AD/A-006 898

FOL: A PROOF CHECKER FOR FIRST-ORDER LOGIC

Richard W. Weyhrauch, et al

Stanford University

Prepared for:

Office of Naval Research
Advanced Research Projects Agency

September 1974

DISTRIBUTED BY:

NTIS
National Technical Information Service
U. S. DEPARTMENT OF COMMERCE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER STAN-CS-74-432	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER AD/A-006898
4. TITLE (and Subtitle) FOL: A PROOF CHECKER FOR FIRST-ORDER LOGIC		5. TYPE OF REPORT & PERIOD COVERED technical, September, 1974
		6. PERFORMING ORG. REPORT NUMBER STAN-CS-74-432
7. AUTHOR(s) R. W. Weyhrauch and A. J. Thomas		8. CONTRACT OR GRANT NUMBER(s) DAHC 15-73-C-0435
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Department Stanford University Stanford, California 94305		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS ARPA/IPT, Attn: Stephen D. Crocker 1400 Wilson Blvd., Arlington, Va. 22209		12. REPORT DATE Sept. 1974
		13. NUMBER OF PAGES 63
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) ONR Representative: Philip Surra Durand Aeronautics Bldg., Rm. 165 Stanford University Stanford, California 94305		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Releasable without limitations on dissemination.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) SEE REVERSE		

PRICES SUBJECT TO CHANGE

This manual describes the use of the interactive proof checker FOL. FOL implements a version of the system of natural deduction described by Prawitz, augmented in the following ways:

(i) it is a many-sorted first-order logic and a partial order over sorts may be declared: this reduces the size of formulas;

(ii) purely propositional deductions can be made in a single step;

(iii) the truth values of assertions involving numerical and LISP constants can be derived by computation;

(iv) there is a limited ability to make metamathematical arguments, and

(v) there are many operational conveniences.

The goal of FOL is to use formal proof techniques as practical tools for checking proofs in pure mathematics and proofs of the correctness of programs. It is also intended to be used as a research tool in modelling common-sense reasoning in the representation theory of artificial intelligence.

STANFORD ARTIFICIAL INTELLIGENCE LABORATORY
MEMO AIM-235
STANFORD COMPUTER SCIENCE DEPARTMENT
REPORT NO. STAN-CS-432

SEPTEMBER 1974

FOL : a Proof Checker for First-order Logic

by
Richard W. Weyhrauch
Arthur J. Thomas

Abstract:

This manual describes the use of the interactive proof checker FOL. FOL implements a version of the system of natural deduction described by Prawitz, augmented in the following ways:

(i) it is a many-sorted first-order logic and a partial order over sorts may be declared: this reduces the size of formulas ;
(ii) purely propositional deductions can be made in a single step;
(iii) the truth values of assertions involving numerical and LISP constants can be derived by computation;
(iv) there is a limited ability to make metamathematical arguments, and
(v) there are many operational conveniences.

The goal of FOL is to use formal proof techniques as practical tools for checking proofs in pure mathematics and proofs of the correctness of programs. It is also intended to be used as a research tool in modelling common-sense reasoning in the representation theory of artificial intelligence.

We are grateful to Ashok Chandra for conceptual help and for implementing the Taut and Tauteq rules.

The research described here was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense under contract DAHC-15-73-c-0435.

TABLE OF CONTENTS

Section 0 THE RATIONALE FOR A FIRST-ORDER PROOF CHECKER

The reader ready to plunge right into making FOL proofs may skip to section 1.

The idea of doing mathematical reasoning mechanically goes back to Leibniz, but it was not until the end of the last century that Frege and Peano developed the first completely formal systems adequate for expressing some kinds of reasoning. Much of the work of Whitehead and Russell was an attempt at demonstrating that large parts of mathematics could actually be expressed within such systems. After these initial successes, however, the interest of logicians changed from proving theorems *within* mathematical systems to proving meta-theorems *about* such systems.

Even before Goedel's work, it was intuitively clear that checking proofs was different from finding them. It is an essential part of the idea of formal system that proofs can be checked mechanically, whereas *finding* proofs mechanically was always regarded as a research problem. This distinction was clarified by the work of Goedel, Tarski, Turing and Church which showed that algorithms for finding proofs can work infallibly only in limited domains and that some mathematical ideas cannot be completely characterized by axiomatic systems.

The advent of computers and the beginning of the study of artificial intelligence gave rise to attempts to explore experimentally what can be proved by machine. There has been steady progress in this endeavour, but twenty years work leaves us a long way from being able to prove important mathematical theorems.

Knowing that mechanical theorem proving has a long way to go justifies a renewed interest in the more straight-forward task of proof-checking by computer. Moreover, while it is not as interesting to check proofs by computer as to make computers prove the theorems, proof-checking has obvious potential applications. The most important of these is proving that computer programs meet their specifications since the reasoning involved is lengthy although usually straightforward - or so our intuition tells us. Since a computer program is a mathematical object whose properties are determined entirely by its symbolic form, it is a mathematical disgrace to have to debug them case by case rather than proving them correct in general. Since the programs are long, the proofs of correctness will be long, and since programmers sometimes think wishfully, it is obviously desirable that the proofs be checked by computer.

It is also interesting to see if we can check the proofs of interesting mathematical theorems even though the problem is of less practical urgency, since the human refereeing process works quite well.

At first sight, computer proof checking seems almost trivial. We know that almost all practical mathematical reasoning can be done in axiomatic set theory which in turn is expressed in first order predicate calculus. Therefore, it would seem that all we need do is to make a proof checker for predicate calculus, choose either the Zermelo-Fraenkel or the Goedel-Bernays-von Neumann axioms for set theory and write and check our proofs. This is one of the things the FOL project is doing, but in order that its formal proofs should not be substantially longer than conventional mathematical proofs, it is necessary to reformulate the usual logical systems. This can be thought of as an effort to produce a formal system in which the rules of inference, as well as the expressive power of the language, is more closely correlated with actual mathematical practice. The use of a computer allows for the introduction of complicated rules of inference whose metamathematics is not simple. FOL provides for the following:

(1) its notion of a first-order language includes function symbols, equality and other usual mathematical notation, such as infix operators, n-tuple notation;
(2) the user can declare sorts and declare variables to range over given sorts. This greatly reduces the length of axioms and theorems and corresponds to the fact that in an informal proof a context is established, and the reader knows that a certain part of the proof is carried out within the context;
(3) the decision procedures for certain simple domains are built into the system. This allows some proofs to be much shorter than usual mathematical proofs, because the computer can go through some quite complex chains of reasoning by itself. At present, propositional deduction and a fragment of the theory of equality have been implemented. The Boolean algebra of sets and elementary commutative algebra are planned;
(4) some facilities for introducing definitions have been implemented;
(5) a facility is provided for defining the interpretations of constants and predicate/function symbols, and for *computing* within a model of the language. This means, for example, that algebraic and LISP functions can be calculated directly, rather than being synthetically derived;
(6) some primitive facilities are available for metamathematical reasoning;
(7) rules of inference for some interesting modal logics are provided.

The domains which are being explored by means of FOL proofs include:

(i) CLASSICAL MATHEMATICS. This is the single most striking success in our ability to *represent* reasoning in terms of formal derivations. How close are these derivations to a mathematician's informal proof? Do they constitute a faithful representation of his reasoning? How are the inference rules of our logic related to the actual rules of evidence he uses when convincing himself of some truth? The answers to these questions are important in determining whether we can make computer-checkable proofs that are not enormously longer than the proofs in mathematical journals. Experiment with the use of FOL in classical mathematics will help answer them. Theoretical studies of the intensional properties of proofs such as those of Kreisel (1971a,1971b) are also relevant. Moreover, it turns out that a large part of many mathematical proofs in the literature are really at the metamathematical level, i.e. they are reasoning about the reasoning in the axiomatic system. Thus it can happen that a simple theorem prover or proof-checker is not even capable of *expressing* the theorems of mathematicians, let alone *proving* them;

(ii) MATHEMATICAL THEORY OF COMPUTATION. (McCarthy 1963, Floyd 1967, Manna 1974)and others have shown how first-order theories can be used in proving properties of programs. Making this into a tool for verifying programs before they are widely distributed is one of the major goals of the FOL project. This will require further research in formalizing the properties of programs, the ability provided by the *attachment* feature of FOL to establish decidable properties of parts of the program by direct calculation rather than step-by-step inference, and a great deal of experiment aimed at making the proofs correspond to the programmer's informal reasoning that his program does what it should;

(iii) REPRESENTATION THEORY. Common sense reasoning is being represented in FOL in the style of (McCarthy and Hayes 1969). As in proving programs correct, purely inferential reasoning must be supplemented by assertions directly computed from the data base representing the environment; again the FOL attachment feature is the key device used. Even more experiment will be required before the formal proofs correspond to informal reasoning than in the case of mathematics, because this area has not been well explored (perhaps only by McCarthy, Hayes 1974, and Sandewall 1970). Particular problems are the axiomatization of time, simultaneity, causality, knowledge, and the geometric reasoning involved in perception. Metamathematics also comes in, particularly when it is necessary to reason about knowledge and belief. We hope that axiomatizing the metamathematics of FOL, i.e. the structure and truth conditions of FOL sentences together with a *reflection principle*, suitably restricted to avoid paradoxes, will enable us to express common sense reasoning about knowledge, belief, truth and falsehood.

FOL is committed to a system of *natural deduction*. The use of the word 'natural' is best explained by Prawitz himself (Prawitz,1965):

> *'Systems of natural deduction, invented by Jaskowski and by Gentzen in the early 1930's, constitute a form for the development of logic that is natural in many respects. In the first place, there is a similarity between natural deduction and intuitive, informal reasoning. The inference rules of the systems of natural deduction correspond closely to procedures common in intuitive reasoning, and when informal proofs -- such as are encountered in mathematics for example -- are formalized within these systems, the main structure of the informal proofs can often be preserved. This in itself gives the systems of natural deduction an interest as an explication of the informal concept of logical deduction.*
>
> *Gentzen's variant of natural deduction is natural also in a deeper sense. His inference rules show a noteworthy systematization, which, among other things, is closely related to the interpretation of the logical signs. Furthermore, as will be shown in this study, his rules allow the deduction to proceed in a certain direct fashion, affording an interesting normal form for deductions. The result that every natural deduction can be transformed into this normal form is equivalent to what is known as Hauptsatz or the normal form theorem, a basic result in proof theory, which was established by Gentzen for the calculi of sequents. The proof of this result for systems of natural deduction is in many ways simpler and more illuminating.*

In this manual, most of the metamathematical notions discussed will be referred to by words in the following font e.g. SYNTYPE, INDVAR, WFF. *These notions will play a greater role in later versions of FOL.*

Section 1 THE NOTION OF AN FOL LANGUAGE

In FOL the user specifies a first-order language by making a set of DECLARATIONs (see Section 4.3). The proof-checking system then generates a proof checker and a collection of rules specific to that system.

An FOL *language* is determined by specifying a way of building up expressions, usually called well formed formulas or WFFs, from collections of primitive symbols. In FOL these classes of symbols are called SYNTYPEs. They are:

1. logical constants:

   a) *sentential constants* - SENTCONSTs: FALSE, TRUE
   b) *sentential connectives* - SENTCONNs: ¬,∧,∨,⊃,≡
   c) *quantifiers* - QUANT: ∀, ∃

2. auxiliary symbols: - AUXSYM: "(" and ")"

3. sets of variable symbols:

   a) *individual variables* - INDVARs.
   b) *individidal parameters* - INDPARs.

