Stanford Artificial Intelligence Laboratory
Memo AIM-288

Computer Science Department
Report No. STAN-CS-76-572

September 1976

An FOL Primer

by

Robert E. Filman
and
Richard W. Weyhrauch

Research sponsored by

Advanced Research Projects Agency
ARPA Order No. 2494

COMPUTER SCIENCE DEPARTMENT
Stanford University

STANFORD ARTIFICIAL INTELLIGENCE LABORATORY
MEMO AIM-288
STANFORD COMPUTER SCIENCE DEPARTMENT
REPORT NO. STAN-GS-76-572

September 2, 1976

An FOL Primer

by
Robert E. **Filman**
and
Richard W. Weyhrauch

Abstract:

This primer is an introduction to FOL, an interact&e proof checker for first order logic. Its **examples** can be used to learn the FOL system, or read independently for a flavor of our **style** of interactive proof checking. Several example proofs are presented, successively **increasing in** the complexity of the FOL commands employed.

FOL runs on the computer at the Stanford Artificial Intelligence Laboratory. It can be used over the ARPA net after arrangements have been made with Richard Weyhrauch (network address R W W **●SU-AI)**.

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

AN'FOL PRIMER

CONTENTS

1 In the beginning.. 1

2 Another Simple Proof 3

3 Declarations 7

3.1 Individual Constants 7

3.2 Predicate declarations 7

3.3 Operator declarations 9

3.4 Other declarations.. 10

4 TERMs and WFFs 10

5 Axioms 11

6 The FOL Proof 12

6.1 Introduction to the Two by Two Colored Grid World 13

6.2 Sample Proof in the Grids World 15

7 Simplification mechanisms 18

7.1 The Grids World Revisited 22

8 Administrative commmands 27

8.1 The show commmand 27

8.2, The Backup File 29

8.3 Erasing Proof Steps.. 29

8.4 Reading in FOL Command Files , 29

8.5 Using FOL from non-Stanford terminals 30

8.6 Saving the state of the proof 30'

9 Pointers to additional information 31

Bibliography 32

Index 33

An FOL Primer

FOL, a checker for first order predicate calculus proofs, has been written by the Stanford formal reasoning group under John McCarthy and Richard Weyhrauch. FOL checks proofs using the formalism of natural deduction described in Prawitz[1965]. This paper is an introductory explanation of how to use FOL. Detailed expositions of predicate calculus can be found in most introductory logic texts, for example, Kleene[1968], Mendelson[1964], and Manna[1974].

The first section contains some very simple examples to help a new user get started playing with FOL. The later chapters are more difficult and explain additional features of FOL. A detailed description of the language can be found in the users manual, Weyhrauch and Glassmire[1976].

section 1 In the beginning

In this section we demonstrate the use of FOL to generate a simple proof. Any reader with with access to the FOL proof checker is urged to actually run the program using the example commands. FOL is invoked on the SAIL operating system with the monitor command, R FOL.

We will begin with what is perhaps the most classical proof in logic, and study its expression in FOL. Even a person who has never had a course in formal logic understands **the** syllogism:

Socrates is a man
All men are mortal
therefore
Socrates is mortal

First we have to express the assertions as a well formed formula (WFF) of first order logic. For this purpose we need an individual constant (INDCONST) Socrates, two predicate constants (PREDCONSTs), MAN and MORTAL, each of one argument, and an individual variable (INDVAR), x, to express the *all men* part of the second line. The exact rules for forming WFFs can be found in the FOL users manual. The three parts above are represented as

```
     MAN(Socrates)
∀x. (MAN(x)⊃MORTAL(x))
   MORTAL(Socrates)
```

The upside down A, "∀", is called a universal quantifier and is read *for all,* i.e. for all x if MAN(x) then MORTAL(x). Our goal is to prove

```
(MAN(Socrates)∧∀x.(MAN(x)⊃MORTAL(x)))⊃MORTAL(Socrates)
```

