**Stanford Artificial** Intelligence Laboratory
Memo AIM-235 1

July 1077

**Computer Science Department**
**Report No. STAN-CS-77-432**

# A Users Manual for FOL

**by**

**Richard W. Weyhrauch**

Research sponsored by

Advanced Research **Projects** Agency

COMPUTER SCIENCE DEPARTMENT
Stanford University

STANFORD ARTIFICIAL INTELLIGENCE LABORATORY
MEMO AIM-235.1
STANFORD COMPUTER SCIENCE DEPARTMENT
REPORT NO. STAN-CS-76-432

August 14, 1977

by
Richard W. Weyhtauch

Abstract:

This manual explains how to use of the proof checker FOL, and supersedes all previous manuals. FOL checks proofs of a natural deduction style formuiation of first order functional calculus with equality augumented in the following ways:

(i) it is a many-sorted first-order logic in which a partiai order over the sorts may be specified;
(ii) conditional expressions are allowed for forming terms
(iii) axiom schemata with predicate and function parameters are allowed
(iv) purely propositional deductions can be made in a single step;
(v) a partial model of the language can be built in a LISP environment and some deductions can be made by direct computation in this model;
(vi) there is a limited ability to make metamathematical arguments;
(vii) there are many operational conveniences.

A major goal of FOL is create an environment where formal proofs can be carefully examined with the eventual aim of designing practical tools for manipulating proofs in pure mathematics and about the correctness of programs. This includes checking proofs generated by other programs. FOL is also a research, tool in modeling common-sense reasoning including reasoning about knowledge and belief.

We are grateful to the many people who have contributed to this project. They include John McCarthy, Bill Glassmire, Ashok Chandra, Chris Goad, Bob Filman, Juan Bulnes, Andrew Robinson and Arthur Thomas.

The research described here was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense under contract MDA903-76-C-0206.

CONTENTS

1 SOME INTRODUCTORY REMARKS . . . . . . . . . . 1

2 WHAT IS AN FOL LANGUAGE? . . . . . . . . . . 2

3 TERMS, AWFFS AND WFFS . . . . . . . . . . 4

3.1 TERMs . . . . . . . . . . 4

3.2 AWFFs . . . . . . . . . . 4

3.3 WFFs . . . . . . . . . . 4

4 PROOFS USING FOL . . . . . . . . . . 6

4.1 An FOL proof of $((P \supset Q) \wedge (P \supset R)) \supset (P \supset Q \wedge R)$ . . . . . . . . . . 6

5 THE COMPUTER PROGRAM FOL . . . . . . . . . . 9

5.1 How to run FOL at Stanford . . . . . . . . . . 9

5.2 General information on the features of FOL . . . . . . . . . . 10

5.2.1 Individual symbols . . . . . . . . . . 10

5.2.2 Prefix and Infix notation . . . . . . . . . . 10

5.2.3 Extended notion of TERMs . . . . . . . . . . 10

5.2.4 The Equality of WFFs . . . . . . . . . . 10

5.2.5 VLs and subparts of WFFs and TERMs . . . . . . . . . . 10

5.2.6 SORTs . . . . . . . . . . 11

5.2.7 Semantic Attachment . . . . . . . . . . 11

5.2.8 Syntactic Simplification . . . . . . . . . . 11

5.2.9 Decision procedures . . . . . . . . . . 11

6 LANGUAGE SPECIFICATION . . . . . . . . . . 12

6.1 Declarations . . . . . . . . . . 12

6.2 SORT manipulation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

6.2.1 Default SORT declarations. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

6.2.2 MOREGENERAL declaration. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

6.2.3 EXTENSION declarations. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

7 THE GENERATION OF NEW DEDUCTION STEPS . . . . . . . . . . . . . . . . . . . . . . . . . . 17

7.1 Axioms . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

7.1.1 Using axioms as axiom schemas . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18

7.2 Assumptions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19

**7.3 Basic introduction and elimination rules** . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19

7.3.1 Summary of the basic rules. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20

7.3.2 AND (∧) rules . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

7.3.3 OR (∨) rules . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

7.3.4 IMPLIES (⊃) rules . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25

7.3.5 FALSE (FALSE) rules . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26

7.3.6 NOT (¬) rules . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

7.3.7 EQUIVALENCE (≡) rules . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

7.3.8 Quantification rules . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30

7.3.8.1 Quantification example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30

7.3.8.2 UNIVERSAL QUANTIFICATION (∀) rules . . . . . . . . . . . . . . . . . . . . . . . . . 31

7.3.8.3 EXISTENTIAL QUANTIFICATION (∃) rules . . . . . . . . . . . . . . . . . . . . . . . 32

7.3.8.4 Quantifier rules with SORTs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

7.4 The TAUT and TAUTEQ commands . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36

7.5 The QUANT Command . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

7.6 The DISTRIB command . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

7.7 The SUBSTITUTION command . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40

7.8 The MONADIC command 41

7.9 Semantic Attachment and Simplification 42

7.9.1 A technical explanation 43

7.9.2 Declaring representation names 45

7.9.3 The ATTACH command 45

7.9.4 Setting default representations 46

7.9.5 The SIMPLIFY command 48

7.9.6 Auxiliary FUNCTION definition 48

7.10 syntactic simplification 49

7.10.1 Making a simplification set 51

7.10.2 Example of syntactic simplification 53

8 ADMINISTRATIVE COMMANDS 56

8.1 The LABEL command 56

8.2 File Handling commands 56

8.2.1 The FETCH command 56

8.2.2 The MARK command 56

8.2.3 The BACKUP command 57

8.2.4 The CLOSE command 57

8.3 The COMMENT command 58

8.4 The CANCEL command 58

8.5 The SHOW command 58

8.6 The EXIT command 59

8.7 The TTY and UNTTY commands 60

8.8 The SPOOL Command 60

Appendix A FORMAL DESCRIPTION OF FOL 61

Appendix B THE SYNTAX OF THE MACHINE IMPLEMENTATION OF FOL . . . . . . . . . . . . . . . . . . . . . . . . 63

BIBLIOGRAPHY 68

## Section 1 SOME INTRODUCTORY REMARKS

FOL is a computer program which checks derivations in an arbitrary first order language. This sounds very technical but it simply means that there are restrictions on the language that we use to write sentences. A description of the allowable ones is given in the following sections. In this section I briefly describe how FOL is *used.* Examples of proofs are given in sections 4.1 and 3.3.8.

FOL can be used in two ways. Proofs can be done interactively using the computer to check each step or commands may be written on a file and processed when FOL reads the file. Usually both modes are used during the same proof. The principal content of this manual is a description of the commands that FOL accepts.

The checking of a proof has several parts. First, the particular language you are going to use must be specified to FOL. This is accomplished by the declaration commands. These have three functions: they specify which identifiers are to be the different kinds of syntactic elements of your language, they describe part of the sorting mechanism, and they tell the scanner about infix operators and binding powers. The details are found in the section on declarations.

After you have specified a language, FOL can read sentences (usually called well formed formulas or WFFs). The first WFFs normally read into FOL are the axioms of the theory you are considering. For example, if you are interested in set theory you might fetch the file KELLEY.AX[AX,RWW]. It contains all the declarations and axioms for Kelley's version of set theory [Kelley 1955]. Of course you are free to make up any system of axioms you want. Notice FOL will not check whether your axioms are consistent; it only checks the correctness of the derivations you make. After you read in (or type at the console) the axioms of your theory, you are ready to check a proof.

The rules of inference of FOL allow you to generate new proof steps from those you already have. The basic set of rules consists of an introduction and an elimination rule for each of the logical connectives and each of the quantifiers. There are also other commands, like TAUT and TAUTEQ which combine some of these basic rules into powerful techniques for producing new proof steps. The basic rules are an implementation of a system of first order logic called natural deduction [Prawitz 1965].

For the new user of FOL a good place to begin reading this manual is section 4.1. There it gives some examples of FOL proofs and some complete dialogues with the program. Other more extensive examples can be found in Filman and Weyhrauch [1976]. The primer can be thought of as a companion volume to this manual, as it contains extensive examples and lots of hints on actually using FOL. This manual (I hope) has a correct and fairly complete description of the facilities of FOL. In addition it contains a detailed description of the syntax of its commands. A description of how to run the FOL program at the Stanford Artificial Intelligence Laboratory is found in section 5.1.

*The metamathematical notions mentioned will be refered 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 2 WHAT IS AN FOL LANGUAGE?

The FOL user specifies a first-order language by making a set of DECLARATIONs (see Section 6.1). The proof-checking system then generates a proof checker and a collection of rules specific to that language.

DECLARATIONs in FOL are similar to declarations in a programming language in that they introduce symbols and tell how they will subsequently be used both syntactically and semantically. FOL does not yet have a block structure so that all declarations are permanent. When block structure allowing declarations local to a block is added, the idea that declarations determine a first order language will have to be modified.

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

I. logical constants:

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

2. sets of variable symbols:

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

3. 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

4. sets of constant symbols:

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

In addition one can

5. declare a PREDCONST P to be a SORT. This means that its ARITY is one and that something has property P, i.e. ∃x.P(x).

6. restrict a symbol to belong to some SORT.

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

8. specify the range and domain of OPCONSTs to range over particular SORTs.

These last four facilities allow the FOL user to talk about different kinds of objects; just as he can' in informal proofs. Consider integers and even integers. By 5 above these can be thought of as two SORTs of objects. 6 allows us to say that all even integers are integers. '7 can be used to declare that plus is a function from integers to integers and therefore from even integers to integers (by 6). Using 5 we can express the result that the sum of two even integers is an even integer (and so by 6 also an integer). The FOL notation for such assertions is given in section 62.3 on SORTs.

## Section 3 TERMS, AWFFS AND WFFS

### Section 3.1 TERMs

t is an FOL TERM if either

1. t is an INDPAR, INDVAR, or an INDCONST, or
2. t is $f(t_1,t_2,...,t_n)$, where f is an OPCONST of ARITY n and $t_i$ is a TERM, or
3. t is (IF A THEN $t_1$ ELSE $t_2$), where A is a WFF and $t_1,t_2$ are TERMs.