4. a set of *n-place predicate parameters* - PREDPARs.

These symbols are used to form those sentences common to all FOL languages. Sometimes a language L may also contain symbols which are intended to have interpretations which are fixed relative to the domain of the interpretation. Examples are: "∈" in set theory, "=" in first order logic with equality, "0" and "Suc" in arithmetic. These are represented by

5. sets of constant symbols:

   a) *individual constants* - INDCONSTs.
   b) *n-place operation symbols* - OPCONSTs.
   c) *n-place predicate constants* - PREDCONSTs.

In addition one can

6. restrict the range of a variable symbol to some PREDCONST by declaring it to be a SORT.

7. designate a *partial order* to hold among some of those PREDCONSTs which have been declared to be SORTs;

TERM, AWFFs (atomic well formed formulas), and WFFs (well formed formulas) are defined in the usual way.

A formal description of these languages and of the notion of SORT is given in appendix 1. The entire extended syntax of FOL is described in appendix 2.

A first-order THEORY is defined by a (possibly empty) set of sentences of L, called AXIOMs. It is the creation of such theories and the checking of valid deductions in them that is the main purpose of the computer program FOL.

Section 2 THE NOTION OF AN FOL DEDUCTION

A derivation (the following description of which is taken almost *verbatim* from Prawitz 1965) begins by inferring a *consequence* from some ASSUMPTIONs or AXIOMs by means of one of the RULEs listed below. We indicate this by writing the formulas assumed on a horizontal line and the formula inferred immediately below this line. On the computer this can be repeated using previous consequences as new hypothesis. This generates a tree. which we call a DERIVATION. Thus if we wish to derive A⊃(B∧C) from (A⊃B)∧(A⊃C) we write:

```
              (A⊃B)∧(A⊃C)                     (A⊃B)∧(A⊃C)
             -------------                   --------------
   A             (A⊃B)               A            (A⊃C)
   -------------------               -------------------
            B                                  C
           ---------------------------------------
                             (B∧C)
```

At each step so far, the configuration is a DERIVATION of the undermost formula from the set of formulas that appear as ASSUMPTIONs. The assumptions are the uppermost formula occurrences, and we say that the undermost formula *depends* on these ASSUMPTIONs. Thus, the example above is a deduction of B∧C from the set of assumptions {(A⊃B)∧(A⊃C),A}, and in this deduction, B∧C is said to depend on the top occurrences of these formulas.

As the result of some inferences, however, the formula inferred becomes independent of some or all assumptions, and we then say that we *discharge* the assumptions in question. There are four ways to discharge assumptions, namely:

(1) Given a deduction of B from {A}∪Γ, we may infer A⊃B and discharge the assumptions of the form A;

(2) Given a deduction of FALSE from {¬A}∪Γ, we may infer A and discharge the assumptions of the form ¬A;

(3) Given three deductions, one of C from {A}∪Γ_1, one of C from {B}∪Γ_2 and one of A∨B, we may infer C and discharge the assumptions of the form A and B that occur in the first and second deductions respectively, i.e. below the end-formulas of the three deductions, we may write C and then obtain a new deduction of C independent of the mentioned assumptions;

(4) Given a deduction of B from {A[x←a]}∪Γ and a deduction of ∃x.A, we may infer B and discharge assumptions of the form A[x←a], provided that a does not occur in ∃x.A, in B, or in any assumption - other than those of the form A[x←a] - on which B depends in the given deduction.

To continue the deduction above, we may write A⊃(B∧C) below B∧C and obtain a deduction of A⊃(B∧C) from {(A⊃B)∧(A⊃C)}.

Section 3 THE RULES OF INFERENCE

The inference rules consist of an *introduction* (I) and an *elimination* (E) rule for each logical constant. The letters within parentheses indicate that the inference rule discharges assumptions as explained above.

$$\wedge I)\quad \frac{A \quad B}{A \wedge B} \qquad\qquad \wedge E)\quad \frac{A \wedge B}{A} \quad \frac{A \wedge B}{B}$$

$$\vee I)\quad \frac{A}{A \vee B} \quad \frac{B}{A \vee B} \qquad\qquad \vee E)\quad \frac{A \vee B \quad \begin{matrix}(A)\\ C\end{matrix} \quad \begin{matrix}(B)\\ C\end{matrix}}{C}$$

$$\supset I)\quad \frac{\begin{matrix}(A)\\ B\end{matrix}}{A \supset B} \qquad\qquad \supset E)\quad \frac{A \quad A \supset B}{B}$$

$$\forall I)\quad \frac{A}{\forall x.A[a \leftarrow x]} \qquad\qquad \forall E)\quad \frac{\forall x.A}{A[x \leftarrow t]}$$

$$\exists I)\quad \frac{A[x \leftarrow t]}{\exists x.A} \qquad\qquad \exists E)\quad \frac{\exists x.A \quad \begin{matrix}(A[x \leftarrow a])\\ B\end{matrix}}{B}$$

$$\neg I)\quad \frac{\begin{matrix}(A)\\ \mathrm{FALSE}\end{matrix}}{\neg A} \qquad\qquad \neg E)\quad \frac{\begin{matrix}(\neg A)\\ \mathrm{FALSE}\end{matrix}}{A}$$

$$F I)\quad \frac{\neg A \quad A}{\mathrm{FALSE}} \qquad\qquad F E)\quad \frac{\mathrm{FALSE}}{A}$$

$$\equiv I)\quad \frac{A \supset B \quad B \supset A}{A \equiv B} \qquad\qquad \equiv E)\quad \frac{A \equiv B}{A \supset B} \quad \frac{A \equiv B}{B \supset A}$$

Restriction on the ∀I-rule: a must not occur in any assumption on which A depends.

Restriction on the ∃E-Rule: a must not occur in $\exists x.A$, in B, or in any assumption on which the upper occurrence of B depends other than $A[x \leftarrow a]$.

Section 3.1 An FOL deduction using the computer

We show here the computer interaction necessary to check the derivation given in Section 2.

*In this and all succeeding sections examples of interactions with the computer will appear in small type. Those lines which are typed by the user will be preceeded by five stars "*****". The other lines are those typed by the computer.*

To derive A⊃(B∧C) from (A⊃B)∧(A⊃C), we proceed as follows.

```
*****DECLARE SENTCONST A,B,C;
*****ASSUME (A⊃B)∧(A⊃C);
1  (A⊃B)∧(A⊃C)   (1)
*****∧E 1,1;
2  (A⊃B)   (1)
*****ASSUME A;
3  A   (3)
*****⊃E 2,3;
4  B   (1 3)
*****∧E 1,2;
5  (A⊃C)   (1)
*****⊃E 3,5;
6  C   (1 3)
*****∧I 4∧5;
7  B∧C   (1 3)
*****⊃I 3⊃7;
8  A⊃(B∧C)   (1)
```

Each LINE typed by the computer contains: 1) a LINENUM, which labels that LINE: 2) the WFF representing the result of applying the RULE typed by the user on the line above: 3) a list of numbers representing those LINEs of the proof on which the WFF depends. Consider the LINE begining with 7 in the above example. 7 is its LINENUM, B∧C is the WFF on this LINE, and the derivation of B∧C on this LINE depends on the assumptions on LINEs 1 and 3. This LINE was generated by the user specifying as a RULE ∧I (AND introduction) using *lines* 4 and 5. This information is typed by the user and in the example appears directly above LINE 7 of the proof.

There are two other things to notice about this example. The first thing typed by the user was a declaration stating that A,B and C are SENTCONSTs. Making declarations is essential. Failure to declare an identifier is the most common reason for a syntax error. Second is that when ⊃I is applied to LINEs 3 and 7, LINE 3 has been removed from the list of dependencies of the new LINE. This corresponds to the description of this rule given on each of the previous two pages. The exact format of the commands a user must type to the computer is explained in section 4.

Section 3.2 Implementation - user oriented features of FOL

There are several differences between the machine implemention of FOL and the description given above and in Appendix 1. These differences are usually for the purpose of making life easier for the user. The description in the Appendix presents a clean version of the logic so that the metamathematics can be discussed in a straight-forward way. The major differences are described briefly below; more detailed descriptions occur in the appropriate sections of the sequel.

Section 3.21 Individual symbols

In Prawitz's logic, individual variables (INDVARs) may only appear bound, and individual parameters only free. In FOL, this restriction is relaxed, and INDVARs may appear free as well as bound in well-formed formulas. INDPARs, however, must always appear free. Additionally, natural numbers are automatically declared to be INDCONSTs of SORT NATNUM.

Section 3.22 Prefix and Infix notation

FOL allows a user to specify that binary predicate and operation symbols are to be used as infixes. The declaration of a unary application symbol to be prefix makes the parentheses around its argument optional. The number of arguments of an application term is called its ARITY. Section 4.1 describes how to make such declarations.

Section 3.23 Extended notion of TERMs

In addition to ordinary application terms, FOL accepts TERMs representing finite sets, comprehension terms, n-tuples and LISP s-expressions. A detailed description of the syntax of these terms is to be found in Appendix 2.

Section 3.24 The Equality of WFFs

The description of subsitution given in Section 4.35 is consistent with FOL's notion of equivalence of WFFs. The proof-checker always considers two WFFs to be equal if they can both be changed into the *same* WFF by making allowable changes of bound variables. Thus, for example, the TAUT rule will accept ∀x.P(x)⊃∀y.P(y) as a tautology.

Section 3.25 VLs and subparts of WFFs and TERMs

FOL as implemented offers very powerful and convenient techniques for referring to objects in a proof: essentially, any well-formed expression has a name, and can be manipulated as a single entity. A VL is a name of a part of a derivation. There are several kinds of VLs: for example, a label represents a line-number, the WFF on that line, and a list of the dependencies of that line in the derivation.

The syntax of VLs is very extensive and a review of it will be left to Appendix 2.

Section 3.26 Axioms and Assumptions

FOL allows the specification of certain WFFs as AXIOMs. The difference between these and ASSUMPTIONs is that the former are not mentioned explicitly as dependencies of any lines of the derivation. Thus every proof checked by FOL tacitly depends on a set of AXIOMs.

Section 3.27 FOL derivations

As opposed to a *tree*, a deduction in FOL consists of a collection of AXIOMs and a *linear sequence* of lines, each line representing either an ASSUMPTION or a DEDUCTION from the previous lines (and axioms).

Section 3.28 SORTs

The addition of SORTs, and specification of a partial order over them, constitutes a major extension of FOL from a computational point of view. Their meaning and use is discussed in the sections on declarations and the quantifier rules.

Section 4 USING THE PROOF CHECKER

FOL is invoked at the Stanford A.I Lab by typing *R FOL* to the monitor. A backup file is automatically opened onto which input is saved; the name of this file may be altered by means of the BACKUP command (*vide infra*). To save an entire core image type the command 'EXIT;' and *SAVE <filename>*; to restart type *RU <filename>* and you will be where you left off.

The commands fall naturally into several classes:

1. Commands for defining the first-order language under consideration: that is to say, commands for making *declarations*;

2. Commands for defining *axioms*;

3. Commands for making *assumptions* and applying the rules of inference to generate new steps in a derivation;

4. Administrative commands, which do not alter the state of the derivations, but enable various book-keeping functions to be carried out.

In this manual the syntax of FOL will be described using a modified form of the MLISP2 notion of pattern. These form the basic constructs of the FOL parser.

