$$sum(x, y) = x + y$$

for $(x, y) \in \mathbf{N}^2$;

- The multiplication function ($n = 2$) defined by

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

for $(x, y) \in \mathbf{N}^2$;

- The projection functions defined by

$$I_k^n(x_1, x_2, \ldots, x_n) = x_k$$

for $(x_1, x_2, \ldots, x_n) \in \mathbf{N}^n$ and $k = 1, 2, \ldots, n$.

Some well known partial numerical functions which are not total include

- The subtraction function ($n = 2$) defined by

$$sub(x, y) = x - y$$

for $(x, y) \in Dom(sub) = \{(x, y) \in \mathbf{N}^2 : y \leq x\}$;

- The partial quotient and partial remainder functions characterized by

$$q = qt_0(x, y) and\ r = rm_0(x, y) \Longleftrightarrow y = qx + r and\ 0 \leq r < x$$

for $(x, y) \in Dom(qt_0) = Dom(rm_0) = \{((x, y) \in \mathbf{N}^2 : x \neq 0\}$ and $q, r \in \mathbf{N}$.

4.3 Extension.

We shall often *extend* partial functions to make them total. For example, we define

- Cut-off subtraction by

$$x \dot{-} y = \begin{cases} x - y & \text{if } y \leq x \\ 0 & \text{if } x < y \end{cases}$$

  for $(x, y) \in \mathbf{N}^2$.

- The quotient function by

$$qt(x, y_= \begin{cases} qt_0(x, y) & \text{if } x > 0 \\ 0 & \text{if } x = 0 \end{cases}$$

  for $(x, y) \in \mathbf{N}^2$.

- The remainder function by

$$rm(x, y) = \begin{cases} rm_0(x, y) & \text{if } x > 0 \\ 0 & \text{if } x = 0 \end{cases}$$

  for $(x, y) \in \mathbf{N}^2$.

WARNING: In the theory of computable functions the domain of a function plays an important role. Typically an n-tuple $(x_1, x_2, \ldots, x_n)$ is *not* in the domain of some computable function f because the program which computes $f(x_1, x_2, \ldots, x_n)$ does not terminate normally when the input is $(x_1, x_2, \ldots, x_n)$: it goes into an infinite loop. It can happen (as we shall see) that the total function F defined from the partial function f by the prescription

$$F(x_1, x_2, \ldots, x_n) = \begin{cases} f(x_1, x_2, \ldots, x_n) & \text{if } (x_1, x_2, \ldots, x_n) \in Dom(f) \\ 0 & \text{otherwise.} \end{cases}$$

will *not* be computable, even though f *is* computable.

4.4 Numerical Relations.

A *numerical relation* is a set R of n-tuples of natural numbers:

$$R \subset \mathbf{N}^n.$$

The positive integer n is called the **arity** of the numerical relation; a numerical relation with arity n is also called an n-**ary numerical relation**. (As with functions *unary* means 1-ary, *binary* means 2 -ary, *ternary* means 3-ary, etc.) The words *predicate* and relation are synonomous.

An n-ary numerical relation determines (and is determined by) its *characteristic function* which is the numerical function c_R defined by

$$c_R(x_1, x_2, \ldots, x_n) = \begin{cases} 1 & \text{if } (x_1, x_2, \ldots, x_n) \in R \\ 0 & \text{if } (x_1, x_2, \ldots, x_n) \notin R \end{cases}$$

for $(x_1, x_2, \ldots, x_n) \in \mathbf{N}^n$.

An important difference between numerical functions and numerical relations is that by convention *relations are always assumed to be totally defined*; i.e. the characteristic function c_R of a numerical relation is *always* a total numerical function.

4.5 The Unlimited Register Machine.

In this section we shall describe an abstract computer called the Unlimited Register Machine (URM). It differs from real computers in two ways.

Firstly, the instruction set of a URM is much smaller than that of a real computer. This makes the URM much easier to study than a real computer (although it also makes the URM less efficient than a real computer), but does not in principle restrict the computing power of the URM: we shall see that the URM can compute anything a more complicated computer can.

Secondly, the URM has an infinite memory (i.e. has infinitely many data registers and has program memory which can hold an arbitrarily large program). Moreover any data register can hold an arbitrarily large number. This idealization also makes the URM easier to study and is not as far removed from reality as one might think: any particular calculation on a URM will use only a finite amount of memory so no particular calculation which can be done by a URM is in principle too difficult to be performed by a real computer.

Thirdly, program memory is disjoint from data.

The *data registers* of the URM are denoted by $R_1, R_2, R_3, \ldots$ and the URM recognizes the following five kinds of instructions:

- *Halt Instruction.* There is a single halt instruction H which causes the URM to stop execution.
- *Zero Instructions.* For each $n = 1, 2, \ldots$ there is a zero instruction Z n which causes the URM to set the register R_n to 0, leaving the other registers unaltered.
- *Successor Instructions.* For each $n = 1, 2, \ldots$ there is a successor instruction S n which causes the URM to increment by 1 the contents of the register R_n, leaving the other registers unaltered.
- *Transfer Instructions.* For each $m = 1, 2, \ldots$ and $n = 1, 2, \ldots$ there is a transfer instruction T m n which causes the URM to replace the contents of the register R_n by the contents of the register R_m (i.e. transfer R_m to R_n), leaving the other registers (including R_m) unaltered.
- *Jump Instructions.* For each $m = 1, 2, \ldots$, each $n = 1, 2, \ldots$, and each $q = 0, 1, 2, \ldots$ there is a jump instruction J m n q which causes the URM to jump to the q-th instruction if the contents of the registers R_m and R_n are equal and execute the next instruction otherwise. A jump instruction does not alter any registers. If a jump is made to an invalide instruction number this is the same as a halt.

A *URM-program* is a finite sequence of such instructions. If a program P is loaded into the URM's program memory, the data registers $R_1, R_2, \ldots$ are given initial values, and the URM is given the command to start computing, the URM responds as follows. It executes the first instruction I_0 modifying the appropiate registers as required. If that instruction was not a jump instruction, or if it was a jump instruction $J(m, n, q)$ but the contents of the registers R_m and R_n are not equal, then the URM continues with the next instruction; if it was a jump instruction $J(m, n, q)$ and contents of the registers R_m and R_n are equal, then the URM continues with the q-th instruction. It repeats this process until it is asked to execute a halt instruction at which point the URM stops.

Note that it is possible (even likely) that a program will not stop at all (for example, the program consisting of the single instruction $I_0 = J(5, 5, 0)$).

Note that program memory is indexed starting at 0, i.e. the program steps are numbered $I_0, I_1, I_2, \ldots$ whereas the data memory is indexed starting at 1, i.e. the registers are numbered $R_1, R_2, \ldots$.

The *program counter* is the number of the next instruction to execute. The program counter is incremented by one by a zero instruction Z n, successor instruction S n, or transfer instruction T m n. On the other hand, in a jump instruction J m n q the program counter is changed to q-th instruction if the contents of the registers R_m and R_n are equal and otherwise incremented by one if they are not equal.

Note that there is no provision for input to or output from the URM. Rather we consider the input to the URM to be the sequence of values in the registers $R_1, R_2, \ldots$ when the URM starts and the output from the URM to be the value in the register R_1 when the URM stops. (Sometimes we allow two outputs say R_1 and R_2 although this should really be regarded as computing two different functions.)

An n-ary numerical function f is called *URM-computable* iff there is a URM-program P which computes f in the following sense: if the registers of the URM are initialized so that the register R_1 holds the number a_1, R_2 holds the number a_2, ..., and R_n holds the number a_n, all other registers hold zero, and the program P is loaded in to the machine and executed (starting with the first instruction of the program) then

if $(a_1, a_2, \ldots, a_n) \in Dom(f)$, then the program halts and the register R_1 holds the value $f(a_1, a_2, \ldots, a_n)$ of the function;

and

if $(a_1, a_2, \ldots, a_n) \notin Dom(f)$, then the program never halts, i.e. computes forever.

4.6 Examples of URM-Computable Functions.

In this section we give some simple examples of URM-computable functions.

Example 4.6.1 *The addition function defined by*

$$add(x, y) = x + y$$

for $(x, y) \in \mathbf{N}^2 = Dom(sum)$ *is URM-computable.*

Example 4.6.2 *The multiply function defined by*

$$mult(x, y) = x * y$$

for $(x, y) \in \mathbf{N}^2 = Dom(mult)$ *is URM-computable.*

Example 4.6.3 *The predescessor function defined by*

$$pred(x) = \begin{cases} x - 1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \end{cases}$$

is URM-computable.

Example 4.6.4 *The cut-off subtraction function, dotminus, defined by*

$$x \dot{-} y = \begin{cases} x - y & \text{if } y \leq x \\ 0 & \text{if } x < y \end{cases}$$

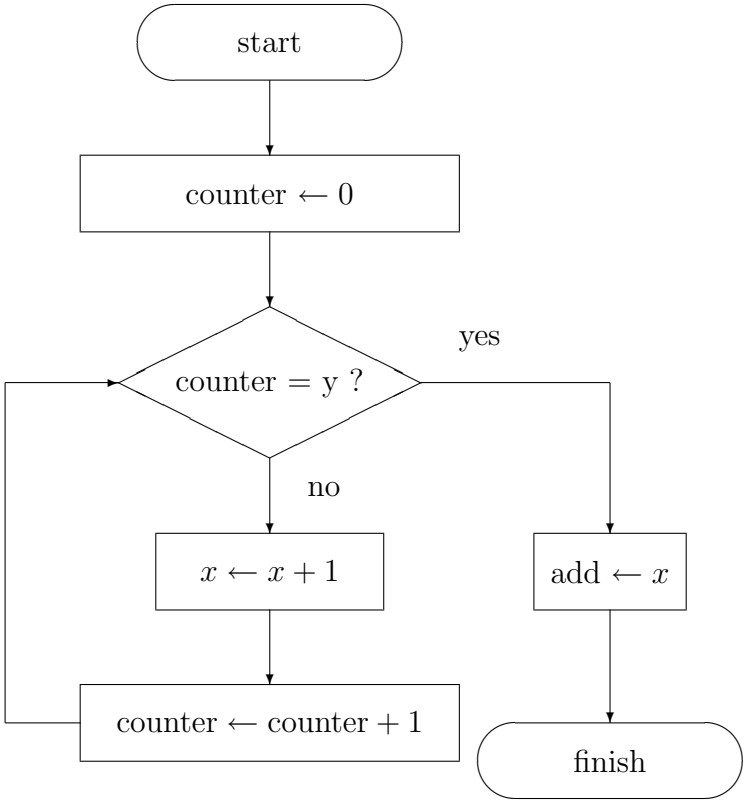
for $(x, y) \in \mathbf{N}^2 = Dom(\dot{-})$ *is URM-computable.*

Example 4.6.5 *The divide with remainder function defined by div(x,y)=q and remain(x,y)=r iff* $x = qy + r$ *where* $0 \leq r < y$ *for* $(x, y) \in \mathbf{N}^2$ *with* $y \neq 0$ *is URM-computable.*

Addition:

```
!  input: x,y        output:  add = x + y
def add(x,y)
  let counter = 0
  do until counter = y
    let x = x + 1
    let counter = counter + 1
  loop
  let add = x
end def
```

Figure 4: flowchart for add



add assembly code

```
        Z   counter
loop:   J   counter   y   done
        S   x
        S   counter
        J   1         1   loop
done:   H
```

register assignment and instruction numbering

```
x          R1
y          R2
counter    R3
add        R1
loop:      1
done:      5
```

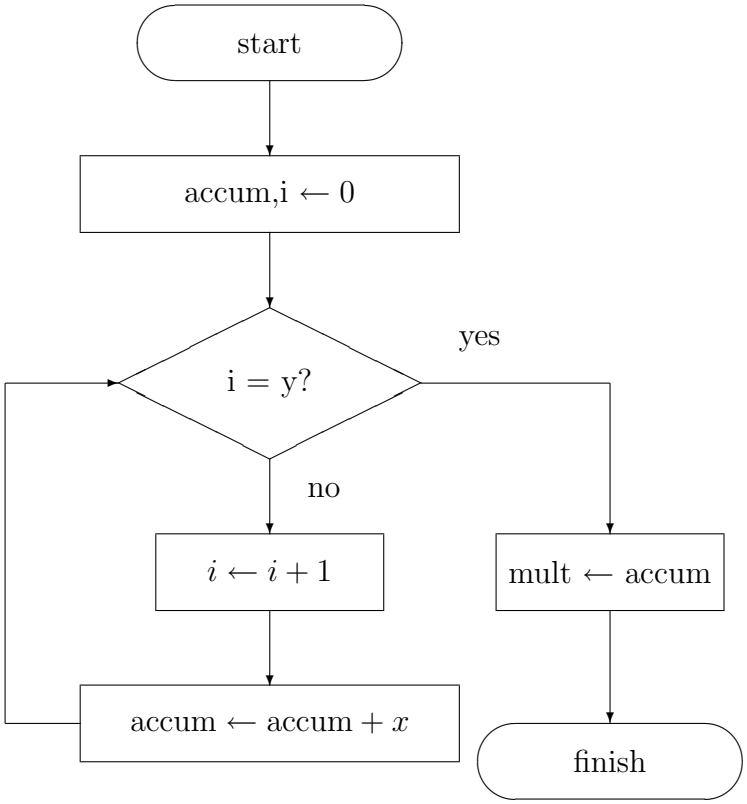
```
0   Z   3
1   J   3   2   5
2   S   1
3   S   3
4   J   1   1   1
5   H
```

Figure 5: ADD.GN

Multiplication:

```
!  input: x,y       output:  mult = x * y
def mult(x,y)
  let accum = 0
  let i = 0
    do  until  i = y
      let accum = add(accum,x)
      let i = i + 1
    loop
    let mult=accum
end def
```

Figure 6: flowchart for multiplication



```
        Z   accum
        Z   i
loop:   J   y       i       done
        macro accum := add(accum,x)
        S   i
        J   1       1       loop
done:   T   accum   mult
        H
```

Expanding macro accum := add(accum,x)

```
        Z   accum
        Z   i
loop:   J   y       i       done
-------------------------------------
        Z   counter
loop2:  J   counter x       done2
        S   accum
        S   counter
        J   1       1       loop2
done2:  S   i
-------------------------------------
        J   1       1       loop
done:   T   accum   mult
        H
```

Figure 7: assembly program for mult

register assignment and instruction numbering

```
x          R1
y          R2
mult       R1
accum      R3
i          R4
counter    R5
loop:      2
done:      10
loop2:     4
done2:     8

 0   Z   3
 1   Z   5
 2   J   2   4   10
 3   Z   5
 4   J   5   1    8
 5   S   3
 6   S   5
 7   J   1   1    4
 8   S   4
 9   J   1   1    2
10   T   3   1
11   H
```

Figure 8: MULT.GN

Predescessor:

```
!  input: x    output:  pred = x - 1   if x > 1
!                               0      otherwise
def pred(x)
  if x = 0 then
     let pred = 0
  else
     let prev = 0
     let next = 1
     do  until  x = next
       let next = next + 1
       let prev = prev + 1
     loop
     let pred = prev
  end if
end def
```

Figure 9: Psuedocode for Predescessor

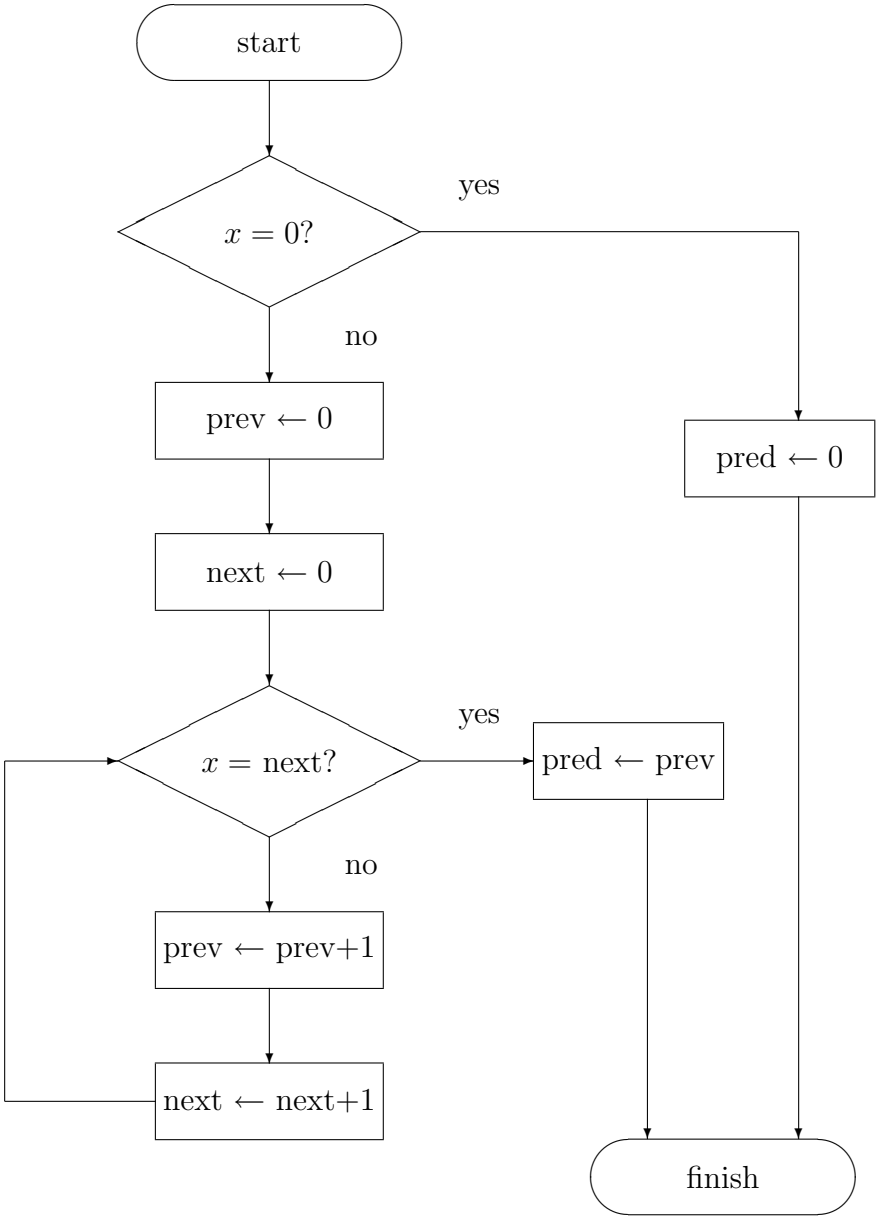


Figure 10: flowchart for predescessor

pred assembly code

```
        Z   prev
        J   x       prev    done
        Z   next
        S   next
loop:   J   x       next    done
        S   next
        S   prev
        J   1       1       loop
done:   T   prev    pred
        H
```

register assignment and instruction numbering

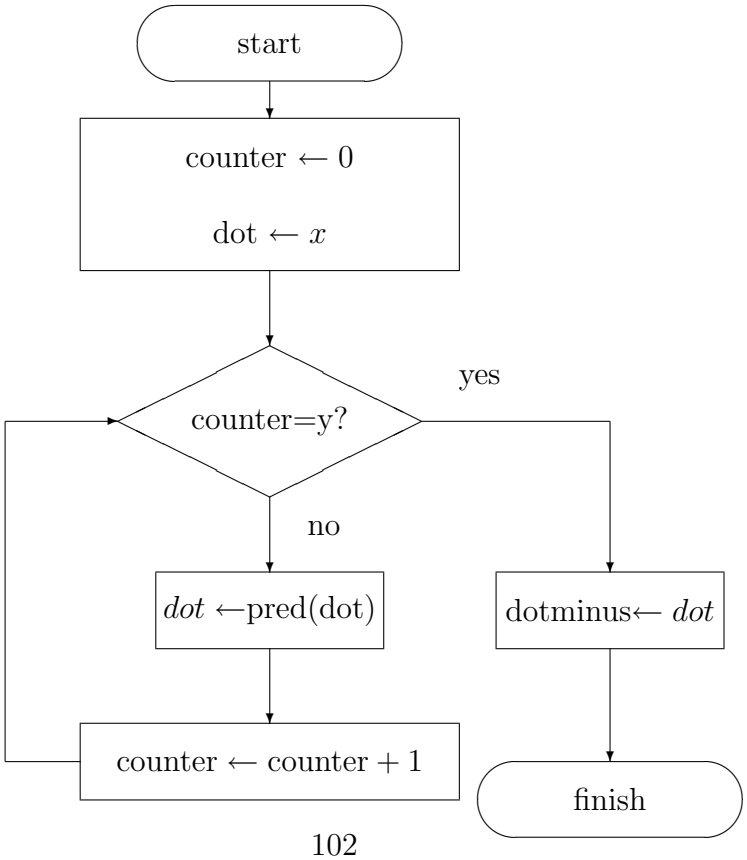
```
        x           R1
        prev        R2
        next        R3
        pred        R1
        loop:       4
        done:       8

 0  Z  2
 1  J  1  2  8
 2  Z  3
 3  S  3
 4  J  1  3  8
 5  S  3
 6  S  2
 7  J  1  1  4
 9  T  2  1
10  H
```

Figure 11: PRED.GN

Dotminus:

```
!  input: x,y   output: dotminus = x-y  if x>y
!                                0     otherwise
def dotminus(x,y)
  let counter = 0
  let dot=x
  do until counter = y
    let dot = pred(dot)
    let counter = counter + 1
  loop
  let dotminus = dot
end def
```

Figure 12: flowchart for dotminus



```
        Z   counter
        T   x          dot
loop:   J   counter    y           done
        macro dot := pred(dot)
        S   counter
        J   1          1           loop
done:   T   x          dotminus
        H
```

Expanding macro x := pred(x)

```
        Z   counter
        T   x          dot
loop:   J   counter    y           done
-------------------------------------------
        Z   prev
        J   x          prev        done2
        Z   next
        S   next
loop2:  J   dot        next        done2
        S   next
        S   prev
        J   1          1           loop2
done2:  T   prev       dot
-------------------------------------------
        S   counter
        J   1          1           loop
done:   T   dot        dotminus
        H
```

Figure 13: dotminus assembly code

register assignment and instruction numbering

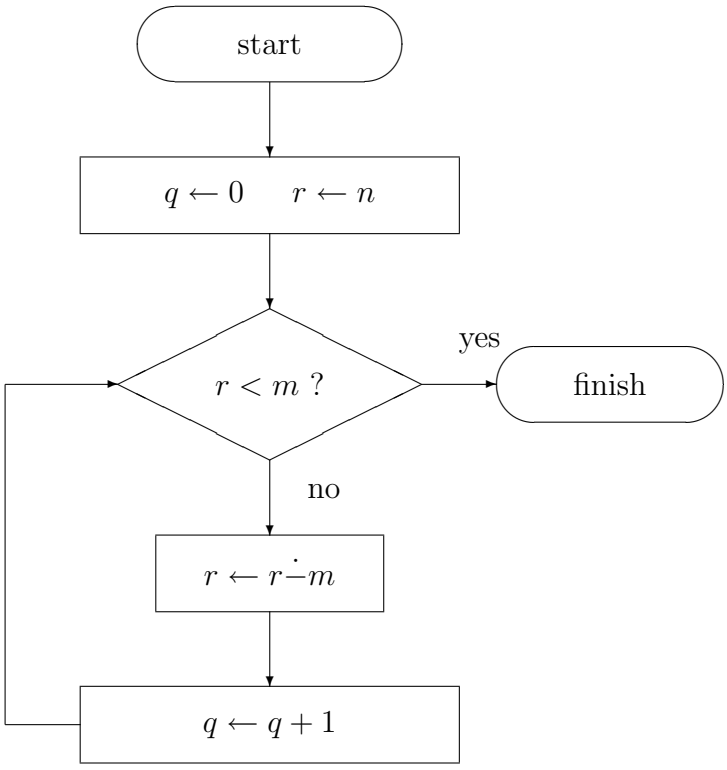
```
x          R1
y          R2
dotminus   R1
dot        R1
counter    R3
prev       R5
next       R4
loop:      1
done:      13
done2:     10
loop2:     6

 0   Z   3
 1   J   3   2   13
 2   Z   5
 3   J   1   5   10
 4   Z   4
 5   S   4
 6   J   1   4   10
 7   S   4
 8   S   5
 9   J   1   1    6
10   T   5   1
11   S   3
12   J   1   1    1
13   H
```

Figure 14: DOTMINUS.GN

Divide with remainder:

```
!  input:n,m   output: q,r  such that  n = qm + r
!  where  0 <= r < m   undefined if m = 0.
sub divrem(n,m,q,r)
  let q = 0
  let r = n
  do until r < m
    let r = dotminus(r,m)
    let q = q + 1
  loop
end sub
```

Figure 15: divide with remainder (divrem)



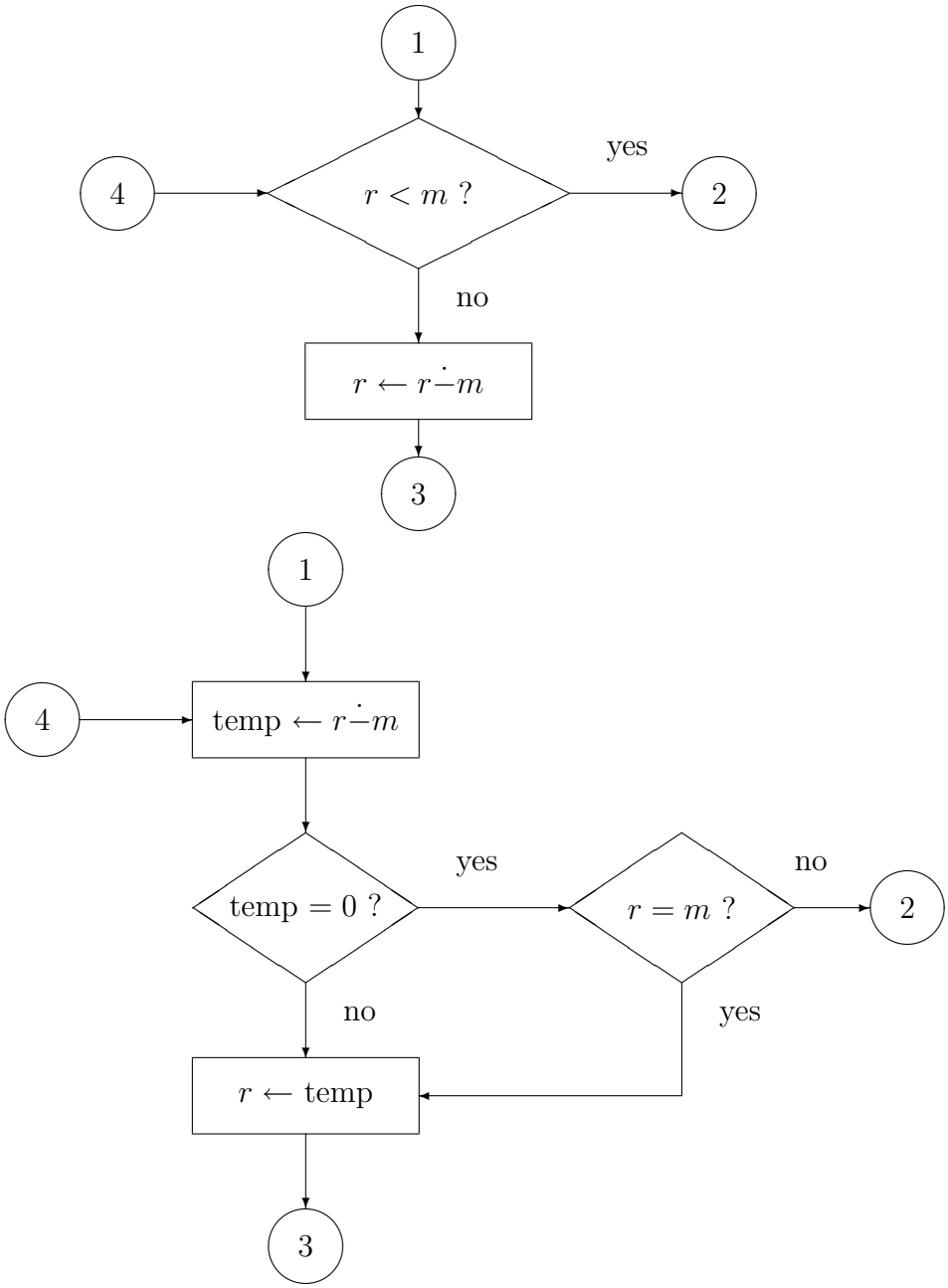


Figure 16: expanding part of the flowchart for divrem

Figure 17: divrem assembly code

```
        Z   zero
        Z   q
        T   n       r
loop:   macro temp := dotminus(r,m)
        J   temp    zero    exit
again:  T   temp    r
        S   q
        J   1       1       loop
exit:   J   r       m       again
        H
```

Expanding temp := dotminus(r,m)

```
        Z   zero
        Z   q
        T   n           r
-------------------------------------------
loop:   T   r           dot
        Z   counter
loop1:  J   counter     m       done
        Z   prev
        J   dot         prev    done2
        Z   next
        S   next
loop2:  J   dot         next    done2
        S   next
        S   prev
        J   1           1       loop2
done2:  T   prev        dot
        S   counter
        J   1           1       loop1
done:   T   dot         temp
-------------------------------------------
        J   temp        zero    exit
again:  T   temp        r
        S   q
        J   1           1       loop
exit:   J   r           m       again
        H
```

Figure 18: DIVREM.GN

	zero		R20		loop:	4
	q		R7		loop1:	5
	n		R1		loop2:	10
	r		R8		done2:	14
	counter		R3		done:	17
	m		R2		again:	19
	prev		R5		exit:	22
	next		R4			
	temp		R6			
	dot		R9			
0	Z	20				
1	Z	7				
2	T	1	8			
3	T	8	9			
4	Z	3				
5	J	3	2	17		
6	Z	5				
7	J	9	5	14		
8	Z	4				
9	S	4				
10	J	9	4	14		
11	S	4				
12	S	5				
13	J	1	1	10		
14	T	5	9			
15	S	3				
16	J	1	1	5		
17	T	9	6			
18	J	6	20	22		
19	T	6	8			
20	S	7				
21	J	1	1	3		
22	J	8	2	19		
23	H					

4.7 Universal URM machines

An URM machine U with two inputs is universal iff for all other URM machines P with one input there is a number e such that for all inputs x the output of U computing on input e,x is the same as the ouput of P computing on input x. (U diverges on e,x just in case P diverges on x.)