### Section 3.2 AWFFs

A is an atomic well-formed formula or AWFF if

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

### Section 3.3 WFFs

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

1. An AWFF is a WFF.
2. If A, B and C are WFFs, then so are:
   $(A \wedge B)$, $(A \vee B)$, $(A \supset B)$, $(A \equiv B)$, $\neg(A)$ and (IF A THEN B ELSE C).
3. If A is a WFF, then so are ∀x. A and ∃x. A provided that x is an INDVAR.

The main symbol or mainsym of a' WFF of the form $(A \wedge B)$, $(A \vee B)$, $(A \supset B)$, $(A \equiv B)$, $\neg(A)$, ∀x. A and ∃x. A is $\wedge$, $\vee$, $\supset$, $\equiv$, $\neg$, ∀, ∃ respectively. The scope of some occurrence of a SENTCONN or a QUANT in a WFF A is that part of A which has this occurrence as its mainsym. An occurence of an INDVAR x in a WFF A, is *bound* or free according as the occurrence belongs or does not belong to the scope of a QUANT that is immediately followed by an x.

The above notations are entirely conventional in mathematical logic except for the conditional expression (IF A THEN $t_1$ ELSE $t_2$). Its value as a term is that of $t_1$ if A is true and that of $t_2$ otherwise. The notation is eiiminable, but it makes the description of computable functions much more straightforward.

The notations $A[t \leftarrow x]$ and $A[t \leftarrow u]$, where A is 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 $A[a \leftarrow x; x \leftarrow t]$.

In FOL there are many ways of referring to **WFFs** and **TERMs** which already appear in a proof. The syntax for these constructs is found in Appendix **B.**

## Section 4 PROOFS USING FOL

An FOL *derivation* is a sequence of proof steps each of which is a valid *consequence* of the collection of facts already asserted. We refer to facts within the context of a given derivation as VLs. Each VL has a name which specifies a WFF W as well as information as to how W came to be part of this particular derivation. Three different types of names for VLs are LINENUMs, LABELs and AXIOM n a mes.

Each RULE listed below has the following form. It takes some set of WFFs and VLs and produces a new step. The LINENUM of this step is the name of this VL and can be used to refer to it.

A derivation starts by making some ASSUMPTIONs or stating AXIOMs and then using the RULEs of inference to generate new steps. We now give an examples to show the structure of FOL proofs. Other proofs can be found throughout the manual. Section 7.3.8 is an example using all of the quantifier rules.

In this and all succeeding-sections examples of interactions with the computer will appear indented. Those l i nes which are typed by the user wi l l be preceeded by f ive stars "*****" and appear in the same font as this sentence. The lines typed by the computer will appear like this.

### Sect ion 4.1 An FOL proof of ((P⊃Q)∧(P⊃R))⊃(P⊃Q∧R)

Below is a proof of the propositional tautology: ((P⊃Q)∧(P⊃R))⊃(P⊃Q∧R). It would usually be done in a single step using the TAUT command (see section 7.4) but is included here to illustrate the use of FOL.

The proof shows that if P implies Cl and P implies R, then P implies Q∧R. The informal argument goes as follows: suppose we know (P⊃Q) n (P⊃R) then we know both P⊃Q and P⊃R. So if'we assume P we can conclude both Q and R, i.e. Q∧R. Therefore from P⊃Q and P⊃R we can conclude P⊃(Q∧R), dropping our assumption of P. Finally we conclude ((P⊃Q)∧(P⊃R)⊃(P⊃(Q∧R))) without any assumptions at all. The FOL proof is written below. Please look at this proof carefully as it is in this section that a detailed description of what FOL prints and what it means is most clearly explained. One way to follow this proof is to actually try it on the computer. How to do this is explained in section 5.1.

```
    *****DECLARE SENTCONST P Q R;
```

This specifies the FOL language we are using has three SENTCONSTs, P, Q and R. Making declarations is essential. Failure to declare an identifier is the most common reason for a syntax error. The second set of five stars is the FOL prompt "character". It means that it understood your last command and it is waiting for you to type more. If you make an error it attempts to say what it thinks is wrong. Don't worry, you can't break it by making errors.

```
    *****ASSUME (P⊃Q)∧(P⊃R);
```

```
1 (P⊃Q)∧(P⊃R) (1)
```

This step says assume I know ((P⊃Q)∧(P⊃R)). FOL responds by printing a LINE. Each LINE typed by the computer contains: I) 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. Note that an assumption only depends on itself. The LINENUM 1 is the VL, or the name for that LINE of the proof.

```
*****∧E 1 1;

2 P⊃Q (1)
```

This is an example of the RULE *AND elimination*. The "∧E" is the rule name. The "1" after the rule name is the VL 1, i.e. the first LINE of the proof. It is the VL that the rule applies to. The second "1" says conclude the first conjunct. All together this command reads do an *and elimination* on line one of the proof picking the first conjunct. FOL then creates a new LINE, which it labels 2, and which, asserts the first conjunct of LINE one. Note that the VL 1 appears in the list of dependencies.

```
*****∧E 1 ↑:#2;

3 P⊃R (1)
```

This is another example of *AND elimination*. It asserts the second conjunct of LINE one. The syntax used is an alternative to the one above and is included here to introduce you to FOL subpart designators. They are explained in detail in Appendix B. The ↑ is a special label for LINENUMs. It means two LINEs from the end of the proof. Similarly for any other number of up arrows. There is more use of this construct in the proofs below. The colon following the "↑" is one of the most important concepts in FOL. It can be thought of as a function on VLs which retrieves the WFF associated with the VL. ↑: is the same as 1: is the same as ((P⊃Q)∧(P⊃R)). Any VL followed by a : is a WFF and NOT a VL. WFFs cannot be used where VLs are expected. This distinction is also explained in appendix B.

```
*****ASSUME P;

4 P (4)

*****⊃E ,↑;

5 Q (1 4)

*****⊃E ↑,↑;

6 R (1 4)

*****∧I 5 6;

7 Q∧R (1 4)

*****⊃I 4 9
```

```
8 P⊃(Q∧R)  (1)

****⊃I 1⊃;

9 ((P⊃Q)∧(P⊃R))⊃(P⊃(Q∧R))
```

Look at the LINE begining with 7 in the above example. 7 is its LINENUM, Q∧R is the WFF on this LINE, and the derivation of Q∧R on' this LINE depends on the assumptions 1 and 3. This LINE was generated by the specifying as a RULE *AND introduction* using LINEs 4 and 5. On LINE 8 when *IMP LIES introduction* is applied to LINEs 3 and 7, LINENUM 3 has been removed from the list of dependencies of the new LINE. This corressponds to the informal idea that the truth of the conclusion no longer need the discharged assumption. There are five rules that discharge assumptions. They are ***IMPLIES introduction***, *OR **elimination***, *NOT **introduction***, *NOT elimination* and EXIST ***introduction***. The exact details of what assumptions are eliminated can be found in each of the individual'descriptions of the RULEs. On LINE 10 assumptions are again discharged and the theorem is proved. I repeat: this theorem is a tautology and therefore can be proved in a single step using the TAUT rule and should usually be done that way when using FOL.

## Section 5 THE **COMPUTER** PROGRAM **FOL**

### Sect ion 5.1 How to run FOL at Stanford

FOL is invoked at the Stanford AI Lab by typing *R* ***FOL*** to' the monitor. To save an entire session you want to continued later type the command '**EXIT;**' to FOL, followed by *SAVE* ***<filename>*** to the monitor. To restart type ***RU <filename>*** to the monitor and you will be where you left off.

FOL 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 creating new' **VLs.** These' include making **AXIOMs,** assumptions, and **applying** the RULEs of inference to generate new steps in a derivation;

3. 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** is described using **the following** notion of pattern. Those form the basic constructs of the FOL parser.

1. Identifiers which appear in **patterns are** to be **taken** Morally.
2. Patterns for syntactic typos **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** 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 **comparatively** standard **Backus-Naur** Form **description.**

## Section 5.2 General information on the features of FOL

### Section 5.2.1 Individual symbols

In FOL INDVARs may appear both free and bound in WFFs. INDPARs, however, must always appear free. Natural numbers are automatically declared INDCONSTs of SORT NATNUM. The only kind of numbers understood by FOL are natural numbers, i.e. non-negative integers. -3 should be thought of not as an individual constant, but rather as the prefix operator -, applied to the INDCONST 3.

### Section 5.2.2 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 6.1 describes how to make such declarations.

### Section 5.2.3 Extended notion of TERMs

In addition to ordinary application terms, FOL accepts several other kinds of TERMs. There are three kinds of bracket TERMs: those surrounded by square brackets [,], those surrounded by curly brackets {,}, and those surrounded by angle brackets <,>. These are the only expressions in FOL that do not have a fixed number of arguments; Quote TERMs are individual constants for s-ex pressions. They appear in proofs as any s-expression preceeded by a "'" symbol. FOL also parses comprehension expressions of the form {x|P(x)}. A detailed description of the syntax of these TERMs and more examples are found in Appendix B.

### Section 5.2.4 The Equality of WFFs

FOL 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 if x and y are of the same SORT.

### Section 5.2.5 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. As explained above a VL is a part of a derivation. The syntax of naming VLs is very extensive and a review of it will be left to Appendix B.

## Section 5.2.6 SORTs

The declaration of SORTs, and specification of a partial order over them, constitutes a major feature of FOL from a computational point of view. It was the first major difference of FOL from the usual formalisms for first order logic.

## Section 5.2.7 Semantic Attachment

The semantic attachment mechanism of FOL is one of its most novel features. It allows a user to describe to the proof checker some computational information about the theory he is examining and allows him to make conclusions using this computational information rather than using the FOL rules of inference.

## Section 5.2.8 Syntactic Simplification

This is a powerful' syntactic simplifier which allows a user to specify a set of equations as simplification rules and then to simplify any expression by continually performing replacements until no more are possible.

## Section 5.2.9 Decision procedures

FOL presently has three decision procedures implemented. TAUT decides if WFFs are propositional' tautologies. TAUTEQ is like TAUT but takes equalities into account. MONADIC decides monadic predicate calculus statements.