1. Identifiers which appear in patterns are to be taken literally.
2. Patterns for syntatic types are surrounded by angle brackets. Thus <wff> is a WFF.
3. Patterns for repetitions are designated by:
   REPn[<pattern>] means n or more repeated PATTERNs.
   If a REPn has two arguments then the second argument is a pattern that acts as a separator. So that REP1[<wff>, ,] means one or more WFFs seperated by commas.
4. Alternatives appear as ALT[<PATTERN1> | ... | <PATTERNn>].
   ALT[<wff> | <term>] means either a WFF or a TERM.
5. Optional things appear as OPT[<pattern>]
   REP2[<wff>,OPT[,]] means a sequence of two or more WFFs optionally separated by commas.

These conventions are combined with the comparatively standard Backus Normal Form description.

Section 4.1 System Specification

The first step in specifying a first-order theory is the description of the language which is to be used. This is done by defining the symbols of the language, using the declaration commands. These commands specify which symbols are to be variables, constants and predicate or function symbols.

Section 4.11 Declarations

As we mentioned above, one of the first things that a user of FOL must do is to define the FOL language to be considered. Every identifier in a proof must be declared to have a SYNTYPE. Only nine of these types can be declared by the user. They are:
1. SYNTYPE1

a) INDVAR (*individual variables*)
b) INDPAR (*individual parameters*)
c) INDCONST (*individual constants*)
d) SENTPAR (*sentential parameters*)
e) SENTCONST (*sentential constants*)

2. SYNTYPE2

a) PREDPAR (*predicate parameters with one or more arguments*)
b) PREDCONST (*predicate constants*)
c) OPPAR (*operation parameters or function parameters*)
d) OPCONST (*operation constants or function constants*)

Declarations are fixed within a proof and once made they *cannot* be changed.

```
DECLARE  ALT[ REP1[<simpldec> OPT[,]] | REP1[<appldec> OPT[,]] ]  ;
```

There are two kinds of SYNTYPEs, those of symbols which take arguments, SYNTYPE2s, and those which do not, SYNTYPE1s.

```
<syntype1>    :=  ALT[ <indsym>  | <sentsym> ]
<syntype2>    :=  ALT[ <predsym> | <opsym>   ]
```

The idea of SORTs is to allow a user of FOL to restrict the ranges of function to some predetermined set. This correspond to the usual practice of mathematicians of saying let f be a function which maps integers into integers. In FOL a SORT is just a PREDCONST of ARITY 1, i.e. a property of individuals. The effect of this informal restriction to integers is achieved in FOL by

```
*****DECLARE PREDCONST INTEGER 1;
```

followed by

```
*****DECLARE OPCONST +((INTEGER,INTEGER)=INTEGER;
```

A PSEUDOSORT is an identifier which has not yet been declared but is assumed to be a PREDCONST of ARITY 1 and is declared such because of the context in which it appears. If INTEGER had not been separately declared above, in its appearance in the second command it would have been considered to be a PSEUDOSORT and declared accordingly. There is one special PSEUDOSORT, i.e. the PREDCONST UNIVERSAL. This represents the most general SORT and is the default option whenever SORT specifications are optional. In declarations it can also be abbreviated by "*". The MOSTGENERAL command explained in the next section, can be used to change the name of the MOSTGENERAL SORT.

```
<pesudosort>  :=  ALT[ <identifier> | * ]
```

Simple declarations

```
<simpldsc>    :=  <syntype1> <idlist>  OPT[ ε <pseudosort> ]
```

Examples of simple declarations:

```
*****DECLARE INDVAR x y z;

*****DECLARE INDVAR a b c ε Set, A B C ε Class;
```

Application declarations

```
<appldsc>     :=  <syntype2> <idlist> <argdec> OPT[ [ <bpdsc> ] ]
<argdec>      :=  ALT[ <Argsort> | <natnum> ]
<argsort>     :=  ALT[ : <sortrsp>  ALT[=|→] <pseudosort>  |
                       ( <sortrsp> ) ALT[=|→] <pseudosort>  ]
<sortrsp>     :=  REP1[ <pseudosort> , OPT[ALT[=|,]] ]

<bpdec>       :=  ALT[ <rbp> | <rbp> <lbp> | <lbp> <rbp> | INF | PRE ]
<rbp>         :=  R  ←  <natnum>
<lbp>         :=  L  ←  <natnum>
```

Examples of application declarations:

```
*****DECLARE OPCONST  EXP(Int,Int)=Int [L←850 R←800] ;
```

The meaning of this declaraion is that EXP is an OPCONST, it has two arguments (ARITY 2), both of which are of SORT Int. It also has a value of SORT Int, and is to be used as in infix operator with a right binding power of 800 and a left binding power of 850. This could also be declared by

```
*****DECLARE OPCONST EXP:Int×Int→Int [L←850 R←800] ;
```

Simpler declarations can be made if you don't wish to specify so much information.

```
*****DECLARE OPCONST EXP:Int*Int→Int [INF] ;
```

declares EXP the same as above but uses the default infix bindings R←500, L←550.

```
*****DECLARE OPCONST EXP(Int,Int)=Int;
```

simply makes EXP an ordinary applicative function, so you must type EXP(a,b) rather than (a EXP b). Further simplificion can be made if less sort information is wanted

```
****DECLARE OPCONST EXP(Int,Int);
```

makes the value of EXP have the SORT **UNIVERSAL** (the MOSTGENERAL SORT), and

```
*****DECLARE OPCONST EXP 2;
```

just says it has ARITY 2. Of course

```
*****DECLARE OPCONST EXP 2 [INF]
*****DECLARE OPCONST EXP 2 [L←850 R←800] ;
```

have the obvious meaning. This section has illustrated most of common ways of making declarations. There are some other examples scattered throughout this manual.

Section 4.12 SORT manipulation

There are several commands which affect the SORT structure:

Section 4.121 NOSORT declaration

```
NOSORT ;
```

The **NOSORT** command turns off SORT checking. If any SORTs have already been declared, an error message will be given.

Section 4.122 MOSTGENERAL, NUMSORT, SETSORT, SEXPRSORT

```
MOSTGENERAL <sort> ;
NUMSORT     <sort> ;
SETSORT     <sort> ;
SEXPRSORT   <sort> ;
```

In FOL certain TERMs come with predeclared SORTs: numerals become INDCONSTs of SORT NATNUM, comprehension terms, set terms and n-tuple terms have SORT SET, quote-terms have SORT SEXPR, and the default MOSTGENERAL SORT is the PREDCONST UNIVERSAL. The effect of the above commands is to replace these default SORTs with those specified by the user. For example, in the case of Goedel-Bernays-von Neumann set theory, the MOSTGENERAL SORT is called CLASS.

Section 4.123 MOREGENERAL declaration

```
MOREGENERAL <sort> ≥ { <sort_list> } ;
```

For example,

```
*****MOREGENERAL chesspiece ≥ {whitepiece,blackpiece};
```

is equivalent to the axioms

$$\forall x.\ (whitepiece(x) \supset chesspiece(x))$$
$$\forall x.\ (blackpiece(x) \supset chesspiece(x))$$

where chesspiece, whitepiece and blackpiece are understood to have been previously declared PREDCONSTs. Although these axioms do not appear explicitly, the quantifier rules behave as if they did (this is explained in detail in section 4.327). This establishes a partial order among the SORTs. Another typical example would be the declaration of classes to be MOREGENERAL than sets.

Section 4.124 EXTENSION declarations

```
EXTENSION <predconst> <ext_set> ;
```

```
<ext_set>     :=    <primext> REP0[ ALT[∪|∩|/] <primext> ]
<primext>     :=    ALT[ <sort> | { <indconstlist> } ]
```

where each of the SORTs in the <primext> already has an EXTENSION defined. For example,

```
*****DECLARE INDCONST BK ∈ BKINGS, WK ∈ WKINGS;

*****DECLARE PREDCONST KINGS 1;

*****EXTENSION BKINGS {BK};

Extension of BKINGS is (BK)

*****EXTENSION WKINGS {WK};

Extension of WKINGS is (WK)

*****EXTENSION KINGS WKINGS U BKINGS;

Extension of KINGS is (WK BK)
```

The initial declaration declares BK to be of SORT BKING, and WK to be of SORT WKING. The command 'EXTENSION BKINGS {BK};' says that BK is the *only* object which satisfies the predicate BKINGS; similarly, the command 'EXTENSION KINGS BKINGS U WKINGS' says that the only objects which satisfy the predicate KINGS are those in the union of the extensions of BKINGS and WKINGS, i.e. BK and WK. This is equivalent to the introduction of the axioms:

$$\forall x. (BKINGS(x) \equiv (x{=}BK))$$
$$\forall x. (WKINGS(x) \equiv (x{=}WK))$$
$$\forall x. (KINGS(x) \equiv ((x{=}BK \lor x{=}WK) \land \neg(BK{=}WK)))$$

By itself, this command has no effect, but the semantic simplification mechanism (see Section 4.4) uses these axioms.

Section 4.13 Predeclared Systems

THEORY <sysname> ;

The THEORY command may be used to call up several pre-declared systems. If no THEORY command is given, the basic FOL system is generated, i.e. the full natural deduction system for classical logic with the extended inference rules. The options which are available are

```
<sysname> := ALT | PRAWITZ | ZF | GBN | S4 | S5 | KBK | KBB ]
```

where PRAWITZ is the system described by (Prawitz 1965), i.e. without SORTs or any of the extended inference rules such as TAUT; ZF is Zermelo-Fraenkel set theory (as defined in Appendix 3); GBN is Goedel-Bernays-von Neumann set theory (as defined in Appendix 4); S4 and S5 are Lewis's classical systems of possibility and necessity (as defined in Appendix 5); and KBK and KBB are Hintikka's systems for Knowledge and Belief respectively (see Appendix 5).

Section 4.2 Axioms

Axioms are only briefly mentioned in the description of FOL. In the machine implemented version they play the same role as assumptions, but they do not appear in the dependency list of any step of a deduction, nor are they printed when you show the proof. Thus derivations are always relative to an unmentioned theory. When a theorem creating mechanism is available this will change. The syntax for defining an axiom is:

```
AXIOM <axiom> ;
```

where

```
<axiom>  := REP1[ <axnam> : <axlist>  ; ]
<axlist> := ALT[ <wfflist> | REP1[<axiom>] ]
```

This allows for a block structured way of naming sets of axioms, so they can be referred to either by some particular name, or as part of a group. Each WFF in WFFLIST is given a name by FOL. This name is generated by taking the AXNAM and concatenating an integer to it. For example if the AXNAM is GROUP then they will be given the names GROUP1, GROUP2,... . These can then be used to refer to each axiom. An AXNAM is like a LINENUM and may be used in any context that requires a LINENUM. If WFFLIST only contains one WFF that axiom is called AXNAM.

NOTE: The syntax calls for multiple *semicolons!*

Examples:

```
*****AXIOM  A: B: ∀X.¬X<X,
                  ∀Y.¬(X<Y∧Y<X); ;
            C:    ∀W.W<W; ;
```

This creates two axioms A and C. Axiom A contains two subaxioms B1=∀X.¬X<X and B2=∀Y.¬(X<Y∧Y<X). If you prefer to think of collections of axioms as theories, then the syntax allows arbitrary nesting of theories, each followed by a semicolon. At the moment no checking is done for the consistency of axiom names. You lose if you create conflicting ones. Axioms cannot be got rid of, so be careful. Numbers are *not* legitimate AXNAMs.