It is the aim of this section to outline the construction of a universal URM machine. We begin with the high level description of U (or univ).

```
dim reg(1 to 100)            ! 100 registers
dim gnum(0 to 100,1 to 4)    ! 100 instructions start=0

! reg array  : holds the simulated machine's sequence
!      of registers.  e.g. reg(3) is the number in register 3
!
! gnum array : holds the simulated machine's sequence of
!      instructions. e.g. gnum(2,3) holds the third number
!      in instruction 2
!
! done       : will be a flag (true=1 false=0) which
!      will be initialized at zero and set to 1 when
!      the HALT instruction is executed.
!
! pc         : program counter-  the number of the
!      instruction currently  being executed.


sub univ
  let pc=0                   ! start with instruction 0
  let done=0                 ! not yet done
  do
    call nextstate           ! nextstate returns done
  loop until done=1
end sub
```

```
! r1,r2      : scratch register numbers

sub nextstate                ! The state of the machine is
    let i=gnum(pc,1)         ! the pc and the register array.
    select case  i
      case 1   !  H     halt
        let done=1
      case 2   !  Z     zero the register
        let r1=gnum(pc,2)
        let reg(r1)=0
        let pc=pc+1
      case 3   !  S    successor the register
        let r1=gnum(pc,2)
        let reg(r1)=reg(r1)+1
        let pc=pc+1
      case 4   !  T    transfer the register contents
        let r1=gnum(pc,2)
        let r2=gnum(pc,3)
        let reg(r2)=reg(r1)
        let pc=pc+1
      case 5   !  J    jump if the register contents
        let r1=gnum(pc,2)       ! are equal
        let r2=gnum(pc,3)
        if reg(r1)=reg(r2) then
            let pc=gnum(pc,4)
          else
            let pc=pc+1
        end if
    end select
end sub
```

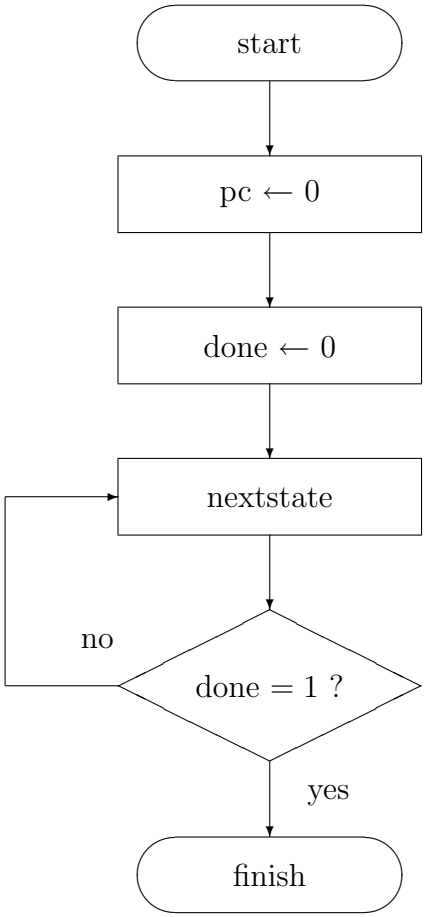


Figure 19: flowchart for univ

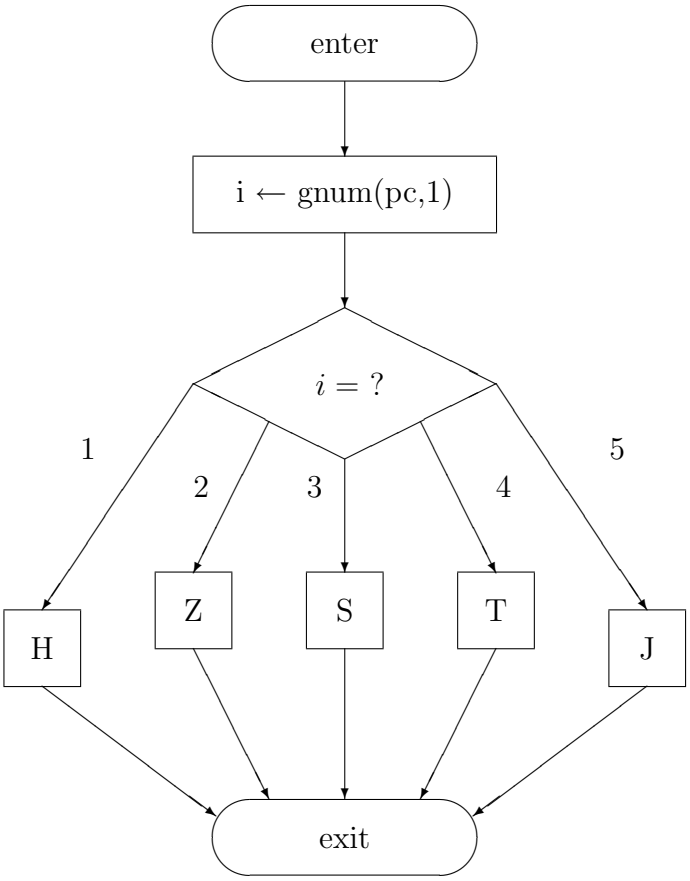


Figure 20: flowchart for nextstate

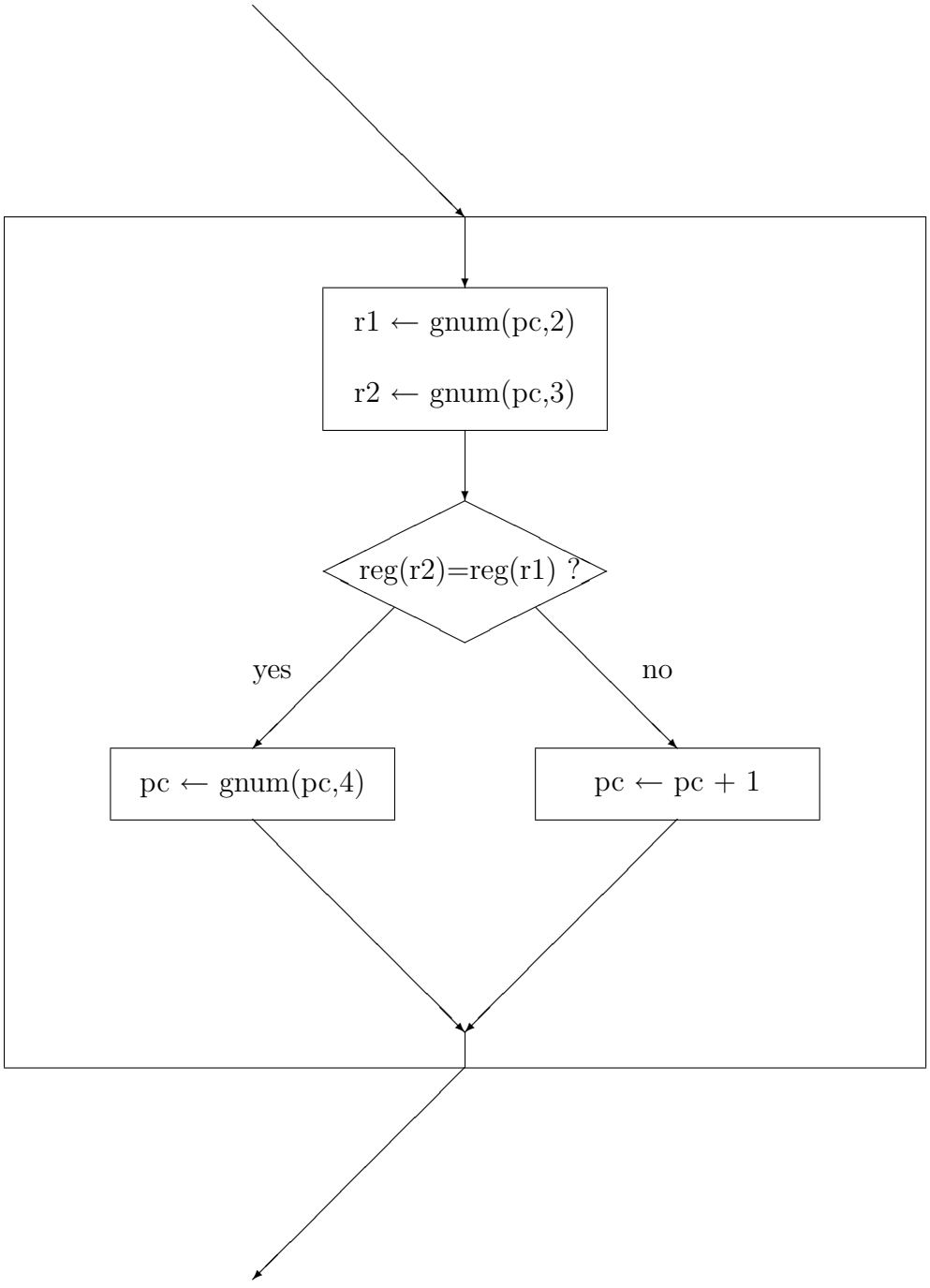


Figure 21: Jump: magnification of J box

The branch i =? in nextstate is coded by:

```
        J   i    two          zero
        J   i   three         succ
        J   i    four         trans
        J   i    five         jump
halt:       HALT code goes here
        J   1    1            exit
zero:       ZERO code goes here
        J   1    1            exit
succ:   SUCCESSOR code goes here
        J   1    1            exit
trans:   TRANSFER code goes here
        J   1    1            exit
jump:       JUMP code goes here
exit:
```

The array assignment statements are handled as follows:

let i=gnum(pc,4)	let temp=extract(gnum,pc) let i =extract(temp,4)	E gnum pc temp E temp 4 i
let reg(r1)=reg(r2)+1	let temp=extract(reg,r2) let temp=temp+1 let reg=put(reg,r1,temp)	E reg r1 temp S temp P reg r1 temp

The computability of the extract and put functions will be shown in section 4.8. The two new operations P and E are added to the basic set, H Z S T J, and are described in section 6.6. The URM code (see section 4.9) for the universal machine will be in this extended language and will also simulate the two new operations P and E.

4.8 Extract and Put

The Godel numbering scheme uses the even decimal positions (starting from 0 on the left) as markers to show where a new term begins, and uses the odd

decimal positions for the digits of the terms in the sequence to be coded. A 2 marker means that a new term is beginning, and a 1 marker means that the old term is continuing. For example, the Godel number of the sequence (G.N.)

54 6 217

is (with the original digits underlined)

$2\underline{5}1\underline{4}2\underline{6}2\underline{2}1\underline{1}1\underline{7}.$

This is a Godel number in standard form. In order to make every number the Godel number of some sequence, the initial marker can be any digit except 0, a marker > 2 is identified with a 2, a 0 marker is identified with a 1, and an extra digit at the end is ignored.

The following functions:

LENGTH(x),DIGIT(x,i),TERMS(x), START(x,y), and PUTEND(x,y)

are register machine computable. They will be used only to show that the two functions EXTRACT(x,y),and PUT(x,y,z) are register machine computable.

At this point the reader should be convinced that given psuedocode for a new function in terms of old functions, and given URM programs for the old function, one can routinely construct an URM program for the new function.

LENGTH(x)= number of decimal digit in x.

LENGHT(0)=LENGTH(1)= ... =LENGTH(9)=1,
LENGHT(10)=LENGTH(11)= ... =LENGTH(99)=2,
LENGHT(100)=LENGTH(101)= ... =LENGTH(999)=3, etc.

DIGIT(x,i)= the i^{th} decimal digit of x starting from i=zero on the left.

DIGIT(907,0)=9,
DIGIT(907,1)=0,
DIGIT(907,2)=7,
DIGIT(907,3)=0,
DIGIT(907,n)=0 for all $n \geq 3$

TERMS(x) = number of terms in the sequence with G.N. x, with the empty sequence having 0 terms. TERMS(251426221117)=3

START(x,y) = position of marker for the start of the yth term in the sequence with G.N x, undefined if TERMS(x) $\leq$ y. (Count terms from 0 on the left).

START(2251426221117,0)=0,
START(2514261221117,1)= 4, and
START(2514262211117,1)= 6.

PUTEND(x,y) = the G.N. of the sequence formed by adding y as one more term to the end of the sequence with G.N x.

PUTEND(251426221117,98)=25142622111729 18

EXTRACT(x,y) = the yth term of the sequence with G.N. x if y < TERMS(x), 0 otherwise.

EXTRACT(25142622111 7,0)=54
EXTRACT(2514261221117,1)=6
EXTRACT(251426221117,2)=217
EXTRACT(251426221117,3)=0

PUT(x,y,z) = the G.N. of the sequence formed by putting x into the yth term of the sequence with G.N. z (first adding as many 0 terms as necessary if z has fewer than y terms).

PUT(251426221117,2,99)=25142622919
PUT(251426221117,4,99)=25142622111720202919 1

The URM programs for LENGTH and DIGIT are assigned as exercises for the student. The others are too long to fit into the 92 instruction limitation of GNUMBER.COM.

```
def length(number)          ! Returns the number of decimal digits
  let len = 1               ! e.g. length(0)=length(9)=1
  let num = div(number,10)  !   length(10)=length(99)=2
  do until num = 0
    let num = div(num,10)   ! div is the quotient q output
    let len = len+1         !  of divrem.
  loop
  let length = len
end def

def digit(number,position)               ! digit(7406,0)=7
  let dig = length(number)-position-1    ! digit(7406,1)=4
  let num = number                       ! digit(7406,2)=0
  if dig < 0 then                        ! digit(7406,3)=6
    let digit = 0               ! digit(7406,n)=0 for n>3
  else
    let times = 0
    do until times = dig
      let num = div(num,10)  ! div is quotient q output
      let times = times + 1  !  of divrem.
    loop
    let digit = remain(num,10) ! remain is remainder r output
  end if                       ! of divrem
end def
```

```
def terms(array)    ! Returns the number of terms in array,
  let count = 0     ! if array is a valid godel number.
  let pos = 0                              ! terms(212223)=3
  do                                       ! terms(271819)=1
    if digit(array,pos) = 2 then let count = count + 1
    let pos = pos + 2
  loop until pos > length(array)
  let terms=count
end def

def start(array,index) ! Returns the position of start marker
  let pos = 0        ! of the term of array with given index.
  let count = 0           ! Undefined if index > terms(array)
  do until count = index        ! start(272829,0)=0
    let pos = pos + 2           ! start(272929,1)=2
    let d = digit(array,pos)    ! start(272929,2)=4
    if  d = 2 then let count = count + 1
  loop
  let start=pos
end def

def putend(array,x)         ! Returns the array with number x
  let arrayx = array        ! concatenated onto end of array.
  let arrayx = arrayx*10 +2 ! putend(2728,9)=272829
  let digx = 0                          ! digit of x
  do until digx = length(x)
     let arrayx = arrayx*10 + digit(x,digx)
     let digx = digx+1
     if digx < length(x) then let arrayx = arrayx*10 +1
  loop
  let putend = arrayx
end def
```

```
def extract(array,index)              ! Returns array[index]
  if index >= terms(array) then       ! extract(272829,0)=7
    let extract = 0                   ! extract(272829,1)=8
  else                                ! extract(272829,2)=9
    let position=start(array,index)
    let term = 0              !  extract(272829,n)=0 for n>2
    do                        !  extract(271819,0)=789
      let position = position + 1
      let d = digit(array,position)
      let term = 10 * term + d
      let position = position + 1
    loop while digit(array,position) = 1
    let extract = term
  end if
end def

def  put(array,index,x)   ! Returns outarray with
  let inarray = array     ! outarray[i]=x if i=index
  let outarray = 0        ! outarray[i]=array[i] otherwise.
  let i = 0               ! If index > terms(array) then
  do until i = index      ! 0 terms are added as
    let term = extract(inarray,i)       ! necessary.
    let outarray = putend(outarray,term)
    let i = i + 1             ! put(2728,4,99)=272820202919
  loop
  let outarray = putend(outarray,x)
  let i=i+1
  do while i < terms(inarray)
    let term = extract(inarray,i)
    let outarray = putend(outarray,term)
    let i = i + 1
  loop
  let put = outarray
end def
```

4.9 UNIV.GN commented listing

Instruction types :

H	:1:	halt
Z a	:2:	set Ra = 0
S a	:3:	increment Ra (add 1 to Ra)
T a b	:4:	transfer Ra to Rb
J a b c	:5:	if Ra = Rb then jump to instruction c
E a b c	:6:	extract (Rb)-th term of Ra and put in Rc
P a b c	:7:	put Ra into (Rb)-th term of Rc

Register usage

R1	: Gödel number of simulated instruction list
R2, R3	: inputs for simulated machine
R4	: Gödel number of simulated register list
R5	: simulated program counter, n
	Special notation:
r(m)	: m-th term of R4 – m-th simulated register
i	: R5-th term of R1 – Gödel number of n-th simulated instruction where n = R5. Has 4 coordinates i(0) .. i(3).
R6	: Gödel number of n-th simulated instruction, i
R7 - R10	: coordinates of n-th simulated instruction, i(0) .. i(3)
R11	: i(1)-st term of R4, r(i(1))
R12	: i(2)-nd term of R4, r(i(2))
R13	: i(3)-rd term of R4, r(i(3))
R14	: used as counter from 0 to 6 to fill R20 - R26, then holds 7
R15	: time count for simulated machine
R20	: holds 0
R21	: holds 1
R22	: holds 2
R23	: holds 3
R24	: holds 4
R25	: holds 5
R26	: holds 6

00	J	20	20	50		jump to initial constant-setting sequence
01	Z	15				set time counter to zero
02	Z	4				clear simulated register sequence, R4:=0
03	P	2	21	4		put R2 into 1st term of R4, r(1):=R2
04	P	3	22	4		put R3 into 2nd term of R4, r(2):=R3
05	Z	5				point to 0-th instruction
06	P	5	20	4	b	put instruction number into 0th term of R4, r(0):=R5
07	E	1	5	6		put code of R5th instruction into R6, R6:=i
08	E	6	20	7		put 0th term of instruction R6 into R7, R7:=i(0)
09	E	6	21	8		put 1st term of instruction R6 into R8, R8:=i(1)
10	E	6	22	9		put 2nd term of instruction R6 into R9, R9:=i(2)
11	E	6	23	10		put 3rd term of instruction R6 into R10, R10:=i(3)
12	E	4	8	11		put i(1)-st term of R4 into R11, R11:=r(i(1))
13	E	4	9	12		put i(2)-st term of R4 into R12, R12:=r(i(2))
14	E	4	10	13		put i(3)-st term of R4 into R13, R13:=r(i(3))
15	J	7	22	z		if i(0) = 2 then jump to zero routine
16	J	7	23	s		if i(0) = 3 then jump to successor routine
17	J	7	24	t		if i(0) = 4 then jump to transfer routine
18	J	7	25	j		if i(0) = 5 then jump to jump routine
19	J	7	26	e		if i(0) = 6 then jump to extract routine
20	J	7	14	p		if i(0) = 7 then jump to put routine
21	J	20	20	h		if i(0) = 1 then jump to halt routine
22	P	11	12	13	p	put r(i(1)) into r(i(2))-nd position of R13
23	J	20	20	u		skip next line
24	E	11	12	13	e	(extract) put r(i(2))-nd position of r(i(1)) in R13
25	P	13	10	4	u	put new R13 into position i(3) of R4, r(i(3)):=R13

Figure 22: UNIV2.GN part1

26	J	20	20	x		jump to routine to add one to instruction pointer
27	Z	11			z	(zero) set R11:=0
28	J	20	20	v		skip next line
29	S	11			s	(successor) add one to R11, R11:=R11+1
30	P	11	8	4	v	put new R11 into position i(1) of R4, r(i(1)):=R11
31	J	20	20	x		jump to routine to add one to instruction pointer
32	P	11	9	4	t	(trans) put R11 into position i(2) of R4, r(i(2)):=R11
33	J	20	20	x		jump to routine to add one to instruction pointer
34	J	11	12	y	j	(jump) if r(i(1)) = r(i(2)) then jump to y
35	E	4	20	5	x	put r(0) into instruction pointer R5, R5:=r(0)
36	S	5				add one to instruction pointer, R5:=R5+1
37	J	20	20	w		skip next line
38	T	10	5		y	put i(3) into instruction pointer R5, R5:=i(3)
39	S	15			w	increase time count by one
40	J	20	20	b		jump to b to begin next simulated instruction
41	E	4	21	1	h	(halt) put output of simulated machine into R1, R1:=r(1)
42	H					halt

.
.

Figure 23: UNIV.GN part 2

```
49  H
50  Z   14              use R14 as counter, R14:=0
51  T   14  20          R20 holds 0
52  S   14              R14 holds 1
53  T   14  21          R21 holds 1
54  S   14              R14 holds 2
55  T   14  22          R23 holds 2
56  S   14              R14 holds 3
57  T   14  23          R23 holds 3
58  S   14              R14 holds 4
59  T   14  24          R24 holds 4
60  S   14              R14 holds 5
61  T   14  25          R25 holds 5
62  S   14              R14 holds 6
63  T   14  26          R26 holds 6
64  S   14              R14 holds 7
66  J   20  20  1       jump to main body of program
```

Figure 24: UNIV.GN initialization routine

4.10 Primitive Recursion.

In this section we define the family of primitive recursive functions and show that every primitive recursive function is URM computable. We say that a relation is primitive recursive iff its characteristic function is primitive recursive.

The set of *primitive recursive functions* is the smallest class of functions which contains the zero function, succesor function, and projection functions and is closed under composition and primitive recursion.

Proposition 4.10.1 *The zero function defined by*

$$\mathbf{0}(x) = 0$$

for $x \in \mathbf{N} = Dom(\mathbf{0})$ *is URM-computable.*

Proposition 4.10.2 *The succesor function defined by*

$$s(x) = x + 1$$

for $x \in \mathbf{N} = Dom(s)$ *is URM-computable.*

Proposition 4.10.3 *For each* $n = 1, 2, \ldots$ *and each* $k = 1, 2, \ldots, n$ *the projection functions defined by*

$$I_k^n(x_1, x_2, \ldots, x_n) = x_k$$

for $(x_1, x_2, \ldots, x_n) \in \mathbf{N}^n = Dom(I_k^n)$ *is URM-computable.*

Let h be an m-ary numerical function and $g_1, g_2, \ldots, g_m$ be n-ary numerical functions. The function n-ary function f defined by the equation

$$f(x_1, \ldots, x_n) = h(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n)) \tag{5}$$

for $(x_1, x_2, \ldots, x_n) \in Dom(f)$ is called the **function defined from** h **and** $g_1, g_2, \ldots, g_m$ **by composition**; here the domain $Dom(f)$ of f is the largest set of n-tuples for which the right hand side of (5) is meaningful, i.e.

$$(x_1, \ldots, x_n) \in Dom(f)$$

if and only if

$$(x_1, \ldots, x_n) \in Dom(g_k)$$

for $k = 1, \ldots, n$ and

$$(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n)) \in Dom(h).$$

Note in particular that f is totally defined (i.e. $Dom(f) = \mathbf{N}^n$) if all the functions $h, g_1, \ldots, g_m$ are totaly defined.

Theorem 4.10.4 *If $h, g_1, \ldots, g_m$ are URM-computable and f is defined from them by composition, then f is URM-computable.*

The following concept and lemma are usefull for proving this and other results. A program P *neatly computes* a function f of n variables if

1. When P starts with $x_1, \ldots, x_n$ in registers $R1, \ldots, Rn$ and arbitrary contents in other registers, P halts with $f(x_1, \ldots, x_n)$ in R! if the function is defined there, and never halts if the function is undefined.

2. P halts at the instruction number just after the last instruction of P.

Lemma 4.10.5 *A function f of n-variables is computable iff it is neatly computable.*

Let g be a totally defined n-ary total numerical function and h be a totally defined $(n+2)$-ary numerical function. Then there is a unique totally defined $(n+1)$-ary numerical function f which satisfies the equations

$$f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n)$$

and

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

for $(x_1, \ldots, x_n, y) \in \mathbf{N}^n$; this function f is called the **function defined from g and h by primitive recursion**.

Note that the function f is computed by iteration: thus, if $n = 1$ we have

$$\begin{aligned} f(x,0) &= g(x) \\ f(x,1) &= h(x,0,f(x,0)) \\ &= h(x,0,g(x)) \\ f(x,2) &= h(x,1,f(x,1)) \\ &= h(x,1,h(x,0,g(x)) \\ f(x,3) &= h(x,2,f(x,2)) \\ &= h(x,2,(h(x,1,h(x,0,g(x)) \end{aligned}$$

and so on.

Theorem 4.10.6 *If g and h are totally defined URM-computable functions, and f is defined from them by primitive recursion, then f is also URM-computable.*

Some examples of functions which can be seen to be primitive recursive are:

addition, multiplication, exponentiation, predescessor, dotminus, and divide with remainder.

Thus by the preceding results we see that every primitive recursive function is URM computable.

4.11 Recursive functions

The class of *recursive functions* is the smallest class of functions containing the primitive recursive functions and closed under the operation of unbounded minimalization.