## Section 6 LANGUAGESPECIFICATION

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 6.1 Declarations

As we mentioned above, the first thing 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. SYNTYPE 1

    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*)
    b) PREDCONST (*predicate constants*)
    c) OPPAR (*o era ion parameters or function parameters*)
    d) OPCONST (*operation constants or function constants*)

All identifiers of SYNTYPE2 require one or more arguments.

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

```
D E C L A R E ALT[REP1[<simpldec> OPT[,]]|REP1[<appldec> OPT[,]] 3 ;
```

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

```
<syn type 1>    :=  ALT[ <indsym>    <sen tsym> ]
<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 corresponds 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 DPCONST +(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.

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

There are two kinds of declarations: simple declarations. and application declarations. Simple declarations define **objects** which do not have arguments; in the present structure of FOL, these objects are **INDVARs, INDPARs, INDCONSTs, SENTPARs,** and **SENTCONSTs**. Application declarations define objects with arguments; this class includes **PREDPARs, PREDCONSTs, OPPARs,** and **OPCONSTs**. The BNF formulation of the declaration syntax is

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

<appldec>       := <syntype2> <idlist> <argdec> OPT[ [ <bpdec> ] ]
<argdec>        := ALT[ <argsort> | <natnum> ]
<argsort>       := ALT[ : <sortrep>   ALT[=|→] <pseudosort> |
                       ( <sortrep> ) ALT[=|→] <pseudosort> ]
<sortrep>       := REP1[ <pseudosort> , OPT[ALT[×|,]] ]

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

Examples of simple declarations:

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

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

*****DECLARE SENTCONST P1 P2 Q;
```

Examples of application declarations:

```
*****DECLARE OPCONST EXP (NATNUM, NATNUM) =NATNUM[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 NATNUM. It also has a value of **SORT** NATNUM, 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:NATNUM⊗NATNUM→NATNUM [L←850 R←800];
```

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

```
*****DECLARE OPCONST EXP:NATNUM⊗NATNUM→NATNUM [I WE;
```

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

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

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

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

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

```
*****DECLARE DPCONST 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 6.2 SORT manipulation

There are several commands which affect the SORT structure.

### Sect ion 6.2.1 Default SORT declarations

```
MOSTGENERAL        • [illegible] ;
NUMERALSORT          <sort> ;
CUBRACKETSORT        <sort> ;
ANRRACKETSORT        <sort> ;
SQBRACKETSORT        <sort> ;
SEXPRSORT            <sort> ;
```

In FOL certain TERMs come'with predeclared SORTs; numerals become INDCONSTs of SORT NATNUM, comprehension terms, curly bracket TERMs (sometimes called finite set TERMs) and angle bracket TERMs (sometimes called n-tuple TERMs) have SORT CLASS, quote TERMs have SORT SEXPR, and the default MOSTGENERAL SORT is the PREDCONST UNIVERSAL. 'This is also the default SORT of square bracket TERMs. The effect of the above commands is to replace these default SORTs with those specified by the user.

### Sect ion 6.2.2 MOREGENERAL declaration

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

For example,

```
*****MOREGENERAL CHESSPIECE ≥{WHITEPIECE,BLACKPIECE};
```

is equivalent to the axioms

$$\forall x . (\mathrm{WHITEPIECE}(x) \supset \mathrm{CHESSPIECE}(x))$$
$$\forall x . (\mathrm{BLACKPIECE}(x) \supset \mathrm{CHESSPIECE}(x))$$

where CHESSP I ECE, WHI TEPI ECE and BLACKPI ECE are previously declared SORTs. Another typical. example would be the declaration of classes to be MOREGENERAL than sets. The MOREGENERAL declarations establish a partial order among SORTs. The effect of this partial order on the quantifier rules is explained in section 7.3.8.4.

### Sect ion 6.2.3 EXTENSION declarations

```
EXTENSION <sort> <ext_set>;

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

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

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

*****DECLARE PREDCONST KINGS 1;

*****EXTENSION BKINGS {BK};

Extension of BKINGS is (BK)

*****EXTENSION WKINGS {WK};

Extension of WKINGS is (WK)

*****EXTENSION KINGS WKINGS ∪ 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 ∪ WKINGS; says that the **only** objects which satisfy the predicate KINGS are those in the union of the extensions of BKINGS and WKI NGS, i.e. BK and WK. This is equivalent to the introduction of the axioms:

```
∀x.(BKINGS(x) ≡ (x=BK))
∀x.(WKINGS(x) ≡ (x=WK))
∀x.(KINGS(x)  ≡ ((x=BK ∨ x=WK) ∧ ¬(BK=WK)))
```

By itself, this command has no effect, but the semantic simplification mechanism (Section 7.9) uses these axioms.

The facts about integers and even integers mentioned in section 2 are expressed by the declarations:

```
*****DECLARE PREDCONST EVENINTEGER(INTEGER);

*****MOREGENERAL INTEGER ≥{EVENINTEGER};

*****DECLARE OPCONST +:INTEGER⊗INTEGER→INTEGER[INF];

*****DECLARE INDVAR e1 e2 e3 ε EVENINTEGER;

*****AXIOM EVEN: ∀e1 e2.∃e3. e1+e2=e3;;

, EVEN: ∀e1 e2.∃e3.(e1+e2)=e3
```

## Section 7 THE GENERATION OF NEW DEDUCTION STEPS

### Section 7.1 Axioms

AXIOMs 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> := <axnam> : <wfflist> ;
```

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 particular axioms. An AXNAM is a VL and may be used in any context that that expects one. If WFFLIST only contains one WFF that axiom is called AXNAM.

*NOTE: The syntax calls for two semicolons!!!!*

Examples:

```
*****DECLARE SENTPAR P,Q,S;

*****AXIOM  P1:    (P⊃(Q⊃P)),
                   (S⊃(P⊃Q))⊃((S⊃P)⊃(S⊃Q)),
                   ((P⊃FALSE)⊃FALSE)⊃P;;
```

This creates the axiom P1. It generates three additional subaxioms P11=(P⊃(Q⊃P)), P12=(S⊃(P⊃Q))⊃((S⊃P)⊃(S⊃Q)) and P13=((P⊃FALSE)⊃FALSE)⊃P. At the moment no checking is done for the consistency of axiom names. You lose if you create conflicting ones. Axioms cannot be gottten rid of, so be careful; Numbers are *not* legitimate AXNAMs.

## Section 7.1.1 **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 **containing** a PREDPAR or an OPPAR.

An axiom can be used anywhere a VL can by using an AXREF. 'This is of the form AXNAM $[PP_1 \leftarrow XX_1,\ldots,PP_n \leftarrow XX_n]$ and its syntax is described in **the** section on **VLs**. An AXREF can appear anywhere a VL can. In the form **AXNAM**$[PP_1 \leftarrow XX_1,\ldots,PP_n \leftarrow XX_n]$, the $PP_i$ are **PREDPARs** or **OPPARs** 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 or TERM preceded by λ, any number of **INDVARs** and a "." (period). Thus, e.g. λ x y z.<wff>. The ARITY, p, of the PREDPAR or OPPAR must be less than or equal to the number of variables following the λ. The indicated X-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 ':' notation, explained in appendix 7.9, can be used to name the WFF associated wih this axiom.' The SUBPART designators can then be used in the same way as they are with other **VLs.**

Example of using axiom schemas:

```
*****DECLARE PREDPAR P 1;DECLARE INDVAR n;

*****DECLARE PREDCONST ≥ 2 [INF];DECLARE OPCONST + 2 [INF];

*****AXIOM INDUCTION: P(0)∧∀n.(P(n)⊃P(n+1))⊃∀n.P(n);;

INDUCTION: P(0)∧∀n.(P(n)⊃P(n+1))⊃∀n.P(n)
*****DECLARE INDVAR a b;
*****∧I INDUCTION [P←λb a. a+b≥b];

1 ∀a.(((a+0)≥0∧∀n.((a+n)≥n⊃(a+(n+1))≥(n+1)))⊃∀n.(a+n)≥n)

*****∧I INDUCTION[P←λb.∀a.a+b≥b];

2 (∀a.(a+0)≥0∧∀n.(∀a.(a+n)≥n⊃∀a.(a+(n+1))≥(n+1)))⊃∀n a.(a+n)≥n

*****∧I INDUCTION[P←λb n.n+b≥b];

3 ∀n.(((n+0)≥0∧∀n1.((n+n1)≥n1⊃(n+(n1+1))≥(n1+1)))⊃∀n1.(n+n1)≥n1)
```

## Section 7.2 Assumptions

ASSUME `<wfflist> ;`

The ASSUME command makes an assumption on **a** new line of the deduction for each **WFF in WFFLIST.** Note that assumptions depend upon themselves.

Examples:

```
*****ASSUME P∧Q;

 1 P∧Q    (1)

*****ASSUME P∧Q, P∧R;

 2 P∧Q    (2)

.3 P∧R    (3)
```

## Section 7.3 Basic introduction and **elimination rules**

The general form of a RULENAM is

```
<rulename> ::= <logconst> ALT[ I | E ]
```

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

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

The **LINENUMINFO** is different for each **RULE.** This is explained below. **We** will use * to stand for an arbitrary VL. 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

```
AI <vl> ∧ <vl> ;
```

-Alternative alphabetic **RULENAMs** will be gtven in parentheses after the standard ones. These ‘usually correspond to other frquently used names for these rules. Thus MP ***(modus ponens)*** or **UG** *(universal generalization)* can be used, instead of **⊃I** or VI.

If there is no syntactic ambiguity any comma appearing in these rules is 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[,]**.

## Sect ion 7.3.1 Summary of the basic rules

The inference rules consist of an ***introduction*** (I) and an ***elimination*** **(E)** rule for each logical constant. This page is included for reference as each rule is discussed **further** on. The letters within parentheses indicate that the inference rule discharges assumptions of **that** form.

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

$$\vee I)\ \frac{a}{A\vee B} \quad \frac{B}{A\vee B} \qquad\qquad \vee E)\ \frac{A\vee B \quad \overset{(A)}{c} \quad \overset{(B)}{c}}{C}$$

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

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

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

$$\neg I)\ \frac{\overset{(A)}{FALSE}}{\neg A} \qquad\qquad \neg E)\ \frac{\overset{(\neg A)}{FALSE}}{a}$$

$$FI)\ \frac{\neg A \quad A}{FALSE} \qquad\qquad FE)\ \frac{FALSE}{A}$$

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

***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 **∃x.A,** in B, or in any assumption on which the upper occurrence of B depends other than **A[x←a].**

## Section 7.3.2 AND (∧) rules .

Introduction rule

```
∧I (∧I)      (#∧#)∧#      ;
```

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.