FOL expects that all identifiers used in a proof be declared, much as a programming language like ALGOL expects identifiers to be declared. Later we will go into greater detail on the syntax and options of declarations. For our present proof, the following declarations will suffice.

```
*****DECLARE INDCONST Socrates;
*****DECLARE PREDCONST MORTAL MAN 1;
*****DECLARE INDVAR x;
```

FOL prompts you for input by typing five stars. Note that all FOL commands except comments end with a semicolon (;). In this paper we will use:

THIS FONT for things typed by a user to FOL,
THIS FONT for things typed by FOL,

FOL knows many commands for generating steps of a proof. These are called *rules of inference.* An easy one to use is the ASSUME command. One can assume any WFF. FOL permits the resulting proof step to be used elsewhere in the proof, and remembers which new lines ***depend*** on which assumptions. A good way to start to prove an implication is to assume its hypothesis, that is, its first half. The FOL command looks like:

```
*****ASSUME MAN(Socrates)∧∀x.(MAN(x)⊃MORTAL(x));

1 MAN(Socrates)∧∀x.(MAN(x)⊃MORTAL(x)) (1)
```

FOL types out the proof step generated by the command, preceded by its line number (one in this case), and followed by a list of the lines in the proof it depends on. Assumptions depend on themselves.

We want to instantiate the second half of line one (the *for all* part) to the particular MAN, Socrates. First we must get this WFF onto a line of its own. The FOL command TAUT is used for deciding tautological consequences of proof steps. Tautologies are WFFs which are the theorems of propositional calculus. We type TAUT followed by the WFF we want, and then the line numbers of those lines from which we think it follows.

```
*****TAUT ∀x.(MAN(x)⊃MORTAL(x)) 1;

2 ∀x.(MAN(x)⊃MORTAL(x)) (1)
```

Note that this line also depends on line one. Proof step two has as its main connective (major symbol), a *for all* sign. The FOL command for specializing statements that are true of all individuals to a specific individual is called ***forall*** *elimination.* To use this command one types the word ∀E followed by the number of the line on which the elimination is to take place, and a list of individuals to replace the variables in the expression. We want to instantiate line two **to** the individual Socrates, so we say,

```
*****∀E 2 Socrates;

3 MAN(Socrates)⊃MORTAL(Socrates) (1)
```

It now follows, tautologically, from lines one and three, that Socrates must be MORTAL. Using the TAUT command again gets this result. Note that more than one line can be given in the reason part of the TAUT command.

```
*****TAUT MORTAL (Socrates) 1,3;

4 MORTAL(Socrates) (1)
```

To some extent, this is the desired result. But we're not home free yet; this line still *depends* upon the original assumption. We eliminate this dependence by creating an *implication* of the first line implying the fourth. FOL has a command for the introduction of implications, **⊃I** (implication introduction). It creates the proof step composed of its first argument implying its second. The dependencies of this new line of proof are those of the second argument to the ⊃I command, less any line whose WFF is the same as the first argument. It is a useful command for eliminating dependencies. Applied to lines one and four, we get

```
*****⊃I 1⊃4;

5 (MAN(Socrates)∧∀x.(MAN(x)⊃MORTAL(x)))⊃MORTAL(Socrates)
```

This is the WFF we want. It has no dependencies; it is a theorem. It is roughly equivalent to the English sentence, If *Socrates is a man,* and *for* ***all x*** if ***x*** *is a man,* then ***x*** *is mortal,* **then** *Socrates* ***is*** *mortal.*

section 2 Another Simple Proof.

Let us next try the expression and solution, in FOL, of the following puzzle, loosely adapted from Gardner [1959].

> **You** *have* ***three*** *boxes, one* **containing** *only black marbles, one only white, and the third, both black marbles and* ***white*** *marbles, The boxes were originally labeled BB, WW, and BW, respectively, to indicate their* ***contents,*** **but** *now someone has come along and switched* the *labels so that every box* ***is*** *labeled* **incorrectly,** ***How*** *can one* ***remove*** *and examine one marbla from only one box, and thereby* **determine** *what the correct labeling of the boxes should be?*

A solution to this problem goes as follows. One selects a marble from the box labeled BW. Let us assume this marble is black. As the box labeled BW is incorrectly labeled, the other marbles in this box must also be black. Therefore, the box labeled BW must have only black marbles. As the box labeled WW must have black marbles (otherwise it would be correctly labeled), and cannot be the box with only black marbles (for that is the box BW), then it must have marbles of both colors. It immediately follows that the box labeled BB must have only white marbles. The reasoning is similar when the marble drawn is white.