Let R be an $(n+1)$-ary numerical relation. The equation

$$w = (\mu y)R(x_1, x_2, \ldots, x_n, y)$$

shall mean

$$R(x_1, x_2, \ldots, x_n, w)$$

and

$$notR(x_1, x_2, \ldots, x_n, y) \text{ for } v = 0, 1, \ldots, w-1,$$

i.e. that w is the smallest y for which $R(x_1, x_2, \ldots, x_n, w)$ holds. It is clear that the condition $w = (\mu y)R(x_1, x_2, \ldots, x_n, y)$ determines w uniquely i.e. that $w_1 = (\mu y)R(x_1, x_2, \ldots, x_n, y)$ and $w_2 = (\mu y)R(x_1, x_2, \ldots, x_n, y)$ imply $w_1 = w_2$. Of course it can happen that $notR(x_1, x_2, \ldots, x_n, y)$ holds for every y; when this happens there will not exist a w satisfying $w = (\mu y)R(x_1, x_2, \ldots, x_n, y)$. Thus the equation

$$f(x_1, x_2, \ldots, x_n) = (\mu y)R(x_1, x_2, \ldots, x_n, y)$$

can be interpreted as defining an n-ary function f with

$$Dom(f) = \{(x_1, x_2, \ldots, x_n) \in \mathbf{N}^n : (\exists y)R(x_1, x_2, \ldots, x_n, y)\}.$$

This function f is called the **numerical function determined from** R *by (unbounded) minimalization.* It is important to remember that this function might not be totally defined.

Proposition 4.11.1 *If R is a URM-computable numerical relation, then the numerical function determined from R by minimalization is URM-computable.*

As a corollary we have that every recursive function is URM computable.

We now prove that the class of recursive functions has several other nice closure properties.

Definition by Cases.

Proposition 4.11.2 *Let $R_1, R_2, \ldots, R_m$ be n-ary recursive relations and $g_1, g_2, \ldots, g_m$ be nary recursive functions. Assume that the relations $R_1, R_2, \ldots, R_m$ are mutually exclusive and exhaustive, i.e.*

$$i \neq j \Longrightarrow R_i \cap R_j = \emptyset$$

and

$$N^n = \bigcup_{i=1}^{m} R_i.$$

Then the n-ary numerical function f defined by

$$Dom(f) = \bigcup_{i=1}^{m} Dom(g_i) \cap R_i$$

and

$$f(x_1, x_2, \ldots, x_n) = \begin{cases} g_1(x_1, x_2, \ldots, x_n) & \text{if } (x_1, x_2, \ldots, x_n) \in Dom(g_1) \cap R_1 \\ g_2(x_1, x_2, \ldots, x_n) & \text{if } (x_1, x_2, \ldots, x_n) \in Dom(g_2) \cap R_2 \\ \ldots & \\ g_m(x_1, x_2, \ldots, x_n) & \text{if } (x_1, x_2, \ldots, x_n) \in Dom(g_m) \cap R_m \end{cases}$$

is recursive.

Bounded Quantification.

Proposition 4.11.3 *Let R be a recursive $(n+1)$-ary numerical relation. Then the $(n+1)$-ary numerical relations A and E defined by*

$$A(x_1, x_2, \ldots, x_n, z) \Longleftrightarrow (\forall y \leq z) R(x_1, x_2, \ldots, x_n, y)$$

and

$$E(x_1, x_2, \ldots, x_n, z) \Longleftrightarrow (\exists y \leq z) R(x_1, x_2, \ldots, x_n, y)$$

for $(x_1, x_2, \ldots, x_n, z) \in N^n$ are also recusive.

Bounded Minimalization.

Let R be an $(n+1)$-ary numerical relation. For $x_1, \ldots, x_n, z \in \mathbf{N}$ define $(\mu y \leq z) R(x_1, \ldots, x_n, y)$ by the conditions

$$(\mu y \leq z) R(x_1, \ldots, x_n, y) = \begin{cases} w & \text{if } w \leq z,\ R(x_1, \ldots, x_n, w), \\ & \text{and } not R(x_1, \ldots, x_n, v) \text{ for } v = 0, 1, \ldots, w-1; \\ 0 & \text{otherwise.} \end{cases}$$

Then the equation

$$f(x_1, \ldots, x_n, z) = (\mu y \leq z) R(x_1, \ldots, x_n, y)$$

for $x_1, \ldots, x_n, z \in \mathbf{N}$ determines a totally defined $(n+1)$-ary numerical function f which is called the **numerical function determined from R by bounded minimalization**.

Proposition 4.11.4 *If R is a recursive relation, then the numerical function determined from R by bounded minimalization is (totally defined and) recursive.*

Coding Finite Sequences.

Proposition 4.11.5 *For each $n = 1, 2, \ldots$ and each $k = 1, 2, \ldots, n$ there there are total recursive functions $\omega^{(n)}, \pi_1^{(n)}, \pi_2^{(n)}, \ldots, \pi_n^{(n)}$ (with $\omega^{(n)}$ n-ary and $\pi_1^{(n)}, \pi_2^{(n)}, \ldots, \pi_n^{(n)}$ unary) satisfying*

$$\pi_k^{(n)}(\omega^{(n)}(x_1, x_2, \ldots, x_n)) = x_k$$

for $k = 1, 2, \ldots, n$ and $(x_1, x_2, \ldots, x_n) \in \mathbf{N}^n$ and

$$\omega^{(n)}(\pi_1^{(n)}(x), \pi_2^{(n)}(x), \ldots, \pi_n^{(n)}(x)) = x$$

for $x \in \mathbf{N}$.

We remark that in the lingo of "mappings" the conditions on $\omega^{(n)}, \pi_1^{(n)}, \pi_2^{(n)}, \ldots, \pi_n^{(n)}$ assert that the mapping

$$\omega^{(n)} : \mathbf{N}^n \longrightarrow \mathbf{N}$$

is bijective with inverse

$$(\omega^{(n)})^{-1} : \mathbf{N} \longrightarrow \mathbf{N}^n$$

given by

$$(\omega^{(n)})^{-1}(x) = (\pi_1^{(n)}(x), \pi_2^{(n)}(x), \ldots, \pi_n^{(n)}(x))$$

for $x \in \mathbf{N}$.

Let $\mathbf{N}^\infty$ to be the set of all infinite numerical sequences which eventually vanish:

$$\mathbf{N}^\infty = \{\rho \in \mathbf{N}^{\mathbf{N}} : \exists k \forall i \geq k : \quad \rho_i = 0\}$$

Define the *projection mapping*

$$proj : \mathbf{N} \times \mathbf{N}^\infty \longrightarrow \mathbf{N}$$

by

$$proj(i, \rho)) = \rho_i$$

Define the *redefinition map*

$$redef : \mathbf{N} \times \mathbf{N}^\infty \times \mathbf{N} \longrightarrow \mathbf{N}^\infty$$

by

$$redef(i, \rho, u) = s$$

where

$$s_j = \begin{cases} \rho_j & \text{if } j \neq i \\ u & \text{if } j = i. \end{cases}$$

Define the *extension by zero map*

$$ext_n : \mathbf{N}^n \longrightarrow \mathbf{N}^\infty$$

by

$$ext_n(a_1, a_2, \ldots, a_n) = \rho$$

where

$$\rho_j = \begin{cases} a_{j+1} & \text{if } j = 0, 1, \ldots, n-1 \\ 0 & \text{if } j \geq n \end{cases}$$

Proposition 4.11.6 *There is a bijective map*

$$\omega^{(\infty)} : \mathbf{N}^\infty \longrightarrow \mathbf{N}$$

and recursive functions $\pi^{(\infty)}, r, e_1, e_2, \ldots$ *such that*

$$\pi^{(\infty)}(i, \omega(\rho)) = proj(i, \rho),$$

$$r(i, \omega(\rho), u) = \omega(redef(i, \rho, u))$$

for $\rho \in \mathbf{N}^\infty$ *and* $i, u \in \mathbf{N}$*, and*

$$e_n(a_1, a_2, \ldots, a_n) = \omega(ext_n(a_1, a_2, \ldots, a_n))$$

for $(a_1, a_2, \ldots, a_n) \in \mathbf{N}^n$.

4.12 The URM-Computable Functions are recursive.

In this section we shall define the URM in purely mathematical terms and use this definition to show that every URM-computable function is recursive.

Firstly, the *URM instructions* $Z(n)$, $S(n)$, $T(m, n)$, and $J(m, n, q)$ are to be viewed as mathematical objects; if it helps the reader can think of H as an abbreviation for the number 0, $Z(n)$ as an abbreviation for the pair $(1, n)$, $S(n)$ as an abbreviation for the pair $(2, n)$, $T(m, n)$ as an abbreviation

for the triple $(3, m, n)$, and $J(m, n, q)$) as an abbreviation for the quadruple $(4, m, n, q)$. A *URM program* is an infinite sequence

$$P = (I_0, I_1, I_2, \ldots)$$

of URM instructions such that all but finitely many of the instructions are halt instructions.

A *URM state* is a sequence

$$\rho = (\rho_0, \rho_1, \rho_2, \ldots)$$

of natural numbers such that $\rho_k = 0$ for all but finitely many k. The element ρ_0 of the sequence ρ is called the **value of the program counter in the state** ρ and for $k = 1, 2, \ldots$ the number ρ_k is called the **value of the register** R_k **in the state** ρ. In the notation of section on coding a URM state is nothing more than an infinite sequence of natural numbers which is eventually 0; i.e. an element of N^∞.

The Next State.

For each URM program $P = (I_0, I_1, I_2, \ldots)$ and each URM-state ρ we define a new URM-state

$$\sigma = \mathcal{S}_P(\rho)$$

called the *next state* obtained from the program P and the URM-state ρ

$$I_{\rho_0} = H \quad \Longrightarrow \quad \sigma = \rho$$

$$I_{\rho_0} = Z(n) \quad \Longrightarrow \quad \sigma_k = \begin{cases} \rho_0 + 1 & \text{if } k = 0 \\ 0 & \text{if } k = n \\ \rho_k & \text{otherwise} \end{cases}$$

$$I_{\rho_0} = S(n) \quad \Longrightarrow \quad \sigma_k = \begin{cases} \rho_0 + 1 & \text{if } k = 0 \\ \rho_n + 1 & \text{if } k = n \\ \rho_k & \text{otherwise} \end{cases}$$

$$I_{\rho_0} = T(m, n) \quad \Longrightarrow \quad \sigma_k = \begin{cases} \rho_0 + 1 & \text{if } k = 0 \\ \rho_m & \text{if } k = n \\ \rho_k & \text{otherwise} \end{cases}$$

$$I_{\rho_0} = J(m, n, q) \quad \Longrightarrow \quad \sigma_k = \begin{cases} \rho_0 + 1 & \text{if } k = 0 \text{ and } \rho_n \neq \rho_m \\ q & \text{if } k = 0 \text{ and } \rho_n = \rho_m \\ \rho_k & \text{if } k > 0 \end{cases}$$

for $\rho \in \mathbf{N}^\infty$. Thus we have defined for each program P a map

$$\mathcal{S}_P : \mathbf{N}^\infty \longrightarrow \mathbf{N}^\infty.$$

This map is called the *next state map* of the program P.

The Computation Determined by a Program and URM-State.

Given $a_1, a_2, \ldots, a_n \in \mathbf{N}$ and a URM-program P we define a sequence

$$P(a_1, a_2, \ldots, a_n) = (\rho^{(0)}, \rho^{(1)}, \rho^{(2)}, \ldots)$$

inductively by the prescription

$$\rho_i^{(0)} = \begin{cases} 0 & \text{for } i = 0 \\ a_i & \text{for } i = 1, 2, \ldots, n \\ 0 & \text{for } i > n. \end{cases}$$

and

$$\rho^{(k+1)} = \mathcal{S}_P(\rho^{(k)}).$$

This sequence is called the **computation determined by P with initial configuration** $(a_1, a_2, \ldots, a_n) \in \mathbf{N}^n$ The computation is said to *terminate* in case $\rho_0^{(k)} = H$ for some k and is called *non-terminating* in the contrary case.

The n-ary Function Computed by a Program.

We then define

- $P(a_1, a_2, \ldots, a_n) \uparrow$ iff the computation $P(a_1, a_2, \ldots, a_n)$ never stops, i.e. is non-terminating. One also says that the computation *diverges* in this case.
- $P(a_1, a_2, \ldots, a_n) \downarrow$ iff the computation $P(a_1, a_2, \ldots, a_n)$ terminates. One also says that the computation *converges* in this case.
- $P(a_1, a_2, \ldots, a_n) \downarrow b$ iff the computation $P(a_1, a_2, \ldots, a_n)$ is terminates, say with $\rho_0^{(l)} \notin \{1, 2, \ldots, s\}$ and we have

  $$\rho_1^{(l)} = b.$$

  One also says that the computation **converges to** b in this case.

The definitions entail that $P(a_1, a_2, \ldots, a_n) \downarrow$ iff $P(a_1, a_2, \ldots, a_n) \downarrow b$ for some $b \in \mathbf{N}$.

We now define for each URM-program P and each natural number n a function $\Phi_P^{(n)}$ by the conditions

$$Dom(\Phi_P^{(n)}) = \{(a_1, a_2, \ldots, a_n) \in \mathbf{N}^n : P(a_1, a_2, \ldots, a_n) \downarrow\}$$

and for $(a_1, a_2, \ldots, a_n) \in Dom(\Phi_P^{(n)})$ and $b \in \mathbf{N}$

$$b = \Phi_P^{(n)}(a_1, a_2, \ldots, a_n) \Longleftrightarrow P(a_1, a_2, \ldots, a_n) \downarrow b.$$

$\Phi_P^{(n)}$ is called the n-**ary numeric function computed by the URM program** P.

Note that the definitions entail the simple identity

$$\Phi_P^{(n)}(a_1, a_2, \ldots, a_n) = \Phi_P^{(n+1)}(a_1, a_2, \ldots, a_n, 0).$$

An n-ary partial numerical function f is called *URM-computable* iff there is a URM program P such that

$$f = \Phi_P^{(n)}.$$

This means that $f(a_1, a_2, \ldots, a_n)$ is defined precisely for those n-tuples $(a_1, a_2, \ldots, a_n)$ for which the computation $P(a_1, a_2, \ldots, a_n)$ is finite and that $f(a_1, a_2, \ldots, a_n)$ is the result of that computation in the sense that it is the value left in the register R_1 when the computation stops.

Gödel Numbers.

Let $\mathcal{I}$ denote the set of all URM instructions and define a bijection

$$\beta : \mathcal{I} \longrightarrow \mathbf{N} \setminus \{0\}$$

by

$$\begin{array}{lcl}
\beta(H) & = & 0 \\
\beta(Z(n)) & = & 4(n-1)+1 \\
\beta(S(n)) & = & 4(n-1)+2 \\
\beta(T(m,n)) & = & 4\omega^{(2)}(m-1, n-1)+3 \\
\beta(J(m,n,q)) & = & 4\omega^{(3)}(m-1, n-1, q)+4
\end{array}$$

for $m, n, q = 1, 2, \ldots$.

Recall that a URM program is a sequence

$$P = (I_0, I_1, I_2, \ldots)$$

of URM instructions such that $\beta(I_j) = 0$ for all but finitely may j. Let $\mathcal{P}$ denote the set of all URM programs and define a bijection

$$\gamma : \mathcal{P} \longrightarrow \mathbf{N}$$

by

$$\gamma(P) = \omega^{(\infty)}(\beta(I_0), \beta(I_1), \beta(I_2), \ldots)$$

When $P = \gamma(e)$ we say that e is the *Gödel number* of the program P.

Universal Recursive Functions.

Recall from section 4.12 that each URM-program P and each natural number n determine a function $\Phi_P^{(n)}$ by the conditions

$$Dom(\Phi_P^{(n)}) = \{(a_1, a_2, \ldots, a_n) \in \mathbf{N}^n : P(a_1, a_2, \ldots, a_n) \downarrow\}$$

and for $(a_1, a_2, \ldots, a_n) \in Dom(\Phi_P^{(n)})$ and $b \in \mathbf{N}$

$$b = \Phi_P^{(n)}(a_1, a_2, \ldots, a_n) \Longleftrightarrow P(a_1, a_2, \ldots, a_n) \downarrow b.$$

The $(n+1)$-ary function $\phi^{(n)}$ determined by

$$Dom(\phi^{(n)}) = \{(e, a_1, a_2, \ldots, a_n) : P(a_1, a_2, \ldots, a_n) \downarrow \text{ where } P = \gamma(e)\}$$

and

$$\phi^{(n)}(e, a_1, a_2, \ldots, a_n) = \Phi_P^{(n)}(a_1, a_2, \ldots, a_n)$$

for $P = \gamma(e)$ and $(e, a_1, a_2, \ldots, a_n) \in Dom(\phi^{(n)})$ is called the *universal function* for n-ary computable functions. We also denote by $\phi_e^{(n)}$ the function computed by the URM program with Gödel number e, i.e.

$$\phi_{\gamma(P)}^{(n)} = \Phi_P^{(n)}$$

for each URM program P. Note that

$$(a_1, a_2, \ldots, a_n) \in Dom(\phi_e^{(n)}) \Longleftrightarrow (e, a_1, a_2, \ldots, a_n) \in Dom(\phi^{(n)})$$

and

$$\phi_e^{(n)}(a_1, a_2, \ldots, a_n) = \phi^{(n)}(e, a_1, a_2, \ldots, a_n)$$

for $(e, a_1, a_2, \ldots, a_n) \in Dom(\phi^{(n)})$.

Theorem 4.12.1 *For each n the universal function $\phi^{(n)}$ is recursive.*

Lemma 4.12.2 *The binary numerical function S determined by the condition*

$$S(\gamma(P), \omega(\rho)) = \omega(\mathcal{S}_P(\rho))$$

for $P \in \mathcal{P}$ and $\rho \in \mathbf{N}^\infty$ is recursive.

Let $\mathcal{S}_P^t$ be the t-th iterate of $\mathcal{S}_P$. Thus

$$\mathcal{S}_P^0(\rho) = \rho$$

and

$$\mathcal{S}_P^{t+1}(\rho) = \mathcal{S}_P(\mathcal{S}_P^t(\rho)).$$

Thus if the URM is running the program P and is started in the state ρ it will be in the state $\mathcal{S}_P^t$ after computing t steps of the program.

Lemma 4.12.3 *The ternary numerical function R determined by the condition*

$$R(t, \gamma(P), \omega(\rho)) = \omega(\mathcal{S}_P^t(\rho))$$

for $P \in \mathcal{P}$,$\rho \in \mathbf{N}^\infty$, and $t \in \mathbf{N}$ is recursive.

Now for each URM program P define

$$\mathcal{T}_P : \mathbf{N}^\infty \longrightarrow \mathbf{N} \cup \{\infty\}$$

by setting $\mathcal{T}_P(\rho)$ = to the number of steps executed by the URM if it is started in state ρ and running the program P. (We set $\mathcal{T}_P(\rho) = \infty$ if the machine computes forever.)

Lemma 4.12.4 *The binary numerical function T determined by the conditions*

$$Dom(T) = \{(\gamma(P), \omega(\rho)) : \mathcal{T}_P(\rho)) \neq \infty\}$$
$$T(\gamma(P), \omega(\rho)) = \omega(\mathcal{T}_P(\rho))$$

for $P \in \mathcal{P}$ and $(\gamma(P), \omega(\rho)) \in Dom(T)$ is URM computable.

Now an explicit formula for the universal function $\phi^{(n)}$ is given by

$$\phi^{(n)}(e, a_1, \ldots, a_n) = \pi(1, R(T(e, e_{n+1}(1, a_1, \ldots, a_n)), e_{n+1}(1, a_1, \ldots, a_n))).$$

Theorem 4.12.5 *The class of recursive functions and the class of URM-computable functions is the same.*

4.13 Decidable Relations.

Let $R \subset \mathbf{N}^n$ be an n-ary numerical relation. We call R a URM- *decidable relation* iff the characteristic function c_R or R is URM-computable. Recall that the characteristic function c_R of R is the numerical function defined by

$$c_R(x_1, x_2, \ldots, x_n) = \begin{cases} 1 & \text{if } (x_1, x_2, \ldots, x_n) \in R \\ 0 & \text{if } (x_1, x_2, \ldots, x_n) \notin R \end{cases}$$

for $(x_1, x_2, \ldots, x_n) \in \mathbf{N}^n$; it is *always* a totally defined function.

We often write $R(x_1, x_2, \ldots, x_n)$ as an abbreviation for $(x_1, x_2, \ldots, x_n) \in R$:

$$R(x_1, x_2, \ldots, x_n) \Longleftrightarrow (x_1, x_2, \ldots, x_n) \in R.$$

Proposition 4.13.1 *The intersection, union, and difference of two URM-decidable relations is a URM-decidable relation.*

4.14 Partial decidability

A numerical relation $R(x_1, ..., x_n)$ is said to be *partially decidable*, or *recursively enumerable*, if there is an RM program P such that for each n-tuple of inputs $(x_1, ..., x_n)$, P has output 1 if $R(x_1, ..., x_n)$ is true and P diverges (never halts) if $R(x_1, ..., x_n)$ is false.

For example, for every RM program P, the set of x such that P halts with input x is partially decidable.

Theorem 4.14.1 *Every decidable relation is partially decidable.*

Theorem 4.14.2 *If both $R(\times)$ and $\neg R(\times)$ are partially decidable, then $R(\times)$ is decidable.*

4.15 The Diagonal Method

We begin Cantor's proof that the set of all functions from **N**to **N**cannot be enumerated.

Theorem 4.15.1 *(Cantor) Suppose that $\{f_n : n \in \mathbf{N}\}$ is a sequence of function each from **N**to **N**. Then there exists a function $f : \mathbf{N} \mapsto \mathbf{N}$ which is not equal to any of the f_n.*

proof: Let f be defined as follows:

$$f(n) = f_n(n) + 1$$

This result proves that the set of all numerical functions is uncountable.

Theorem 4.15.2 *The total binary function F defined by*

$$F(e,x) = \begin{cases} \phi^{(1)}(e,x) & \text{if } (e,x) \in Dom(\phi^{(1)}) \\ 0 & \text{otherwise} \end{cases}$$

is not URM computable.

Proof: Suppose not, i.e. that the binary function F is computable. Then the unary function g defined by

$$g(x) = F(x,x) + 1$$

for $x \in \mathbf{N}$ is also URM computable. The function F is by definition total (that's the whole point) and hence so is g. Let P be a program which computes g and let $e = \gamma(P)$ be its Gödel number. Thus

$$(e,x) \in Dom(\phi^{(1)})$$

for every $x \in \mathbf{N}$ and

$$g(x) = \phi^{(1)}(e,x).$$

In particular,

$$(e,e) \in Dom(\phi^{(1)})$$

and

$$g(e) = \phi^{(1)}(e,e).$$

Hence

$$\begin{aligned} \phi^{(1)}(e,e) &= g(e) \\ &= F(e,e) + 1 \\ &= \phi^{(1)}(e,e) + 1 \end{aligned}$$

which is a contradiction. Thus the assumption that F is URM computable is wrong; i.e. F is not URM computable.

4.16 The Halting Problem.

For each n define the *halting problem* to be the relation $H^{(n)} \subset \mathbf{N}^{n+1}$ defined by

$$(e, a_1, a_2, \ldots, a_n) \in H^{(n)} \Longleftrightarrow (a_1, a_2, \ldots, a_n) \in Dom(\phi_e^{(n)})$$

is Thus $(e, a_1, a_2, \ldots, a_n) \in H^{(n)}$ says that the program with Gödel number e halts on input $(a_1, a_2, \ldots, a_n)$.

Theorem 4.16.1 *The halting problem is undecidable.*

Proof: We shall do the case $n = 1$. If the relation $H^{(1)}$ is decidable then by the theorem on definition by cases (see 4.11.2) the total binary function F defined by

$$F(e, x) = \begin{cases} \phi^{(1)}(e, x) & \text{if } (e, x) \in H^{(1)} \\ 0 & \text{otherwise} \end{cases}$$

is also URM computable. But this is the function F of 4.15.2.

Corollary 4.16.2 *There is a URM program P such that the predicate*

$$H_P = \{a \in \mathbf{N} : P(a) \downarrow\}$$

is not decidable.

For[12] another view of the halting problem see footnote 12.

4.17 The Gödel Incompleteness Theorem

The Gödel incompleteness theorem shows that Peano arithmetic is not complete, that is, there are sentences which are true in the standard model of number theory but are not provable from Peano arithmetic. The proof of the theorem uses Gödel numbers of formulas and of tableau proofs.

Throughout this section, all formulas and sentences will be understood to be in the vocabulary of Peano arithmetic. We shall say that a sentence A is **true** if A is true in the standard model of arithmetic, (which has the set N of natural numbers as a universe and the usual functions $0, s, +$, and $*$).

[12] for another view of the halting problem see footnote 12

Gödel numbers of formulas: We first identify each symbol of predicate logic with the vocabulary of Peano arithmetic with a positive integer. Let us identify the symbols

$$\neg, \vee, \wedge, \Rightarrow, \Leftrightarrow, [,], \forall, \exists, (,), =, 0, s, +, *$$

with the integers 1 through 16, and identify the variables in order with the integers from 17 on. Each finite sequence[13] of symbols, and in particular each wff, then has a Gödel number.

Gödel numbers of finite tableaus. Let T be a finite tree with parent function π, and for each nonroot node t of T let $\Phi(t)$ be a wff. Identify the nodes of T with the numbers $1, \ldots, n$, with 1 being the root. We give the labeled tree (T,Φ) the Gödel number of the sequence of Gödel numbers of the triples

$$(2, \pi(2), \phi(2)), ..., (n, \pi(n), \phi(n))$$

where $\phi(i)$ is the Gödel number of $\Phi(i)$. If n = 1, that is, T has only a root, the Gödel number is 0.

A set S of sequences of symbols is said to be decidable if the set of Gödel numbers of elements of S is decidable. Similarly for sets of finite labelled trees.

EXAMPLES. Each of the following sets is decidable:

The set of wffs.

The set of sentences.

The set of axioms of Peano arithmetic.

The *Church-Turing Thesis* is the statement that every computable function is URM-computable. One evidence for this Thesis is that all models of computation that have been considered (e.g. URM's, recursive functions, Turing machines, lambda calculus, etc.) yield the same class of of functions. It can be used to prove that the set of wffs of a predicate logic with finite vocabulary is decidable.

[13]In fact we can Gödel number this book by assigning a number to each symbol (one popular method is called ASCII coding). The Gödel number of the book is then: $27172 \ldots 21010$ which codes: $77, 97, 116, 104 \ldots 101, 110, 100$ which codes: Math ... end. Note that we can refer to the Gödel number of this book but we can't actually include it here!

Lemma 4.17.1 *Let R(x) be a partially decidable relation. There is a wff A with only x free such that*

$$R = \{n \in N \ : \ A(x//n) \textit{is true}\}.$$

Lemma 4.17.2 *The set of all true sentences of arithmetic is undecidable, and is not even partially decidable.*

From now on let H be any set of sentences such that:
(i) H is decidable.
(ii) H contains the set of axioms of Peano arithmetic.
(iii) Every sentence in H is true.
For instance, properties (i) - (iii) hold for Peano arithmetic itself, and also for Peano arithmetic plus one more true sentence.

Lemma 4.17.3 *The set of pairs (T,A) such that T is a tableau proof of A from H is decidable.*

Lemma 4.17.4 *The set of sentences A which are tableau provable from H is partially decidable.*

Theorem 4.17.5 *GODEL INCOMPLETENESS THEOREM. For any set H with properies (i) - (iii) above, there is a sentence A which is true but is not provable from H.*

4.18 The Undecidability of Predicate Logic.

Theorem 4.18.1 *(Church) For every URM program there is a wff $\sigma_P(x)$ such that for every $a \in \mathbf{N}$ the sentence $\sigma_P(\mathbf{a})$ is valid if and only if P halts on input a:*

$$\vdash \sigma_P(\mathbf{a}) \Longleftrightarrow P(a) \downarrow .$$

(Here **a** *is the constant symbol corresponding to a.*

For contrast, it should also be stated that the set of valid wffs of propositional logic is decidable.