```
*****ASSUME P,Q;

1  P  (1)

2  Q  (2)

*****∧I 1,2;

3 P∧Q  (1 2)

*****∧I 1 (2 1) ;

4  P∧(Q∧P) (1 2)
```

Elimination rule

```
∧E(AE)  #    OPTC ALT[,|:] 3 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 B. The dependencies of the result are the same as those of *.

```
*****ASSUME P∧(Q∧R);

1  P∧(Q∧R)  (1)

*****∧E 1 1;

2 P  (1)

*****∧E ↑ 2;

3  Q∧R  (1)

*****∧E 1: #2#2;
```

```
    4  R  (1)  .
```

Note the various **possible** syntaxes. Each of these commands could be replaced by an appropriate TAUT command; **e.g.,** the above command **^E** 1 **: #2#2;** could be replaced by TAUT **Q^R 1;**.

## Section 7.3.3 OR (v) rules

Introduction rule

```
VI(OI)   (#v<wff>v<wff>)   ;
```

ORs may be parenthesized just like ANDs, 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.

```
*****ASSUME   P;

1 P  (1)

*****vI 1,(PvR);

2 Pv(PvR)   ( 1 )
```

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 thoee'of #1 which are not equal to A and those of #2 not equal to 8. Remember two WFFs are equal if they differ only by a change of bound variable. In the example two different commands are g i ven. Note how the dependencies are treated in each case.

```
*****ASSUME PvQ,P,Q;

1 PvQ  (1)

2 P  (2)

3 Q  (3)

*****⊃I ¬Q, 2;

4 ¬Q⊃P  (2)

*****ASSUME ¬Q;

5 ¬Q  (5)
```

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

6 FALSE  (3 5)

*****FE  P ;

7 P  (3 5)

*****⊃I 5⊃;

8 ¬Q⊃P  (3)

*****∨E 1,4,8;

9 ¬Q⊃P  (1)

*****∨E 1,8,4;

10 ¬Q⊃P  (1 2 3)
```

TAUT could replace the introduction command, but if a TAUT were used in the elimination rule, the resulting VL would have extra dependencies

```
*****TAUT ¬Q⊃P 1,4,8;

11  (¬Q⊃P)  (1 4 8)
```

## Section 7.3.4 IMPLIES (⊃) rules

Introduction rule

⊃I (DED) ALT[ #⊃# | <wff>⊃# ] ;

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 often called the deduction theorem.

```
*****ASSUME P;

1 P (1)

*****⊃I P,1;

2 P⊃P (1)

*****⊃I 1⊃1;

3 P⊃P
```

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.

```
*****ASSUME P⊃Q,P;

1 P⊃Q (1)

2 P (2)

*****⊃E 1, 2;

3 Q (1 2)
```

The elimination rule can be replaced by TAUT, but TAUT will not remove those dependencies removed bt the ⊃I rule.

## Section 7.3.5 FALSE (FALSE) rules

<u>Introduct ion 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.

```
*****ASSUME ¬P,P;

1 ¬P (1)

2 P (2)

*****FI ↑,↑;

3 FALSE (1 2)

*****¬I ↑,¬P;

4 ¬¬P (2)

*****⊃I ↑⊃;

5 P⊃¬¬P
```

<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 contradict ion. The dependencies (there had better be some or your theory is inconsistant) are just those of #.

```
*****ASSUME FALSE;

1 FALSE (1)

*****FE P∧¬P;

2 P∧¬P (1)
```

## Section 7.3.6 NOT (¬) rules

Introduction rule

```
¬I (NI)   #   .,   ALT[  #1  |<wff> ]   ;
```

# 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.

```
*****ASSUME ¬P;

1 ¬P  (1)

*****ASSUME P;

2 P  (2)

*****FI ↑,;

3 FALSE (1 2)

*****-I ' ¬P;

4 ¬¬P ( 2 )

*****⊃I ↑⊃;

5 P⊃¬¬P
```

Elimination rule

```
¬E (NE)   #     ,   ALT[ #1 | <wff>  3  ;
```

# 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 ¬¬P, ¬P;

1 ¬¬P  (1)

2  ¬P  (2)

*****FI 1 2;

3 FALSE (1 2)
```

```
*****NE  2;
4  P  (1)
*****⊃I  1,;
5  ¬¬P⊃P
```

## Section 7.3.7 EQUIVALENCE (≡) rules

### Introduction rule

```
≡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.

```
*****ASSUME FALSE⊃P;

1 FALSE⊃P (1)

*****ASSUME P⊃FALSE;

2 P⊃FALSE (2)

*****≡I 1 2;

3 FALSE≡P (1 2)
```

### Elimination rule

```
≡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 #.

```
*****ASSUME P≡¬¬P;

1 P≡¬¬P (1)

*****≡E 1;

2 P⊃¬¬P (1)

*****≡E ↑ ⊂;

3 ¬¬P⊃P (1)
```

## Section 7.3.8 Quantification rules

### Section 7.3.8.1 Quantification example

```
*****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 7.3.8.2 UNIVERSAL QUANTIFICATION (V) rules

Introduction rule

```
V I (UG) # , REP1 [ OPT [ALT [<indvar>|<indpar>] ← ] <indvar> , OPT [,]] ;
```

Several simultaneous universal generalizations on • can be carried out with this command. For each element of the list (either x or **a←x)** a new universal quantifier (Vx) 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 step **10** is a universal generalization of step **9.** There is nothing free in the **WFF** on line 1 (line 9's only dependency) so the generalization is legal. Notice that the "a" was changed to an "x". "a" cannot serve as a bound variable, as it is an INDPAR.

Elimination rule

```
VE (US) # , <termlist> ;
```

Universal specialization 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 7.3.8.3 EXISTENTIAL QUANTIFICATION (3) rules

Introduction rule

```
31 (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 <natnumlist>
```

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 PREOCONST F 1;

*****DECLARE INDVAR x y;

*****TAUT F(x)v¬F(x);

1 F(x)v¬F(x)

*****∃I 1,x←y OCC 2 ;

2 ∃y.(F(x)v¬F(y))

*****∀I 2, x;

3 ∀x. ∃y.(F(x)v¬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 dependencies 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 $\exists y.P(a,y)$

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 dependencies 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 7.3.8.4 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 Qand **Qis 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)
        -------          -M-w---      ------m
         error            A(q)         A(r)

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

A s an example, consider the following FOL proof:

```
*****DECLARE PREDCONST CHESSPIECE WHITEPIECE BLACKPIECE 1;

*****DECLARE INDCONST black white ∈Color;

*****DECLARE OPCONST color:CHESSPIECE→Color;

*****DECLARE INDVAR p ∈ CHESSPIECE,wp ∈ WHITEPIECE,bp ∈ BLACKPIECE;

*****AXIOM COLOR: ∀wp. (color(wp)=white),
*                 ∀bp,(color(bp)=black);;

    COLOR: COLOR1:∀wp.color(wp)=white
           COLOR2: ∀bp.color(bp)=black

*****∀e COLOR1 wp;

1 color(wp)=white

*****∀e COLOR1 p;

2 WHITEPIECE(p)⊃color(p)=white
```

In general, if universal specialization is applied to a formula with a term whose SORT is

MOREGENERAL than the quantified variable, the result of the specialization is an implication asserting that if the term is of the proper SORT, then the specialization holds. if the variable is MOREGENERAL than the term, then the usual WFS is returned. Corresponding results hold for the other quantifier rules.

## Sect ion 7.4 The TAUT and TAUTEQ commands

### 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 f1;
*****DECLARE INOVAR a b;
*****TAUTEQ a=b⊃(P(a)≡P(b));
1 a=b⊃(P(a)≡P(b))
*****TAUT a=b⊃(P(a)≡P(b));
Not a tautology
*****TAUTEQ a=b⊃f(a)=f(b):
Not a tautology
```

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.

As mentioned before, any inference by one of the basic propositional rules can also be performed by TAUT. The difference is that TAUT sometimes handles dependencies unsatisfactorily, as in the following example:

```
*****DECLARE PREOCONST P Q 1; DECLARE INDVAR X Y Z;

*****DECLARE SENTCONST A B;

*****ASSUME AvB,A⊃∀X.P(X),B⊃∀X.P(X),A,B;

1  AvB  (1)

2  A⊃∀X.P(X) (2)

3  B⊃∀X.P(X) (3)

4  A  (4)"

5  B  (5)

*****⊃E 4 2;

6  ∀X.P(X) (2 4).

*****⊃E 5 3;

7  ∀X.P(X) (3 5)

*****∀E ↑ X ;

8  P(X) (2  4)

*****∀E ↑ X ;

9  P(X)  (3  5)

*****vE 1,8,9;

10  P(X)  (1  2  3)

*****TAUT P(X) 1,2,3,8,9;

11  P(X)  (1 2 3 4 5)

*****TAUT P(X) 1,2,3;

Not a tautology
```

## Sect ion 7.5 The QUANT Command

### Quantification rules

There are three new FOL commands which affect **WFFs** with quantifiers. They are PUSH, PULL, and QUANT. PUSH works on **WFFs** with an initial negation sign followed by any **number** of quantifiers. It pushes this, and any other negation symbols it might find, through these quantifiers making the necessary changes until the matrix of the formula is reached. PULL does the opposite,. namely it pulls negations out to the front of the formulas.