To express these lines in predicate calculus, we need to establish several conventions. There are eight **individuals** in this problem, the three boxes, which we will call ONE, TWO and THREE, respectively, the three labels, BB, WW, and BW, and the two kinds of marbles, BLACKS and WHITES. The following declaration creates these eight individual constants (INDCONSTs).

```
*****DECLARE INDCONST ONE, TWO, THREE, BB, WW, BW, BLACKS, WHITES;
```

Some of these objects are boxes, others are not. We will naturally want to distinguish between the boxes and the not boxes. We can conveniently do this by defining a predicate constant (PREDCONST), BOX that will be true when its argument is a box.

```
*****DECLARE PREDCONST BOX 1;
```

The "1" at the end of the predicate declaration indicates that this predicate has only one argument.

There are also three **PREDCONSTs** with two arguments in this problem, IS, LABELED and HAS. The predicate IS will describe the true contents of the box, basically, what the box would be labeled if it were correctly labeled. We will think of a box being LABELED with a label, and it HAS white and black marbles. We declare these predicates:

```
*****DECLARE PREDCONST IS LABELED HAS 2;
```

It is also useful to have a few individual variables (INDVARs).

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

These are the necessary primitives for translating the puzzle. The first phrase of the puzzle states that there are three boxes in the world. We have decided to call our three boxes ONE, TWO and THREE. A WFF expressing part of the equivalence of the predicate BOX and these individuals is:

```
∀x.(BOX(x) ≡ (x=ONE ∨ x=TWO ∨ x=THREE))
```

We can use this in our proof by making it into an axiom. Axioms can be used just like lines in a proof, but axioms have names, instead of line numbers. They are assumptions which do not introduce any dependencies into the proof. To make the above WFF an AXIOM, we say:

```
*****AXIOM  AREBOX:
*        ∀x.(BOX(x) ≡ (x=ONE ∨ x=TWO ∨ x=THREE));;
```

This axiom is now called **AREBOX.** Note the *pair* of semicolons at the end of the axiom. This is one of the more unpleasant syntactic constructs of FOL.

The meaning of each label is expressed in the axiom LABEL. This defines the predicate IS on **BOXes** and marbles.

```
*****AXIOM LABEL:
*      ∀x.(IS(x,BB) ≡ (HAS(x,BLACKS)∧¬HAS(x,WHITES))),
*      ∀x.(IS(x,WW) ≡ (HAS(x,WHITES)∧¬HAS(x,BLACKS))),
*      ∀x.(IS(x,BW) ≡ (HAS(x,WHITES)∧ HAS(x,BLACKS)));;
```

Note the use of one name for these three axioms. When we wish to refer to the second axiom, we will call it LABEL2.

If boxes were labeled correctly, we could determine which marbles the box has from the above axiom. However, the problem states the opposite; all labels are wrong.

```
*****AXIOM WRONG-LABEL< ,
*       V x y.  (LABELED (x,y)⊃¬IS(x,y));;
```

We also know that there is one box of each kind.

```
*****AXIOM IS-EACH:
*       3 x.   (BOX(x) A IS(x,BB)),
*       3 X.   (BOX(x) A IS(x,WW)),
*       3 x.   (BOX(x) A IS(x,BW));;
```

Once again, the parts of this axiom will be refered to as IS_EACH1, IS_EACH2, and IS_EACH3.

The "3" in this axiom is called an existential quantifier and is read *there exists*, i.e. there exists a BOX, x which has only black marbles in it, i.e., IS (x,BB).