The Completeness theorem can be used to prove that the set of valid sentences of full predicate logic is partially decidable.

4.19 Computer problem

This is the first of two problem sets using the GNUMBER program. In this assignment you only need the SIMPLE form of the program, which you start by pressing the space bar when you see the title screen.

Your diskette has the following sample register machine programs:
ADD, MULT, PRED, DOTMINUS, and DIVREM.

Your problem assignment is to type in register machine programs which compute the following functions. Test your answers out using the GNUMBER program, then file your answers on your diskette and give them the names indicated.

In the formulas, x,y are the numbers in registers R1,R2 before running the program, and a,b are the numbers in these registers after running the program.

EQUAL:	$a = 1$ if $x = y, a = 0$ if not $x = y$
SQUARE:	$a = x * x$
ROOT:	a = square root of x if x is a perfect square, undefined otherwise
LESS:	$a = 1$ if $x < y$, $a = 0$ otherwise.
FACTRL:	a = x! ($a = 1 * 2 * ... * x$ if $x > 0, a = 1$ if $x = 0$)
EXP:	a = x raised to the y-th power if $x > 0$, $a = 0$ if $x = 0$ undefined if $x = y = 0$
PRIME:	$a = 1$ if x is prime $(2, 3, 5, 7, 11, \ldots)$, otherwise $a = 0$
LENGTH:	a = the number of decimal digits in x (For example, 7402 has length 4)
DIGIT:	a = the y-th digit in x, counting from 0 on the left (For example, the 0-th digit of 7402 is 7)

In solving your problems, you may load in the sample programs and use them as building blocks if you wish.

4.20 Computer problem

This assignment uses the ADVANCED form of the GNUMBER program, which you start by pressing the G key when you see the title screen.

The problems in this assignment deal with Godel numbers (G.N.'s) of Register machine programs. Each RM instruction is a sequence consisting of an instruction letter and up to four numbers. The instruction letters are identified with numbers as follows:

$$H = 1, Z = 2, S = 3, T = 4, J = 5, E = 6, P = 7.$$

Each instruction, being a finite sequence of numbers, has a Godel number. An RM program P is a finite sequence of instructions $p1, \ldots, pn$. If instruction pm has Godel number gm, then the Godel number of the whole program P is the Godel number of the sequence $g1, \ldots, gn$.

Your diskette has the following sample register machine programs. In the formulas, x,y,z,t are the numbers in registers R1,R2,R3,R4 before running the program, and a,b are the numbers in these registers after running the program.

TERMS: If x is the G.N. of a sequence in standard form, then a is the number of terms of the sequence.

FIVE: Puts the constants 0 through 5 in registers 20 through 25. (It is often convenient to put this at the start of a program).

JOIN: If x and y are G.N.'s of RM programs P and Q in standard form, z and t the numbers of instructions in P and Q, and registers 20 through 25 already contain 0 through 5, then a is the GN of the program P followed by Q with each jump target of Q increased by the number of instructions in P. Extra bonus: this program ends with $z + t$ in R8.

PARAM: If x is the G.N. of a program which neatly computes a function $f(.,.)$ of two variables, then a is the G.N. of a program which neatly computes the function $g(.) = f(y,.)$ of one variable. A commented listing is in section F.

NXSTATE: If x is the G.N. of an RM program in standard form, and y is the G.N. of a sequence representing the register state, then b will be the next state. (y and b are in R4)

UNIV2: The universal program in 2 variables. If x is the G.N. of a RM program P, then a is the output of the program P with inputs y and z in R1 and R2 and zero inputs elsewhere. A commented listing of this program is in section 4.9.

Your problem assignment is to type in register machine programs which compute the following functions. Test your answers out using the GODEL

and UNGODEL commands in the GNUMBER program, then file your answers on your diskette and give them the names indicated. The approximate number of steps required for the program is shown. When you need small constants, it is recommended that you start your program with FIVE to put 0 through 5 in registers 20 through 25.

CONCAT: If x and y are G.N.'s of sequences of numbers in standard form, and z and t are the numbers of terms in these sequences, then a is the G.N. of the first sequence followed by the second sequence. (Concatenation of two sequences). (7 steps)

CONST: a is the G.N. of an RM program which puts the constant x in R1. (19 steps)

SUCC: If x is the G.N. of a program in standard form which computes a function $f(.)$, then a is the G.N. of a program which computes the function $f(.) + 1$. (23 steps)

TOPREG: If x is the G.N. in standard form of a program P, then a is the largest number of a register mentioned in the first y instructions of x. (28 steps)

COMPOSE: If x and y are G.N.'s in standard form of programs which neatly compute functions $g(.)$ and $h(.)$ of one variable, then a is the G.N. of a program which neatly computes the composition function $f(.) = g(h(.))$. (61 steps)

BEFORE: $a = 1$ if the RM program with G.N. x, inputs y and z in R1 and R2, and zero inputs elsewhere, halts before t steps, and a = 0 otherwise. (Hint: This can be done by slightly modifying the RM program UNIV2. UNIV2 puts the time in R15.) (63 steps)

RECUR: If x is the G.N. in standard form of a program P which neatly computes a total function $h(.,.)$, y is the largest register mentioned by P, and z is the number of instructions of P, then a is the G.N. of a program which computes the function $f(.)$ obtained from h by primitive recursion in the form

$$f(0) = 1, f(u + 1) = h(f(u), u).$$

(It is possible but difficult to do this in 92 or fewer steps. If you run out of space, write the shorter program RECUR0 which does what RECUR should do provided that registers 20 through 27 already contain the constants 0 through 7.)

In solving your problems, you may load in the sample programs and use

them as building blocks if you wish. Remember that the LOAD command has been improved so that you can load RM programs from the diskette without erasing the old instruction list.

4.21 Exercises

1. a) Write a register machine program which diverges for every input.
b) Write a register machine program P such that: $P(x, y) \uparrow$ if $x = y$, $P(x, y) \downarrow 0$ if $x \neq y$.

2. Write a register machine program which uses only the instructions Z, S, and J, and has the effect of placing the number in register 3 into register 7. (This shows that the T command can always be avoided in register machine programs).

3. Suppose instead of using the URM instruction set we use JN,S,Z,T,H where

JN 1 2 6

would mean Jump to instruction 6 if the contents of register 1 is not equal to the contents of register 2. Prove that every computable function is computable in this new sense.

4. Suppose we consider programs that only use the instructions S,Z,T,H; i.e. no jump instructions at all. Show that not every computable function is computable in this sense.

5. Write a register machine program which computes the function
f(x) = Gödel number of the sequence (0,1,...,x).
(You may use the Extract or Put instructions in this problem).

6. Suppose R(x,y) is a decidable relation, and f(x) is the function obtained from R by unbounded minimalization,

$$f(x) = y \Leftrightarrow [R(x, y) \wedge \forall z < y \neg R(x, z)].$$

Prove that f is computable by drawing a diagram showing how to get an RM program for f from an RM program for the characteristic function of R.

7. Prove that the undecidable relation

$$\{(x, y) : \text{the RM program with G.N. x and input y halts}\}$$

is partially decidable.

8. Prove that the relation

$$\{x : \text{for some y, the RM program with G.N. x and input y halts}\}$$

is partially decidable.

9. Suppose the numerical relation R(x,y) is decidable. Use a flow diagram to show that the relation

$$\exists z[z \leq y \wedge R(x, z)]$$

is also decidable.

10. Prove that the relation

$$R = \{(x, y) : UNIV(x, y) \downarrow 0\}$$

is undecidable, where UNIV is the universal program which computes the ouput of the register machine program with Gödel number x and input y.

11. Let A be a sentence which is true in the standard model N of Peano arithmetic, (i.e. the model with universe $\{0, 1, 2, ...\}$ and the usual $0, S, +, *$). Using both the Gödel completeness and incompleteness theorems, prove that there is a model M of A and a sentence B such that B true in N and false in M.

12. Prove that the relation
[U(x) iff x is the G.N. of a program which computes the characteristic function of a unary relation]
is undecidable.

13. Suppose $P : \mathbf{N} \mapsto \mathbf{N}$ is a partial recursive function and X is the domain of P, i.e.

$$X = \{n \ : \ P(n) \downarrow\}$$

Show that if X is nonempty, then there exists a total recursive function $f : \mathbf{N} \mapsto \mathbf{N}$ such that X is the range of f, i.e.

$$X = \{f(n) \ : \ n \in \mathbf{N}\}.$$

[Recursion theorists would say that X is recursively enumerable.]

14. Define $D(x) \doteq U(x, x) + 1$ where U is the universal program. Let e be a Gödel number of D. Prove that $D(e) \uparrow$.

15. Define $E(x) \doteq U(x, x)$. Let e be a Gödel number of E. Prove that $E(e) \uparrow$. Prove or disprove: $U(e, e) \downarrow$.

16. Do **not** do this problem.

17. Prove that no one will be able to do this problem set completely. (Hint: see the previous problem.)

18. Give a tableau proof of that the Halting problem is undecidable.

19. Reading assignment: **Gödel, Escher, Bach: an Eternal Golden Braid**, Douglas R. Hofstadter, Basic Books 1979.

5 Mathematical Lingo

This chapter defines the most fundamental terminology used by modern mathematicians. The treatment is informal and descriptive. The importance of this lingo cannot be overestimated: it forces us to make certain vital distinctions without which confusion reigns. In this chapter we explain most of the notations and terminology which are in common use and indicate those which are synonomous with those used in this book. The reader should use the index to find the definition of a new term.

There are three kinds of mathematical objects: individuals, sets, and functions. Loosely speaking, an **individual** is an object (like a number or point) which has no further structure, a **set** is an object (like the set of integers or the set of odd integers) which is comprised of other (simpler) objects called its **elements**, and a **function** is an operation (like addition) which assigns to one or more "input" objects (called the **arguments** of the function) an "output" object (called the **value** of the function for the given arguments).

5.1 Sets.

A set X divides the mathematical universe into two parts: those objects x which **belong** to X and those which don't. The notation $x \in X$ means x belongs to X, the notation $x \notin X$ means that x does not belong to X. The objects which belong to X are called the **elements** of X or the **members** of X. Other words which are roughly synonmous with the word set are **class**, **collection**, and **aggregate**. These longer words are generally used to avoid using the word **set** twice in one sentence. (The situation typically arises when an author wants to talk about sets whose elements are themselves sets; he/she might say " the collection of all finite sets of integers" rather than "the set of all finite sets of integers".) Authors typically try to denote sets by capital letters (e.g. X) and their elements by the corresponding small letters (e.g. $x \in X$) but are not required to do so by any commonly used convention. Thus when reading a mathematics book which is discussing a set X and individuals x and y you should *never* assume that $x \in X$ and $y \notin X$ unless the author has explicitly asserted (or assumed) these relations.

The simplest sets are **finite** and these are often defined by simply listing **(enumerating)** their elements between curly brackets. Thus if $X = \{2, 3, 8\}$

then $3 \in X$ and $7 \notin X$. Often an author uses dots as a notational device to mean "et cetera" and indicate tha the pattern continues. Thus if

$$A = \{a_1, a_2, \ldots, a_n\} \tag{6}$$

then for any object b, the phrase "$b \in A$" and the phrase "$b = a_i$ for some $i = 1, 2, \ldots$" have the same meaning; i.e. one is true if and only if the other is. Having defined A by (6) we have

$$b \in A \Longleftrightarrow b = a_1 \text{ or } b = a_2 \text{ or } \ldots \text{ or } b = a_n,$$

i.e. the shorter phrase "$b \in A$" has the same meaning as the more cumbersome phrase "$b = a_1$ or $b = a_2$ or $\ldots$ $b = a_n$".

The device of listing some of the elements with dots between curly brackets can also be used to define infinite sets provided that the context makes it clear what the dots stand for. For example we can define the set of **natural numbers** by

$$\mathbf{N} = \{0, 1, 2, 3, \ldots\}$$

and the set of **integers** by

$$\mathbf{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$$

and hope that the reader understands that $0 \in \mathbf{N}$, $5 \in \mathbf{N}$, $-5 \notin N$, $\frac{3}{5} \notin \mathbf{N}$, $0 \in \mathbf{Z}$, $5 \in \mathbf{Z}$, $-5 \in \mathbf{Z}$, $\frac{3}{5} \notin \mathbf{Z}$, etc..

Certain sets are so important that they have names:

$\emptyset$	(the empty set)
$\mathbf{N}$	(the natural numbers)
$\mathbf{Z}$	(the integers)
$\mathbf{Q}$	(the rational numbers)
$\mathbf{R}$	(the real numbers)
$\mathbf{C}$	(the complex numbers)

These names are almost universally used by mathematicians today, but in older books one may find other notations. Here are some true assertions: $0 \notin \emptyset$, $\frac{3}{5} \in \mathbf{Q}$, $\sqrt{2} \notin \mathbf{Q}$, $\sqrt{2} \in \mathbf{R}$, $x^2 \neq -1$ for all $x \in \mathbf{R}$, and $x^2 = -1$ for some $x \in \mathbf{C}$ (namely $x = \pm i$).

If X is a set and $P(x)$ is a property which either holds or fails for each element $x \in X$, then we may form a new set Y consisting of all $x \in X$ for which $P(x)$ is true. This set Y is denoted by

$$Y = \{x \in X : P(x)\} \tag{7}$$

(some authors write | instead of : here) and called "the set of all $x \in X$ such that $P(x)$". For example, if $Y = \{x \in \mathbf{N} : x^2 < 6 + x\}$, then $2 \in Y$ (as $2^2 < 6 + 2$), $3 \notin Y$ (as $3^2 \not< 6 + 3$), and $-1 \notin Y$ (as $-1 \notin \mathbf{N}$). This is a very handy notation. Having defined Y by (7) we may assert that for all x

$$x \in Y \Longleftrightarrow x \in X \text{ and } P(x)$$

and that for all $x \in X$

$$x \in Y \Longleftrightarrow P(x).$$

Since the property $P(x)$ may be quite cumbersome to state, the notation $x \in Y$ is both shorter and easier to understand.

Let Y and X be two sets. Then Y is a **subset** of X, written $Y \subset X$ iff every element of Y is an element of X; i.e. iff for all x, $x \in Y$ implies $x \in X$. Symbolically:

$$Y \subset X \Longleftrightarrow \forall x\, [x \in Y \Rightarrow x \in X].$$

Two sets are **equal**, written $X = Y$ iff $X \subset Y$ and $Y \subset X$, i.e. iff every element of X is an element of Y and every element of Y is an element of X. Symbolically:

$$X = Y \Longleftrightarrow \forall x\, [x \in X \Leftrightarrow x \in Y].$$

Example 5.1.1 *Let* $X = \{x \in \mathbf{N} : x^2 + 7 < 6x\}$ *and* $Y = \{2, 3, 4\}$. *Then* [14] $X = Y$.

It follows from the definitions that the set defined by an enumeration is unaffected by the order of the enumeration and by any repetitions in the enumeration. Thus

$$\{1, 3, 7\} = \{3, 1, 7\} = \{3, 1, 7, 1, 3\}.$$

[14] See section A for detailed proof.

5.2 Functions.

A **function** is a mathematical object f consisting of a set X called the **domain** of f, a set Y called the **codomain** of f, and an operation which assigns to every element $x \in X$ a unique value $f(x) \in Y$. This is summarized by the notation

$$f : X \longrightarrow Y$$

A function is also called a **map** or **mapping** and sometimes a **transformation**, while domain and codomain are often called **source** and **target** respectively. (Note that the arrow goes from source to target.) The unique value assigned to $x \in X$ is usually denoted by $f(x)$ but in some contexts other notations such as f_x or fx are customary. One calls $f(x)$ the **value** of f for **argument** x. The function f can be viewed as a computer program which takes an **input** $x \in X$ and produces an **output** $f(x) \in Y$. The input must be an element of the set $X = Dom(f)$ or the program bombs (either produces an error message or goes into an infinite loop).

When we don't want to give it a seperate name we denote the domain of a function f by $Dom(f)$. Some authors call a function Y**-valued** when its target is Y. (For these authors e.g. a "real valued function f defined on X " is a function $f : X \longrightarrow \mathbf{R}$.) The set of values actually assumed by f is called the **range** of f and denoted $Ran(f)$:

$$Ran(f) = \{f(x) \in Y : x \in Dom(f)\}.$$

Thus

$$f : X \longrightarrow Y \Longleftrightarrow \begin{cases} f \text{ is a function,} \\ Dom(f) = X, \\ Ran(f) \subset Y. \end{cases}$$

The notation $f : X \longrightarrow Y$ conveys the idea that $f(x) \in Y$ when $x \in X$ i.e. that f takes its input from X and returns as output an element of Y. It is important to note that the notation $f : X \longrightarrow Y$ entails that $Dom(f)$ is equal to X (i.e. that $f(x)$ is defined for every $x \in X$) but does not entail that $Ran(f)$ is all of Y but only a subset. Thus if $x \in X$ we may conclude that $f(x) \in Y$ and if $y \in Ran(f)$ we may conclude that $y = f(x)$ for some $x \in X$ but from $y \in Y$ we should not conclude that $y = f(x)$ without further hypothesis.

Any numerical expression involving a real variable defines a function. For example, the equation

$$f(x) = \frac{1}{1-x}$$

defines a function $f : X \longrightarrow \mathbf{R}$ whose domain is given by

$$X = Dom(f) = \{x \in \mathbf{R} : x \neq 1\}$$

and whose output is $5 \in \mathbf{R}$ when the input is $\frac{4}{5} \in X$. (In elementary mathematics books the domain of a function defined by an explicit formula in this fashion is always assumed to be the largest set where the formula is meaningful and the codomain is assumed to be the set $\mathbf{R}$ of real numbers. In more advanced books it is customary to specify domain and codomain as part of the definition.)

Sometimes one wishes to refer to a function without giving it a name. A good way to do this is with the symbol $\mapsto$. Thus one could refer to the function f defined above as the map

$$\{x \in \mathbf{R} : x \neq 1\} \longrightarrow \mathbf{R} : \ x \mapsto \frac{1}{1-x}.$$

Two points must be emphasized. First, the value of the function is unique. [15] For $x_1, x_2 \in Dom(f)$ we have

$$x_1 = x_2 \Longrightarrow f(x_1) = f(x_2).$$

For example, by convention, $\sqrt{x}$ is defined for real numbers $x \geq 0$ and denotes the non-negative square root of x. Thus y
$sqrtx \Longleftrightarrow x^2 = y$ and $x \geq 0$. In particular, $\sqrt{4} = 2$ and $\sqrt{4} \neq -2$. Secondly, the notation $f(x)$ is meaningful if and only if $x \in Dom(f)$; it is wrong to write $f(x)$ without proving or assuming that $x \in Dom(f)$. (Some authors will not specify $Dom(f)$ explicitly but will note that $f(x)$ is "not defined" for certain values of x: what they mean is that such values of x are not in $Dom(f)$.)

Two functions $f_1 : X_1 \longrightarrow Y_1$ and $f_2 : X_2 \longrightarrow Y_2$ are **equal** iff their domains and codomains are equal: $X_1 = X_2 \qquad Y_1 = Y_2$ and they return

[15] In older mathematics books this convention was not always followed, leading to so-called "multi-valued" functions. In effect such books declared that the meaning of $y = \sqrt{x}$ is $y^2 = x$. But how do you avoid the conclusion that $\sqrt{4} = 2$ and $\sqrt{4} = -2$ so that $2 = -2$?

the same output for any input: $f_1(x) = f_2(x)$ for all $x \in X_1$. This may be summarized symbolically by:

$$f_1 = f_2 \qquad \Longleftrightarrow \qquad \begin{cases} X_1 = X_2, \, Y_1 = Y_2, \text{ and} \\ f_1(x) = f_2(x) \text{ for all } x \in X_1. \end{cases}$$

We caution the reader that according to this definition of equality the two functions $f : \mathbf{Z} \longrightarrow \mathbf{Z}$ and $g : \mathbf{Z} \longrightarrow \mathbf{N}$ defined by

$$f(x) = g(x) = x^2$$

for $x \in \mathbf{Z}$ are not equal since their targets are not equal. It may seem like nit-picking to distinguish these two (and indeed until recently most authors did not) but failure to make the distinction sometimes leads to confusion.

5.3 Inverses.

Given functions $f : X \longrightarrow Y$ and $g : Y \longrightarrow Z$ the **composition** of f and g is denoted $g \circ f$ (read " *g after f*") and defined by $g \circ f : X \longrightarrow Z$ with

$$(g \circ f)(x) = g(f(x))$$

for $x \in X$.

A function which returns its argument unchanged is called an *identity function*; more precisely the function

$$id_X : X \longrightarrow X$$

defined by

$$id_X(x) = x$$

for $x \in X$ is called the **identity function** of X.

A function $f : X \longrightarrow Y$ is a **injection** (adjective: **injective**) iff its output determines its input uniquely; [16] i.e. iff for all $x_1, x_2 \in X$ we have $x_1 = x_2$ whenever $f(x_1) = f(x_2)$. A function $f : X \longrightarrow Y$ is a **surjection** (adjective: **surjective**) iff every point of Y is the output of some input; i.e.

[16] Of course, for any function its input determines its output uniquely; that is the definition of a function.

iff for every $y \in Y$ there is an $x \in X$ such that $f(x) = y$. A function is a **bijection** (adjective: **bijective**) iff it is both injective and surjective. In older terminology an injective function is called **one-to-one**, a surjective function is called **onto**, and a bijection is called a **one-to-one correspondence**.

For example, let $\mathbf{R}^+$ denote the set of non-negative real numbers:

$$\mathbf{R}^+ = \{x \in \mathbf{R} : x \geq 0\}$$

and consider the four functions:

$$\begin{array}{ll} f_1 : \mathbf{R} \longrightarrow \mathbf{R} & f_1(x) = x^2 \text{ for } x \in \mathbf{R}; \\ f_2 : \mathbf{R} \longrightarrow \mathbf{R}^+ & f_2(x) = x^2 \text{ for } x \in \mathbf{R}; \\ f_3 : \mathbf{R}^+ \longrightarrow \mathbf{R} & f_3(x) = x^2 \text{ for } x \in \mathbf{R}^+; \\ f_4 : \mathbf{R}^+ \longrightarrow \mathbf{R}^+ & f_4(x) = x^2 \text{ for } x \in \mathbf{R}^+. \end{array}$$

Then f_1 is neither injective nor surjective, f_2 is surjective but not injective, f_3 is injective but not surjective, and f_4 is bijective.

Let $f : X \longrightarrow Y$. A **left inverse** to f is a map $g : Y \longrightarrow X$ such that

$$g \circ f = id_X$$

and a **right inverse** to f is a map $g : Y \longrightarrow X$ such that

$$f \circ g = id_Y.$$

A **two-sided inverse** to f is a map which is both a left inverse to f and a right inverse to f. The word *inverse* unmodified means two-sided inverse.

Proposition 5.3.1 *The following three assertions relate the notions of inverse functions to the notions of injective, surjective, and bijective.*

injective *A map $f : X \longrightarrow Y$ is injective if and only if there is a left inverse $g : Y \longrightarrow X$ to f. If f is injective but not surjective the left inverse is not unique.*

surjective *A map $f : X \longrightarrow Y$ is surjective if and only if there is a right inverse $g : Y \longrightarrow X$ to f. If f is surjective but not injective the right inverse is not unique.*

bijective *A map $f : X \longrightarrow Y$ is bijective if and only if there is a two-sided inverse $g : Y \longrightarrow X$ to f. If f is bijective then the inverse is unique since it is characterized by the condition that*

$$y = f(x) \Longleftrightarrow x = g(y)$$

for $x \in X$ and $y \in Y$. (In fact, when f is bijective, the only left inverse is the inverse g and the only right inverse is g.) The inverse of a bijective map $f : X \longrightarrow Y$ is often denoted by $f^{-1} : Y \longrightarrow X$:

$$y = f(x) \Longleftrightarrow x = f^{-1}(y).$$

To understand the meaning of this proposition think of the equation $y = f(x)$ as a problem to be solved for x. Then the function $\left\{ \begin{array}{c} \text{injective} \\ \text{surjective} \\ \text{bijective} \end{array} \right\}$ iff for every $y \in Y$ the equation $y = f(x)$ has $\left\{ \begin{array}{c} \text{at most} \\ \text{at least} \\ \text{exactly} \end{array} \right\}$ one solution $x \in X$. Now if $g : Y \longrightarrow X$ is a right inverse to f the problem $y = f(x)$ has at most one solution for if $y = f(x_1) = f(x_2)$ then $g(y) = g(f(x_1)) = g(f(x_2))$ whence $x_1 = x_2$ since $g(f(x) = id_X(x) = x$. Conversely, if the problem $y = f(x)$ has at most one solution, then any map $g : Y \longrightarrow X$ which assigns to $y \in Y$ a solution x of $y = f(x)$ (when there is one) is a left inverse to f. (It does not matter what value g assigns to y when there is no solution x.) Similarly, if $g : Y \longrightarrow X$ is a right inverse to $f : X \longrightarrow Y$ then $x = g(y)$ is a solution to $y = f(x)$ since $f(g(y)) = id_Y(y) = y$. The converse assertion that there is a right inverse $g : Y \longrightarrow X$ to any surjective map $f : X \longrightarrow Y$ may not seem obvious to someone who thinks of a function as a computer program: even though the problem $y = f(x)$ has a solution x, it may have many, and how is a computer program to choose? (If $X \subset \mathbf{N}$ one could define $g(y)$ to be the smallest $x \in X$ which solves $y = f(x)$ but this will not work if $X = \mathbf{Z}$ for in this case there may not be a smallest x.) In fact, this converse assertion is generally taken as an axiom: the so called *axiom of choice*, and cannot be proved from the other axioms of mathematics. [17]

Here are some familiar bijections and their inverses. (Note how carefully the source and target of each function are specified.)

[17] Although it can be proved in certain cases; e.g. when $X \subset \mathbf{N}$.

1. The linear map $\mathbf{R} \longrightarrow \mathbf{R} : x \mapsto ax + b$ is bijective if $a \neq 0$; its inverse is the map $\mathbf{R} \longrightarrow \mathbf{R} : y \mapsto (y - b)/a$. For $x, y \in \mathbf{R}$:
$$y = ax + b \Longleftrightarrow x = (y - b)/a.$$

2. The restricted square function $\mathbf{R}^+ \longrightarrow \mathbf{R}^+ : x \mapsto x^2$ is bijective; its inverse is the square root function $\mathbf{R}^+ \longrightarrow \mathbf{R}^+ : y \mapsto \sqrt{y}$. For $x, t \in \mathbf{R}^+$:
$$y = x^2 \Longleftrightarrow x = \sqrt{y}.$$

3. The cube function $\mathbf{R} \longrightarrow \mathbf{R} : x \mapsto x^3$ is bijective; its inverse is the cube root function $\mathbf{R} \longrightarrow \mathbf{R} : y \mapsto y^{\frac{1}{3}}$. For $x, y \in \mathbf{R}$:
$$y = x^3 \Longleftrightarrow x = y^{\frac{1}{3}}$$

4. The exponential map $\mathbf{R} \longrightarrow \mathbf{R} \setminus \{0\} : x \mapsto e^x$ is bijective; its inverse is the natural logarithm $\mathbf{R} \setminus \{0\} \longrightarrow \mathbf{R} : y \mapsto \ln(y)$. For $x, y \in \mathbf{R}$ with $y > 0$:
$$y = e^x \Longleftrightarrow x = \ln(y).$$

5. The restricted sine function $\sin : \{\theta \in \mathbf{R} : -\frac{\pi}{2} \leq \theta \leq \frac{\pi}{2}\} \longrightarrow \{y \in \mathbf{R} : -1 \leq y \leq 1\}$ is bijective; its inverse is the inverse sine function [18] $\sin^{-1} : \{y \in \mathbf{R} : -1 \leq y \leq 1\} \longrightarrow \{\theta \in \mathbf{R} : -\frac{\pi}{2} \leq \theta \leq \frac{\pi}{2}\}$. For $-\frac{\pi}{2} \leq \theta \leq \frac{\pi}{2}\}$ and $-1 \leq y \leq 1$:
$$y = \sin(\theta) \Longleftrightarrow \theta = \sin^{-1}(y).$$

6. The restricted cosine function $\cos : \{\theta \in \mathbf{R} : 0 \leq \theta \leq \pi\} \longrightarrow \{x \in \mathbf{R} : -1 \leq x \leq 1\}$ is bijective; its inverse is the inverse cosine function [19] $\cos^{-1} : \{x \in \mathbf{R} : -1 \leq x \leq 1\} \longrightarrow \{\theta \in \mathbf{R} : 0 \leq \theta \leq \pi\}$. For $0 \leq \theta \leq \pi$ and $-1 \leq x \leq 1$:
$$y = \sin(\theta) \Longleftrightarrow \theta = \sin^{-1}(y).$$

7. The restricted tangent function $\tan : \{\theta \in \mathbf{R} : -\frac{\pi}{2} \leq \theta \leq \frac{\pi}{2}\} \longrightarrow \mathbf{R}$ is bijective; its inverse is the inverse tangent function [20] $\tan^{-1} : \mathbf{R} \longrightarrow \{\theta \in \mathbf{R} : -\frac{\pi}{2} \leq \theta \leq \frac{\pi}{2}\}$ For $-\frac{\pi}{2} \leq \theta \leq \frac{\pi}{2}$ and $u \in \mathbf{R}$
$$u = \tan(\theta) \Longleftrightarrow \theta = \tan^{-1}(u).$$

[18] Sometimes called the arcsine and denoted arcsin.
[19] Sometimes called the arccosine and denoted arccos.
[20] Sometimes called the arccosine and denoted arctan.