Using axioms as axiom schemas.

There are no special rules for axiom schemas, merely an extension of the use of the rules already given. Namely, an *axiom schema* is simply an axiom with a predicate parameter (PREDPAR) in it.

An axiom can be used anywhere a step can by using an AXREF. This is of the form AXNAM[PP_1←XX_1,...,PP_n←XX_n] and its syntax is described in the section on VLs. An AXREF can appear anywhere a VL can. In the form AXNAM[PP_1←XX_1,...,PP_n←XX_n], the PP_i are predicate parameters (PREDPARs) appearing in the axiom, and the XX_i are propositional functions assigned to these parameters. The assignments are done successively rather than simultaneously.

An XX_i is a WFF preceded by λ, any number of INDVARs and a "." (period). Thus e.g. λ x y z.<wff>. The ARITY, p, of the PREDPAR must be less than or equal to the number of variables following the λ. The indicated λ-conversion on the first p variables is done automatically. The error message "NOT ENOUGH LAMBDA VARIABLES" means p is too large. The remaining variables are treated as parameters of the entire axiom, and the instance of the axiom returned is the universal closure of the axiom with respect to these parameters.

The :≡ (SUBPART) mechanism (see Appendix 2) can be used to take pieces out of the resulting formula in the usual way.

Example of using axiom schemas:

```
*****DECLARE PREDPAR F 1;

*****INDVAR X;

*****AXIOM INDUCTION: F(0)∧∀X.(F(X)⊃F(X+1))⊃∀X.F(X);;

INDUCTION: F(0)∧∀X.(F(X)⊃F(X+1))⊃∀X.F(X)

*****DECLARE INDVAR a b;

*****∧I INDUCTION[F←λb a.a+b=b+a];

1 ∀a.((a+0)=(0+a)∧∀X.((a+X)=(X+a)⊃(a+(X+1))=((X+1)+a))⊃∀X.(a+X)=(X+a))

*****∧I INDUCTION[F←λb.∀a.a+b=b+a];

2 ∀a.(a+0)=(0+a)∧∀X.(∀a.(a+X)=(X+a)⊃∀a.(a+(X+1))=((X+1)+a))⊃∀X a.(a+X)=(X+a)

*****∧I INDUCTION[F←λb X.X+b=b+X];

3 ∀X.(X+0)=(0+X)∧∀X1.((X+X1)=(X1+X)⊃(X+(X1+1))=((X1+1)+X))⊃∀X2(X+X2)=(X2+X))
```

Section 4.3 The generation of new deduction steps

Note: when the variables A,B and C are mentioned in this section, they refer to the description of the basic Prawitz logic in section 3.

Section 4.31 Assumptions

```
ASSUME  <wfflist>  ;
```

The ASSUME command makes an assumption on a new line of the deduction for each WFF in WFFLIST. Note that the dependencies of a line appear in parentheses at the end of a line, and that assumptions depend upon themselves

```
Examples:

*****ASSUME ∀x.x<x;

1 ∀x.x<x   (1)

*****ASSUME ∀y.y<y, ¬∀y.y<y;

2 ∀y.y<y   (2)

3 ¬∀y.y<y   (3)
```

Section 4.32 Introduction and Elimination rules

The general form of a RULENAME is

```
<rulename> := <logconst> ALT( I | E )
```

where I stands for Introduction and E for elimination. The format of a command is:

```
<rule_of_inference> := <rulename> <linenuminfo> ;
```

The LINENUMINFO is different for each rule. This is explained below. We will use * to stand for an arbitrary VL (see section 3.25). In the description of some of the rules it is necessary to distinguish among several VLs. In this case we write *1,*2,... . We will write

```
∧I *∧* ;
```

rather than

```
∧I <vl> ∧ <vl> ;
```

Alternative alphabetic RULENAMEs will be given in parentheses after the standard ones. These usually correspond to other frequently used names for these rules. Thus MP (*modus ponens*) or UG (universal generalization) can be used, instead of ⊃I or ∀I.

All commas in these rules are optional. This will not be mentioned explicitly in the following sections. Thus a "," appearing in a rule specification it is to be thought of as OPT[,].

Section 4.321 AND (∧) rules

__Introduction rule__

```
∧I (AI)      (*∧*)∧*        ;
```

The LINENUMINFO for ∧I is any parenthesized conjunctive expression in which all conjuncts are VLs. If no parentheses appear (even in a subexpression) association is to the right, thus *∧(*∧*∧*)∧* means *∧((*∧(*∧*))∧*). AND is always a binary connective. The "&" and "," are alternatives to the "∧" symbol. The dependencies of a line are those LINENUMs mentioned.

__Elimination rule__

```
∧E(AE)  *   OPT[ ALT[,|:] ]  ALT[1|2| <subpart> ]   ;
```

1 picks out the first conjunct, 2 picks out the second conjunct and SUBPART picks the appropriate subpart. For the definition of SUBPART see Appendix 2. The dependencies of the result are the same as those of *. The first command in the example could have also been written "AE 4 1;" or "∧E 4:1;" or "AE 4:*1;".

```
******∧E 4,1;

5 (∀x.Class(x)∧∀a.¬(a∈MT))

******AE 4:#1#2;

6 ∀a.¬(a∈MT)

******AE 4:#1#1#1;

The main symbol of  ∀x.Class(x) is not an ∧

******∧E 4:#3;

In the <subpart> :#3 , 3 is too large
```

Section 4.322 OR (v) rules

Introduction rule

```
vI (OI)     (*v<wff>v<wff>)     ;
```

OR's may be parenthesized just like AND's, but at least *one* disjunct *must* be a VL. Any VLs given will cause the dependencies of that line to be included in those of the conclusion. As with AND, association is to the right and OR is binary.

Elimination rule

```
vE (OE)     * , *1 , *2     ;
```

* is the VL on which a disjunction AvB appears *1 and *2 are both VLs such that *1: and *2: are both equal to the WFF C. The conclusion of this rule is the WFF C. The dependencies of the conclusion are those of * along with those of *1 which are not equal to A and those of *2 not equal to B. Remember two WFFs are equal if they differ only by a change of bound variable. In the example two different commands are given. Note how the dependencies are treated in each case.

```
*****ASSUME 1:v3:;

9 ∀x.x<xv¬∀y.y<y   (9)

*****OI 1v3:;OI 2:v3;

10 ∀x.x<xv¬∀y.y<y   (1)

****
11 ∀y.y<yv¬∀y.y<y   (3)

******vE 9,10,11;

12  ∀x.x<xv∀y.y<y   (9)

******vE 8,10,11;

13 ∀x.x<xv¬∀y.y<y   (3)

******vE 9,11,10;

14 ∀x.x<xv¬∀y.y<y   (1 3 9)
```

Section 4.323 IMPLIES (⊃) rules

Introduction rule

```
⊃I(DED)   ALT[ *⊃* | <uff>⊃* ] ;
```

The difference between *⊃* and <wff>⊃* is that in the former case dependencies of the conclusion which are equal to the hypothesis are deleted. A comma is an alternative to the "⊃" symbol. In other styles of presenting first order logic this rule is called the deduction theorem.

```
******⊃I 1⊃1;

15 ∀x.x∈x⊃∀x.x∈x

*****:DED 1:⊃1;

16 ∀x.x∈x⊃∀x.x∈x   (1)

***** ⊃I 2,1;

17 ∀y.y∈y⊃∀x.x∈x
```

Elimination rule

```
⊃E(MP)  *  , *   ;
```

The order in which the arguments are specified is irrelevant. This is the classical rule *modus ponens*. The dependencies of the conclusion are the union of the dependencies of both VLs.

```
******⊃E  1,17;

18 ∀x.x∈x    (1)
```

Section 4.324 FALSE (FALSE) rules

<u>Introduction rule</u>

```
FI  *1  , *2   ;
```

If *1 is of the form A, then *2 must be of the form ¬A (or the other way around). The conclusion is just the WFF "FALSE". Its dependencies are the union of those of *1 and *2.

```
*****FI 1,3;

19 FALSE   (1 3)
```

<u>Elimination rule</u>

```
FE  *  ,  ALT[ *1 | <wff> ]  ;
```

* must be of the WFF "FALSE". A new line is created with either *1: or the WFF specified by the alternative. This rule says that anything follows from a contradiction. The dependencies (there had better be some) are just those of *.

```
*****FE 19 6:#1(A←x);

20 ¬(x<MT)   (1 3)
```

Section 4.325 NOT (¬) rules

Introduction rule

```
¬I (NI)    *    ,    ALT[ *1 | <suff> ]    ;
```

* must be the WFF "FALSE". The conclusion of the rule is the negation of *1: or the WFF. The dependencies of the conclusion are those of * minus the ones equal to *1: or WFF.

```
*****¬I 19,3;

21 ¬¬∀y.yεy   (1)

*****DED 1⊃21;

22 ∀x.xεx⊃¬¬∀y.yεy
```

Elimination rule

```
¬E (NE)    *    ,    ALT[ *1 | <suff> ]    ;
```

* must be the WFF "FALSE". *1 or WFF must have the form ¬A. The conclusion is A. The dependencies are those of *, minus any equal to ¬A. If this rule is omitted (or simply not used) and only the introduction and elimination rules are used the proof is intuitionisticly valid.

```
*****ASSUME ¬3;;

23 ¬¬∀y.yεy   (23)

*****FI 23,3;

24 FALSE   (3 23)

*****¬E 24,3;

25 ∀y.yεy   (23)

*****DED 23⊃25;

26 ¬¬∀y.yεy⊃∀y.yεy
```

Section 4.326 EQUIVALENCE (≡) rules

<u>Introduction rule</u>

```
≡I (EI)    ≡1  ,  ≡2   ;
```

Either ≡1 is of the form A⊃B and ≡2 is of the form B⊃A or vice versa. The conclusion is A≡B. The dependencies are the union of the dependencies of ≡1 and ≡2.

```
*****≡I 26,22;

27 ¬¬∀y.y<y≡∀y.y<y
```

<u>Elimination rule</u>

```
≡E (EE)    ≡   ,  ALT[ ALT[⊃|1] | ALT[⊂|2] ]   ;
```

If ≡ is of the form A≡B then the first alternative produces A⊃B, the second B⊃A. The dependencies are those of ≡.

```
*****≡E 27 ⊂;

28 ∀y.y<y⊃¬¬∀y.y<y
```

Section 4.327 QUANTIFICATION rules

This is an example of a proof using all the quantification rules.

```
*****DECLARE INDVAR x y; DECLARE INDPAR a b; DECLARE PREDPAR P 2;

*****ASSUME ∀x.∃y.P(x,y)∧∀x y.(P(x,y)⊃P(y,x));

1 ∀x.∃y.P(x,y)∧∀x y.(P(x,y)⊃P(y,x))  (1)

*****∧E 1 1;

2 ∀x.∃y.P(x,y)  (1)

*****∧E 1 2;

3 ∀x y.(P(x,y)⊃P(y,x))  (1)

*****∀E 2 a;

4 ∃y.P(a,y)  (1)

*****∀E 3 a b;

5 P(a,b)⊃P(b,a)  (1)

*****∃E 4 b;

6 P(a,b)  (6)

*****⊃E 5,6;

7 P(b,a)  (1 6)

*****∧I 6 7;

8 P(a,b)∧P(b,a)  (1 6)

*****∃I 8 b←y;

9 ∃y.(P(a,y)∧P(y,a))  (1)

*****∀I 9 a←x;

10 ∀x.∃y.(P(x,y)∧P(y,x))  (1)

*****⊃I 1⊃10;

11 (∀x.∃y.P(x,y)∧∀x y.(P(x,y)⊃P(y,x)))⊃∀x.∃y.(P(x,y)∧P(y,x))
```

Section 4.3271 UNIVERSAL QUANTIFICATION (∀) rules

Introduction rule

∀I (UG) ● , REP1[OPT[ALT[<indvar>|<indpar>] ←] <indvar> , OPT[,]] ;

Several simultaneous universal genalizations on ● can be carried out with this command. For each element of the list (either x or a←x) a new universal quantifier (∀x) is put at the front of ●: (with x for all free occurrences of a in the second case) and a new line of the derivation is created.