AXIOM names can be used in proof generating commands just as line numbers can. To instantiate the AXIOM WRONG-LABEL to the individual ONE and the label BB, we say:

```
*****VE WRONG-LABEL ONE BB;

1 LABELED(ONE,BB)⊃¬IS(ONE,BB)
```

Similarly, to apply this axiom to the other two box label pairs, we give FOL the commands:

```
*****VE WRONG-LABEL TWO WW;

2 LABELED(TWO,WW)⊃¬IS( TWO,WW)

*****VE WRONG-LABEL THREE BW;

3 LABELED(THREE,BW)⊃¬IS(THREE,BW)
```

Note that we have arbitrarily decided which box is labeled with which tag.

We will also need to know what it means for a box to be WW. The axiom LABEL provides this definition .

```
*****VE LABEL1 ONE:

4 IS(ONE,BB)≡(HAS(ONE,BLACKS)∧¬HAS(ONE,WHITES))

*****VE LABEL2 TWO:

5 IS(TWO,WW)≡(HAS(TWO,WHITES)∧¬HAS(TWO,BLACKS))

*****VE LABEL3 THREE;

6 IS(THREE,BW)≡(HAS(THREE,WHITES)∧HAS(THREE,BLACKS))
```

We know from the axiom IS-EACH of the existence of some boxes. We want to use this axiom, but we cannot until we remove the existential quantifiers of its parts. The command to accomplish this is 3E or *existential elimination.*

WFFs with the major connective of V refer to all individuals; we can apply them to anything at all. WFFs quantified by an 3, on the other hand, refer to some nameless individual; all we can do is give it a name. To say, in effect, *Let's call it x*. The form of the 3E command is the same as VE; but the created line has the form of an assumption and depends only on itself. When a future consequence of this assumption no longer mentions this new name, FOL will automatically remove this assumption as a dependency and replace it with those of the existential statement.

```
*****∃E IS_EACH1 x ;

7 BOX(x)∧IS(x,BB) (7)

*****∃E IS_EACH2 y;

8  BOX(y)∧IS(y,WW) (8)

*****∃E IS-EACH3 z;

9 BOX(z)∧IS(z,BW) (9)
```

We can now talk about these x, y, and z. Note that we used different names in each 3E. To have called them the same would be to imply their equality; this is clearly not so, and FOL would remember such a tacit assumption. As each of x, y, and z is .a box, the axiom AREBOX can be applied to them.

```
*****∀E AREBOX x;

1 0 BOX(x)≡(x=ONEv(x=TWOvx=THREE))

*****∀E AREBOX y ;

11 BOX(y)≡(y=ONEv(y=TWOvy=THREE))

*****∀E AREBOX z;

12  BOX(z)c(z=ONEv(z=TWOvz=THREE))
```

The TAUT command doesn't know about the predicate = *(equals),* but there is a command, TAUTEQ, which can decide (some) tautologies using knowledge of the equality predicate. Its form is the same as the TAUT command used in the last proof; note that axioms can be used, just as line numbers in its list of reasons. One invocation of the TAUTEQ command will yield us the solution to the puzzle.

```
*****TAUTEQ
*       (LABELED(ONE BB)∧LABELED(TWO WW)∧LABELED(THREE BW))⊃(
*       (HAS(THREE BLACKS)>
*               (IS(THREE BB)∧IS(ONE WW)∧IS(TWO BW))) ∧
*       (HAS(THREE WHITES)>
*               (IS(THREE WW)∧IS(ONE BW)∧IS(TWO BB)))) 1:12;

13 (LABELED(ONE,BB)∧(LABELED(TWO,WW)∧LABELED(THREE,BW)))⊃(
    (HAS(THREE,BLACKS)⊃(IS(THREE,BB)∧(IS(ONE,WW)∧IS(TWO,BW))))∧
    (HAS(THREE,WHITES)⊃(IS(THREE,WW)∧(IS(ONE,BW)∧IS(TWO,BB)))))
```

FOL automatically deletes dependencies created by the 3E command when the introduced variables are no longer in the proof. As line 13 doesn't mention x, y or z, it is not dependent upon the fact that we used x in line 7, and not some other INDVAR. As line 13 has no dependencies, it is a theorem; Note the explicit assumption in this-theorem that box ONE is labeled BB, etc. Also note the use of the expression 1:12 in the line number part of the command. This tells FOL to use all of the lines between line one and line twelve, inclusive, in trying to decide this tautology.

section 3 Declarations

We now consider, in turn, most of the FOL commands in greater detail.