5.4 Cartesian Product.

Let $n \in \mathbf{N}$ be a natural number and X be a set. An **n-tuple** of elements of X is a finite sequence

$$x = (x_1, x_2, \ldots, x_n)$$

where $x_1, x_2, \ldots, x_n \in X$. The set of all n-tuples of elements of X is denoted by X^n. More generally, if $X_1, X_2, \ldots, X_n$ are sets then the **cartesian product** $X_1 \times X_2 \times \ldots \times X_n$ is the set of all n-tuples $(x_1, x_2, \ldots, x_n$ with $x_1 \in X_1$, $x_2 \in X_2$, ..., $x_n \in X_n$:

$$X_1 \times X_2 \times \ldots \times X_n = \{(x_1, x_2, \ldots, x_n) : x_i \in X_i \text{ for } i = 1, 2, \ldots, n\}.$$

Thus

$$X^n = \underbrace{X \times X \times \ldots \times X}_{n}$$

The Cartesian product is also called the **direct product** by some authors.

Let X be a set. A subset $R \subset X^n$ is called an n-ary **relation** [21] **unary** means 1-ary, **binary** means 2-ary, **ternary** means 3-ary. (The word "relation" unmodified usually means "binary relation".) In some contexts it is customary to write $R(x_1, x_2, \ldots, x_n)$ rather than $(x_1, x_2, \ldots, x_n) \in R$.

Similarly, a function $f : X^n \longrightarrow Y$ is sometimes called an n-ary function. When $X = Y$ the word *operation* is often used in place of the word function; thus a **unary operation** on a set X is a function with domain and codomain X, a **binary operation** on X is a function with domain X^2 and codomain X, a **ternary operation** on X is a function with domain X^3 and codomain X, etc..

Given a function $f : X_1 \times X_2 \times \ldots \times X_n \longrightarrow Y$ it is customary to denote the value of f for argument $(x_1, x_2, \ldots, x_n)$ by $f(x_1, x_2, \ldots, x_n)$ rather than $f((x_1, x_2, \ldots, x_n))$. More generally, one often tacitly omits or inserts parantheses to promote legibility.

Sometimes the value of a function or relation for given arguments is denoted in other ways. For example, we write $x + y$ rather than $+(x, y)$ ($+$ is really a binary function) and $x < y$ rather than $<(x, y)$ ($<$ is really a binary relation). Here, parentheses play the crucial role of indicating the order in which the operations are performed ($x - (y + z) \neq (x - y) + z$) and

[21] Logicians would say that a predicate denotes a relation in the same way that a numeral denotes a number.

when parentheses are omitted this order is determined according to some em convention (e.g. $x - y + z$ means $(x - y) + z$ and not $x - (y + z)$).

The notation where the name of a binary function is placed between (rather than in front of) the arguments is called **infix** notation. Occassionally, the name of the function is placed after the operation – one writes $(x, y)f$ rather than $f(x, y)$ – this is called **postfix** notation. The notation $f(x, y)$ is thus called **prefix** notation. It is possible to omit parantheses unambiguously when using postfix (or prefix notation) and some calculators (e.g. those made by Hewlett-Packard) and programming languages (e.g. APL) do this. (Thus $x - y + z$ is denoted $xy - z+$ in postfix notation.) [22]

Given a subset $A \subset X$ the function

$$\chi_A : X \longrightarrow \{0, 1\}$$

defined by

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

is called the **characteristic function** of A in X. (Some authors call it the *indicator function.*)

Given a function $f : X \longrightarrow Y$ the **graph** of f is the set of all pairs $(x, f(x))$ with $x \in X$:

$$graph(f) = \{(x, y) \in X \times Y : y = f(x)\}.$$

Remark 5.4.1 *Let $G \subset X \times Y$. Then there is a function $f : X \longrightarrow Y$ such that $G = graph(f)$ if and only for every $x \in X$ there exists a unique $y \in Y$ with $(x, y) \in G$. Moreover, given $f_1, f_2 : X \longrightarrow Y$ we have that $f_1 = f_2$ if and only if $graph(f_1) = graph(f_2)$.*

The remark is obvious. When $G = graph(f)$ the unique $y \in Y$ such that $(x, y) \in G$ is $f(x)$ and conversely given G and $x \in X$ we may define $f(x)$ to be the unique $y \in Y$ such that $x \in X$. Because of this remark, some authors identify functions with their graphs, thereby reducing the number of kinds of mathematical objects from three (individuals, sets, and functions) to two (individuals and sets).

[22] The observation that parantheses are not needed with prefix (or postfix) notation is due to a Pole named Łukasiewicz so parentheses-free notation is sometimes called Polish (or reverse Polish) notation.

5.5 Set theoretic operations.

These include

Intersection of two sets. Given two sets A and B, their **intersection** is denoted by $A \cap B$ and defined to be the set of all x in both A and B:

$$A \cap B = \{x | x \in A \text{ and } x \in B\}.$$

Union of two sets. Given two sets A and B, their **union** is denoted by $A \cup B$ and defined to be the set of all x in either A or B:

$$A \cup B = \{x | x \in A \text{ or } x \in B\}.$$

Set-theoretic difference of two sets. Given two sets A and B, their **difference** is denoted by $A \setminus B$ and defined to be the set of all x in A and not in B:

$$A \setminus B = \{x | x \in A \text{ and } x \notin B\}.$$

Image of a set by a function. Given a function $f : X \longrightarrow Y$ and a subset $A \subset X$ the **image** of A by f is the set of all $f(x)$ as x ranges over A:

$$f(A) = \{f(x) | x \in A\}.$$

Range of a function. Given a function $f : X \longrightarrow Y$ the **range** of f is denoted by $Ran(f)$ and is the image of X by f:

$$Ran(f) = f(X).$$

Preimage of a set by a function. Given a function $f : X \longrightarrow Y$ and a subset $B \subset Y$ **preimage** of B by f is denoted by $f^{-1}(B)$ and is defined to be the set of all $x \in X$ such that $f(x) \in B$:

$$f^{-1}(B) = \{x \in X | f(x) \in B\}.$$

Preimage of a point by a function. [23] Given a function $f : X \longrightarrow Y$ and $y \in Y$ the **preimage** of y by f is

$$f^{-1}(y) = f^{-1}(\{y\}) = \{x \in X : f(x) = y\}.$$

[23] In situations like this many authors do not distinguish between a point $y \in Y$ and the corresponding singleton $\{y\} \subset Y$. Thus if f happens to be a bijection, $f^{-1}(y)$ might denote value $f^{-1} : Y \longrightarrow X$ for argument $y \in X$ (i.e. the unique $x \in X$ such that $f(x) = y$) or it might denote the corresponding singleton (i.e. $\{x\}$). The reader must deduce from the context which is meant.

Restriction of a function to a subset of its domain. Given a function $f : X \longrightarrow Y$ and a subset $A \subset X$ the **restriction** of f to A is denoted by $f|A$ and is defined to be the function $A \longrightarrow Y$ whose output for $x \in A$ is $f(x)$.

Union of a collection of sets. Let $\{A_i\}_{i\in I}$ be a collection of sets indexed by I. [24] A point x belongs to the **union** of the sets $A_i : i \in I$ iff it belongs to at least one of them:

$$\bigcup_{i\in I} A_i = \{x | x \in A_i \text{ for some } i \in I\}.$$

Intersection of a collection of sets. Let $\{A_i\}_{i\in I}$ be a collection of sets indexed by I. A point x belongs to the **intersection** of the sets $A_i : i \in I$ iff it belongs to all of them:

$$\bigcap_{i\in I} A_i = \{x | x \in A_i \text{ for all } i \in I\}.$$

Power set of a set. Given a set X the **power set** of X is denoted by 2^X and is defined to be the set of all subsets of X:

$$2^X = \{A | A \subset X\}.$$

Set of functions from one set to another. Given sets X and Y the set of all maps from X to Y is denoted by Y^X:

$$Y^X = \{f | \quad f : X \longrightarrow Y\}.$$

Proposition 5.5.1 *Let A, B, C be sets. Then the following laws hold:*

associative	$(A \cap B) \cap C = A \cap (B \cap C)$
	$(A \cup B) \cup C = A \cup (B \cup C)$
commutative	$A \cap B = B \cap A$
	$A \cup B = B \cup A$
distributive	$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
	$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
De Morgan's	$A \setminus (B \cap C) = (A \setminus B) \cup (A \setminus C)$
	$A \setminus (B \cup C) = (A \setminus B) \cap (A \setminus C)$

[24] i.e. $i \mapsto A_i$ is a function defined on I whose output A_i for input $i \in I$ is a set A_i.

Proposition 5.5.2 *Let $\{A_i\}_{i\in I}$ be a collection of sets indexed by I and X be a set. Then we have the following infinite distributive laws:*

$$X \cap \bigcup_{i\in I} A_i = \bigcup_{i\in I}(X \cap A_i)$$

$$X \cup \bigcap_{i\in I} A_i = \bigcap_{i\in I}(X \cup A_i)$$

and the following infinite De Morgan's laws:

$$X \setminus \bigcap_{i\in I} A_i = \bigcup_{i\in I}(X \setminus A_i)$$

$$X \setminus \bigcup_{i\in I} A_i = \bigcap_{i\in I}(X \setminus A_i)$$

Proposition 5.5.3 *Let $\{B_i\}_{i\in I}$ be an indexed collection of subsets of a set Y, $B \subset Y$, and $f : X \longrightarrow Y$. Then*

$$f^{-1}\left(\bigcup_{i\in I} B_i\right) = \bigcup_{i\in I} f^{-1}(B_i)$$

$$f^{-1}\left(\bigcap_{i\in I} B_i\right) = \bigcap_{i\in I} f^{-1}(B_i)$$

and

$$f^{-1}(Y \setminus B) = X \setminus f^{-1}(B).$$

Proposition 5.5.4 *Let $\{A_i\}_{i\in I}$ be an indexed collection of subsets of a set X, $A \subset X$, and $f : X \longrightarrow Y$. Then*

$$f\left(\bigcup_{i\in I} A_i\right) = \bigcup_{i\in I} f(A_i)$$

$$f\left(\bigcap_{i\in I} A_i\right) \subset \bigcap_{i\in I} f(A_i)$$

5.6 Finite Sets

A set X is called *finite* iff there is a natural number $n \in \mathbf{N}$ and a bijective mapping

$$f : \{1, 2, \ldots n\} \longrightarrow X.$$

Note that the empty set is a finite set (with $n = 0$) for the set $\{1, 2, \ldots n\}$ is also the empty set (since there are no natural numbers greater than or equal to 1 and less than or equal to 0) and the empty mapping is certainly a bijection between the empty set and itself.

We define the *cardinality* of a finite set X to be the natural number n of the definition. There is however a logical difficulty with this kind of definition: in general, whenever we define something to be "the c which satisfies P(c)" we must verify that c is "well-defined" by the condition $P(c)$ i.e. that there is one and only one c which does in fact satisfy P(c). In the present context to show that the notion of cardinality is well-defined we must verify that if $f : \{1, 2, \ldots n\} \longrightarrow X$ and $g : \{1, 2, \ldots m\} \longrightarrow X$ are both bijections, then $m = n$. Since the composition of bijections is a bijection, and the inverse of a bijection is a bijection, the bijections f and g give rise to a bijection $f^{-1} \circ g : \{1, 2, \ldots, m\} \longrightarrow \{1, 2, \ldots n\}$ The justification for the definition of cardinality is the third assertion in the following

Proposition 5.6.1 *Let m and n be natural numbers and*

$$h : \{1, 2, \ldots m\} \longrightarrow \{1, 2, \ldots, n\}$$

a mapping. Then:

- *if h is injective, then $m \leq n$; and*
- *if h is surjective, then $n \leq m$; hence*
- *if h is bijective, then $n = m$.*

The following proposition shows why certain notations were chosen. Denote by $\#(X)$ the cardinality of the finite set X.

Proposition 5.6.2 *Let X and Y be finite sets. Then*

$$\#(X \times Y) = \#(X)\#(Y),$$

$$\#(2^X) = 2^{\#(X)},$$
$$\#(Y^X) = \#(Y)^{\#(X)},$$

and

$$\#(X \cup Y) = \#(X) + \#(Y) - \#(X \cap Y).$$

5.7 Equivalence Relations.

Let X be a set and $\equiv$ a binary relation on X. In this section we use the infix notation $x \equiv y$ instead of $(x, y) \in \equiv$ or $\equiv (x, y)$. The relation $\equiv$ is

1. **reflexive** iff $x \equiv x$;
2. **symmetric** iff $x \equiv y \implies y \equiv x$;
3. **transitive** iff $x \equiv y, y \equiv z \implies x \equiv z$;]

for all $x, y, z \in X$. An **equivalence relation** is a binary relation which is reflexive, symmetric, and transitive. Of course the usual relation of equality $(x = y)$ is an equivalence relation, and in general equivalence relations are relations which behave much like equality.

In fact, there is a device for changing an equivalence relation $\equiv$ into equality. Namely for each $x \in X$ define the equivalence class $[x] = [x]_{\equiv}$ by

$$[x] = \{y \in X : y \equiv x\}.$$

Then for $x, y \in X$ the following are equivalent"

- $x \equiv y$;
- $[x] = [y]$;
- $[x] \cap [y] \neq \emptyset$.

We can form the space of equivalence classes:

$$(X/\equiv) = \{[x] : x \in X\};$$

it is called the **quotient space** of X by the equivalence relation $\equiv$. Since

$$x \equiv y \iff [x] = [y]$$

we have in effect converted the equivalence relation into ordinary (set-theoretic) equality.

One important property of equality is that equals may be substituted for equals. The analog for general equivalence relations is called "respect". More precisely let $fX^n \longrightarrow X$ be an n-ary function on X. We say that the equivalence relation $\equiv$ **respects** the relation f iff for $x_1, y_1, x_2, y_2, \ldots, x_n, y_n \in X$ we have:

$$x_1 \equiv y_1, \ldots, x_n \equiv y_n \Longrightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n).$$

When this happens we define an n-ary function $[f] : (X/\equiv)^n \longrightarrow (X/\equiv)$ by the condition:

$$[f]([x_1], [x_2], \ldots, [x_n]) = [f(x_1, x_2, \ldots, x_n)].$$

The definition is consistent (one says that $[f]$ is *"well-defined"* since

$$[f(x_1, x_2, \ldots, x_n)] = [f(y_1, y_2, \ldots, y_n)]$$

whenever $[x_1] = [y_1], [x_2] = [y_2], \ldots, [x_n] = [y_n]$ so that the definition of $[f]([x_1], \ldots, [x_n])$ does not depend on the particular elements $x_1, \ldots, x_n$ representing the equivalence classes $[x_1], \ldots, [x_n]$. The function $[f] : (X/\equiv)^n \longrightarrow (X/\equiv)$ the function **induced** by f.

Similarly, let $P \subset X^n$ be an n-ary relation on X. We say that the equivalence relation $\equiv$ **respects** the relation P iff for $x_1, y_1, x_2, y_2, \ldots, x_n, y_n \in X$ we have:

$$x_1 \equiv y_1, \ldots, x_n \equiv y_n) \Longrightarrow P(x_1, \ldots, x_n \Leftrightarrow P(y_1, \ldots, y_n).$$

Again we can define an n-ary relation $[P] = [P]_{\equiv} \subset (X/\equiv)^n$ by

$$[P] = \{([x_1], \ldots, [x_n]) \in (X/\equiv)^n : (x_1, \ldots, x_n) \in P\}$$

As before we have for $x_1, \ldots, x_n \in X$ that

$$(x_1, \ldots, x_n) \in P \Leftrightarrow ([x_1], \ldots, [x_n]) \in [P]$$

and we call $[P]$ the n-ary relation induced by P.

The simplest example (aside from equality itself) of an equivalence relation is afforded by "modular arithmetic". Choose a non-zero integer m

(the "modulus") and for $x, y \in \mathbf{Z}$ write $x \equiv y \,(mod\, m)$ iff $x - y$ is divisible by m; e.g. $5 \equiv 19 \,(mod\, 7)$ since 7 divides -14. The quotient space of this equivalence relation is usually denoted $\mathbf{Z}_m$ or sometimes $\mathbf{Z}/m\mathbf{Z}$ and is finite:

$$\mathbf{Z}_m = \{[0], [1], [2], \ldots, [m-1]\}.$$

Equivalence modulo m respects the arithmetic operations of addition, subtraction, and multiplication: namely, if $x_1 \equiv y_1 \,(mod\, m)$ and $x_2 \equiv y_2 \,(mod\, m)$ then $x_1 + x_2 \equiv y_1 + y_2 \,(mod\, m)$, $x_1 - x_2 \equiv y_1 - y_2 \,(mod\, m)$, and $x_1 \cdot x_2 \equiv y_1 \cdot y_2 \,(mod\, m)$. (For example, $2 \equiv 9 \,(mod\, 7)$ and $17 \equiv 3 \,(mod\, 7)$ so $19 \equiv 12 \,(mod\, 7)$, $-15 \equiv 6 \,(mod\, 7)$, and $34 \equiv 27 \,(mod\, 7)$.) An example of a relation which is *not* respected by equivalence modulo m is the usual order relation: thus $3 \equiv 10 \,(mod\, m)$ and $4 \equiv 4 \,(mod\, m)$ but $3 < 4$ and $10 \not< 4$.

An equivalence relation on a set X and a surjective function $\pi : X \longrightarrow W$ are much the same thing. Thus the map π determines an equivalence relation $\equiv$ on X via

$$x \equiv y \Longleftrightarrow \pi(x) = \pi(y)$$

for $x, y \in X$ while an equivalence relation $\equiv$ on a set X determines a map

$$\pi : X \longrightarrow (X/\equiv)$$

called the **projection** via

$$\pi(x) = [x]$$

for $x \in X$.

5.8 Induction on the Natural Numbers

A fundamental principle in mathematics is

The Principle of Mathematical Induction. *Let $S \subset \mathbf{N}$ be a set of natural numbers satisfying*

- $0 \in S$, *and*
- *for all n: $n \in S \Longrightarrow n + 1 \in S$.*

Then S is the set of all natural numbers: $S = \mathbf{N}$.

The principle may be justified as follows. Suppose that $S \subset \mathbf{N}$ satisfies (5.8) and (5.8). Then by (5.8) $0 \in S$. But by (5.8) $0 \in S \Longrightarrow 1 \in S$ so (as $0 \in S$) $1 \in S$. But by (5.8) $1 \in S \Longrightarrow 2 \in S$ so (as $1 \in S$) $2 \in S$. etc.

The principle of mathematical induction is generally used as a method of proof. Typically, a proof which uses mathematical induction will have the following form:

Theorem $S = \mathbf{N}$.
Proof: By mathematical induction. First we show that $0 \in S$. (Some argument will appear here.) Hence $0 \in S$ as required.

Next we assume that $n \in S$. (Some further argument will appear here. In this argument the hypothesis $n \in S$ is referred to as the "induction hypothesis".) Therefore $n + 1 \in S$. This completes the proof.

Students sometimes have trouble with this kind of proof for it looks like one is assuming what has to be proved. The point is that one proves $n+1 \in S$ from the hypothesis that $n \in S$ and then concludes that $n \in S$ from $0 \in S$ and the principle of induction. This is very different from proving that $n \in S$ from the hypothesis that $n \in S$ (a rather trivial activity).

As an example we prove the formula

$$(P_n) \qquad 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

for $n \in \mathbf{N}$. The formula is true for $n = 0$ (since the empty sum is 0); i.e. (P_0) is true. Next assume that (P_n) is true. Then adding $n + 1$ to both sides we obtain

$$\begin{aligned} 1 + 2 + \ldots + n + (n+1) &= \frac{n(n+1)}{2} + (n+1) \\ &= \frac{(n+1)n + 2(n+1)}{2} \\ &= \frac{(n+1)((n+1)+1)}{2} \end{aligned}$$

This is (P_{n+1}). Thus we have proved (P_{n+1}) from the hypothesis (P_n), i.e. we have shown that $(P_n) \Longrightarrow (P_{n+1})$. But according to the principle of mathematical induction, if (P_0) and if for all n we have $(P_n) \Longrightarrow (P_{n+1})$, then for all n we have (P_n). This completes the proof. (Note that we have proved

in particular that $1 + 2 + \cdots + 100 = 5050$ a fact that Euler's schoolmaster thought would require an hour to establish.)

Sometimes the principle of mathematical induction is expressed in a slightly different form: viz.

The Principle of Strong Mathematical Induction. *Let $S \subset \mathbf{N}$ be a set of natural numbers satisfying*

(*) For all $n \in S$ if $\{0, 1, 2, \ldots, n-1\} \subset S$, then $n \in S$.

Then S is the set of all natural numbers: $S = \mathbf{N}$.

This principle is really no different from the other form. Indeed, suppose $S \subset \mathbf{N}$ satisfies (*) and define

$$S' = \{n \in \mathbf{N} : \{0, 1, 2, \ldots, n-1\} \subset S\}.$$

Then clearly $0 \in S'$ (since $\emptyset \subset S$ while (*) asserts that $n + 1 \in S'$ whenever $n \in S'$. Thus by the ordinary form of the principle of mathematical induction, $S' = \mathbf{N}$. This clearly implies that $S = \mathbf{N}$ since if $n \in \mathbf{N}$, then $n + 1 \in \mathbf{N}$ so $n + 1 \in S'$ so $\{0, 1, 2, \ldots, n\} \subset S$ so $n \in S$. The reason this is called the principle of "strong" mathematical induction is that when we construct a proof using this principle we get to assume that $k \in S$) for all $k < n$; we have to prove $n \in S$ from this. (When we use ordinary mathematical induction we have to prove $n \in S$ from a weaker hypothesis: viz. $n = 0$ or $n - 1 \in S$.) The difference however is really that some proofs (like the proof of the addition formula given above) are expressed more naturally using the ordinary principle while others are expressed more naturally using the strong principle.

6 Computer program documentation

6.1 TABLEAU program documentation

TABLEAU helps you write down a tableau proof in predicate logic. It has three modes of operation:

Hypothesis mode builds the formula to be proved and a list of hypotheses. The program will only allow well formed formulas to be entered.

Tableau mode builds a semantic tableau, and shows the current branch and its two neighbors. The program will only allow trees which follow the rules for a semantic tableau.

Map mode shows the whole semantic tableau but with abbreviated formulas.

The program starts in Hypothesis mode. You can change from one mode to another with the commands H, T, and M. The command Q is used to quit the program. To protect against accidental quitting, the program asks you to type Q a second time to be sure you really meant to quit. It gives you a chance to return to the previous state or to start a new tableau instead of quitting.

EQUIPMENT NEEDED

TABLEAU will run on an IBM PC (TM) or compatible computer with at least 256 K of memory and one disk drive. With more memory, you will have room to build a larger tableau, up to 1000 nodes. The files

TABLEAU.COM, TABLEAU.000, TABLEAU.001, and TABLEAU.002

must be in the current directory for the program to run. If you want to load problems and file answers, you will need the problem diskette in drive slot A. When the title screen appears, you are given the choice of running the program on a color or monochrome display. You may experiment to see which choice looks best on your screen.

A QUICK PREVIEW

Here is a short example which will give you an idea of what the program does. Type the keys in square brackets exactly as shown and watch the screen. Begin with the title screen showing and the problem diskette in drive slot A.

[c] (for color), or [m] (for monochrome) (You are now in Hypothesis mode)
[L] (A list of .TBU files is shown)
[sample][RETURN key] (A formula to prove and hypothesis list will appear)
[t] (You are now in Tableau mode)
[Down arrow][Down arrow][g] (The Get command)
[End key][e] (The Extend command) (2 new nodes appear)
[Down arrow][Right arrow][g][e] (2 more nodes appear)
[h][L][asample][RETURN key] (A new tableau is loaded)
[t] (A completed tableau proof is on the screen)
[PgDn] (Another part of the tableau)
[m] (You are now in map mode)
[q][q] (You have quit the TABLEAU program)

HYPOTHESIS MODE

In this mode you can enter the formula to be proved and/or a list of hypotheses. You can either type these in at the keyboard or load them from the diskette. Figure 25 is a sample screen.

MOVING THE CURSOR TO A NEW LINE.

Up or Down arrow : up or down one line.
PgUp : go to the top of the screen.
PgDn : go to the line marked "next".

COMMANDS IN HYPOTHESIS MODE.

The message in the window at the bottom of the screen lists the following commands, except when you are in the process of typing in a new formula.

E : Edit (change) the formula in the current line. This command will not work after a tableau has been built.

K : Kill. The formula in the present line will be erased. This command will not work after a tableau has been built.

L : Load. This command displays a list of files which contain sample hypothesis lists and tableaus. If you type the name of one of these files and press Return or Enter (the key has different names on different computers) a list of hypothesis will appear on the screen. If you press RETURN without a

```
------------------------------------------------------------
                    FORMULA TO BE PROVED
wff  | r \/ s
============================================================
                       HYPOTHESIS LIST
------------------------------------------------------------
wff  | p \/ q
wff  | p->r
wff  | q->s
next |




------------------------------------------------------------
             TABLEAU PROGRAM -- HYPOTHESIS MODE
 Type in the formula to be proved on the line marked "here"
   or type in a new hypothesis on the line marked "next".
 E(dit) K(ill hyp) L(oad) M(ap) P(ull) Q(uit) T(ableau)
------------------------------------------------------------
```

Figure 25: Hypothesis Mode

file name, you will get back to the regular command list with no change. The files have names up to eight characters long followed by the suffix ".TBU". You should not enter the suffix, only the name as it appears in the window.

P : Pull. The hypothesis in the current line is pulled from its present position and put at the end of the hypothesis list. This command will not work after a tableau has been built. You can use this command to easily change the order in which the hypotheses are listed.

M : Change to Map mode.

Q : Quit the program.

T : Change to Tableau mode.

STARTING A FORMULA. You can type in formulas yourself instead of loading a list of formulas from the diskette. You may also add new formulas to a list which has been loaded in. If you move the cursor to a free line, the window at the bottom of the screen will change to a new background color and tell you that you may type in a formula.