The syntax for these commands is

```
PUSH <vl>;
FULL <vl>;
```

The QUA NT command is much harder to explain. It tries to do "correct quantifier manipulations", but the phrase in quotes is not **clearly** defined. Its syntax is

```
QUANT  <wff>  ,  <vl>  ;
```

The meaning of this command is similar to TAUT. It says verify that the WFF follows from the given **VL** by **quantifier** manipulations. PUSH and PULL are just special cases of this rule. First *there* are **some** restrictions on the form of the WFF compared to that of the VL. They must be *propositionally similar* or there is no hope of applying this rule. If there are no equivalences, this means that the two must be identical when

1) quantifiers are dropped
2) terms are replaced by *'s.
3) negations are pushed in to **AWFFs**
4) **implications (A⊃B)** are changed to disjunctions **(¬AvB)**

Thus ¬( A( tl )vB( x) ) is **propositionally** similar to **¬A(f(x))∧¬B(t3)** but not to **¬(B(x)vA(t1))**.

**∃m.S(m)⊃∃m.(∀k.(k<m⊃¬S(k))∧S(m))** follows from **¬∀m.¬S(m)⊃¬∀m.(∀n.(n<m⊃S(n))⊃S(m))** by QUANT.

## Sect ion 7.6 The DISTRIB command

Since FOL accepts the following alternatives to **WFFs** and **TERMs**.

```
<wff>   :=  <condw>  := I F <wff> THEN <wff> ELSE <wff>
<term>  :=  <condt>  := I F <wff> THEN <term> ELSE <term>
```

the DISTRIB rule can be used to distribute function and predicate symbols over conditional expressions.

```
DISTRIB λ <indvar> .<applexp>   <condt> ;
```

Where <indvar> is an INDVAR, <applexp> is an application expression, i.e. either a PREDSYM or an OPSYM followed by an argument list of TERMs, and <cond t> is a conditional expression which is a TERM,

The effect of this rule is to distribute the application symbol over the conditional expression on the arguments specified by the individual variable.

Examp l es:

```
*****DISTRIB λX.F(X) IF TRUE THEN Y ELSE Z;

1  F( IF TRUE THEN Y ELSE Z)≡IF TRUE THEN F(Y) ELSE F(Z)

*****DISTRIB λX.P(Y,X,X) IF TRUE THEN F(Y) ELSE F(Z);

2 P(Y,IF TRUE THEN Y ELSE Z,IF TRUE THEN Y ELSE Z)≡
                    IF TRUE THEN P(Y,F(Y),F(Y)) ELSE P(Y,F(Y),F(Y));
```

## Section 7.7 The SUBSTITUTION command

This command allows you to take a line with an equation on it and substitute its right side for its left side in some other line. Its syntax is

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

#1 can have either = or ≡ as its major connective. If no occurence list is specified then all possible substitutions are made. If you want to substitute the left side of #1 for the right side the command is

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

In order to replace $t_1$ by $t_2$ within the occurrence of $t_3$ in (IF A THEN $t_3$ ELSE $t_4$), it isn't necessary to prove that $t_1 = t_2$, but only $A \supset t_1 = t_2$, and the SUBSTITUTION command uses this fact in a generalized form.

Namely, if #1 has the form $wff \supset wff_1 \equiv wff_2$ or $wff \supset t_1 = t_2$ the substitution is made only if TAUTEQ proves that $P \supset wff$, where P is the precondition of the left hand side of the equality.

The precondition of any subexpression of an FOL expression is then the conjunction of the preconditions of those parts of the. conditionals which contain the subexpression. In a conditional, IF P THEN Q ELSE R, the precondition of the THEN part is P and the precondition of the ELSE part is $\neg P$.

For example, in the WFF IF P THEN (IF Q THEN a ELSE b) ELSE b The first occurrence of b has precondition $P \land \neg Q$, the second occurrence $\neg P$.

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 occurrences of a bound variable which will cause a conflict are c-hanged. Thus, if one tries to substitute f(x) for y in ∀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.

The 'new' variabte 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 7.8 The MONADIC command

MONADIC rule

```
MONADIC <wff> , <vllist> ;
```

This rule implements a decision procedure for the monadic predicate calculus; i.e., it will decide whether WFF follows from VLLIST whenever the formulas involved contain only unary predicates. More generally, this command will always attempt to decide whether VLLIST implies WFF. Of course, this will not generally work, but it does work in many cases. If the decision procedure succeeds, WFF is concluded and dependencies are the union of the dependencies of each WFF in the VLLIST.

```
*****DECLARE PREDCONST P 1;DECLARE SENTCONST A;

****
*****DECLARE INDCONST C;DECLARE INDVAR X;

****
*****MONADIC ∀X.P(X)⊃P(C);

1 ∀X.P(X)⊃P(C)

*****MONADIC ∀X.(A≡P(X))∧∃X.P(X) ⊃ A;

2 (∀X.(A≡P(X))∧∃X.P(X))⊃A
```

## Sect ion 7.9 **Semantic Attachment and Simplification**

FOL is intended to express a variety of methods of human reasoning. Though the word "reasoning" usually connotes a logical deductive process of using facts and assertions to obtain conclusions, much of human intelligence relies more upon observation than upon deduction. We look at a book. The book is seen to be "green", as an immediate observation, not as a deduction involving, say, analysis of wavelengths of light and sensory receptors in the eye. Similarly, humans cross streets without conscious analysis of the traffic flow, add numbers without resorting to basic set theory, and play chess without considering each move in terms of the geometry of the board.

Any system which hopes to express a variety of reasoning processes therefore needs a method of doing purely computational tasks. In FOL, the simplification mechanism provides this ability. These routines have two parts. First, **FOL's ATTACH** command permits the user to define a correspondence between the various constants (function symbols, predicate constants, individual constants) of his language and corresponding objects in the programming language LISP. Second, facts about the LISP structure can be used directly in the proof via the **SIMPLIFY** command, eliminating the necessity of a possibly complicated deduction. For example, obvious attachments to the function symbol + and to the individual constants **17,34,51** would allow one to conclude **17+34=51** in one step, instead of computing 34 successors of 17. In order to explain this more clearly we first give an informal account of the technical details.

The declarations made by an **FOL** user specify a first order language **L=<P,F,C>**, where P is the list of PREDCONSTs, F the list of **OPCONSTs**, and C is the list of **INDCONSTs**. **A** model for such a language is a structure **M=<D,P',F',C'>** where D is a set, and **P',F'**, and **C'** are lists of predicates' over **D**, functions on D, and individuals of D such that the **ARITYs** of the symbols in P and F match the **ARITYs** of the predicates and functions at the corresponding positions in P' and F'. The idea here is that the language L is used for making statements, about structures such as M. In particular, when the user writes down a theory in FOL, he generally has in **mind some** particular model for his language, and the axioms of his theory are intended to express the properties of this **particular** model. The **fact** that FOL' is actually a **LISP program** running in a LISP environment inspires the following idea: some parts of a models for an **FOL** languages can often be expressed computationally in the sense that the elements of D can be represented by s-expressions, and the predicates and functions on D can be represented by **LISP** functions and predicates. It should then be possible to use the tomputationai representation to aid FOL deductions concerning the model. For example, suppose the theory we are interested in is first order number theory, and the model that we have in mind is the set of natural numbers together with the operations of successor, addition and multiplication. The numerals have natural representations as LISP numbers, and the functions in question have *PLUS I, *PLUS,and *TIMES as their LISP counterparts. As mentioned above it should then be possible to use the computational representation to provide swift deductions of such statements as 25+37=52.

The semantic attachment facility in FOL allows the user to set up these computational representations of his subject matter, and to use this representation to aid deduction in FOL. This ability is achieved by using the **ATTACH** and **SIMPLIFY** commands. The ATTACH command allows FOL **OPCONSTs, PREDCONTs**, and **INDCONSTs** to be attached to the corresponding kinds of LISP objects. **The SIMPLIFY** command **allows** the attachment information to be used in deduction:

when the user gives a TERM as the argument to SIMPLIFY, any attachments which may exist to the symbols in that TERM are looked up, and if possible, the value of the TERM in the computational representation is computed; finally, if an FOL TERM with that value can be found, the equality of the TERM with its simplified version is asserted as the next line of the proof: SIMPLIFY behaves in an analogous manner if given a WFF rather than a TERM as its argument. With the above overview in mind, let us proceed to the details.

## Section. 7.9.1 A technical explanation

Given a language L=<P,F,C> and a model M=<D,P',F',C'>, we define an interpretation function I which gives, for each TERM t of L in which no free variable occurs, the individual in D which t denotes. In particular we define the interpretation of an INDCONST c to be the individual c' in D, and where f is an OPCONST, and the interpretations of TERMs $t_1,...,t_n$ are defined, we inductively define the interpretation of the TERM $f(t_1, t_2, \ldots, t_n)$ to be $f'(I(t_1), I(t_2), \ldots, I(t_n))$. We may extend the interpretation function to formulas (again without free variables) over L by defining I(w) to be the object TRUE exactly" when the formula w is true of the model (for a technical definition see Kleene [ 19681). When f' is the function in a model corresponding to the OPCONST f in L, we will also say that f' is the interpretation of f, and similary for PREDCONSTs. Now we define a computational model to be an object K=<D',P",F",C">, where it is understood that D' is a set of s-expressions, and P",F",and C" are lists of LISP predicates, functions, and s-expressions respectively, with the appropriate restrictitions on ARITYs. From the extensional point of view, a computational model is for 'a language is just like a set-theoretic model for a language, except that we do not require that the functions and predicates concerned be total; that is functions and predicates may be undefined (non-terminating) for some elements of D'. We define an attachment map rtt from terms and formulas of L into K in a manner exactly analogous to the definition of I above. We have one last map to worry about, the map rep which gives, for each object in the domain D' of the computational model M, the object it represents in the domain 0 of the model M. Now we may define precisely the meaning of attachments made in the FOL system: The attachment of an INDCONST c to an SEXPR C signifies that c and C represent the same object in the model, that is to say, I(c)=rep(C). Similarly, the attachment of an OPCONST f to a LISP EXPR or SUBR F signifies that the result of applying F to an SEXPR C which represents an individual c in the model, is a SEXPR which represents the individual f'(c) in the model. The analogous statements hold for attachments to PREDCONSTs. The above conditions are equivalent to the statement that the following diagram commutes.

$$\begin{tikzcd}
\text{FOL TERMs} \arrow[d, "\text{att}"'] \arrow[dr, dash] & \\
\text{LISP sexpr} \arrow[r, "\text{rep}"'] & \text{Domain of model}
\end{tikzcd}$$

The semantic simplifier given an FOL TERM, attempts to compute its attachment, and to find a simpler TERM with the same attachment. if it succeeds, the simplified TERM is returned. 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 LISP representation of the model would give the number 3 as an answer - the simplifier would then return the INDCONST 3.