As we have previously stated, FOL expects that all individuals, predicates, and operators used in **a** proof be declared. The general form of a declaration is the word DECLARE, followed by the type (SYNTYPE) of the declaration, followed by the identifiers to be declared of that type, often with other useful information, such as the number of arguments of a predicate or operator, or the SORT to which a individual belongs. A common cause of syntax errors in FOL is an incorrect or missing declaration.

section 3.1 Individual Constants

Following the scheme above, we declare individual constants KingofSpades, QueenofHearts and JackofDiamonds with the command:

```
*****DECLARE INDCONST KingofSpadesQueenofHeartsJackofDiamonds;
```

All natural numbers come as predeclared INDCONSTs in FOL. Any LISP s-expression, **proceeded by** the quote character **', is** treated by FOL as an INDCONST.

Remember that all FOL commands, except the COMMENT command, are terminated by semi-colons.

section 3.2 Predicate declarations

The declaration of a predicate constant is similar. One gives the predicate's name, followed by the number of arguments it takes. Thus to have predicates PLAYINGCARD and BLACKCARD, each of one argument, we would declare:

```
*****DECLARE PREDCONST PLAYINGCARD BLACKCARD 1;
```

The number of arguments of a predicate or operator is its ARITY. FOL will treat any predicate constant of arity 1 as a SORT, that is, as denoting the collection of individuals for which it is true, FOL also assumes that every SORT is non-empty; there at least one individual which satisfies every predicate of arity 1.

More information than the mere existence of a constant can be conveyed to FOL by a declaration. To tell FOL that the individual constant NineofSpades was of the SORT PLAYINGCARD we would state:

```
*****DECLARE INDCDNST NineofSpades ϵ PLAYINGCARD;
```

the ϵ symbol denoting membership, much as an individual can be the element of a set.

To state in its declaration that all REDCARDs are of the SORT PLAY I NCCARD, we give the commands:

```
*****DECLARE PREDCONST REDCARD 1;

*****MOREGENERAL PLAYINGCARD≥{REDCARD};
```

The MOREGENERAL command tells FOL that every REDCARD is a PLAYINGCARD. While a given predicate can be declared only once, it can appear in a MOREGENERAL statement as often as desired. MOREGENERAL may be abbreviated MG.

If we want the PREDCONST SPADE to take its argument without parentheses, we can declare it to be a prefix predicate, with the command:

```
*****DECLARE PREDCONST SPADE 1 [PRE];
```

A typical FOL command using the prefix sort SPADE would be:

```
*****ASSUME SPADE NineofSpades;

1 SPADE NineofSpades (1);
```

The PREDCONST SAMESUI T can be declared by:

```
*****DECLARE PREDCONST SAMESUIT 2;
```

To declare the symbol "<" to be an *infix* PREDCONST, used between its arguments, without paren thesis, we state:

```
*****DECLARE PREDCONST < 2 [INF];
```

Other example of predicate declarations are:

```
*****DECLARE PREDCONST ACE,JACK,TEN,NINE 1 [PRE];
*****DECLARE PREDCONST ROYALFLUSH 5:
```

```
*****DECLARE PREDCONST CAPTURES 2;
```

PREDCONSTs can be declared to range over other sorts. Thus the predicate ≤ on the natural numbers **can** be declared:

```
*****DECLARE PREDCONST ≤(NATNUM,NATNUM)[INF];
```

Declaring a predicate on a domain has no effect **on** the **proof.** Making declarations in this manner, however, leads to a more easily understood proof; it is a convention we will use in the rest of this paper.

Several PREDCONSTs come predeclared in FOL. The most important is the arity **2** infix **PREDCONST** = (equals). **FOL** has a command for the substitution of equals for equals. This command will be explained **in a** later section of this primer.