Propositional Logic: The symbols which are allowed in formulas are:

AND & (shift 4) /\ (forslash backslash)

OR | (shift backslash) \/ (backslash forslash)

NOT ¬ (Ctrl N)

IMPLIES IFTHEN ONLYIF $->$ (minus, greater than)

IFF $<->$ (less than, minus, greater than)

[ ] (brackets)

The computer will accept either the symbols or words as shown (in paranthesis we give some indication of the key strokes which word on some keyboards). Any other string of letters and numbers which begins with a letter can be used as a propositional symbol.

Predicate Logic: The following additional symbols are allowed:

ALL Å(Ctrl A)

EXIST É (Ctrl E)

$=$ $<$ $<=$ $>$ $>=$ (common infix relations)

$+$ $-$ $*$ (common infix functions)

( ) (parentheses)

, (comma)

Any other string (of letters and numbers) which begins with a letter can be used as a variable, relation symbol or function symbol. The type of symbol and the number of argument places are determined by the first use of the

symbol. A string which begins with a number can be used only as a constant symbol.

MOVING WITHIN A FORMULA. You can move within a formula using the Right and Left arrow keys. New symbols are inserted at the cursor position. The Backspace and Del keys can be used to erase symbols as usual. The Home key will jump to the beginning of the line and the End key will jump to the end of the line. The Esc key will erase the entire line.

FINISHING A FORMULA. The computer will say "new" or "here" if the line is empty, "wff" if the line contains a Well Formed Formula, and will say "bad" otherwise. When you have a wff or an empty line, you may leave the line by pressing RETURN, the Up or Down arrow key, or the PgUp or PgDn key (depending on which line you want to go to next). You cannot leave the line when the computer says "bad". This makes sure that only well formed formulas are allowed. When the computer says " bad", you can type the ? key to get a message telling you what is wrong with the formula.

CHANGING A FORMULA. If the tableau has not been built, you can change an existing formula by moving to its line and pressing the E key (for Edit). Then make changes in the usual way. If you delte the whole line, the computer will say "nil".

TABLEAU MODE

In this mode you can build a semantic tableau. The tableau is a tree which has a formula at each node. The top node has the negation of the formula to be proved, and the next nodes have the hypotheses. If every branch through a node is contradictory, the formula is shown in red (or written between : symbols on a monochrome screen). When every node of the tableau is red, the tableau is a completed proof. Your current location in the tableau is the node which has the blinking cursor and blue background (or reversed text on a monochrome screen). Figure 26 is a sample screen, but with the current location indicated by asterisks ** ** instead of a blue background.

The tableau is built one step at a time. To extend the tableau, you move the cursor to a formula, type G to Get the formula into a box in the window at the bottom of the screen, move the cursor to the end of a branch, and then type E to Extend the tableau. The program will only allow tableau extensions which are legal according to the formal definition of a tableau in the course.

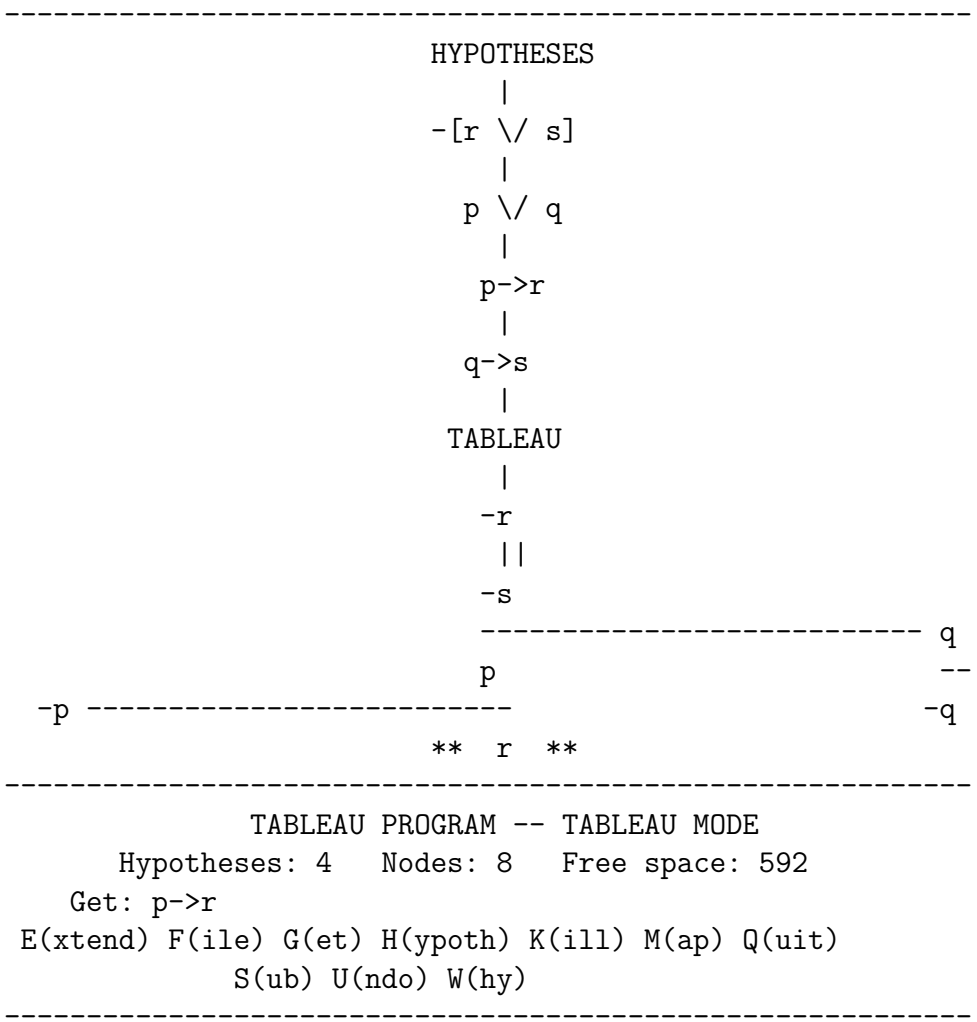


Figure 26: Tableau Mode

MOVING WITHIN THE TABLEAU. The screen shows the current branch of the tableau and the neighboring branches to the right and left. If the tree is too large, only part of the tableau can be seen on the screen at one time. The cursor can be moved within the tableau using the arrow keys in the following ways:

Up arrow : Move up one line.

Down arrow : Move down one line along the current branch.

Right arrow : Move one branch to the right. This is active only when there is a branching directly above the present node.

Left arrow : Same as right arrow but moves one branch to the left.

Home : Move to the top of the tableau.

End : Move to the end of the current branch.

Page Up : Move up one screen (9 lines) .

Page Down : Move down one screen (9 lines).

COMMANDS IN TABLEAU MODE. The list of commands is shown in the window at the bottom of the screen.

Propositional Logic:

E : Extend. The tableau is extended using the tableau rule for the formula in the "Get" box. This command is available only when the cursor is at the bottom of a branch. Nothing happens if the "Get" formula is atomic or negated atomic.

F : File. Saves the hypothesis list and tableau in its present state into a file on the diskette. The computer asks you to type in the name of the file, in the form XXXXXXXX.TBU. (You don't enter the suffix ". TBU"; the computer will add it automatically). Illegal names are ignored, and you are warned if you try to use a name which is already on the diskette. To get back to the program without saving, just type RETURN without typing a file name.

To ERASE an unwanted .TBU file from the diskette, Quit and start an empty tableau (no hypotheses and no formula to be proved), go to Tableau mode, use the File command, and type the name of the file you want to erase.

G : Get. The formula at the cursor is put into the "Get" box in the bottom window. (The formula is then shown in green in the tableau and has a green background in the bottom window.) If you later change branches above that formula, it will drop out of the box. This makes sure that the formula can only be used below the place where it appears in the tableau.

H : Change to Hypothesis mode.

K : Kill. This command erases everything below the cursor, and is used to correct mistakes.

M : Change to Map mode.

Q : Quit the program.

U : Undo. This command undoes the last Kill or Extend command, and goes back to the previous position.

W : Why. This command tells you which formula was used to add the current formula to the tableau. It does this by putting the formula which was used into the "Get" box and writing it in green in the tableau.

Predicate Logic:

When the tableau is extended using a quantified formula, the variable in the quantifier is replaced by a term. In this program, you must tell the computer which term to use. This is taken care of by an extra provision in the Extend command. E : Extend (continued). If the formula in the "Get" box starts with a quantifier or negated quantifier, the bottom window turns red and asks you for a term to substitute for the quantified variable. The rules for entering terms here are the same as the rules for entering formulas in Hypothesis Mode. The computer will not let you enter a bad term and will explain what is wrong when you press the ? key. Press RETURN when you are finished entering the term.

Predicate Logic with Equality:

A second box, the "Sub" box, is added in the bottom window to provide for the equality substitution rule. A new command is added which puts a formula into this box.

S : Substitution. This command is available only when the current formula is an equation. The bottom window turns red and you are asked to either accept the equation as given (Return key), or to reverse it (Right arrow key). The equation will then appear in the "Sub" box with a cyan (blue-green) background and will be written in cyan in the tableau.

E : Extend (continued). If the formula in the "Get" box is an atomic or negated atomic formula, the equality substitution rule will be used. To do this the "Sub" box must contain an equation between two terms. The new formula is formed by taking the "Get" formula and replacing the first

term in the "Sub" box by the second term in the "Sub" box. Nothing will happen if there is no possible substitution. If there is exactly one possible substitution, the bottom window will turn red, the computer will highlight the substitution position, and you will be asked to accept (Return key) or cancel (Esc key). If there is more than one place to substitute, the computer will highlight the first one and ask you to accept, cancel, or go to the next place (Right arrow key).

W : Why (continued). If the current formula was added to the tableau by an equality substitution, the substitution equation will be put into the "Sub" box and the target of the substitution will be put into the "Get" box.

= : The computer will ask you for a term in the bottom of window. When you type int the term t and press return, the wff $t = t$ will be added to the tableau. This command is available only at the end of a branch.

MAP MODE

This mode displays the tableau in a smaller scale by showing only the main connective of the formulas. If the tableau is too large to fit on the screen in Tableau mode, use Map mode to see where you are in the big picture. Figure 27 is a sample screen in this mode.

The current location in the tableau is again shown by the blinking cursor and blue background, and the current formula is displayed in full in the bottom window. You can still use the arrow and page keys to move within the tableau. However, you cannot change the tableau in Map mode. Sometimes the tableau is so complex that it will not fit on the screen even in Map mode. A sharp symbol, #, is used to indicate a portion of the tableau which is too complicated to fit on the screen. The Zoom command can be used to enlarge a portion of the tableau to see what is inside the #.

The commands, shown in the bottom window, are as follows.

H : Change to Hypothesis mode.

Q : Quit the program.

T : Change to Tableau mode.

Z : Zoom. Redraws part of the tableau in a larger scale with the present cursor position at the top of the screen. This command is useful when the tableau is so large that it will not fit on the screen even in Map mode, so that # symbols appear on the screen. It is best to use this command with the cursor at a node which is below the point where the central branch splits and above the # symbol.

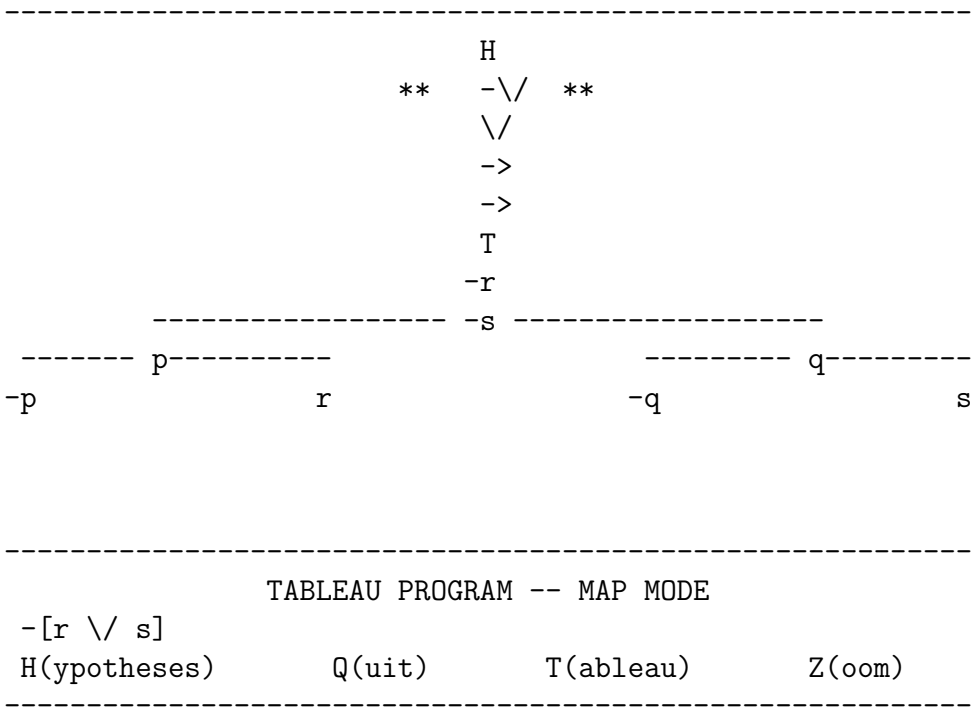


Figure 27: Map Mode

6.2 COMPLETE program documentation

This program is designed for classroom demonstrations of complete (or finished) tableaus in propositional logic. It makes tableaus with the property that every branch is either contradictory or is such that every node of the branch is used on the branch. It works like the TABLEAU program and uses .TBU files, but with the following differences. 1. There is no Hypothesis mode, only Tableau and Map modes. 2. The program starts by listing the .TBU files on the diskette in drive slot A. (These files must be prepared with the TABLEAU program). 3. Only propositional rules are recognized. Wffs beginning with quantifiers and atomic wffs are treated as propositional symbols. 4. The E(xtend) command uses the current wff to extend every noncontradictory branch through the current node. 5. Contradictory nodes are shown in red, and noncontradictory nodes which have been used are shown in cyan. When every node is either red or cyan, the tableau is complete. 6. There is no provision for monochrome display. 7. At least 360 K of memory is required.

6.3 PREDCALC program documentation

INTRODUCTION. This program demonstrates the rules of formation for formulas of first order predicate logic, and the corresponding inductive definition of the truth value of a formula. It works like a Hewlett- Packard (TM) (or reverse Polish notation) calculator, but operates on formulas of predicate logic instead of numbers. There are nine formulas in a stack and the last four formulas which were put on the stack are visible on the screen. The calculator has 24 "keys" which allow you to add an atomic formula to the bottom of the stack or to make a new formula by applying a logical connective or quantifier to formulas in the stack.

At all times you can switch back and forth between two modes, a TEXT mode and a GRAPHICS mode. In text mode the formulas in the stack are shown in the usual way. This mode corresponds to the syntax of predicate logic. By the graph of a formula we mean the set of valuations for which the formula is true. The graphics mode displays the graphs of the bottom four formulas in the stack. This mode corresponds to the semantics of predicate logic. Since the graphs have three dimensions, only the three variables x, y, and z are allowed in a formula.

The VIEW command gives an alternative, more detailed display of the graphs of the formulas.

EQUIPMENT NEEDED. The program runs on an IBM PC (TM) or compatible computer with at least 160K of memory and one disk drive. With a color- graphics card both text and graphics modes are available. Without this card, only the text mode is available, but the graphs can still be seen with the VIEW command.

STARTING THE PROGRAM. When you see the title screen, you have two options for starting out. The first option is to specify a universe size by typing a number between 1 and 8. The calculator will then appear on the screen in text mode. Initially, the stack contains nine false formulas designated by the f symbol.

The second option is to type the letter L (for LOAD), to load a problem or previous session from the diskette. A list of files which contain problems will appear on the screen. To get started you type the name of one of these files and press Return or Enter. The files have names up to eight characters long followed by the suffix ".PRD". You should not enter the suffix, only the name as it appears on the screen. The calculator will then appear on the screen, and the universe size and starting mode (Text or Graphics) will be determined by the problem you select.

You can also choose between a color graphics, monochrome graphics, or text only display at the title screen. If you choose the monochrome display, the graphics mode will be disabled. You may experiment to see which one works best on your screen.

A QUICK PREVIEW. Here is a short example which will give you an idea of what the program does. Press the keys shown within square brackets [] exactly as shown. Start out with the title screen for the PREDCALC program. Put your problem diskette in drive slot A.

[L] (A list of files will appear)

[preview][ENTER key] (The text screen will appear)

[LEFT ARROW key][z][x][ENTER key] ($z < x$ will appear in position 1)

[=][x][2][ENTER key] ($z < x$ in position 2 and x=2 in position 1)

[&][ENTER key] ($z < x$ & x=2 will appear in position 1)

[PgDn key][x][ENTER key] (The formula in position 1 now matches the goal)

[g][ENTER key] (The screen is now in Graphics mode)

[m][ENTER key] (You are now in memory mode)

[ENTER key][ENTER key][ENTER key][ENTER key] (Replay in Graphics mode)

[q][ENTER key] (The quitting screen appears)

[q] (You have quit the PREDCALC program)

GOALS. When you start the program by selecting a problem from the diskette, a Goal will be shown next to the stack. There are two types of problems, Text and Graphics problems.

A Text problem starts out in Text mode, and the goal is a formula which is displayed above the stack. The object of the problem is to match the goal formula in position one of the stack by using the calculator. In order to do this, you must begin with atomic formulas and build up to the goal formula through a parsing sequence.

A Graphics problem starts out in Graphics mode, and the goal is shown in the form of a graph. The object is to think of a formula which has the required graph and to get the formula into position one of the stack by using the calculator keys. This type of problem is more difficult and requires an understanding of the interpretation of quantifiers in a model.

No goal is shown if you start the program by selecting a universe size. You can use the calculator "keys" to build and look at formulas in either text or graphics mode.

THE CALCULATOR. When you begin the program with a text problem, a text screen will appear. At the top is the goal formula, then the bottom four formulas of the stack (initially all false), then an array of 24 "calculator keys" and a help window. In graphics mode you will see graphs of the goal formula and the four formulas in the stack, and a similar array of calculator keys. There is no room for a help window in graphics mode.

THE TIME COUNTER. The number 0 in the lower right corner of the calculator is a TIME COUNTER. Each time you carry out a calculator command which changes the formulas in the stack, the time counter increases by one.

MOVING WITHIN THE CALCULATOR KEYBOARD. To distinguish between the computer keyboard and the "calculator keys" in the screen picture, we shall call the "calculator keys" BOXES. The currently active box, in this case the Quit box, is enclosed in a double border (which is green in the computer display). The lower right part of the screen has a help window explaining what the currently active box does. There are two ways to change the active box. One way is to use the keys on the computer's numeric pad

to move to another box (the four arrow keys and the four corner keys move in the indicated direction). The other way is to press the first letter of the desired calculator box. For example, if you type N the active box will be the box labeled Not. Typing the $=, <, +, \&$, and * keys will also move you to the corresponding box. By moving within the calculator keyboard and reading the help messages, you can discover what all the calculator boxes do.

USING THE CALCULATOR COMMANDS. You CALL a calculator command by pushing the RETURN key when the box with the command is marked with the double green border. Here is a brief overview of the calculator commands. The commands in the first four columns cause changes in the formulas in the stack, and are counted by the time counter. The commands in the right two columns of the calculator are used for a variety of purposes which do not change the formulas in the stack and are not counted by the time counter.

Six of the boxes can be used to put an atomic formula on the stack: the four boxes in the left column and the boxes labeled ".=." and ".<."; you will usually begin a session with one of these. The periods represent argument places which must be replaced by variables or constants. As an example, when the ".=." box is active, you can enter the atomic formula "x=3" by typing x, then typing 3, and then typing RETURN. If you change your mind, you can cancel the variables or constants by pressing the Esc key instead of Return. The five boxes labeled "&", "Or", "$- >$", "$< - >$", and "Not" can be used to put a new formula in the stack by combining old formulas with logical connectives. The "All." and "Exi." boxes can be used to make a new formula by applying a quantifier to a formula in the stack. To call the quantifier commands you type one of the variables x, y, or z to replace the dot and then type RETURN. The "Dup" and "Pik." boxes are used to rearrange the formulas which are already in the stack.

Figures 28 and 29 are reproductions of two of the help messages which go with the calculator boxes.

Except for the Mem box, every help box in the first four columns has a table showing exactly how the command affects the formulas in the stack. For example, the "&" box will put the conjuction of the formulas from positions 1 and 2 in position 1, copy the formulas from positions 9 to 3 into postions 8 to 3, and leave postion 9 unchanged. In the boxes which have periods which are to be replaced by variables, the only variables accepted are x, y, and z. The random relation box labeled "R(...)" introduces a new predicate symbol

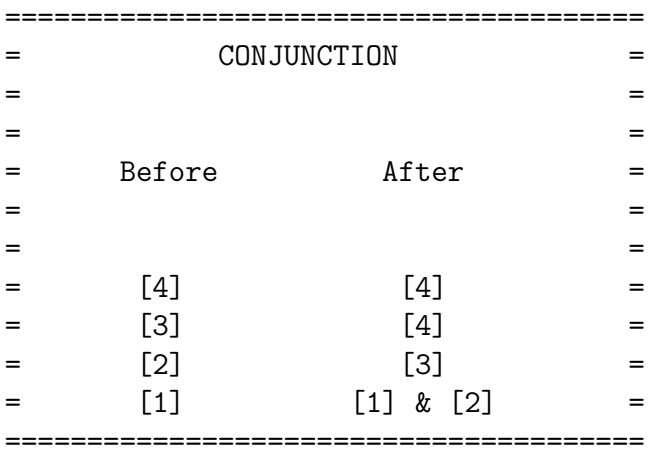


Figure 28: Help box for the & key

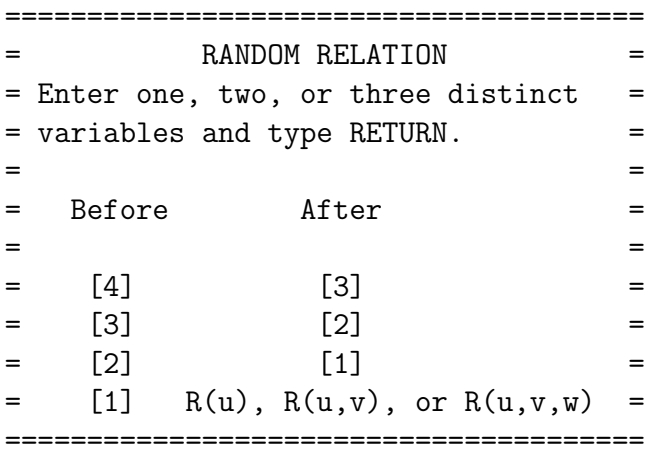


Figure 29: Help box for the R(...) key

with one, two, or three argument places each time it is called, starting with A. Its graph will be chosen randomly by the computer, with the variables you enter. The random function box ".=h(..)" introduces a new function symbol with one or two arguments each time it is called, starting with a. Again, its graph is chosen randomly with the variables you enter.

The "Mem" box is a toggle switch (like a light switch) which allows you to go back and forth between two modes of operation, Memory mode and Calculator mode. You are normally in calculator mode. When you switch to memory mode, the calculator reverts to its starting position at time 0. You can then replay your previous sequence of commands by pressing the Return key. You can continue pressing the Return key until the time counter reaches its previous count, or else go back to Calculator mode partway through the sequence by moving to another box and changing the sequence of commands. In Memory mode, the "Mem" box is replaced by the "Calc" box, which will cause an immediate jump back to the state at the end of the original command sequence.

Here is a detailed description of these commands for the last two columns of calculator boxes.

The "Grph" box is a toggle which switches the calculator between Text and Graphics modes. In graphics mode you will see the graphs of the four formulas in the stack and the goal formula. One of the main objects of the program is to allow you to see both the syntax of the formulas in text mode and the semantics in graphics mode. You are encouraged to switch back and forth between the two modes as often as possible. You can always do this without changing anything else. Each formula has, at most, the three variables x, y, and z, and its graph is a set of points in a cube whose side is the size of the universe. The cube has three axes labeled with the variables x, y, and z. All of the calculator boxes work in the same way in graphics mode as in text mode. In graphics mode, the "Grph" box is replaced by the "Text" box, which returns you to text mode.

The "Both" box lets you see the formulas and their graphs together.

The "View" box shows the graph of one formula at a time in an expanded form, and is useful for examining a graph in more detail. It lets you view (one at time) any of the nine formulas in the stack or the goal formula.

The "Quit" box can be used either to quit the program entirely, or to restart the program. It will put a menu on the screen which will be discussed later on.

The "Undo" box will Undo the last command. It cannot be used twice in a row.

The "Load" box can be used to load in a new problem or previous session from the diskette. It is the same as the Load option which is available when you start the program.

The "File" box is used to file your session in its current state on the diskette, so that it can be loaded in and replayed or revised at a later time, or handed in on the diskette as a completed assignment. To ERASE an unwanted .PRD file from the diskette, use the Quit box to start an empty session (no goal, time 0, no memory, any universe size) and then use the File box and type the name of the file you want to erase.

There is one extra command intended for instructors preparing problems for students, the "Goal" command. This command is called by holding the Ctrl key down and pressing the G key. If you call the Goal command when you are in Text mode, a problem file will be created with the formula currently in stack position one as a Text Goal. If you call the Goal command when you are in Graphics mode, a problem file will be created with the formula in stack position one as a Graphics goal.

ACHIEVING YOUR GOAL. When you succeed in your task of getting a copy of the goal formula into position 1 of the stack, the computer will reward you by replacing the word "Goal" with the word "Done". In a text problem, you must match the formula exactly. A different formula which has the same graph as the goal formula does not count. In a graphics problem, you will achieve the goal when you match the given graph in stack position one.

QUITTING OR RESTARTING THE PROGRAM. When you call the "Quit" command, a menu appears on the screen.

If there has been any change since the last time you filed your session, you will get a warning message, and have a chance to file it by pressing the F key. (If you leave the program without filing your session, all record of it is lost). The screen will also inform you if your current session has already been filed. As protection against accidentally quitting the program, you must press Q a second time to quit. There are three options for returning to the program. Pressing R will clear all the formulas from the stack and set the time counter to 0, but will keep the previous goal and universe size. Pressing S will completely restart the program and go back to the original opening screen. Pressing any other key will return the program to the position before

you called the "Quit" command.

6.4 BUILD program documentation

This program can be used to look for finite models of a set of sentences of predicate logic. It is based on the Finished Branch Lemma. Given a finite set of hypotheses and a finite universe, you try to extend the hypothesis set to a finished set of sentences with a constant symbol for each element of the universe. The program is similar to the TABLEAU program. The differences between the two programs are described here.

There are three modes.

HYPOTHESIS MODE builds a list of hypotheses and a universe set. The program will only allow sentences to be entered.

BUILD MODE builds a set of sentences. The program automatically adds witnesses for sentences like A & B and Åx A, and asks you to supply witnesses for sentences like A | B and É x A. It also keeps track of which sentences already have witnesses, and whether the current set is contradictory, finished, or neither.

MAP MODE displays the graphs of the predicates in the current potential model. Truth values which are not yet determined are indicated with question marks.

The program starts in Hypothesis mode. You can change from one mode to another with the commands B, H, and M. The command Q is used to quit the program. To protect against accidental quitting, the program asks you to type Q a second time to be sure you really meant to quit.

Although the following pages describe the program as it appears on a color display, it can also be run on a monochrome display.

HYPOTHESIS MODE

In this mode you can enter the size of the universe and a list of hypotheses. You can either type these in at the keyboard or load them from the diskette. The program will not let you change to another mode until it has been given a universe size and at least one hypothesis.

In addition to the commands in the TABLEAU program, there is one new command, U, which is used to choose the universe size. A number between 1 and 20 will be accepted as a universe size, and the elements of the universe will be the natural numbers from 0 to one less than the universe size.

The hypotheses must be sentences of predicate logic without function symbols or constant symbols.