The attachment mechanism allows several representation of the model by LISP SEXPRs to be in force at the same time. I will seek to motivate this aspect of the.attachment facility by means of an example: consider a theory of chess which includes a general theory of lists as a subtheory (this subtheory would be applied in arguments about lists of pieces, lists of game positions, and so on). The intended model of such a theory includes at least two kinds of objects: chess positions, and lists. Lists and positions form disjoint domains in the model, though it may be possible to build lists of chess positions. If we are going to build a computational representation of this model, we will need to represent positions and lists by s-expressions in such a way that no s-expression represents both a list and a position. The natural representation of a chess position as an s-expression is as a list of eight lists, each of which--is a list of eight piece names (one of which is "empty" or some such), and the natural representation of lists as s-expressions is the direct representation as LISP lists. This representation scheme cannot be used, since it will not be possible to decide whether a given list of eight lists of eight piece names represents a chess board or a list of list of pieces. That is to say, the map rep will not be well defined. It is of course not hard to solve this problem by the use of some slightly fancier coding, but a general solution to the problem of disambiguating computational representations is available: Suppose that the intended model of an FOL theory T includes the disjoint domains $D_1,...,D_n$, and suppose further that we have a different coding function for each of these domains. That is we have n different representation functions rep, which map the domain of s-expressions into the domain of the model, with the property that the range of $rep_i$ is a subset of $D_i$. Then it is possible that a single s-expression s codes two different objects $d_i,d_j$ in the model, but as long as we know what coding function $rep_i$ to apply, there is no ambiguity. Then the definition of the att map may be extended to take account of the possibility of multiple representations in the folloing way: The domain of the att map will still consist of the set of FOL terms and formulas, but its range will now lie in the set of pairs of the form, <representation function,s-expression>. The soundness condition for the att map is now that, when att(t)=<rep,s>, we have rep(s)=I(s). In order to specify this new more complicated att map, the user of the FOL system must give representation information concering his attachments. Specifically, each representation function must be given, a name, and when the attachment to an INDCONST is given, the name of the associated representation' function must be given as well. Similarly, when the attachment F to an OPCONST f is specified, the (names of the) representations of its arguments and of the value it returns must be given, and when the attachment to a PREDCONST is specified, the representations of its arguments must also be specified. The significance of specifying that the representations of the arguments and value of the attachment F to an OPCONST f are $R_1,R_2,...,R_n$, and $R_v$ respectively, is that $R_v(F(A_1,A_2,...A_n))=f'(R_1(A_1),R_2(a_2),...,R_n(A_n))$, where f' is the interpretation of f, whenever $A_1,...,A_n$ are SEXPRs in the domains of $R_1,...,R_n$. The same holds for attachments to PREDCONSTs, mutatis mutandis. Given the attachments with representation information for individual symbols, the map att on the domain of terms and formulas is defined inductively in the obvious way: If f is attached to F,and the declared representations of the arguments of F are $R_1,R_2,...,R_n$, and terms $t_1,t_2,...,t_n$ have attachments with representations $R_1,R_2,...,R_n$ then $att(f(t_1,t_2,...,t_n))=F(att(t_1),att(t_2),...att(t_n))$. Under **this** definition the diagram above commutes for each individual representation function.

Note that if the representation of the attachment. of any term **t** does not match that of its place in the. argument list, then $F(att(t_1),att(t_2),...,att(t_n))$ cannot be expected to represent the interpretation of $f(t_1,...,t_n)$. The reason for this is that the correctness of a computation which purports to represent a mathematical function depends on the representation of the arguments of the function as data objects. For example, no one would expect a floating point multiplication algorithm to behave correctly if its arguments were encoded as integers rather than floating point numbers.

Finally, note that the attachment map, as well as the **EXPRs** which represent functions, may be partial. The user is never required. to provide an attachment for any FOL symbol, nor is any attachment to an OPCONST or PREDCONST required to be complete. The simplification mechanism will use whatever **information.is** available, but it never dies because of insufficient information.

## Sect ion 7.9.2 Declaring representation names

The representation maps from LISP objects to the intended model may be given names by use of the declaration command. Representation names may be any sequence of characters which is accepted by the FOL parser as a token (the user would do well not give his representations weird **names** which might interfere with the parsing of the statements in which the name might appear. For example "]" doesn't make it as a REPNAM.) The following syntax is used:

```
DECLARE REPRESENTATION REP1 [<randomtoken>];
```

Since the model itself appears no where in the FOL system, there is no need for the user to give any detailed information about the nature of the representation maps which he has in mind. All that is necessary is that he give each such map a name so that he may refer to it at will.

## Section 7.9.3 **The ATTACH command**

Attachments to FOL symbols are made using. the ATTACH command. The syntax for this command is:

```
*ATTACH ALT[<predconst>|<opconst>|<indconst>]
        OPTIALT   [TO   | to |→|↔|*]]
        OPTE "[" ALT [<REPNAM>]|                               ; for INDCONSTS
                  [<REPNAM1>,...,<REPNAMn>]   |                ; for PREOCONSTS
                  [<REPNAM1>,...,<REPNAMn> = <REPOUT>]         ; for OPCONSTS
                        "]"     ]
        <sexpr>;
```

where

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

The effect of the command is that the FOL symbol appearing as the first argument is attached to the SEXPR. If the FOL symbol is a **PREDCONST** or OPCONST, then the SEXPR must be either an atom which names an already existing LISP function or predicate (i.e. the atom has an EXPR or SUBR on its property list), or **a** LAMBDA expression. The **ARITY** of the FOL symbol in these cases should match the number of arguments accepted by the attached LISP function.

There are two optional arguments to the ATTACH command. The first specifies whether or not the attachment should be regarded as "going in both directions", and is only meaningful if the FOL symbol is an INDCONST. **A** two way attachment has the effect of telling the simplifier that, whenever SEXPR is computed as the **LISP** representation of a TERM, then the attached FOL symbol should be returned as the simplified version of that TERM. That is to say, if the FOL INDCONST A is attached "both ways" to the SEXPR **S,** then, not only is S the LISP representation of A, but A is the preferred FOL name of the (model value denoted by the) LISP object S. **The** manner in which the argument specifies whether the **attachment** goes both ways is as follows: TO,to, and → indicate **a** one-way attachment, while ↔ and * indicate a two-way attachment. If the argument is left out, then a one-way attachment is assumed.

The second optional argument specifies the representation information associated with the attachment: If the attachment represents an individual, then **[<REPNAM>]** specifies that the name of the representation map for that attachment is **<REPNAM>**. If the attachment represents a predicate, then **[<REPNAM1>,..<REPNAMn>]** gives the names of the representations expected for the arguments of the attachment. If the attachment represents a function, **then** **[<REPNAM1>,..<REPNAMn>=<REPOUT>]** specifies that the names of the representations expected for the arguments of the attachment are **<REPNAM1>,...,<REPNAMn>** respectively, and that the name of the representation of the output is **<REPOUT>**. The character * may occur anywhere where a representation name is expected. The effect is that the default representation name for the context in which the representation name occurs is used. The default specification facilities for representation **names** are described in the next section.

## Section 7.9.4 **Setting default representations**

The REPRESENT command may be used to associate representation names with SORTs, with, the effect that the representation name associated with a SORT is used whenever 'an attachment is made to a symbol "involving" the given SORT, and no representation name is specified directly. To be more precise, each FOL symbol has a collection of slots: an INDCONST has one slot, whereas an OPCONST of **ARITY** N **has N+1** slots,: its output, and its arguments. At the present time each symbol may have one piece of SORT information and one piece of representation information associated with each of its slots. The result of associating a SORT **s** with a representation **r** via the REPRESENT command is that, whenever an attachment **is** made where no representation is given directly for a slot of the symbol being attached to, and the SORT of that slot is s, then representation of that slot is set to r. The purpose of this command is to allow the user to set up a convenient set of defaults for representation information; nothing can be accomplished with the command that could not- be accomplished without it, given sufficient patience on the part of the user. The syntax for the command is:

```
REPRESENT ALT ['*|'{ REP1 [<SORTSYM>]   '}] AS <REPNAM>;
```

The effect of REPRESENT commands is cumulative; at any given time a SORT has the default representation most recently assigned by a REPRESENT command. Note that the effect of one represent command can overide that of a previous REPRESENT command. If a * appears instead of a list of SORTs, then <REPNAM> becomes the "default default". The effect of this is that whenever an attachment is made to a symbol involving a given SORT, and no representation name is specified, and there is no defualt representation for the SORT, then the default default ,if any, is used. If no default default has been assigned, and no representation name has been specified in any other way, then an error message will be printed out at the time of the attempted attachment. The REPRESENT * command can be repeated with the effect that the effect of the last such command is overridden.

There are two sets of canonical attachments to INDCONSTs in effect in any FOL system. Each of the numerals (i.e. the INDCONSTs 0,1,2,...) has the LISP integer which it denotes as its canonical attachment; the representation name for all canonical attachments to numerals "NATNUMREP". Similarly each of the quote INDCONSTs (e.g. '(A B)) is attached to the s-expression which it denotes,with the representation name "SEXPREP". The canonical attachments are two-way attachments.