Remember there is a restriction on the application of this rule, namely the newly quantified variable must not appear free in any of the dependencies of ●.

In the example ∀I occurs on line 9. There is nothing free in the WFF on line 1 (line 9s only dependency) so the generalization is legal. Notice that the "a" was changed to an "x". "a" cannot be generalized, as it is an INDPAR.

Elimination rule

∀E (US) ● , <termlist> ;

Universal specification uses the terms in the <termlist> to instantiate the universal quantifiers in the order in which they appear. If a particular term is not free for the variable to be instantiated a bound variable change is made and then the substitution is made. The variable created is declared to be an INDVAR of the correct SORT.

Line 4 and 5 of the example were created by this rule.

Section 4.3272 EXISTENTIAL QUANTIFICATION (∃) rules

Introduction rule

```
∃I(EG) ● , REP1[OPT[<term> ←] <indvar> OPT[<occlist>],OPT[,]]  ;
```

The list following ● tells which TERMs are to be generalized. If the optional <term> is present, it is first replaced by <indvar> at each occurrence mentioned in the <occlist>. The WFF on ● is then generalized and the next thing in the list is considered. Notice that no use can be made of an <occlist> if there is no TERM present. The machine will ignore such a list in this case. The dependencies of the conclusion are just those of ●.

<occlist> := OCC <ordernatnumlist>

The <ordernatnumlist> is a list of natural numbers in increasing order.

In the example existential introduction is done on line 9 of the proof. This is the most interesting line of this example. You will note that the dependencies of this line are *not* as described above because of the previous existential elimination. This is explained below.

```
*****:DECLARE PREDCONST F 1;TAUT F(x)∨¬F(x);

****
****
27 F(x)∨¬F(x)

***** ∃I 27,x←y OCC 2;

28     ∃y.(F(x)∨¬F(y))

***** ∀I 28, x;

29     ∀x.∃y.(F(x)∨¬F(y))
```

Elimination rule

```
∃E(ES)  ● ,  REP1[ALT[ <indvar> | <indpar> ],OPT[,]]   ;
```

The implementation of this rule is the most radically different from the formal statement given above. This rule corresponds in informal reasoning to the following kind of argument. Suppose we have shown that something exists with some particular property, e.g. ∃y.P(a,y). Then we say "call this thing b". This is like saying ASSUME P(a,b). Then we can reason about b. As soon as we have a sentence, however, that no longer mentions b, it is a theorem which does not depend on what we called "y" but only on the dependancies of the existential statement we started with. Thus we can eliminate P(a,b) from the assumptions of this theorem and replace them with those of the assumptions of ∃y.P(a,y)

The machine implementation thus makes the correct assumption for you, remembers it and *automatically* removes it at the first legitimate opportunity. Several eliminations can be done at once.

In the example an existential elimination was done creating step 6. This line actually has as its REASON that it was ASSUMEd. Line 8 thus depends on it. When the existential generalization was done on the next line, b no longer appeared and so line 6 was removed from the dependancies of line 9. A user should try to convince himself that this is equivalent to the rule stated at the beginning of this manual.

Section 4.3273 Quantifier rules with SORTs

The following table describes the effect of the quantifier rules in the presence of SORT and MOREGENERAL declarations, such that p is of SORT P, q is of SORT Q and r is of SORT R, and R is MOREGENERAL than Q and Q is MOREGENERAL than P

```
∀E      ∀q.A(q)        ∀q.A(q)      ∀q.A(q)
        -------        -------      -------
         A(p)           A(q)        Q(r)⊃A(r)

∀I       A(q)           A(q)         A(q)
        -------        -------      -------
        ∀p.A(p)        ∀q.A(q)       error

∃E      ∃q.A(q)        ∃q.A(q)      ∃q.A(q)
        -------        -------      -------
         error          A(q)         A(r)

∃I       A(q)           A(q)         A(q)
        -------        -------      -------
      P(q)⊃∃p.A(p)     ∃q.A(q)      ∃r.A(r)
```

As an example, it is possible that you might try to instantiate a variable to a term whose SORT is MOREGENERAL than the quantified variable. In this case the result of the specialization is to create an implication asserting that if the term were of the proper SORT then the specialization holds. If the variable is MOREGENERAL than the term then the usual WFF is returned.

Section 4.33 TAUT and TAUTEQ

TAUTOLOGY rule

```
TAUT  <wff>  , <vllist> ;
```

This rule decides if the WFFs follows as a tautological consequence of the WFFs mentioned in the VLLIST (the notion of VLLIST is defined in Appendix 2). In this case WFF is concluded and its dependencies are the union of the dependencies of each WFF in the VLLIST. We think this algorithm is fairly efficient and thus should be used whenever possible.

TAUTEQ rule

TAUTEQ implements a decision procedure for the theory of equality and n-ary predicates, n>0. Its syntax is the same as the TAUT rule:

```
TAUTEQ <wff> , <vllist> ;
```

This rule decides if WFF follows from the WFFs mentioned in VLLIST in the above-mentioned theory. Thus, anything that can be proven by TAUT can also be proven by TAUTEQ, but TAUTEQ runs more slowly than the TAUT rule.

```
*****DECLARE PREDCONST P 1 Q 1;

*****DECLARE OPCONST f 1;

*****DECLARE INDVAR a b;

*****TAUTEQ a=b⊃(P(a)≡P(b));

1 a=b⊃(P(a)≡P(b))

*****TAUT a=b⊃(P(a)≡P(b));

TOUGH LUCK

*****TAUTEQ a=b⊃f(a)=f(b);

TOUGH LUCK
```

The formula a=b⊃(P(a)≡P(b)) cannot be proven propositionally: TAUT would simply rename (a=b) to a new PREDPAR with ARITY 0, say P1, P(a) to P2, and P(b) to P3, and then try to prove P1⊃(P2≡P3). The formula (a=b)⊃f(a)=f(b) cannot be proven by TAUTEQ since TAUTEQ does not know about the arguments of functions.

Section 4.34 The UNIFY Command

UNIFY <wff> • ;

This command tries to establish whether the WFF is a consequence he VL are

This rule of Inference is best described by first presenting some examples:

```
*****ASSUME ∀X.P(X);
1  ∀X.P(X)
*****UNIFY P(f(0)) 1;
2  P(f(0))
*****UNIFY ∃X.P(X) 1;
3  ∃X.P(X)
```

In step 2, the UNIFY mechanism recognised that P, applied to any TERM followed from ∀X.P(X). More aggressively, on line 3, it recognised the that ∀X.P(X) implies that ∃X.P(X). These are two simple cases of the use of this command. A more complicated example is:

```
*****ASSUME ∃X.∀Y.(P(X)∨Q2(X,Y)) ;
1  ∃X.∀Y.(P(X)∨Q2(X,Y))  (1)
*****UNIFY ∃W.P(W)∨∃W.∀Z.Q2(W,Z) 1;
2  ∃W.P(W)∨∃W.∀Z.Q2(W,Z)  (1)
```

Notice that, in both of the examples above, the propositional structure of WFF was the same as that of the VL. This rule is designed to handle exactly this case; namely, it is designed to handle the quantifier manipulations involved in implications between WFFs with similar propositional forms.

Section 4.35 SUBSTITUTION rule

```
SUBST ▪1 IN ▪2 OPT[ OCC <ordernatnumlist> ] ;
```

If the major connective in ▪1 is = or ≡ then (making allowances for bound variable changes) the occurences of the left hand side of ▪1 which appear in ▪2 will be replaced by the right hand side of ▪1. If an occurrence list appears only those listed will get substituted.

```
SUBSTR ▪1 IN ▪2 OPT[ OCC <ordernatnumlist> ] ;
```

does the same as SUBST but substitutes the left hand side of ▪1 for the right hand side of ▪1 in ▪2.

Ordinarily, f(x) cannot be substituted for y in ∀x.F(x,y) as the x in f(x) would then become bound, i.e. f(x) is not *free for* y in ∀x.F(x,y). FOL automatically handles this conflict of bound variables in a substitution: those occurences of a bound variable which will cause a conflict are changed. Thus, if one tries to substitute f(x) for ∀x.F(x,y) the generated substitution instance will be ∀x1.F(x1,(f(x)). Here the newly created variable will have the same SORT as x if SORTs are being used.

The 'new' variable is created by considering the 'old' variable to have two parts: a prefix which is the identifier up to and including its last alphanumeric character, and an index, either empty or a positive integer. The new variable which is generated will have the same prefix, and an incremented index. For this purpose, an empty index is considered to be '0'.

Section 4.4 Semantic Attachment and Simplification

FOL is concerned with checking theorems in a first-order language, which the user specifies by making declarations. This language is then a structure L=<P,F,C>, where P is a set of predicate symbols, F a set of function symbols, and C a set of constant symbols. A *model* of L is a structure M=<D,P',F',C'>, with D a non-empty set, P' a set of n-ary predicates on D, F' a set of functions mapping D^n into D, and C' a subset of D. An *interpretation* of L in M is a map which specifies which symbols in P correspond to which predicates in M, similarly for F and C. The implementation of semantic attachment has two aspects:
(a) the *attachment* mechanism which allows the user to specify the objects in the model which correspond to symbols in the language and *vice versa*, and
(b) the *simplifier* which tries to compute, in the model, the values of FOL expressions, i.e. it uses the notion of *satisfiability*.

For example, we might associate with function symbols the corresponding LISP functions. The OPCONST '+' might be semantically attached to the LISP function, PLUS, and the INDCONSTs '1' and '2' (i.e. the *numerals*) attached to the *numbers* 1 and 2, so that an evaluation of '1+2' in the model would give the *number* 3 as an answer - the simplifier would then return the INDCONST '3'.

Note carefully that the map from L into M and that from M back to L may be *partial*, i.e. there may be symbols in L which have no defined interpretation in M, and the process of simplification with respect to M may generate objects in M which have no canonical symbol in L. The FOL simplifier simplifies sentences to the maximal possible extent, using the results of computation within the model, as well as any relevant information about the EXTENSION and SORT structures which the user has defined on L.

FOL allows the assignment of arbitrary LISP functions or lambda-expressions as the interpretations of predicate and function symbols.

Section 4.41 The ATTACH command

```
ATTACH OPT[*] ALT[ <predconst> | <opconst> | <indconst> ] <s_expr> ;

        <s_expr>        := ALT[ <atom> | ( <s_exprlist> OPT[<dotend>] ) ];
        <s_exprlist>    := REP1[ <s_expr> ]
        <dotend>        := . <sexpr>
        <atom>          := ALT[ <identifier> | <natnum> ]
```

This command allows for the definition of the maps from the FOL language that the user has defined into the LISP environment which he wishes to take as the model of his language (and *vice versa* if the ATTACH* option taken).