BUILD MODE

In this mode you can build a set of sentences (which we shall call a branch), starting with the hypotheses. The sentences are displayed in a column. If there is a contradiction in the branch, that is, a pair of sentences of the form A, $\neg A$, then every sentence below the contradiction is shown in red. If there is no contradiction, every basic sentence and every sentence which has witnesses is shown in cyan, and all other sentences are shown in yellow. If every sentence is shown in cyan, the branch is finished. In this case, any model of the basic sentences is a model of the whole branch, and thus a model of the original set of hypotheses.

Your current location is the sentence which has the blinking cursor and blue background. The sentences in the branch are numbered to help you find your way around a large branch.

The current status of the branch is shown in the window at the bottom of the screen: FINISHED (all cyan), CONTRADICTORY (red terminal node), and UNFINISHED (with the number of the first yellow node) if the branch is neither finished nor contradictory.

The branch is built one step at a time. To extend the branch, you move the cursor to a sentence, and type E for Extend. What happens next depends on the type of sentence. If the current sentence is basic, you will be told that no extension can be made. If it has the form A & B, both A and B will be added to the branch. If it has the form A v B, you will be asked to choose either A or B, and your choice will be added to the branch. If it has the form Åx A, each of the sentences A(x//c) will be added where c is a constant in the universe. If it is of the form É x A, you will be asked to choose a number c between 0 and the largest number in the universe, and the sentence A(x//c) for your choice of c will be added to the branch.

MOVING WITHIN THE BRANCH. If the branch is too large, only part of the branch can be seen on the screen at one time. The cursor can be moved within the branch using the arrow keys in the following ways:

Up arrow : Move up one line.

Down arrow : Move down one line.

Home : Move to the top of the branch.

End : Move to the end of the branch.

Page Up : Move up one screen.

Page Down : Move down one screen.

COMMANDS IN BUILD MODE. The list of commands is shown in the window at the bottom of the screen.

A : Run on AUTOMATIC. The program will automatically extend the branch using all nodes from the current node to the end as much as possible. Nodes which require you to make a choice are left alone. You can stop the automatic process at any time by pressing any key.

E : EXTEND. The branch is extended using the current sentence, as explained in the preceding paragraph. When this command is invoked, the color of the current sentence becomes cyan unless it is already red.

F : FILE. Saves the universe size, the hypothesis list, and the branch in its present state into a file on the diskette. The computer asks you to type in the name of the file, in the form XXXXXXXX.BLD. (You don't enter the suffix ".BLD"; the computer will add it automatically). Illegal names are ignored, and you are warned if you try to use a name which is already on the diskette. To get back to the program without saving, just type RETURN without typing a file name.

H : Change to HYPOTHESIS mode.

K : KILL. This command erases everything below the current sentence, and is used to correct mistakes. Sentences which were added at the same time as the current sentence with the Extend command will not be erased.

M : Change to MAP mode.

Q : QUIT the program.

S : SPLIT screen. This command is useful when the branch gets so large that you cannot see both the current sentence and the new sentences which are added at the end of the branch on the same screen. When you invoke this command, the screen splits in two. The top half of the screen shows the part of the branch around the current sentence, and the bottom half of the screen shows the end of the branch. This lets you see both the current sentence and the new sentences at the end of the branch when you Extend the branch. If the screen is already split, this command will change back to the original single screen.

U : UNDO. This command undoes the last Kill or Extend command, and goes back to the previous position.

W : WHY. This command checks to see whether the current sentence has

a witness. It is a good idea to use this command before trying to extend the branch. It is possible that the current sentence is colored yellow but has a witness which appeared when some other sentence was used in an extension. If the current sentence is colored yellow but a witness is found with this command, the color is changed to cyan, telling you that no extension is needed. The command also tells you two things about the current sentence: (1) which sentence was used to add the current sentence to the branch, and (2) the first witness, if any, for the current sentence. The sentence used and the witness are displayed in the window at the bottom of the screen.

MAP MODE

This mode starts out with a one or two dimensional graph of the first predicate in the current "partially defined" model. If the branch has a basic sentence which determines a truth value, a T or F is shown in the graph, and otherwise a ? is shown. (If the branch is contradictory, the last truth value in the branch is shown). All the predicate symbols are listed in the window at the bottom of the screen.

The Finished Branch Lemma tells us that if the branch is finished, then any way of replacing the ?'s by either T's or F's will produce a model of the set of hypotheses.

Commands in Map mode:

B: Change to BUILD mode.

C: Next CONSTANT. If the current predicate symbol has more than two argument places, the graph of the predicate with a constant 0 in all but the first two places is shown first. C command will then show the graph for the next value of the constants. You can see the full graph by repeatedly pressing C until you get back to the starting position.

H: Change to HYPOTHESIS mode.

N: NEXT predicate. If there is more than one predicate symbol, this command shows the graph of the next predicate symbol.

Q: QUIT the Build program.

6.5 MODEL program documentation

In this program you can define and change predicates and functions in a finite model of predicate logic, and ask the program for the values of sentences and

terms. When you start the program, you are asked for a number between 1 and 32 for the size of the universe. Initially, the program will work with a vocabulary containing a constant for each element of the universe (numbers starting from 0), the binary predicates <=, <, >=, >, =, <>, and the binary functions +, -, *. The predicates will have the usual meaning, and the functions will be interpreted as addition, subtraction, and multiplication modulo the size of the universe. If you enter a sentence, the computer will tell you its truth value. If you enter a term, the computer will tell you its value as an element of the universe. You can introduce new relation or function symbols, or change the interpretations of old ones, by entering definitions with curly brackets and colons. The available commands are shown on the screen and are self-explanatory. The program has a help screen with examples showing how to enter sentences, terms, and definitions. This help screen is reproduced below.

If you type a wff or term the program returns its value in the model. If you type a definition the program modifies the model accordingly. Here are some definitions:

$q := \{x : x > 0\}$
$p := \{(x, y) : x < y$ OR EXIST $z[q(z)$ AND $z <> y]\}$
$f := \{(x, y) : x * y + x - y\}$
$* := \{(x, y) : x * y + x - y\}$
$p(1, 2)$:=FALSE
$f(1, 2) := 3$
Here are some wffs and terms:
ALL x EXIST y $x < y <->$ EXIST y ALL x $x < y$
ALL x [EXIST y $x <> y->$ EXIST $yy <> y$]
$2 + 3$
ALL x EXIST y [q(x) OR p(x,y)]

6.6 GNUMBER program documentation

INTRODUCTION

GNUMBER simulates a register machine, which is the basic tool in the study of computable functions. The title screen asks you to select either the SIMPLE or the ADVANCED form of GNUMBER. The simple form can be used to enter instructions and register values and watch a register

machine program run. The advanced form has additional features which let you manipulate Godel numbers of register machine programs and get a close look at register machine programs which refer to themselves. Programs which refer to themselves lead to the striking results of Godel showing that some problems are unsolvable.

The program works on an IBM PC or compatible computer with at least 256K of memory. With more memory, there will be room for longer registers. At the title screen you can select either a color or monochrome display. You may experiment to see which one looks best on your screen.

There are three modes of operation, the Instruction Editor, the Register Editor, and the Main Control Panel. GNUMBER starts out in the Instruction Editor. You can change from one mode to another with the commands I, R, and M. At any time the command Q can be used to quit the program. To protect against accidental quitting, the program asks you to type Q a second time to be sure you really meant to quit. GNUMBER can be run on either a monochrome or color display, but some of the information stands out more clearly on a color display.

The next few pages first explain what you can do in each mode with the simple form of GNUMBER. Then the additional features of the advanced form are described. If you are only using the simple form, you can skip the material on the advanced form.

A QUICK PREVIEW

Here is a short sample session which will give you an idea of what the GNUMBER program does. Begin with the title screen showing. Type each key within square brackets exactly as shown, and watch the screen.

[SPACE key] (You are now in the Simple Instruction Editor)
[z][1] (The Register machine instruction Z 1 appears)
[Return][S][1][0] (Instruction S 10 appears)
[Return][j][Return] (Instruction J 1 1 1 appears)
[r] (You are now in the Register Editor)
[4] (Register 1 now has value 4)
[m] (You are now in the Main Control Panel)
[s] (You have chosen to run at slow speed)
[SPACE bar] (Watch the program run. Notice registers 1 and 10)
[SPACE bar] (The register machine program stops)
[q][q] (You have quit the GNUMBER program)

INSTRUCTION EDITOR

You always start out in the Instruction Editor, and can get there from other modes with the I command. In the Instruction Editor, you can type in a register machine program, load a sample register machine program from the disk, or save a register machine program. There are places for 92 instructions, numbered from 0 to 91. Any instruction beyond 91 is assumed to be a Halt. The next instruction number is shown at the top of the screen. On the right side of the screen is a help window which has a list of the available register machine instructions and editor commands. The letters H,J,S,T,Z are used for register machine instructions, and the letters C,D,F,L,O,M,Q,R are used for editor commands.

Moving within the screen. The UP, DOWN, RIGHT, and LEFT arrow keys the spacebar, and the HOME, END, RETURN, and TAB keys can be used to move within the instruction editor.

Register machine instructions. The following register machine instructions can be entered in your programs. The table shows what each instruction does when r, s, and t are the numbers following the instruction letter and [r] is the number in register r.

INSTRUCTION		EFFECT
H	(Halt)	Stop.
Z r	(Zero)	[r] := 0.
S r	(Successor)	[r] := [r]+1.
T r s	(Transfer)	[s] := [r].
J r s t	(Jump)	if [r] = [s] then jump to instruction t.

Entering register machine instructions. When you start the GNUMBER program, there is a H command for Halt at every position. Any instruction can be changed by simply typing over it. When an instruction letter is typed at an instruction column, the correct number of places are filled in with question marks. You can then use the RIGHT and LEFT arrow keys or the spacebar to go to a number column and type the number you want. (The DEL key changes a number back to a question mark.) In the third place of the J command, an instruction number is needed, and the computer will let you type in any value from 0 to 92. At any other place, a register number is needed, and the computer will let you type in a value from 1 to 45. You can

also place “break points” in the column after the instruction letter, which will cause the register machine program to stop when it is being run. The ! key will insert or remove a brack point. The computer will ignore any attempt to enter an illegal instruction. When you are finished with the instruction, you may leave the line by typing an arrow key, RETURN, HOME, or END. Any remaining question marks will change to 1's.

Commands in the Instruction Editor.

Q : QUIT the Gnumber program.

M : Go to the MAIN Control Panel.

R : Go to the REGISTER Editor.

C : CLEAR all instructions to H, and set the time counter and next instruction number to zero.

D : DELETE the current instruction line, move all later lines up one, and adjusts all J (Jump) instructions accordingly.

O : OPEN a line. This command moves all instructions below the current line down one, adjusts all J (Jump) instructions accordingly, and writes an H in the current line. Use this command when you want to insert a new instruction at the current line.

F : FILE a register machine program. This command saves the current register machine program in a file on the disk. The computer will ask you to type in a file name of up to 8 letters and then press RETURN. The suffix .GN will automatically be added to the name you choose. You will get a warning if you type in a file name which is already used. You can get back to the Instruction Editor without saving by pressing RETURN without a file name.

You can ERASE an unwanted .GN file by clearing all instructions to H (with the Clear command), then invoking the File command, and then typing the name of the file which you want to erase.

L : LOAD a register machine program. In the bottom window of the screen you will see the message

LOAD A REGISTER MACHINE PROGRAM AT LINE nn

where nn is the current line of the cursor in the Instruction editor. The computer will show you a list of all files on the disk whose names have

the suffix .GN, and ask you to type in a file name and press RETURN. The register machine program described in the file will then be put into the instruction list, starting at the line nn. All old instructions from line nn to the end will be moved ahead to the end of the new program, and all jump instructions will be adjusted in the correct way. The next instruction number and time counter will be set to 0. You can get back to the Instruction Editor without loading a new program by pressing RETURN without a file name. After you load a register machine program, its name will be displayed at the top of the screen. The name will stay there until you change a program instruction, file a program, or load a new program. If the file name you type is not on the diskette, or if there is not enough room to load the new program starting at line nn, you will be informed by a message and will return to the Instruction Editor with no change.

This command can be used either to load an RM program by itself, or to load an RM program somewhere in the middle of an old program. To load a program by itself, first press Home to get to instruction line 0, then press C to clear out the old instruction list, and then press L. To load a new program in the middle or at the end of an old program, move the cursor to the line where you want the new program to begin and then press the L key.

U : UNDOES the most recent change in the instruction list. The register list is returned to what is was before the most recent entry of an instruction letter or number, or one of the commands C, D, L, or O. Use this instruction to recover if you accidentally press the wrong key.

Advanced Instruction Editor.

By means of a Godel numbering scheme, each natural number is also the code of a finite sequence of natural numbers. The Godel numbering scheme uses the even decimal positions (starting from 0 on the left) as markers to show where a new term begins, and uses the odd decimal positions for the digits of the terms in the sequence to be coded. A 2 marker means that a new term is beginning, and a 1 marker means that the old term is continuing. For example, the Godel number of the sequence

5034 6 217

is (with the original digits underlined)

$2\underline{5}1\underline{0}1\underline{3}1\underline{4}2\underline{6}2\underline{2}1\underline{1}7.$

This is a Godel number in standard form. In order to make every number the Godel number of some sequence, the initial marker can be any digit except 0, a marker > 2 is identified with a 2, a 0 marker is identified with a 1, and an extra digit at the end is ignored.

Two new 3-placed instructions, E and P, are available in the advanced form. Remember that [s] stands for the number in register s. The E command EXTRACTS the [s]-th term from the sequence coded by [r] and places it in register t. (All terms beyond the last term of the sequence are considered to be 0). The P command PUTS the number [r] into the [s]-th term of the sequence coded by register t. The effect of these commands may be summarized symbolically, where (r) denotes the sequence with Godel number [r].

```
INSTRUCTION                EFFECT
E r s t  (Extract)         [t] := the [s]-th term of (r).
P r s t  (Put)             The [s]-th term of (t) := [r].
```

REGISTER EDITOR

You can get to the Register Editor from other modes with the R command. In the Register Editor you can put numbers into the registers. There are 45 registers, numbered 1 through 45. The register editor displays the next instruction number and registers 1 through 15 on the right side of the screen, and instructions 0 through 45 on the left side of the screen. (The remaining instructions 46 through 91 still exist but are no longer visible on the screen.) GNUMBER starts with 0 in every register. The help window below the registers lists the available commands.

Moving within the Register Editor.

The PGDN and PGUP keys display the next or previous group of 15 registers, 1-15, 16-30, and 31-45. The UP and DOWN arrow keys move the cursor up and down one row, and the HOME key moves the cursor to register one. You can also get to the NEXT INSTRUCTION REGISTER (register 0) by going to register 1 and pressing the up arrow key.

Entering a number into a register.

A number is entered into a register by typing the digits 0,...,9 as usual. You can enter a number into the Next Instruction Register as well as the ordinary registers. The backspace key works in the usual way. When you are finished entering the number, type ENTER, an UP or DOWN arrow, HOME, PGUP, PGDN, or one of the commands I, M, or Q. You can enter up to 2,000 digits. While you are entering a number which is more than one line long, the screen shows you how many digits have scrolled off the left edge of the window.

Exploring a register.

After you or the computer finish entering a number, its total length (measured in digits) is shown at the extreme right of the screen. If a register contains more that a full line of digits, you can explore the contents of the register by using the RIGHT and LEFT arrow keys and the END key (which displays the last 39 digits). This will cause the number to scroll horizontally and be displayed in white. The ENTER, UP, DOWN, HOME, PGUP, and PGDN keys and the I, M, and Q commands will leave the register and behave in the usual way.

Register Editor Commands.

Q : QUIT this program.
C : CLEAR all registers (put a zero in every register).
I : Go to the INSTRUCTION Editor.
M : Go to the MAIN Control Panel.

Advanced Register Editor.

There are four new commands which involve Godel numbers. A Godel number is assigned to a register machine program in the following way. Each register machine instruction is a sequence consisting of a letter and from 0 to 3 numbers. The instruction letters H,Z,S,T,J,E,P are assigned the codes 1 through 7 respectively. This makes each register machine instruction a sequence of from 1 to 4 numbers, and this sequence is assigned its Godel number. The instruction list is considered to end one step past the last nonhalt instruction. The register machine program is a finite sequence of

instructions, which gives rise to a finite sequence of Godel numbers that in turn has a Godel number.

G : Put the GODEL number of the register machine program shown in the current instruction list into the current register. This command also sets the time counter and next instruction to 0, and leaves all other registers unchanged.

U : (UNGODEL) Put the register machine program whose Godel number is in the current register into the instruction list. (A term in the current register sequence which is not a code of an instruction is treated as the end of the instruction list). This command also sets the time counter and next instruction register to 0, and leaves all other registers unchanged.

S : Change to SEQUENCE display. This command causes any number which has more than 3 digits and is the Godel number of a sequence in standard form to be displayed as a sequence of numbers enclosed by parentheses and separated by commas, other numbers are still displayed in the normal way. When you explore a register containing a sequence, the RIGHT and LEFT arrow keys move to the beginning of the next or preceding term of the sequence, and the END key moves to the beginning of the last term of the sequence.

N : Change to NUMBER display. This command causes the numbers in all registers to be displayed in the usual way as ordinary numbers.

The last line in the help window tells you whether the Number or Sequence display is being used. The next to the last line in the help window displays more information about the current register. If the computer has enough memory, the first few registers will have room for 12,000 digits instead of 2,000 digits. The amount of room in the current register is reported. If you are exploring a register in a number display, the number of digits and the place of the first visible digit are reported. If you are exploring a register in a sequence display, the number of terms, the place of the first visible term, and the length (number of digits) of the first visible term are reported.

MAIN CONTROL PANEL

You can get to the Main Control Panel from other modes with the M command. The Main Control Panel is the place where you run a register machine program. It has a variety of commands which allow you to start and stop the register machine program and control its speed. While the register

machine program runs at slow or one-step speed, the next instruction label will be shown with a white background, and its motion will give an indication of what the program is doing. The register contents also keep changing, and the time counter at the top of the screen shows the number of instruction steps in the run. When running at fast speed, the register contents are hidden and the time is only shown in multiples of 100.

While the register machine program is running, it can be interrupted by pressing any key. You can go to the Instruction or Register Editor and make changes or explore the contents of a long register, and then return to the Main Control Panel and continue running the program.

The screen display of the Main Control Panel and the Register Editor are the same except for the help window at the bottom of the screen.

Main Control Panel Commands.

I : Go to the INSTRUCTION Editor.
R : Go to the REGISTER Editor.
SPACE : Run the current register machine program.
Q : QUIT the Gnumber program.
S : Make the register machine program run at SLOW speed.
F : Make the register machine program run at FAST speed.
O : Make the register machine program run ONE step at a time.
T : Set the TIME counter and next instruction number to zero.

A A Simple Proof.

We present a detailed proof of a simple fact so that the reader can study the logical structure of a simple proof. Note that certain keywords like *choose* and *assume* are italicized in the proof. Study how these words are used.

Example A.0.1 *Let* $X = \{x \in \mathbf{N} : x^2 + 7 < 6x\}$ *and* $Y = \{2, 3, 4\}$. *Then* $X = Y$.

Recall that for sets the equality $X = Y$ means that both $X \subset Y$ and $Y \subset X$. Thus are proof breaks up into two parts.

First we show that $Y \subset X$, i.e. that for all x, $x \in Y \Rightarrow x \in X$. *Choose* x. *Assume* $x \in Y$. Then either $x = 2$, or $x = 3$, or $x = 4$. If $x = 2$, then $x^2 + 7 = 11 < 12 = 6x$. If $x = 3$, then $x^2 + 7 = 16 < 18 = 6x$. If $x = 4$, then $x^2 + 7 = 23 < 24 = 6x$. In all three cases, $x \in \mathbf{N}$ and $x^2 + 7 < 6x$. *Therefore* $x \in X$. We have shown that every $x \in Y$ satisfies $x \in X$; *Therefore* $Y \subset X$.

Now we show $X \subset Y$, i.e. that for all x, $x \in Y \Rightarrow x \in X$. *Choose* $x \in X$. Then $x \in \mathbf{N}$ and $(x - 3 - \sqrt{2})(x - 3 + \sqrt{2}) = x^2 - 6x + 7 < 0$ so either $(x - 3 - \sqrt{2}) < 0 < (x - 3 + \sqrt{2})$ or $(x - 3 - \sqrt{2}) > 0 > (x - 3 + \sqrt{2})$. The latter case implies (subtract $x - 3$) that $-\sqrt{2} > \sqrt{2}$ which is false so the former case must hold. From $(x - 3 - \sqrt{2}) < 0 < (x - 3 + \sqrt{2})$ we conclude $3 - \sqrt{2} < x < 3 + \sqrt{2}$ and since $1 < 3 - \sqrt{2} < 2$, $4 < 3 + \sqrt{2} < 5$, and x is an integer, it follows that $x = 2$ or $x = 3$ or $x = 4$, i.e. that $x \in Y$. We have shown that every $x \in X$ satisfies $x \in Y$, i.e. that $X \subset Y$.

B A lemma on valuations.

Our purpose in this section is to give a detailed proof of the important fact that if two valuations v and w agree on every variable which occurs freely in $\mathbf{A}$, then

$$\mathcal{M}, v \models \mathbf{A} \text{ if and only if } \mathcal{M}, w \models \mathbf{A}.$$

This fact is obvious in view of the meaning of the notation $\mathcal{M}, v \models \mathbf{A}$ (viz. that $\mathbf{A}$ is true in the model $\mathcal{M}$ when the free variables of $\mathbf{A}$ are assigned values via v) but the proof we give is of interest both because it illustrates an important method (the method of induction on the structure of a wff) and because it justifies simplified notation which we introduced in the section 2.4.

We begin with an inductive definition of the set $FREE(\mathbf{A})$ of variables which occur freely in a wff $\mathbf{A}$. This definition agrees with the notion of free occurrence introduced earlier (in section 2.2) in that a variable $\mathbf{x}$ has a free occurrence in $\mathbf{A}$ if and only if $\mathbf{x} \in FREE(\mathbf{A})$.

Definition B.0.2 *The set $FREE(\mathbf{A})$ of variables which* **occurr freely** *in a wff $\mathbf{A}$, also called the set of* **free variables** *of $\mathbf{A}$, is defined inductively as follows:*

(basis) If $\mathbf{A}$ is an atomic wff, then $FREE(\mathbf{A})$ is the set of all variables which occur in $\mathbf{A}$.

($\neg$) For each wff $\mathbf{A}$,

$$FREE(\neg\mathbf{A}) = FREE(\mathbf{A}).$$

($\wedge, \vee, \Rightarrow, \Leftrightarrow$) For each binary connective $*$ (i.e. $*$ is one of $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$) and all wffs $\mathbf{A}$ and $\mathbf{B}$,

$$FREE([\mathbf{A} * \mathbf{B}]) = FREE(\mathbf{A}) \cup FREE(\mathbf{B}).$$

($\forall, \exists$) For each wff $\mathbf{A}$ and variable $\mathbf{x}$,

$$FREE(\forall\mathbf{x}\mathbf{A}) = FREE(\mathbf{A}) \setminus \{\mathbf{x}\}$$

$$FREE(\exists\mathbf{x}\mathbf{A}) = FREE(\mathbf{A}) \setminus \{\mathbf{x}\}$$

If v is a valuation in a model $\mathcal{M}$ and $\mathbf{A}$ is a wff, then $v|FREE(\mathbf{A})$ stands for the **restriction** of v to the set of variables which occur freely in $\mathbf{A}$. That is, $v|FREE)\mathbf{A})$ is the function $FREE(\mathbf{A}) \to U_{\mathcal{M}}$ which has the same values as v for each $\mathbf{x} \in FREE(\mathbf{A})$.

Theorem B.0.3 *Let $\mathcal{M}$ be a model and Å be a wff. Let v and w be valuations in $\mathcal{M}$. If*

$$v|FREE(\mathbf{A}) = w|FREE(\mathbf{A}) \tag{8}$$

then

$$\mathcal{M}, v \models \mathbf{A} \text{ if and only if } \mathcal{M}, w \models \mathbf{A}. \tag{9}$$

This theorem is often stated verbally as: $\mathcal{M}, v \models \mathbf{A}$ depends only on the values of v at the free variables of $\mathbf{A}$.
Proof: Let $S(\mathbf{A})$ be the statement we wish to prove, that for all valuations v and w in $\mathcal{M}$, if (8), then (9). The proof is by induction on the wff $\mathbf{A}$. The model $\mathcal{M}$ stays the same throughout the proof.
(basis) If $\mathbf{A}$ is an atomic wff, then $FREE(A)$ is the set of all variables which occur in $\mathbf{A}$, and by the rule (M:n) for truth values in predicate logic, $\mathcal{M}, v \models \mathbf{A}$ depends only on the values of v at variables which occur in $\mathbf{A}$. Thus $S(\mathbf{A})$ is true when $\mathbf{A}$ is atomic.
($\neg$) Assume $S(\mathbf{A})$. Suppose (8) holds for $\neg\mathbf{A}$:

$$v|FREE(\neg\mathbf{A}) = w|FREE(\neg\mathbf{A}).$$

Since $FREE(\neg\mathbf{A}) = FREE(\mathbf{A})$, (8) holds for $\mathbf{A}$:

$$v|FREE(\mathbf{A}) = w|FREE(\mathbf{A}).$$

Now by $S(\mathbf{A})$, (9) holds for $\mathbf{A}$:

$$\mathcal{M}, v \models \mathbf{A} \text{ if and only if} \mathcal{M}, w \models \mathbf{A}$$

so by the rule (M:$\neg$), (9) holds for $\neg\mathbf{A}$:

$$\mathcal{M}, v \models \neg\mathbf{A} \text{ if and only if} \mathcal{M}, w \models \neg\mathbf{A}.$$

($\wedge, \vee, \Rightarrow, \Leftrightarrow$) Assume $S(\mathbf{A})$ and $S(\mathbf{B})$. Suppose (8) holds for $\mathbf{A} * \mathbf{B}$ where $*$ is one of binary connectives $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$:

$$v|FREE(\mathbf{A} * \mathbf{B}) = w|FREE(\mathbf{A} * \mathbf{B}).$$

Since $FREE(\mathbf{A} * \mathbf{B}) = FREE(\mathbf{A}) \cup FREE(\mathbf{B})$, (8) holds for both $\mathbf{A}$ and $\mathbf{B}$:

$$v|FREE(\mathbf{A}) = w|FREE(\mathbf{A}) \text{ and } v|FREE(\mathbf{B}) = w|FREE(\mathbf{B})$$

Now by $S(\mathbf{A})$, (9) holds for $\mathbf{A}$:

$$\mathcal{M}, v \models \mathbf{A} \text{ if and only if} \mathcal{M}, w \models \mathbf{A}$$

and by $S(\mathbf{B})$ (9) holds for $\mathbf{B}$:

$$\mathcal{M}, v \models \mathbf{B} \text{ if and only if} \mathcal{M}, w \models \mathbf{B}$$

so by the rule (M:$*$), (9) holds for $\mathbf{A} * \mathbf{B}$:

$$\mathcal{M}, v \models \mathbf{A} * \mathbf{B} \text{ if and only if} \mathcal{M}, w \models \mathbf{A} * \mathbf{B}.$$

($\exists$) Assume $S(\mathbf{A})$ and that (8) holds for $\exists \mathbf{x}\mathbf{A}$:

$$v|FREE(\exists \mathbf{x}\mathbf{A}) = w|FREE(\exists \mathbf{x}\mathbf{A}.$$

We must show that (9) holds for $\exists \mathbf{x}\mathbf{A}$.