## Section 7.9.5 The **SIMPLIFY** command

The SIMPLIFY command makes use of information concerning attachments, sorts, and extensions in computing a simplified expression which is **equivalent** to its argument. The syntax of the command is:

```
SIMPLIFY [ALT<wff>|<vl>| <term> 3 ;
```

The simplifier then attempts to find an expression in the language which corresponds to this evaluated entity. In the case of **VLs** and TERMs, the original expression is returned, set equal to 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** if the result of simplification is a truth-value, the **WFF** or its negation is returned, whichever is appropriate.

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

Examples of the use of these commands are found in the primer.

The method employed for simplification is roughly as follows: if A is a TERM having the form $f(t_1,t_2,...,t_n)$, then (recursively), the sorts, attachments, and simplified FOL expressions of $t_1,t_2,...,t_n$ are computed. (Of **course,it** is not always the case that all of this information can be determined). The same information concerning A is computed in the following manner: if f has an attachment whose argument representations match the representations of $t_1,t_2,...,t_n$then the attachment to A is computed by applying the attachment to f to the attachments of $t_1,t_2,...,t_n$. The sort of A is determined in the obvious manner: if the sorts of $t_1,f_2,...,t_n$ match the argument sorts of $f$, then A has the output sort of $f$. The simplified FOL expression for A is the "inverse" attachment to the attachment to A if such exists, and f applied to the simplified versions of $t_1,t_2,...,t_n$ otherwise. Thus when simplifying a complicated TERM, we first simplify its subparts, and then use the information so obtained to simplify **the TERM.**

## Section 7.9.6 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.

## Sect ion 7.10 syntactic simplification

The basic idea of syntactic simplification is repeated substitution of selected equalities and equivalences into a given expression. More "precisely, let E be a set of universally quantified equations and equivalences, so members of E look like $\forall x.(t_1=t_2)$ or $\forall y.(F_1\equiv F_2)$, where x and y represent variable sequences, $t_1$ and $t_2$ represent FOL TERMs, and $F_1$ and $F_2$ represent FOL WFF. A match, or immediate simplification, of an FOL expression EXPR consists of replacing an occurrence of $t_1[x\leftarrow u](F_1[y\leftarrow v])$ in EXPR by $t_2[x\leftarrow u](F_2[y\leftarrow v])$, where u (v) is a sequence of TERMs.

The following example from a correctness proof for the McCarthy-Painter compiler, is' (the formalization of the correctness statement for constant expressions, where the variables have the following intended meanings:

c represents constants of the source language;
i and j represent machine locations;
ssv and osv represent source language state vectors
  and object language state vectors, respectively;
vl represents variables of the source language.

Consider half of the base case of the induction:

```
     ∀c i ssv osv. (∀v1.(v1 OCCURSIN c⊃(loc(v1)<i∧ssv●v1=osv●loc(v1)))
(*)      .             ⊃(compute(compile(c,i),osv)●ac=ssv●c
                       ∧∀j.(j<i⊃compute(compile(c,i),osv)●j=osv●j)))
```

(*) is a direct consequence of elementary logical facts together with the following axioms defining source language state vectors, the compiler, and the "load immediate" instruction of the object language:

```
∀ssv c. ssv●c=c;
∀c i.compile(c,i)=mkli(c);
∀c osv.compute(mkli(c),osv)●ac=c;
∀c osv j.compute(mkli(c),osv)●j=osv●j.
```

The direct proof can be thought of as reducing (*) to TRUE by the following sequence of left-to-right substitutions (immediate simplifications):

```
compile(c,i)=> mkli(c)
compute(mkli(c),osv)●ac => c
ssv●c => c
c=c => TRUE
compile(c,i)=> mkli(c)
compute(mkli(c),osv)●j => osv●j
osv●j=osv●j => TRUE
j<i⊃TRUE => TRUE
∀j.TRUE => TRUE
TRUE∧TRUE => TRUE
∀v1.(v1 OCCURSIN c⊃(loc(v1)<i∧ssv●v1=osv●loc(v1)))⊃TRUE => TRUE
∀osv.TRUE => TRUE
```

```
∀ssv.TRUE => TRUE
∀i.TRUE   =>   TRUE
∀c. TRUE => TRUE
```

FOLs syntactic simplification commands implement- (a version of) this repeated substitution algorithm. There are essentially two subtleties involved in formalizing the procedure exemplified above: (1) There may be more than one equation (or equivalence) whose left half matches a given expression, so one has to establish a precedence hierarchy for matching. (2) What order does the algorithm use to consider the subexpressions of a given expression e?

FOLs solution to the first problem is the following ordering on expressions:

Each simplification expression (i.e., left half of an equation or equivalence) is regarded as a linear string of atoms. Each atom is either:

(1) a constant (which is not bound by the universal quantifiers in the prefix);
(2) an old variable (which is bound by the universal quantifiers in the prefix and which has occurred before in the linear string);
(3) a new variable (which is bound by the universal quantifiers in the prefix and which has not occurred before in the linear string).

If we think of concatenating different atoms to a given initial string, then the atoms have the precedence ordering

constants c old variables < new variables

and expressions are ordered lexicographically in accordance with the ordering on atoms.

Let's consider, for example, the precedence relations among the simplification expressions f(a,b,b), f(a,b,c), f(a,a,x), f(a,x,x), f(a,x,y), f(x,x,x), and f(x,x,y), where f,a,b,c are constants and x,y are variables. The last four expressions are linearly ordered:

$$f(a,x,x) < f(a,x,y) < f(x,x,x) < f(x,x,y)$$

and each of the first three expressions is less than f(a,x,x) and incomparable to the other two of the first three expressions:

$$f(a,b,b) < f(a,x,x)$$
$$f(a,b,c) < f(a,x,x)$$
$$f(a,a,x) < f(a,x,x)$$

Together with transitivity, these inequalities completely define the precedence relation.

FOLs syntactic simplification code basically considers subexpressions of e in the usual left-to-right order. The exceptions occur after a subexpression e' has been matched (and substituted for). The algorithm **then** begins again at the subexpression one level above e'. Consider the above example from the McCarthy-Painter compiler. After making the match `compile(c,i) => mkli(c)`, **the** algorithm begins again **with** the expression e"=compute `(mkli(c),osv).e"` does not simplify, and **the** algorithm attempts (unsuccessfully) to match all the subexpressions of e" before considering the expression compute `(mkli(c),osv)●ac`. Then, after making the match compute `(mkli(c),osv)●ac => c`, the algorithm starts again at the expression `c=ssv●c`. The subexpression `ssv●c` matches (and is replaced by c), whereupon the algorithm begins again with the reduced expression `c=c`.

The syntactic simplification algorithm has the usual problems of rewrite rules. A typical difficulty is the possibility of infinitely recurring substitutions; e.g., if one uses `1=1+0` as a simplification equation, **the** algorithm will attempt to make this substitution without end. Longer less obvious loops are also possible. An example that actually occurred **is** the equations

```
      1=succ(0)
∀n.succ(n)=n+1
   ∀n.0+n=n
```

which cause any occurrence of "1" to be replaced by "I" forever.

## Section X10.1 **Making a simplification** set

**One thing a user must do** is to explain which **VLs** will be used as rewrite rules. The **set** of rewrite **rules is called either the** match tree or the simplification set. There are two commands for manipulating match trees.

```
DECLARE SIMPSET <token>;
```

creates an empty match tree, i.e., one with no rewrite rules, which has <token> as its name.

```
<match-tree-name> ←<simpset-expr>;
```

creates a' **match tree** containing the **specified** rewrite rules. Existing simplification sets can be augmented usind **a** command like

```
MTREE ← MTREE ∪<simpset-expr>;
```

Simplification set expressions are defined by the syntax below, where "," means to take the union of the given expressions. The binding powers of ",", "∪" and "\" are that "," binds least strongly, "\" has an intermediate binding power, and "∪" is strongest.

```
<simpset-expr> ::= { <vl list> } | <simpset> |
                   <simpset-expr> , <simpset-expr> |
                   <simpset-expr> U <simpset-expr> |
                   <simpset-expr> \ <simpset-expr>
```

A VL which is a universally quantified equation or equivalence will be used as a rewrite rule in the obvious way; 'that is, in simplifying an, expression, every instance of the left-hand side of the equation will be replaced by the corresponding instance of the right-hand side. A VL, v of some other form will be used as a rewrite rule **v≡TRUE**. If v is also of the form **∀x.,M,** where Vx represents **the** (maximal) prefix of universal quantifiers and **M** is the matrix (so that M is NOT an equation or equivalence), then **M≡TRUE will** be used as a rewrite rule.

There is a standard match tree, LOGICTREE which' contains the rewrite rules corresponding to the following basic logical equivalences:

```
P      A TRUE  ≡ P
P      A FALSE ≡ FALSE
TRUE   A P     ≡ P
FALSE  A P     ≡ FALSE
P      v TRUE  ≡ TRUE
P      v FALSE ≡ P
TRUE   v P     ≡ TRUE
FALSE  v P     ≡ P
P      ⊃ TRUE  ≡ TRUE
P      ⊃ FALSE ≡ ¬P
TRUE   ⊃ P       E P
FALSE  ⊃ P ·   ≡ TRUE
       ¬ TRUE  ≡ FALSE
       ¬ FALSE ≡ TRUE
       X = x   ≡ TRUE
VX. TRUE       ≡ TRUE
VX. FALSE      ≡ FALSE
3X. TRUE .'    ≡ TRUE
3X. FALSE      ≡ FALSE
```

Once an appropriate match tree has been defined, the user may invoke the simplification routines by the command

```
REWRITE ALT[<vl>|<term>|<wff> 3 OPT[ BY <simpset-expr>];
```

The different alternatives have significantly different effects on the proof: (1) rewriting a VL generates a new! proof step which **is** the maximally rewritten form of the given VL; (2) rewriting a TERM t generates a proof step **t=t'**, where **t'** is the maximally simplified form of t; (3) rewriting a WFF w **generates** a proof step **w≡w'**, where **w'** is the maximally simplified form of w, except that if w simplifies to TRUE, the new proof step is simply w. In the latter two cases, the dependencies of the new proof step are the dependencies of the **VLs** which were actually used in the simplification; in the first case the dependencies also include the dependencies of the given VL. If the command does not specify a simplification set expression, the given expression will be simplified according to the basic logical rewrite rules contained in LOGICTREE.