PREDCONSTs and OPCONSTs may be attached either to atoms which are the names of already-defined LISP functions (i.e. ones which have a SUBR, EXPR or MACRO property, including of course all the standard ILISP functions) or legal LISP function, lambda-expression or macro definitions. The attachment mechanism checks that the functions (except SUBRs) being attached have the correct number of arguments corresponding to the ARITY of the PREDCONST or OPCONST to which the attachment is being made. INDCONSTs may be attached to any S-expression.

```
*****DECLARE INDCONST ZERO, ONE ε INTEGER

*****DECLARE OPCONST +(INTEGER,INTEGER)=INTEGER [INF];

*****ATTACH ZERO 0;

ZERO attached to 0

*****ATTACH ONE 1;

ONE attached to 1

*****DECLARE OPCONST CAR CDR(LIST)=LIST;

*****DECLARE OPCONST CONS(SEXPR,SEXPR)=SEXPR;

*****ATTACH CAR CAR;

*****ATTACH CONS CONS;

*****DECLARE INDVAR A B L ε SEXPR;
```

Section 4.42 The SIMPLIFY command

```
SIMPLIFY [ALT <wff> | <vl> | <term> ] ;
```

This command effects the simplification of an FOL sentence by computing within its model, i.e. the simplification mechanism attempts to find, in the model, objects (LISP S-expressions) which correspond to syntactic symbols in the sentence. If any are found, they are EVALuated in the normal way. The simplifier then attempts to find a term in the language which corresponds to this evaluated entity. In the case of VLs and TERMs, the original expression is returned, together with its maximally simplified form: if a term exists in the language for the simplification, then that forms the right hand of the equality. (The simplifier is aware that NATNUMs and LISP numbers correspond to each other). In the case of WFFs, additionally, if the result of simplification is a truth-value, the WFF or its negation is returned, whichever is appropriate. The simplification is carried out to the maximal extent.

If a LISP error is encountered during simplification, an error message is given.

In the model defined by the attachments made above, the following occurs:

```
*****SIMPLIFY ZERO + ONE;

ZERO+ONE=1

*****SIMPLIFY CAR '(A B);

CAR('(A B))=A
```

In addition, the simplification mechanism takes into account any information that is available about the SORT and EXTENSION declarations that have been made. For example, remembering the example on extensions given in section 4.124:

```
*****DECLARE INDCONST BK ∈ BKINGS, WK ∈ WKINGS;

*****DECLARE PREDCONST KINGS 1;

*****EXTENSION BKINGS (BK);

Extension of BKINGS is (BK)

*****EXTENSION WKINGS (WK);

Extension of WKINGS is (WK)

*****EXTENSION KINGS WKINGS U BKINGS;

Extension of KINGS is (WK BK)

*****SIMPLIFY WK=BK;

¬(WK=BK)
```

Section 4.43 Auxiliary FUNCTION definition

```
FUNCTION <function-s_expr> ;
```

This allows the definition of <function-s expr> as an auxiliary LISP function. If the function definition is a legal <s expr> which is not a legal LISP function definition of the DE or DEFPROP sort, an error message will be given.

Section 4.5 Administrative Commands

These commands manipulate the proof checker but do not directly alter the current deduction.

Section 4.51 The LABEL command

```
LABEL  ALT[ <ident> | <ident> = <linenum>]  ;
```

In the first case the next line the proof checker generates will get the label IDENT. **In the second the** LINENUM **mentioned will become labeled by** IDENT. **Labels are alternatives to** VLs **and can be used in any place that the syntax expects them.**

Section 4.52 File Handling commands

Section 4.521 The FETCH command

```
FETCH  <filename> OPT[ FROM <mark1> ] OPT[ TO <mark2> ] ;
```

The FETCH command reads the file <filename>, and executes any FOL commands in this file. FOL accepts standard Stanford file designators. If mark specifications are present, the file is only read within the limits which they specify. The default FROM/TO are the beginning and the end, respectively, of the file. The commands read during a fetch are not printed in the backup file. FETCHes may be nested to a depth of 10.

Section 4.522 The MARK command

```
MARK <token> ;
```

This command has no effect on the proof, but simply places a mark in the file which the FETCH command can use to delimit reading of the file.

Section 4.523 The BACKUP command

```
BACKUP <file name> ;
```

When FOL is initialized, a file called BACKUP.TMP is automatically created. All console input from the user is saved on this file. This command closes the current backup file, and opens a new one with the specified file name.

Section 4.524 The CLOSE command

```
CLOSE ;
```

This closes and reopens the backup file. Normally the backup file is written every five steps in the proof, but this command enables the user to save the state of his deduction at any point.

Section 4.525 The COMMENT command

```
COMMENT <delimiter> <text> <delimiter>
```

When typed at the top-level, this inserts any text between the delimters into the backup file; if it appears in a FETCHed file, the text is ignored. Of course, the delimiter must not appear in the text.

Section 4.53 The CANCEL command

```
CANCEL OPT[ <linenum> ];
```

This cancels all steps of a deduction with LINENUMs greater than or equal to LINENUM. Thus you can remove unwanted steps from a deduction provided they are all at the *end* of the PROOF. If no LINENUM is specified, only the last line is cancelled.

Section 4.54 The SHOW command

The SHOW command is used to display information generated by FOL. The intent of the present command is to allow you to display information about a derivation at the console and save it on a file. The integer after the FILENAME becomes the linelength while this command is active.

```
SHOW <showtype> OPT[                                   <filename> OPT[ <integer> ]] ;

<showtype>  := ALT[ PROOF        OPT[ <rangelist> ] |
                    STEPS        OPT[ <rangelist> ] |
                    AXIOM        OPT[ <axnamlist> ] |
                    DECLARATIONS OPT[ <decinfo>   ] |
                    GENERALITY   OPT[ <geninfo>   ] |
                    LABELS       OPT[ <labelinfo> ] ]

<rangelist> := REP1[<rangespec>,OPT[,]]
<rangespec> := ALT[ OPT[ <linenum> ] : OPT[ <linenum> ] | <linenum> ]
<decinfo>   := REP1[ ALT[ <syntype> OPT[ ( <sort>] |
                          <iolsym>                 |
                          SORTS                    ], OPT[,]]
<geninfo>   := REP1[ <sort>, OPT[,] ]
<labelinfo> := REP1[ ALT[ <label> | <rangespec> ] , OPT[,] ]
```

RANGESPEC may be of the form 23 or 23:65 or :65 or 34: or even :. Its meaning is either a single LINENUM or a range of LINENUMs. If a number stands alone it simply means this number. If there are two numbers separated by a colon, the range is from the first to the second. If numbers do not appear on either side of the colon then the default of 0 or the last line is assumed. An FOLSYM is any declared identifier and show returns its SORT identifier and show returns appropriate syntactic information.

Examples are:

```
****SHOW PROOF 1,2:5,16: FOO.BAZ[SET,RWW] 22;
```

this writes lines 1, 2 to 5, 16 to the last line of the proof onto the file FOO.BAZ[SET,RWW] with a linelength of 22.

```
****SHOW PROOF;
```

displays the proof on the console.

The next example, taken from an actual test file, shows the kind of syntactic information displayed by a "show declarations" command.

```
****SHOW DECLARATIONS EMPTY x + ≤ carry front binaryplus;

EMPTY is INDCONST of sort BYTES

x is INDVAR of sort INTEGER

+ is OPCONST
The domain is INTEGER * INTEGER, and the range is INTEGER[L+650 R+600]

≤ is PREDCONST
The domain is INTEGER * INTEGER[L+350 R+300]

carry is OPCONST
The domain is BYTES * BYTES, and the range is BYTES

front is OPCONST
The domain is BYTES, and the range is BYTES[R+950]

No declaration for binaryplus

****SHOW DECLARATION SORTS;
```

shows all the PREDCONSTs of ARITY 1 (i.e. all of theSORTs)

SHOW commands do the obvious thing in conjunction with the display features turned on by DISPLAY.

Section 4.55 The DISPLAY command

DISPLAY OPT[<displaytype>] ;

```
<displaytype> := ALT[ PROOF          |
                      STEPS          |
                      AXIOM          |
                      ATTACHMENTS    |
                      DECLARATIONS   |
                      LABELS         |
                      STATUS          ]
```

FOL may take advantage of the display features of the Stanford DataDisc system by means of this command.

For example:

```
*****DISPLAY ;
```

creates a display window of full-screen width, into which the steps of the proof are displayed as the derivation continues. The page-printer is restricted to the bottom eight lines of the screen. If the argument is non-null then the 'proof' window is restricted to half-screen width, and a second window, appropriately labelled, occupies the other half of the screen e.g.

```
*****DISPLAY AXIOMS ;
```

causes an 'axiom' window to be opened, and all axioms are printed to that window, rather than to the 'proof' window or the page-printer.

Whatever the current state of the display, 'DISPLAY <null>' causes the 'proof' window to be regenerated, together with the last five lines of the proof, if any. Any other windows which may be present are flushed. This method is slow and cannot be used from teletypes, but provides a much more convenient way of displaying the steps of the proofs and other information.

```
*****UNDISPLAY ;
```

restores the screen to normal teletype mode.

Section 4.56 The EXIT command

EXIT ;

This command returns the user to the monitor in a state appropriate for saving his core-image.

Section 4.58 The SPOOL Command

```
SPOOL <filename> ; XSPOOL <filename> ;
```

These cause the <filename> to be spooled on the appropriate device (LPT or XGP)

Section 4.58 The TTY Command

```
TTY ;
```

This resets the printing routines so that they are teletype rather than display oriented. In this mode, the logical connectives are represented by NOT, OR, & or AND, → or IMP, ↔ or EQUIV, FORALL, EXISTS.

Appendix 1

FORMAL DESCRIPTION OF FOL

The non-descriptive symbols of FOL divide into SYNTYPEs as follows:

1. Individual variables - INDVAR. There are denumerably many individual variable symbols. We use x,y,z as meta-variables for them;

2. Individual parameters - INDPAR. There are denumerably many individual parameter symbols. As meta-variables we use a,b,c;

3. n-place predicate parameters - PREDPAR. For each n there are denumerably many predicate parameter symbols. An n-place PREDPAR is said to have ARITY n;

4. Logical constants:

   a) Sentential constants - SENTCONST: **FALSE** and **TRUE**.
   b) Sentential connectives - SENTCONN: $\neg,\wedge,\vee,\supset,\equiv$.
   c) Quantifiers - QUANT: $\forall$ and $\exists$;

5. Auxiliary signs - AUXSYM: parenthesis (,).

A particular FOL language is distinguished from a pure first order language by declaring certain constant symbols. These have the SYNTYPEs:

1. Individual constants - INDCONST;

2. n-place predicate constants - PREDCONST. Each n-place PREDCONST has ARITY n;

3. n-place operation symbols - OPCONST. Like PREDPARs each has an ARITY. Some authors call OPCONSTs function symbols;

Each SYNTYPE is assumed to be disjoint from all others.

TERMs

t is a TERM in FOL if either
1. t is an INDPAR, INDVAR, or an INDCONST, or
2. t is $f(t_1,t_2,\ldots,t_n)$, where f is an OPCONST of ARITY n and t_i is a TERM.

WFFs

A is an atomic well-formed formula or AWFF if

1. A is one of the symbols "FALSE" or "TRUE".
2. A is $P(t_1,\ldots,t_2)$ where P is a PREDPAR or a PREDCONST of ARITY n.