Assume that

$$\mathcal{M}, v \models \exists \mathbf{x}\mathbf{A}.$$

Then there is a valuation $v' \in VAL(v, \mathbf{x})$ such that $\mathcal{M}, v' \models \exists \mathbf{x}\mathbf{A})$. Let w' be the valuation obtained from w by changing the value at $\mathbf{x}$ to $v'(x)$, that is, $w' \in VAL(w, \mathbf{x})$ and $w'(\mathbf{x}) = v'(\mathbf{x})$. Since

$$FREE(\exists \mathbf{A}) = FREE(\mathbf{A}) \setminus \{\mathbf{x}\}$$

and v agrees with w on $FREE(\exists \mathbf{x}\mathbf{A})$, v' agrees with w' on $FREE(\mathbf{A})$:

$$v'|FREE(\mathbf{A})\ w'|FREE(\mathbf{A}).$$

Then by $S(\mathbf{A})$ we obtain $\mathcal{M}, w' \models \mathbf{A}$ so by the rule (M:$\exists$)

$$\mathcal{M}, w \models \exists \mathbf{x}\mathbf{A}.$$

Since we proved $\mathcal{M}, w \models \exists \mathbf{x}\mathbf{A}$ from the assumption $\mathcal{M}, v \models \exists \mathbf{x}\mathbf{A}$ we have proven that $\mathcal{M}, v \models \exists \mathbf{x}\mathbf{A}$ implies $\mathcal{M}, w \models \exists \mathbf{x}\mathbf{A}$. Reversing the roles of v and w shows that $\mathcal{M}, v \models \exists \mathbf{x}\mathbf{A}$ implies $\mathcal{M}, w \models \exists \mathbf{x}\mathbf{A}$. Thus

$$\mathcal{M}, v \models \exists \mathbf{x}\mathbf{A} \text{ if and only if } \mathcal{M}, w \models \exists \mathbf{x}\mathbf{A}$$

as required.

($\forall$) Assume $S(\mathbf{A})$ and that (8) holds for $\forall \mathbf{x}\mathbf{A}$:

$$v|FREE(\forall \mathbf{x}\mathbf{A}) = w|FREE(\forall \mathbf{x}\mathbf{A}).$$

We must show that (9) holds for $\forall \mathbf{x}\mathbf{A}$.

Assume that

$$\mathcal{M}, v \models \forall \mathbf{x}\mathbf{A}.$$

Then for every valuation $v' \in VAL(v, \mathbf{x})$ we have $\mathcal{M}, v' \models \exists \mathbf{x}\mathbf{A}$. Choose $w' \in VAL(w, \mathbf{x})$ and let v' be obtained from v by changing the value at $\mathbf{x}$ to $w'(x)$. Then $v' \in VAL(v, \mathbf{x})$ and $v'(\mathbf{x}) = w'(\mathbf{x})$. Since

$$FREE(\forall \mathbf{A}) = FREE(\mathbf{A}) \setminus \{\mathbf{x}\}$$

and v agrees with w on $FREE(\forall \mathbf{x}\mathbf{A})$, v' agrees with w' on $FREE(\mathbf{A})$:

$$v'|FREE(\mathbf{A}) = w'|FREE(\mathbf{A}).$$

Then by $S(\mathbf{A})$ we obtain $\mathcal{M}, w' \models \mathbf{A}$. As $w' \in VAL(w, \mathbf{x})$ was arbitrary the (M:$\forall$) rule yields

$$\mathcal{M}, w \models \forall \mathbf{x}\mathbf{A}.$$

Since we proved $\mathcal{M}, w \models \forall \mathbf{x}\mathbf{A}$ from the assumption $\mathcal{M}, v \models \forall \mathbf{x}\mathbf{A}$ we have proven that $\mathcal{M}, v \models \forall \mathbf{x}\mathbf{A}$ implies $\mathcal{M}, w \models \forall \mathbf{x}\mathbf{A}$. Reversing the roles of v and w shows that $\mathcal{M}, v \models \forall \mathbf{x}\mathbf{A}$ implies $\mathcal{M}, w \models \forall \mathbf{x}\mathbf{A}$. Thus

$$\mathcal{M}, v \models \exists \mathbf{x}\mathbf{A} \text{ if and only if } \mathcal{M}, w \models \exists \mathbf{x}\mathbf{A}$$

as required.

This completes the proof of the theorem. The proof is really quite routine (and presented in far more detail than is customary) for it consists merely of unraveling the definitions. Nonetheless, the reader should examine its structure carefully, for this kind of of argument is quit common in mathematical logic.

What has this to do with section 2.4? There we said that the meaning of

$$\mathcal{M} \models \exists y P(3, y)$$

is

$$\mathcal{M}, v \models \exists y P(x, y) \text{ where } v(x) = 3.$$

The careful reader may ask for more precision here. Exactly which v is to be used? There are many valuations which satisfy the condition that $v(x) = 3$ since the value of v on the variables other than x is unspecified. The point is that it doesn't matter. For

$$FREE(\exists y P(x, y)) = \{x\}$$

so that according to theorem B.0.3 we have

$$\mathcal{M}, v \models \exists y P(x,y) \text{ if and only if } \mathcal{M}, w \models \exists y P(x,y)$$

whenever v and w are valuations with $v(x) = 3 = w(x)$. The device we have used is a common one in mathematical exposition; one makes a definition which appears to depend on a choice (in this case the choice is of a valuation satisfying $v(x) = 3$) and then shows that the definition is independent of the choice.

Finally in section 2.3 we introduced the notation $\mathcal{M} \models \mathbf{A}$ when $\mathbf{A}$ is a sentence (that is a wff with no free variables: $FREE(\mathbf{A}) = \emptyset$). This is justified by

Corollary B.0.4 *Let* $\mathbf{A}$ *be a sentence and* $\mathcal{M}$ *be a model for predicate logic. Then if* $\mathcal{M}, v \models \mathbf{A}$ *for some valuation* v*, then* $\mathcal{M}, v \models \mathbf{A}$ *for every valuation* v*. In other words, either* $\mathcal{M}, v \models \mathbf{A}$ *for every valuation* v *or* $\mathcal{M}, v \models \neg\mathbf{A}$ *for every valuation* v*.*

C Summary of Syntax Rules

A *vocabulary* is a disjoint union $\mathcal{P} \cup \mathcal{F}$ of disjoint unions

$$\mathcal{P} = \bigcup_{n=0}^{\infty} \mathcal{P}_n, \qquad \mathcal{F} = \bigcup_{n=0}^{\infty} \mathcal{F}_n.$$

The elements of $\mathcal{P}_n$ are called n-ary *predicate symbols* (*proposition symbols* for $n = 0$) and the elements of $\mathcal{F}_n$ are called n-ary *operation symbols* (*constants* for $n = 0$). There is a set of symbols VAR whose elements are called *variables.*

The set $TERM(\mathcal{F})$ of all *terms* constructed using $\mathcal{F}$ is smallest set of strings satisfying the following rules:

(T: VAR**)** $VAR \subset TERM(\mathcal{F})$;

(T:$\mathcal{F}_0$**)** $\mathcal{F}_0 \subset TERM(\mathcal{F})$;

(T:$\mathcal{F}_n$**)** $\mathbf{f} \in \mathcal{F}_n,\ \tau_\mathbf{1}, \tau_\mathbf{2}, \ldots \tau_\mathbf{n} \in TERM(\mathcal{F}) \Longrightarrow \mathbf{f}(\tau_\mathbf{1}, \tau_\mathbf{2}, \ldots \tau_\mathbf{n}) \in TERM(\mathcal{F})$.

The set $WFF(\mathcal{P}, \mathcal{F})$ is the smallest set of strings satisfying

$(\mathbf{W}{:}\mathcal{P})$ $\mathcal{P}_0 \subset \mathit{WFF}(\mathcal{P}_0)$;

$(\mathbf{W}{:}\mathcal{P}_n)$ $\mathbf{p} \in \mathcal{P}_n,\ \tau_\mathbf{1}, \tau_\mathbf{2}, \ldots,\ \tau_\mathbf{n} \in \mathit{TERM}(\mathcal{F}) \Longrightarrow \mathbf{p}(\tau_\mathbf{1}, \tau_\mathbf{2}, \ldots, \tau_\mathbf{n}) \in \mathit{WFF}(\mathcal{P})$;

$(\mathbf{W}{:}\neg)$ $\mathbf{A} \in \mathit{WFF}(\mathcal{P}) \Longrightarrow \neg\mathbf{A} \in \mathit{WFF}(\mathcal{P})$;

$(\mathbf{W}{:}\wedge)$ $\mathbf{A}, \mathbf{B} \in \mathit{WFF}(\mathcal{P}) \Longrightarrow [\mathbf{A} \wedge \mathbf{B}] \in \mathit{WFF}(\mathcal{P})$;

$(\mathbf{W}{:}\vee)$ $\mathbf{A}, \mathbf{B} \in \mathit{WFF}(\mathcal{P}) \Longrightarrow [\mathbf{A} \vee \mathbf{B}] \in \mathit{WFF}(\mathcal{P})$;

$(\mathbf{W}{:}\Rightarrow)$ $\mathbf{A}, \mathbf{B} \in \mathit{WFF}(\mathcal{P}) \Longrightarrow [\mathbf{A} \Rightarrow \mathbf{B}] \in \mathit{WFF}(\mathcal{P})$;

$(\mathbf{W}{:}\Leftrightarrow)$ $\mathbf{A}, \mathbf{B} \in \mathit{WFF}(\mathcal{P}) \Longrightarrow [\mathbf{A} \Leftrightarrow \mathbf{B}] \in \mathit{WFF}(\mathcal{P})$;

$(\mathbf{W}{:}\forall)$ $\mathbf{A} \in \mathit{WFF}(\mathcal{P}),\ \mathbf{x} \in \mathit{VAR} \Longrightarrow \forall\mathbf{x}\mathbf{A} \in \mathit{WFF}(\mathcal{P})$;

$(\mathbf{W}{:}\exists)$ $\mathbf{A} \in \mathit{WFF}(\mathcal{P}),\ \mathbf{x} \in \mathit{VAR} \Longrightarrow \exists\mathbf{x}\mathbf{A} \in \mathit{WFF}(\mathcal{P})$;

The set $\mathit{FREE}(\tau)$ of variables which occur in the term τ is defined inductively by:

(VAR) $\mathbf{x} \in \mathit{VAR} \Longrightarrow \mathit{FREE}(\mathbf{x}) = \{\mathbf{x}\}$;

$(\mathcal{F}_0)$ $c \in \mathcal{F}_0 \Longrightarrow \mathit{FREE}(c) = \emptyset$;

$(\mathcal{F}_n)$ $\mathit{FREE}(\mathbf{f}(\tau_\mathbf{1}, \ldots, \tau_\mathbf{n})) = \mathit{FREE}(\tau_\mathbf{1}) \cup \ldots \cup \mathit{FREE}(\tau_\mathbf{n})$.

The set $\mathit{FREE}(\mathbf{A})$ of variables which occur freely in the wff $\mathbf{A}$ is defined inductively by:

$(\mathcal{P}_0)$ $\mathit{FREE}(\mathbf{p}) = \emptyset$;

$(\mathcal{P}_n)$ $\mathit{FREE}(p(\tau_\mathbf{1}, \ldots, \tau_\mathbf{n})) = \mathit{FREE}(\tau_\mathbf{1}) \cup \ldots \cup \mathit{FREE}(\tau_\mathbf{n}))$;

$(\neg)$ $\mathit{FREE}(\neg\mathbf{A}) = \mathit{FREE}(\mathbf{A})$;

$(\wedge, \vee, \Rightarrow, \Leftrightarrow)$ $\mathit{FREE}([\mathbf{A} * \mathbf{B}]) = \mathit{FREE}(\mathbf{A}) \cup \mathit{FREE}(\mathbf{B})$;

$(\forall, \exists)$ $\mathit{FREE}(\mathbf{Q}\mathbf{x}\mathbf{A}) = \mathit{FREE}(\mathbf{A}) \setminus \{\mathbf{x}\}$.

The condition that a term τ is *free for* a variable $\mathbf{x}$ in a wff $\mathbf{A}$ is defined inductively as follows:

($\mathcal{P}_0$) τ is free for $\mathbf{x}$ in $\mathbf{p} \in \mathcal{P}_0$;

($\mathcal{P}_n$) τ is free for $\mathbf{x}$ in $p(\tau_\mathbf{1}, \tau_\mathbf{2}, \ldots, \tau_\mathbf{n})$;

($\neg$) τ is free for $\mathbf{x}$ in $\neg\mathbf{A}$ iff τ is free for $\mathbf{x}$ in $\mathbf{A}$.

($\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$) τ is free for $\mathbf{x}$ in $\mathbf{A} * \mathbf{B}$ iff τ is free for $\mathbf{x}$ in $\mathbf{A}$ and τ is free for $\mathbf{x}$ in $\mathbf{B}$.

($\forall$, $\exists$) τ is free for $\mathbf{x}$ in $\mathbf{Qy A}$ iff

either $\mathbf{x} \neq \mathbf{y}$ and τ is free for $\mathbf{x}$ in $\mathbf{A}$

or else $\mathbf{x} = \mathbf{y}$

The term $\tau(\mathbf{x}//\sigma)$ which results from the term τ by substituting the term σ for the variable $\mathbf{x}$ is defined inductively by:

(**S:***VAR*) If τ is a variable, then $\tau(\mathbf{x}//\sigma)$ is σ if τ is the variable $\mathbf{x}$ and is τ if τ is some other variable;

(**S:**$\mathcal{F}_0$) If τ is $\mathbf{f}(\tau_\mathbf{1}, \tau_\mathbf{2}, \ldots, \tau_\mathbf{n})$ then $\tau(\mathbf{x}//\sigma)$ is $\mathbf{f}(\tau_\mathbf{1}(\mathbf{x}//\sigma), \tau_\mathbf{2}(\mathbf{x}//\sigma), \ldots, \tau_\mathbf{n}(\mathbf{x}//\sigma))$

The wff $\mathbf{C}(\mathbf{x}//\sigma)$ which results from the wff $\mathbf{C}$ by substituting the term σ which is free for the variable $\mathbf{x}$ in $\mathbf{C}$ is defined inductively by:

(**S:**$\mathcal{P}_0$) If $\mathbf{C}$ is a proposition symbol $\mathbf{p} \in \mathcal{P}_0$, then $\mathbf{C}(\mathbf{x}//\sigma)$ is $\mathbf{p}$.

(**S:**$\mathcal{P}_n$) If $\mathbf{C}$ is $\mathbf{p}(\tau_\mathbf{1}, \tau_\mathbf{2}, \ldots, \tau_\mathbf{n})$, then $\mathbf{C}(\mathbf{x}//\sigma)$ is $\mathbf{p}(\sigma_\mathbf{1}(\mathbf{x}//\sigma), \tau_\mathbf{2}(\mathbf{x}//\sigma), \ldots, \tau_\mathbf{n}(\mathbf{x}//\sigma))$.

(**S:**$\neg$) If $\mathbf{C}$ is of form $\neg\mathbf{A}$, then $\mathbf{C}(\mathbf{x}//\sigma)$ is $\neg\mathbf{A}(\mathbf{x}//\sigma)$.

(**S:**$\wedge, \vee, \Rightarrow, \Leftrightarrow$) If $\mathbf{C}$ is of form $\mathbf{A} * \mathbf{B}$, then $\mathbf{C}(\mathbf{x}//\sigma)$ is $\mathbf{A}(\mathbf{x}//\sigma) * \mathbf{B}(\mathbf{x}//\sigma)$.

(**S:**$\forall, \exists$) If $\mathbf{C}$ is of one of the forms $\forall\mathbf{yA}$ or $\exists\mathbf{yA}$ and if $\mathbf{y} \neq \mathbf{x}$, then $\mathbf{C}(\mathbf{x}//\sigma)$ is respectively $\forall\mathbf{zA}(\mathbf{x}//\sigma)$ or $\exists\mathbf{zA}(\mathbf{x}//\sigma)$.

(**S:**$\forall\exists$) If $\mathbf{C}$ is of one of the forms $\forall\mathbf{xA}$ or $\exists\mathbf{xA}$ then $\mathbf{C}(\mathbf{x}//\sigma)$ is just $\mathbf{C}$.

D Summary of Tableaus.

A *tree* $\mathbf{T}$ is a system consisting of a set of points called the *nodes* of the tree, a distinguished node $r_{\mathbf{T}}$ called the *root* of the tree, and a function π, or $\pi_{\mathbf{T}}$, which assigns to each node t distinct from the root another node $\pi(t)$ called the *parent* of t; it is further required that for each node t the sequence of nodes

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

defined inductively by

$$\pi^0(t) = t$$

and

$$\pi^{k+1}(t) = \pi(\pi^k(t))$$

terminates for some n at the root:

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

A node of the tree for which $\pi^{-1}(t) = \emptyset$ is called *terminal* and a sequence

$$\mathbf{\Gamma} = (t, \pi(t), \pi^2(t), \ldots, r_{\mathbf{T}})$$

starting at a terminal node t is called a *branch.*

A *labeled tree* is a function Φ defined on a tree $\mathbf{T}$ which assigns a wff $\Phi(t)$ to each nonroot node t and which assigns a set of wffs $\mathbf{H} = \Phi(r_{\mathbf{T}})$ (called the *hypothesis set* of the tableau) to the root $r_{\mathbf{T}}$. For each node t let $\Phi^*(t)$ be defined by

$$\Phi^*(t) = \{\Phi(t)\} \cup \{\Phi(\pi(t))\} \cup \{\Phi(\pi^2(t))\} \cup \ldots.$$

The definition entails that

$$\mathbf{H} = \Phi^*(r_{\mathbf{T}}) \subset \Phi^*(t)$$

for every node t; in fact, $\Phi^*(\pi(t)) \subset \Phi^*(t)$. When the node is terminal we write $\Phi^*(\mathbf{\Gamma})$ instead of $\Phi^*(t)$ where t is the terminal node of the branch $\mathbf{\Gamma}$.

A labeled tree $(T, \mathbf{H}, \Phi)$ is a *tableau* iff at each nonterminal node t of T one of the following conditions holds:

$\boxed{\neg\neg}$ $\Phi(w) = \mathbf{A}$ where $\neg\neg\mathbf{A} \in \Phi^*(t)$;

$\boxed{\wedge}$ $\Phi(w) = \mathbf{A}$ and $\Phi(w') = \mathbf{B}$ where $[\mathbf{A} \wedge \mathbf{B}] \in \Phi^*(t)$;

$\boxed{\neg\wedge}$ $\Phi(u) = \neg\mathbf{A}$ and $\Phi(v) = \neg\mathbf{B}$ where $\neg[\mathbf{A} \wedge \mathbf{B}] \in \Phi^*(t)$;

$\boxed{\vee}$ $\Phi(u) = \mathbf{A}$ and $\Phi(v) = \mathbf{B}$ where $[\mathbf{A} \vee \mathbf{B}] \in \Phi^*(t)$;

$\boxed{\neg\vee}$ $\Phi(w) = \neg\mathbf{A}$ and $\Phi(w') = \neg\mathbf{B}$ where $\neg[\mathbf{A} \vee \mathbf{B}] \in \Phi^*(t)$;

$\boxed{\Rightarrow}$ $\Phi(u) = \neg\mathbf{A}$ and $\Phi(v) = \mathbf{B}$ where $[\mathbf{A} \Rightarrow \mathbf{B}] \in \Phi^*(t)$;

$\boxed{\neg\Rightarrow}$ $\Phi(w) = \mathbf{A}$ and $\Phi(w') = \neg\mathbf{B}$ where $\neg[\mathbf{A} \Rightarrow \mathbf{B}] \in \Phi^*(t)$;

$\boxed{\Leftrightarrow}$ $\Phi(u) = \mathbf{A} \wedge \mathbf{B}$ and $\Phi(v) = \neg\mathbf{A} \wedge \neg\mathbf{B}$ where $[\mathbf{A} \Leftrightarrow \mathbf{B}] \in \Phi^*(t)$;

$\boxed{\neg\Leftrightarrow}$ $\Phi(u) = \mathbf{A} \wedge \neg\mathbf{B}$, and $\Phi(v) = \neg\mathbf{A} \wedge \mathbf{B}$ where $\neg[\mathbf{A} \Leftrightarrow \mathbf{B}] \in \Phi^*(t)$;

$\boxed{\forall}$ $\Phi(w) = \mathbf{A}(\mathbf{x}//\tau)$ where $\forall\mathbf{x}\mathbf{A} \in \Phi^*(t)$;

$\boxed{\neg\forall}$ $\Phi(w) = \neg\mathbf{A}(\mathbf{x}//\mathbf{z})$ where $\neg\forall\mathbf{x}\mathbf{A} \in \Phi^*(t)$;

$\boxed{\exists}$ $\Phi(w) = \mathbf{A}(\mathbf{x}//\mathbf{z})$ where $\exists\mathbf{x}\mathbf{A} \in \Phi^*(t)$;

$\boxed{\neg\exists}$ $\Phi(w) = \mathbf{A}(\mathbf{x}//\tau)$ where $\neg\exists\mathbf{x}\mathbf{A} \in \Phi^*(t)$;

$\boxed{=1}$ $\Phi(w) = \mathbf{p}(\ldots\tau\ldots)$ where $\tau = \sigma$, $\mathbf{p}(\ldots\sigma\ldots) \in \mathbf{\Phi}^*(\mathbf{t})$

$\boxed{=2}$ $\Phi(w) = \mathbf{p}(\ldots\tau\ldots)$ where $\sigma = \tau$, $\mathbf{p}(\ldots\sigma\ldots) \in \mathbf{\Phi}^*(\mathbf{t})$

$\boxed{=1\neg}\Phi(w) = \neg\mathbf{p}(\ldots\tau\ldots)$ where $\tau = \sigma$, $\neg\mathbf{p}(\ldots\sigma\ldots) \in \mathbf{\Phi}^*(\mathbf{t})$

$\boxed{=2\neg}\Phi(w) = \neg\mathbf{p}(\ldots\tau\ldots)$ where $\sigma = \tau$, $\neg\mathbf{p}(\ldots\sigma\ldots) \in \mathbf{\Phi}^*(\mathbf{t})$

In items $\boxed{\neg\neg}$, $\boxed{\forall}$, $\boxed{\neg\forall}$, $\boxed{\exists}$, $\boxed{\neg\exists}$, $\boxed{=1}$, $\boxed{=2}$, $\boxed{=1\neg}$, $\boxed{=2\neg}$

$$\Phi^{-1}(t) = \{w\}.$$

In items $\boxed{\wedge}$, $\boxed{\neg\vee}$, and $\boxed{\neg\Rightarrow}$,

$$\Phi^{-1}(t) = \{w\},\ \Phi^{-1}(w) = \{w'\}.$$

In items $\boxed{\vee}$, $\boxed{\neg\wedge}$, $\boxed{\Rightarrow}$, $\boxed{\Leftrightarrow}$, and $\boxed{\neg\Leftrightarrow}$

$$\Phi^{-1}(t) = \{u, v\}.$$

In items $\boxed{\forall}$ and $\boxed{\neg\exists}$ the term τ is free for $\mathbf{x}$.

In items $\boxed{\neg\forall}$ and $\boxed{\exists}$ the variable $\mathbf{z}$ has no occurrence in $\Phi^*(t)$.

In items $\boxed{=1}$, $\boxed{=2}$, $\boxed{=1\neg}$, $\boxed{=2\neg}$ the wffs $\mathbf{p}(\ldots\tau\ldots)$ and $\mathbf{p}(\ldots\sigma\ldots)$ are atomic wffs and the latter is obtained from the former by substituting the term σ for the one occurrence of the term τ.

E Finished Sets.

We call a set $\mathbf{\Delta}$ of wffs full predicate logic *contradictory* iff either (1) a pair of wffs of form $\mathbf{A}$, $\neg\mathbf{A}$ both in $\mathbf{\Delta}$ or (2) $\mathbf{\Delta}$ contains a wff of form $\tau \neq \tau$.

A wff is *atomic* iff it is either a proposition symbol or else of form $\mathbf{p}(\tau_\mathbf{1}, \tau_\mathbf{2}, \ldots, \tau_\mathbf{n})$ where $\mathbf{p}$ is a predicate symbol and $\tau_\mathbf{1}, \tau_\mathbf{2}, \ldots, \tau_\mathbf{n}$ are terms. A wff is *basic* iff it is either atomic or else of form $\neg\mathbf{B}$ where $\mathbf{B}$ is atomic.

Let τ and σ be terms and $\mathbf{A}$ and $\mathbf{B}$ be wffs. We say that $\mathbf{B}$ is **obtained from A** *by an equality substitution on* $\tau = \sigma$ iff both the wffs $\mathbf{A}$ and $\mathbf{B}$ are basic and $\mathbf{B}$ results from $\mathbf{A}$ either by replacing some occurrence of τ by σ or else by replacing some occurrence of σ by τ. (It is not necessary that the occurrence being subsituted for be an argument; it may be part of an argument.) We say that a set $\mathbf{\Delta}$ of wffs is *closed under equality substitution* iff whenever $[\tau = \sigma] \in \mathbf{\Delta}$, $\mathbf{A} \in \mathbf{\Delta}$ and $\mathbf{B}$ is obtained from $\mathbf{A}$ by an equality substitution on $\tau = \sigma$, we have that $\mathbf{B} \in \mathbf{\Delta}$ and for every variable free term τ we have that $\tau = \sigma$ is in $\mathbf{\Delta}$.

Let $U_{\mathbf{\Delta}}$ denote the set of terms τ such that τ has no variables (i.e. $FREE(\tau) = \emptyset$) and τ has an occurrence in some wff of $\mathbf{\Delta}$.

We call $\mathbf{\Delta}$ *finished* iff $\mathbf{\Delta}$ is not contradictory, closed under equality substitution, and for each wff $\mathbf{C} \in \mathbf{\Delta}$ either $\mathbf{C}$ is a basic wff or else one of the following is true:

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

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

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

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

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

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

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

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

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

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

$[\neg\forall]$ $\mathbf{C}$ has form $\neg\forall\mathbf{x}\mathbf{A}$ where $\neg\mathbf{A}(\mathbf{x}//\sigma) \in \mathbf{\Delta}$ for some term $\sigma \in \mathbf{U_{\Delta}}$;

[$\exists$] $\mathbf{C}$ has form $\exists\mathbf{A}$ where $\mathbf{A}(\mathbf{x}//\sigma) \in \mathbf{\Delta}$ for some term $\sigma \in \mathbf{U_\Delta}$;

[$\neg\exists$] $\mathbf{C}$ has form $\neg\exists\mathbf{xA}$ where $\neg\mathbf{A}(\mathbf{x}//\tau) \in \mathbf{\Delta}$ for every $\tau \in \mathbf{U_\Delta}$.

F Commented outline of the PARAM program

Inputs: x = the G.N. of a program Q in R1, y in R2
Output: The G.N. of the program (T 1 2, Z 1, S 1 y times, Q) in R1

Register usage: R7 = term number t
R8 = first part of answer a
R9 = G.N. of T 1 2 or Z 1 or S 1 instruction

00 - 10	FIVE				put 0 to 5 in R20 to R25
11	T	1	5		save x in R5
12	T	2	6		save y in R6
13	S	6			y := y + 1
14	Z	7			t := 0
15	Z	8			a := 0
16	Z	9			
17	P	24	20	9	
18	P	21	21	9	
19	P	22	22	9	R9 := G.N. of T 1 2
20	P	9	7	8	put T 1 2 in 0th term of a
21	S	7			t := t + 1
22	Z	9			
23	P	22	20	9	
24	P	21	21	9	R9 := G.N. of Z 1
25	P	9	7	8	put Z 1 in 1st term of a
26	P	23	20	9	R9 := G.N. of S 1
27	J	6	7	31	if t = y then jump to 24
28	S	7			t := t + 1
29	P	9	7	8	put S 1 in t-th term of a
30	J	1	1	27	jump to 27

31 - 39	TERMS	R1 := number of terms of x
40	T 1 4	R4 := number of terms of x
41	T 8 1	R1 := a
42	T 5 2	R2 := x
43	S 6	y := y + 1 (no. of terms of a)
44	T 6 3	R3 := y
45 - 68	JOIN	R1 := G.N. of the program with G.N. a followed by the program with G.N. x

G Index

the end