At present there is no FOL command for showing the rewrite rules contained in a match tree.

## Section 7.10.2 Example of syntactic simplification

The following is an example using the syntactic simplification commands.

```
*****DECLARE SENTCONST P;

*****DECLARE INDCONST A B;

*****DECLARE INDVAR X Y;

*****DECLARE OPCONST F 2 G 1;

*****AXIOM F: VX.F(X,A)=A,
*             VX.F(X,X)=G(X),
*             VX Y.F(X,Y)=Y;;

F : F1: VX.F(X,A)=A
   F2:  VX.F(X,X)=G(X)
   F3: VX Y.F(X,Y)=Y

*****ASSUME F1:,F2:,F3:;

1 VX.F(X,A)=A (1)

2 VX.F(X,X)=G(X) (2)

3 VX Y.F(X,Y)=Y (3)

*****REWRITE F(A,A) BY {F1,F2,F3}I

4 F(A,A)=A

*****REWRITE F(A,A) B Y {F2,F3};

5. F(A,A)=G(A)

*****REWRI TE F (A, A) BY {F3};

6 F(A,A)=A

*****REWRITE F(A,A) B Y {1,2,3};

7    F(A,A)=A    (1)

*****REWRI TE F (A,A) BY {2,3};

8 F(A,A)=G(A) (2)
```

```
*****REWRITE F(A,A) BY {3};
9 F(A,A)=A (3)
*****REWRITE F(B,B) BY {1,2,3};
10 F(B,B)=G(B) (2)
*****REWRITE F(B,B) BY {1,3};
11 F(B,B)=B (3)
*****REWRITE F(B,B) BY {1};
This expression does not simplify. Sorry.
*****DECLARE SIMPSET MTTEST;
*****REWRITE ¬TRUE BY NTTEST;
This expression does not simplify. Sorry.
*****REWRITE ¬TRUE BY LOGICTREE;
12 ¬TRUE≡FALSE
*****REWRITE TRUE⊃(P⊃X=X) BY LOGICTREE;
13 TRUE⊃(P⊃X=X)
*****MTTEST←{1,2,3};
*****REWRITE F(A,A) BY MTTEST;
14 F(A,A)=A (1)
*****REWRITE F(A,A)=A BY MTTEST;
15 F(A,A)=A≡A=A (1)
*****REWRITE F(A,A)=A BY MTTEST ∪ LOGICTREE;
16 F(A,A)=A (1)
*****REWRITE F(A,A)=G(A) BY MTTEST ∪ LOGICTREE;
17 F(A,A)=G(A)≡A=G(A) (1)
*****REWRITE F(B,B) BY MTTEST;
```

```
18 F(B,B)=G(B) (2)
*****REWRITE F(B,B)=G(B) BY HTTEST U LOGICTREE:
19 F(B,B)=G(B) (2)
*****REWRITE F(B,B)=G(B)∧F(A,A)=A BY HTTEST U LOGICTREE;
20 F(B,B)=G(B)∧F(A,A)=A (1 2)
*****REWRITE F(A,A) BY MTTEST\{1};
21 F(A,A)=G(A) (2)
*****REWRITE F(A,A) BY MTTEST\{1,2};
22 F(A,A)=A (3)
*****REWRITE F(A,A)=A BY (MTTEST\{1,2})U LOGICTREE;
23 F(A,A)=A (3)
*****REWRITE F(A,A)=A BY MTTEST\{1,2}U LOGICTREE;
24 F(A,A)=A≡A=A (3)
```

## Section 8 ADMINISTRATIVE COMMANDS

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

### Section 8.1 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. When you use the same label in this command twice the second LINENUM specified is the one used from then on.

### Section 8.2 File Handling commands

#### Section 8.2.1 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. An example of a FETCH command is shown in the description of the MARK command.

#### Section 8.2.2 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. For example, suppose that the file A[FOL,RWW] contains the following commands:

```
DECLARE SENTCONST P Q;
ASSUME P∧Q;
MARK 1;
∧E 1;
MARK 2:
∧E ↑ 2;
```

One can invoke these commands in the sequence shown below. Note that it is also possible to produce the following proof with the single command FETCH A [FOL,RWW]; in which case the MARK commands will simply be ignored.

```
*****FETCH A [FOL,RWW] TO 1;

***
****

1  P∧Q (1)

****
*****FETCH A [FOL,RWW] FROM 1 TO 2;

****

2  P  (1)

****
*****FETCH A [FOL,RWW] FROM 2 :

****

3  Q (1)

****
```

## Section 8.2.3 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. ***Caution:*** *it* ***deletes*** *any* ***file*** *of* ***the given name.***

## Section 8.2.4 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 8.3 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 8.4 The CANCEL command

**CANCEL OPT [clinenum>];**

This cancels **all** steps of a deduction with LINENUMs greater than or equal to **LINENUM.** For example, CANCEL 23; deletes step 23 and all later steps. 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 8.5 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> OPTE → <filename> OPT[ <NATNUM> ]] ;

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

<rangelist> := REP1[<rangespec>,OPT[,]]
<rangespec> := ALT[ OPT[ <linenum> ] : OPT[ <linenum> ] | <linenum> ]
<dec info>  := REP1[ ALT[ <syntype> OPT[ ε <sort>] |
                          <folsym>                 |
                          SORTS                    ], OPT[,]]
. <geninfo>  := REP1[ <sort>, OPT[,]]
. <label info> := 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 the SHOW command 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 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 the SORTs)

## Section 8.6 The EXIT command

EXIT;

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

## Section 8.7 The TTY and UNTTY commands

```
TTY OPT [<new name list>];
```

This command makes it possible for FOL to be used from terminal without the full Stanford character set and over the ARPA network. It creates synonyms for the FOL sentential connectives and quantifiers. If a <new name list> appears it must contain seven names, which then become the default input and output names for A, v, ⊃, ¬, ≡, V, and 3, respectively. The original quantifiers and connectives will still be accepted for input, but all output will use the new names.

If the <new name list> is omitted, the last used <new name list> is assumed. If no <new name list> has be used in this proof, then the following default <new name list> is assumed.

| Original symbol | New symbol |
|---|---|
| Λ | & |
| v | OR |
| 3 | IMP |
| ¬ | NOT |
| E | IFF |
| V | FA |
| 3 | EX |

for example,

```
TTY  * + → - ↔ ALL EXISTS:
```

would declare * as a synonym for A, + for v, etc.

```
UNTTY ;
```

This command returns the user to the original names for the connectives and quantifiers, and deletes any the new definitions.

## Sect ion 8.8 The SPOOL Command

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

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

# Appendix A **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: ¬, ∧, ∨, ⊃, ≡.**
   c) Quantifiers - QUANT: V and 3;

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 $f(t_1,t_2,...,t_n)$, 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 $P(t_1,...,t_2)$ 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∧B), (A∨B), (A⊃B), (A≡B), and ¬(A).
3. If A is a WFF, then so are ∀x.A and ∃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∧C, B∨C, B⊃C, B≡C, or ¬B is a SUBFORMULA of A so are B and C.
3. If ∀x.B or ∃x.B is a SUBFORMULA OF A, so is B[t←x].

The notations A[t←x] and A[t←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←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←t], denotes the result of substituting t for all free occurrences of x.

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

# Appendix B 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[ <identifier> |                 ;declared INDCONST
                      <integer>    ]                 ;no declaration necessary


<opsym>      :=  ALT [ <oppar>     <opconst> ]
<oppar>      :=  <identifier>                        ;dsc laced OPPRR
<opconst>    :=  <identifier>                        ;declared OPCONST
<preop>      :=  <opsym>                             ;ARITY 1 and declared PREFIX
<infop>      :=  <opsym>                             ;ARITY 2 and declared INFIX
<applop>     :=  <opsym>                             ;ARITY n and not dsclarsd
                                                     ;INF or PRE dsc


<predsym>    :=  ALT[ <predpar> | <predconst> ]
<predpar>    :=  <identifier>                        ;declared PREDPAR
<predconst>  :=  <identifier>                        ;declared PREOCONST
<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 dsc
```

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

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

<quant>     :a  ALT[ ∀ | FORALL | 3 |  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} U {y|∀x.P(g(x,y))}
powerset(<A,B,C>)
```

```
<term> :=  ALT[  <indsym>
                 <appl term>          |
                 <prefixterm>         |
                 <infixterm>          |
                 <setterm>            |
                 <n_tupleterm>        |
                 <compterm>           |
                 ( <term> )           ]
<applterm>      := <applop> ( <terml ist> )
<prefixterm>    :=  <preop> <term>
<infixterm>     :=  <term>  <infop>  <term>
<setterm>       := a   | <termlist> α}
<ntupleterm>    :=  <  <termlist>  >
<compterm>      :=  α{  <indvar>  |  <wff>  α}

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

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

## AWFFs

**AWFFs** are formed similarly, but cannot be nested,

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

```
<baseawff>    :=  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*** **∀x.P(x)⊃P(x)** ***will*** *parse as* **(∀x.P(x))⊃P(x)** *not as* **∀x.(P(x)⊃P(x))** *!*

We can write adjacent quantifiers of the same type together, so Vx **.∀y.P(x,y)** can be written **∀x y.P(x,y)**. FOL also accepts **(∀x)(∀y)P(x,y)** or (Vx **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).
10:#1#1#1#1#1   =  x
```

Substitutions in WFFs and TERMs

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

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

Examples:

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

## Bibliography.

Filman, R.E. & Weyhrauch, R.W.(1976) 'A FOL Primer' *Stanford University: Artificial Intelligence Laboratory Memo 288.*

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

Kelley, J.L.( 1955) *'General Topology',* (Princeton, New Jersey: D. Van Nostrand Company, Inc.)

Kleene, S.C. (1968) *Mathematical Logic,* John Wiley & Sons, Inc. New York

Kreisel, G.( 197 la) 'Five notes on the application of proof theory to computer science', *Stanford University: IMSSS Technical Report 182*

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

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,7* (Edinburgh: Edinburgh U.P.)

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