The notion of well-formed formula or WFF is defined inductively by:

1. An AWFF is a WFF.
2. If A and B are WFFs, then so are $(A\wedge B)$, $(A\vee B)$, $(A\supset B)$, $(A\equiv B)$, and $\neg(A)$.
3. If A is a WFF, then so are $\forall x.A$ and $\exists x.A$ provided that x is an INDVAR.

The usual definitions of free and bound variables apply and can be found in any standard logic text (e.g. *Mathematical Logic* by S.C. Kleene). Below the usual conventions for omitting parentheses will be used.

SUBFORMULAS

The notion of SUBFORMULA is defined inductively

1. A is a SUBFORMULA of A.
2. If $B\wedge C$, $B\vee C$, $B\supset C$, $B\equiv C$, or $\neg B$ is a SUBFORMULA of A so are B and C.
3. If $\forall x.B$ or $\exists x.B$ is a SUBFORMULA OF A, so is $B[t\leftarrow x]$.

The notations $A[t\leftarrow x]$ and $A[t\leftarrow u]$, where A represents a WFF, t, u TERMs and x an INDVAR are used to denote the result of substituting x or u, respectively, for all occurrences of t in A (if any). In contexts where a notation like $A[t\leftarrow x]$ is used, it is always assumed that t does not occur in A within the scope of a quantifier that is immediately followed by x. The notation $A[x\leftarrow t]$, denotes the result of substituting t for all free occurrences of x.

The notation $A[a\leftarrow x,x\leftarrow t]$ means the result of first substituting x for a and then t for x. To denote simultaneous substitution we use $A[a\leftarrow x;x\leftarrow t]$.

Appendix 2

THE SYNTAX OF THE MACHINE IMPLEMENTATION OF FOL

In this manual the syntax of FOL will be described using a modified form of the MLISP2 notion of pattern. These form the basic constructs of the FOL parser.

1. Identifiers which appear in patterns are to be taken literally.
2. Patterns for syntactic types are surrounded by angle brackets.
3. Patterns for repetitions are designated by:

REP0[<pattern>] means 0 or more repeated PATTERNs,
REPn[<pattern>] means n or more repeated PATTERNs.

If a REP0 or a REPn has two arguments then the second argument is a pattern that acts as a separator. So that REP1[<wff> ,] means one or more WFFs separated by commas.
4. Alternatives appear as ALT[<PATTERN1>|...|<PATTERNn>].
ALT[<wff>|<term>] means either a WFF or a TERM.
5. Optional things appear as OPT[<pattern>]

REP2[<wff>,OPT[,]] means a sequence of two or more WFFs optionally separated by commas.

These conventions are combined with the standard Backus Normal Form notation.

Basic FOL symbols

In an attempt to make life easier for users, the FOL parser makes more careful distinctions about the kinds of symbols that it sees than the previous description indicated.

```
<indsym>     := ALT[ <indvar> | <indpar> | <indconst> ]
<indvar>     := <identifier>                        |declared INDVAR
<indpar>     := <identifier>                        |declared INDPAR
<indconst>   := ALT[ <id[illegible]       > |       |declared INDCONST
                     [illegible]              ]     |no declaration necessary

<opsym>      := ALT[ <oppar>   | <opconst> ]
<oppar>      := <identifier>                        |declared OPPAR
<opconst>    := <identifier>                        |declared OPCONST
<preop>      := <opsym>                             |ARITY 1 and declared PREFIX
<infop>      := <opsym>                             |ARITY 2 and declared INFIX
<applop>     := <opsym>                             |ARITY n and not declared
                                                    |  INF or PRE dec

<predsym>    := ALT[ <predpar> | <predconst> ]
<predpar>    := <identifier>                        |declared PREDPAR
<predconst>  := <identifier>                        |declared PREDCONST
<prepred>    := <predsym>                           |ARITY 1 and declared PREFIX
<infpred>    := <predsym>                           |ARITY 2 and declared INFIX
<applpred>   := <predsym>                           |ARITY n and not declared
                                                    |  INF or PRE dec

<sentsym>    := ALT[ <sentpar> | <sentconst> ]
<sentpar>    := <identifier>                        |declared SENTPAR
<sentconst>  := ALT[ FALSE          |
                     TRUE           |
```

```
                  <identifier>    ]                 ;declared SENTCONST
                                                    ;  INF or PRE dec

<sentconn>  :=  ALT[  ¬  | NOT |                    ;negation
                      ∨  |  OR |                    ;disjunction
                      ∧  |  &  |  AND    |          ;conjunction
                      ⊃  |  →  |  IMP    |          ;implication
                      ≡  |  ↔  |  EQUIV  ]          ;equivalence

<prelog>    :=  ALT[ ¬ | NOT ]
<inflog>    :=  ALT[ ∨ | OR | ∧ | & | AND | ⊃ | → | IMP | ≡ | ↔ | EQUIV ]

<quant>     :=  ALT[ ∀ | FORALL | ∃ | EXISTS ]
```

TERMs

The FOL syntax for TERMs allows for both prefix operators and binary infix operators, as well as the usual function application notation. Any undeclared identifier can be declared an operation constant (OPCONST) using the **DECLARE** command. With proper declaration the following are TERMs:

```
        f(x+-y,g(x*y+z))
        CAR
        car(x,y)
        {ROBOT,BOX1,DOOR}∪{y|∀x.P(g(x,y))}
        powerset(<A,B,C>)

<term> :=  ALT[  <indsym>            |
                 <appterm>           |
                 <prefixterm>        |
                 <infixterm>         |
                 <setterm>           |
                 <n_tupleterm>       |
                 <compterm>          |
                 ( <term>  )         ]
<appterm>        :=  <applop> (  <termlist>  )
<prefixterm>     :=  <preop> <term>
<infixterm>      :=  <term>  <infop>  <term>
<setterm>        :=  {  <termlist>  }
<n_tupleterm>    :=  <  <termlist>  >
<compterm>       :=  {  <indvar> | <wff>  }

<termlist>       :=  REP1[ <term> , OPT[,] ]
```

These are illustrated above and may be used at any time. Other additions may occur from time to time.

Of course, the appropriate restrictions on the SORTs of the arguments of the OPSYMs must be met.

AWFFs

AWFFs are formed similarly, but cannot be nested.

```
<awff> := ALT[ <basawff>    |
               <applawff>   |
               <preawff>    |
               <infawff>    ]

<basawff>   :=  ALT[ <sentsym> |
                     <predpar> ]          ;with ARITY 0
<applawff>  :=  <applpred> ( <termlist> )
<preawff>   :=  <prepred> <term>
<infawff>   :=  <term> <infpred> <term>
```

Examples of AWFFs are

```
{A,B,W}⊂{X|∃Z.W⊂Z∧Z⊂X}
<a,b> = {a,{a,b}}
f(x,y)= 'car(cons(x,y))
```

Equality is treated as any other predicate constant, but the system knows about the substitution of equals for equals. It does not know that A≠B is usually interpreted as ¬(A=B), but treats it as any other predicate symbol.

WFFs

```
<wff> := [ALT <standard first order logic formula> |
              <vl> : [OPT <subpart>] [OPT <subst_oper>] ]
```

The syntax for WFFs allows the following abbreviations and options.

The primitive logical symbols are:

```
<wff>          :=  ALT[ <primwff> | <prewff> | <infwff> ]

<primwff>      :=  ALT[ <awff> | <quantwff> | ( <wff> ) ]
<prewff>       :=  <prelog> <primwff>
<infwff>       :=  <primwff> <inflog> <primwff>
<quantwff>     :=  <quantprefix> <smallwff>
<quantprefix>  :=  ALT[    <quant> REP1[ <indvar> ] . |
                         ( <quant> REP1[ <indvar> ] ) ]
<smallwff>     :=  REP0[ <prelog> ] <primwff>
```

Parentheses may be omitted and then association is to the right. As is usual conjunction binds the strongest, followed by disjunction, implication and equivalence. Negation, as well as both quantifiers, bind to the shortest WFF on their right. Thus $\forall x.P(x) \supset P(x)$ will parse as $(\forall x.P(x)) \supset P(x)$ not as $\forall x.(P(x) \supset P(x))$!

We can write adjacent quantifiers of the same type together, so ∀x.∀y.P(x,y) can be written ∀x y.P(x,y). FOL also accepts (∀x)(∀y)P(x,y) or (∀x y)P(x,y) for ∀x.∀y.P(x,y).

Subparts of WFFs and TERMs

Within a deduction there is a completely general way of specifying any subpart of any TERM or

WFF already mentioned. We accomplish this by means of a SUBPART designator. Derivations consist of WFFs, each of which has a LINENUM. The WFF which appears on this line is designated by following it with a colon. If

```
10.     ∀x y.(P(f(x))⊃Q(h(x,y)))
```

is line 10 of some derivation then 10: represents the WFF on that line, i.e. ∀x y.(P(f(x))⊃Q(h(x,y))). Furthermore, subparts of such a WFF can be designated by a SUBPART designator.

```
<subpart> := REP1[ # <integer> ]
```

The integer denotes which branch of the subpart tree you wish to go down. Quantified formulas and negations have only one immediate subpart, called #1. The other sentential connectives each have two. For predicates and function symbols the number of immediate subparts is determined by their ARITYs. Any conflict with these will produce an error. Thus

```
10:#1          ≡   ∀y.(P(f(x))⊃Q(h(x,y)))
10:#2          ≡   ERROR
10:#1#1#2#1    ≡   h(x,y)
10:#1#1#1#2    ≡   ERROR   (P has ARITY 1).
```

<u>Substitutions in WFFs and TERMs</u>

Once you have named a WFF, you can use a substitution operator to perform an arbitrary substitution.

```
<subst_oper> := [  REP1[<substlist1>,OPT[ ;]]  ]
<substlist1>  :=  ALT[ <term> ← <term> | <wff> ← <wff> ]
```

Examples:

```
10:#1[x←ROBOT]  ≡  ∀y.(P(f(ROBOT))⊃Q(h(ROBOT,y)))
10:#1#1[((x)←ROBOT;Q(h(x,y))←P(x))  ≡  P(ROBOT)⊃P(x)
10:#1#1#1#1[f(10:#1#1#2#1#1)←ROBOT) ≡  ROBOT
10:#1[x←f(y)] ≡  ∀y1.(P(f(f(y)))⊃Q(h(f(y),y1))).
```

Note: the substitution operator changed the bound variable in the last example. This prevented the y in f(y) from becoming bound. See section on substitutions.

WFFs and TERMs thus have the following alternative syntax:

```
<wff>   := <vl> : OPT[ <subpart>  OPT[ <subst_oper> ]]

<term>  := <vl> : OPT[ <subpart>  OPT[ <subst_oper> ]]
```

There is an ambiguity as SUBPART may produce only a WFF where a TERM is necessary (or the other way around). FOL checks for this and will not allow a mistake. Such a subpart designator can be used whenever the syntax calls for a WFF or TERM.

Another label for handling well-formed expressions is the VL

```
<vl> := ALT[ <integer> | <label> OPT[ALT[ +|-] <integer>]  |
                         <axref> | REP1[-] ]
```

The optional + or - <integer> after a label designates an offset from the mentioned label by the amount designated.

The last alternative has not been previously mentioned. Its meaning is the n-th previous line, where n is the number of "-" signs.

Appendix 3

AXIOMS FOR ZERMELO FRAENKEL SET THEORY

The axioms presented here and in appendix 4 are examples of the expression in FOL of the conventional Zermelo-Fraenkel and Goedel-Bernays-von Neumann set theories. We believe that the practical use of set theory for mathematical and computer science proofs will require an extended *practical* system.

```
DECLARE PREDCONST ∈ 2[INF];
DECLARE PREDCONST ⊂ 2[INF];
DECLARE OPCONST ∪ 2[INF];
DECLARE INDVAR r s t u v w x y z;
DECLARE PREDPAR A 2 B 1;

AXIOM ZF:
       EXT:    ∀x y.(∀z.(z∈x≡z∈y)≡x=y);                 % Extensionality
       EMT:    ∃x.∀y.¬y∈x;                              % Null set
       PAIR:   ∀x y.∃z.∀w.(w∈z≡w=x∨w=y);                % Unordered pair
       UNION:  ∀x.∃y.∀z.(z∈y≡∃t.(z∈t∧t∈x));             % Sum set
       INF:    ∃x.(0∈x∧∀y.(y∈x⊃(y∪{y})∈x));             % Infinity
       REPL:   ∀x.∃y ∀z.(A(x,z)≡y=z) ⊃                  % Replacement
               ∀u.∃v.(∀r.(r∈v ≡ ∃s.(s∈u∧A(s,r))));
       SEP:    ∀x.∃y.∀z.(z∈y≡z∈x∧B(z));                 % Separation
       POWER:  ∀x.∃y.∀z.(z∈y≡z⊂x);                      % Power set
       REG:    ∀x.∃y.(x=0∨(y∈x∧∀z.(z∈x⊃¬z∈y)));;        % Regularity

        % Replacement is equivalent to                                    %
        %       ∀x.(∃y.A(x,y)∧∀y z.(A(x,y)∧A(x,z)⊃y=z)) ⊃                  %
        %             ∀u.∃v.(∀r.(r∈v ≡ ∃s.(s∈u∧A(s,r))))                  %
        % or    ∀x.∃!y.A(x,y) ⊃ ∀u.∃v.∀r.(r∈v ≡ ∃s.(s∈u∧A(s,r)))           %

        % Separation is a consequence of and weaker than replacment.      %

% Definitions %
DECLARE PREDCONST FUN 1,INTO 2,PSUBSET 2[INF];
DECLARE       OPCONST rng 1 dom 1;

AXIOM
        SUBSET:        ∀x y.(x⊂y≡∀z.(z∈x⊃z∈y));
        PROPSUBSET:    ∀x y.(PSUBSET(x,y)≡x⊂y∧¬x=y);
        PAIRFUN:       ∀x y z.(z∈{x,y}≡z=x∨z=y);
        UNITSETFUN:    ∀x.( {x}={x,x} );
        OPAIRFUN:      ∀x y.( <x,y>={{x},{x,y}} );
        FUNCTION:      ∀w.(FUN(w)≡∀z.(z∈w⊃∃x y.(z=<x,y>))∧
                          ∀x y z.(<x,y>∈w∧<x,z>∈w⊃y=z) );
        DOMAIN:        ∀w x.(x∈dom(w)≡FUN(w)∧∃y z.(y∈w∧y=<x,z>));
        RANGE:         ∀w x.(x∈rng(w)≡FUN(w)∧∃y z.(y∈w∧y=<z,x>));
        INTO:          ∀w x.(INTO(w,x)≡rng(w)⊂x);
        UNION:         ∀x y z.(z∈x∪y≡z∈x∨z∈y);;
```

Appendix 4

AXIOMS FOR GOEDEL-BERNAYS-VON NEUMANN SET THEORY

```
MOSTGENERAL Class;
DECLARE PREDCONST Class Set 1;
DECLARE PREDCONST ε(Class,Class)[INF];
DECLARE PREDCONST ⊂(Set,Class)[INF];
DECLARE INDVAR A B C ε Class,x y u v w ε Set;
DECLARE PREDCONST Empty OneMany(Class),Disjoint(Class,Class);

AXIOM NGB:

    KLASS:          ∀x.Class(x);
    ISSET:          ∀A B.(AεB⊃Set(A));
    EQUAL:          ∀A B C.((CεA≡CεB)≡A=B);
    EMPTY:          ∃x.∀y.¬yεx;
    PAIRS:          ∀x y.∃u.∀v.(vεu≡v=x∨v=y);

  CLASS:
          EPI:      ∃A.∀u v.(<u,v>εA≡uεv);
          INT:      ∀A B.∃C.∀u.(uεC≡uεA∧uεB);
          COMP:     ∀A.∃B.∀u.(uεB≡¬uεA);
          PROJ:     ∀A.∃B.∀u.(uεB≡∃v.<u,v>εA);
          PROD:     ∀A.∃B.∀u v.(<u,v>εB≡uεA);
          CONV:     ∀A.∃B.∀u v.(<u,v>εB≡<v,u>εA);
          TRI1:     ∀A.∃B.∀u v w.(<u,v,w>εB≡<v,w,u>εA);
          TRI2:     ∀A.∃B.∀u v w.(<u,v,w>εB≡<u,w,v>εA);;
  SET:
          INF:      ∃u.(¬Empty(u)∧∀v.(vεu⊃∃w.(wεu∧¬v=w∧vεw)));
          UNION:    ∀u.∃v.∀w x.(wεx∧xεu⊃wεv);
          POWER:    ∀u.∃v.∀w.(w⊂u⊃wεv);
          REPL:     ∀u A.(OneMany(A)⊃∃v.∀w.(wεv≡∃x.(xεu∧<w,x>εA)));

  FND:              ∀A.(¬Empty(A)⊃∃u.(uεA∧Disjoint(u,A)));

  AC:       ∃A.(OneMany(A)∧∀u.(¬Empty(u)⊃∃v.(vεu∧<v,u>εA)));;
```

Appendix 5

INTUITIONISTIC MODAL LOGICS

<u>Modal Logic</u>:

The best known modalities are the so called *'alethic'* ones, involving *necessity(N)* and *possibility(M)*: but many other sentential operators which display modal characteristics have been studied, e.g. C for causality (Burks,1951), K and B for knowledge and belief (Hintikka,1962), P for perception (Hintikka,1969). These latter modalities are the subject of intensive research in logic at the moment, and a comprehensive semantics has been evolved for some of them (Kripke,1964, Hintikka,1969). There are still many difficult problems, especially in the case of quantification into modal contexts, where the traditional rules of substitutability of equivalents and of existential generalization do not seem to hold. This has led to a reformulation of many ontological notions in quantification theory(see, for example, (Hintikka,1955) and (Follesdal,1968).

(Note that modal operators are sentential operators of a rather special kind, not PREDICATES. It is not possible to regard modal operators as applying to names of sentences or formulae without losing the powerful semantics.(see, for example (Montague, 1963))

In the current implementation, the user may define non-standard modal systems and operators. Lewis S4 and S5, Hintikka's KBK and KBB(*op.cit.*) are already available, together with the operators N(necessarily), M(possibly), K(knows), B(believes).

(a) The Classical Systems T, S4 and S5

von Wright's system, T (von Wright,1951) is got from LPC by adding:

A5: $N.p \supset p$

A6: $N.(p \supset q) \supset (N.p \supset N.q)$

Lewis's system S4 (Lewis&Langford,1932) is got from T by adding:

A7: $N.p \supset NN.p$

Lewis's S5 by adding:

A8: $M.p \supset NM.p$

(b) Natural Deduction Systems of Modal Logic

(1) These are based on minimal, classical and intuitionistic logics;

(2) A formula is said to be *modal* if its principal sign is a modal sentential operator;

(3) Necessity systems:

Prawitz has two inference rules for S4:

```
NI)    a           NE)   N.a
     ......             .......
      N.a                  a
```

and a corresponding deduction rule for NI, when the proof or deduction of 'a' depends *only* on modal formulas.

In S5, N.a⊃a may be inferred also when every formula in the dependency set is either a modal formula or the negation of a modal formula. begin indent 5.0 (4) Possibility systems:

The possibility operator, M, may be added by means of the rules

```
MI)    a           ME)   M.a & b
     ......              .......
      M.a                   b
```

When these rules are added, the deduction rule for NI must be modified to be similar to the rule ME.

In the classical Lewis systems, M and N may be interdefined, e.g. M.a⊃¬N¬.a and N.a ⊃ ¬M¬.a, but in the Prawitz system this is not possible.

The syntax for modal formulae is identical to that of standard formulae, except that WFFs may be preceded by 1 or more modal operators(and imbedded ¬), followed by a '.'. So a period

```
<modalwff>       ::= <modalprefix> <primwff>
<modalprefix>    ::= <identifier> '.
```

For example, NMN−MMNNMNMNM.A and ∀x.M.P(x)⊃MM.p(x) are well-formed.

When scanning for modal formulae is turned on using the 'THEORY' command (see Section 4.13), the following rules then become available:

NECI <line number>, NECE <line-number>
POSSI <line-number>, POSSE <line-number>

as defined by the conditions above.(Note carefully the dependency restrictions)

Bibliography.

Burks, A.W.(1951) 'The logic of causal propositions', *MIND, 60:363-82*

Floyd, R. (1963) 'Assigning meanings to programs' in (J.T.Schwartz,ed.) *Proceedings of a symposium in applied mathematics, vol 19* (New York: American Mathematical Society)

Follesdal, D.(1968) 'Knowledge, Identity and Existence', *Theoria, 33:1*

Hayes, P.J.(1974) 'Some problems and non-problems in representation theory' in *Proceedings AISB Conference, Sussex, England*

Hintikka, J.(1955) 'Form and content in quantification theory', *Acta Phil.Fennica, 8:7*

Hintikka, J.(1962) '*Knowledge and Belief - an introduction to the logic of the two notions*', (Ithaca: Cornell U.P.)

Hintikka, J.(1969) '*Models for Modality*', (New York: D.Reidel)

Kreisel, G.(1971a) 'Five notes on the application of proof theory to computer science', *Stanford University IMSSS Technical Report 182*

Kreisel, G.(1971b) 'A survey of proof theory,II' in (J.E.Fenstad,ed.) *Proceddings of the Second Scandinavian Logic Symposium*,(Amsterdam: North-Holland)

Kripke, S.A.(1964) 'Semantical considerations on modal logic', *Acta Phil.Fennica, 16:83*

Lewis, C.I. & Langford, C.(1932) '*Symbolic Logic*', (New York: Dover)

Manna, Z. (1974) *Mathematical Theory of Computation*, (New York: McGraw-Hill)

McCarthy, J.(1963) 'A basis for a mathematical theory of computation', in *Computer Programming and Formal Systems*, (Amsterdam: North-Holland)

McCarthy, J. & Hayes, P.J.(1969) 'Some Philosophical Problems from the viewpoint of Artificial Intelligence', in (D.Michie,ed.) *Machine Intelligence,4* (Edinburgh: Edinburgh U.P.)

Montague, R.(1963) 'Syntactical treatments of modality', *Acta Phil.Fennica, Symposium on modal and many-valued logics*.

Prawitz, D.(1965) '*Natural Deduction - a proof-theoretical study*', (Stockholm : Almqvist & Wiksell)

Sandewall, E.(1970) 'Representing Natural-language information in predicate calculus', *Stanford A.I.Memo 128*

von Wright, G.H.(1951) '*An Essay on Modal Logic*',(Amsterdam: North-Holland)