Other predefined **ARITY** 1 **PREDCONSTs (SORTs)** include the NATNUMs (natural numbers and zero) and the **SEXPRs** (LISP s-expressions).

It is possible to declare a parsing hierarchy stating which infix or prefix predicate or operator is to be evaluated first. Details on this procedure can be found in the FOL manual.

While FOL declarations may be inserted anywhere in **a** proof, it is a good practice to make all declarations at the begining of the proof. It is also good practice to declare **ARITY** one **PREDCONSTs (SORTs)** first, as it is possible to obtain an incorrect default declaration for **a** SORT by using it in another declaration prior to its own.

section 3.3 Operator declarations

Operator (function) declarations are the similar to predicate declarations. One can, for instance, declare one argument operators SIN and COS with:

```
*****DECLARE OPCONST SIN COS 1;
```

And, like **PREDCONSTs, OPCONST** can be declared to range **over** certain sorts. For example, the operator CONS on s-expressions and s-expressions can be declared:

```
*****DECLARE DPCONST CONS (SEXPR,SEXPR);
```

One'can also specify the range of an operator in a declaration. Thus, to declare an infix **OPCONST +,** of two arguments, on the domain of **NATNUMs** ⊗ NATNUMs into the set of NATNUMs, we say:

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

The operator-P I ECEON which returns the chesspiece on a given square of a given board is declared:

```
*****DECLARE OPCONST PIECEON(BOARD,SQUARE)=CHESSPIECE;
```

Unlike PREDCONSTs, declaring the domain and the range of an OPCONST imparts information to the proof checker. Such a declaration assures FOL that whenever the arguments of an operator are in the declared domain SORTs, the result of the function will be of the SORT of the range. Thus, with the declaration above, FOL knows that whenever the arguments of PIECEON are a BOARD and a SQUARE, respectively, the value of PIECEON will be a CHESSPIECE.

section 3.4 Other declarations

While FOL accepts several other SYNTYPEs in declarations, only two others are of interest to us. Individual variables are are declared like individual constants, with the word INDVAR substituted for INDCONST. Giving a SORT for the variable informs FOL that whenever this variable is used, it may be presumed to be of that SORT. INDVARs are used both as variables, bound in quantified WFFs, and as parameters, free in WFFs. Examples of declarations of INDVARs are:

```
*****DECLARE INDVAR x,y,z;
*****DECLARE INDVAR day date month ∈ TIME;
*****DECLARE INDVAR C1 C2 C3 ∈ PLAYINGCARDS:
```

The other important type of FOL identifier is the predicate parameter (PREDPAR). Any predicate may be substituted for a predicate parameter, provided their arguments match. Predicate parameters are found in induction axioms and the like; we will touch briefly upon their use later in this chapter.

section 4 TERMs and WFFs

In FOL, individual constants and variables, and result of the application of operators are examples of TERMs. Thus, with the obvious declarations, KingofSpades, Black_Kings_Rook_Pawn, 3+4, and Sin (Cos (Tan (5+3)*2)) are all examples of FOL TERMS. The arguments of predicates and operators are always terms. FOL also accepts TERMs representing sets, n-tuples, and LISP s-ex pressions. Details on these features may be found in the FOL manual.

A predicate, with its arguments, forms an ATOMIC WELL FORMED FORMULA, or AWFF, for short. Examples of AWFFs are:

```
SAMESUIT(KingofSpades,QueenofHearts)
3=Black_Kings_Rook_Pawn
RED(Sin(Cos(3*4)-2))
```

Any AWFF is a WELL FORMED FORMULA, (or WFF for short). WFFs may also be formed by joining WFFs with the infix sentential connectives ∧ (and), ∨ (or), ≡ (equivalence), and ⊃ (implication), and by prefixing the connective ¬ (not) to any WFF. These connectives have their usual propositional calculus meaning when used in a WFF. Examples of WFFs include:

```
KINGS(KingofSpades)∧SPADE(KingofSpades)
(TELEPHONE(x)⊃(IS_RINGING(x))⊃SOMEONEHASCALLED(x)))
¬(3+4=5+2⊃4*5*y=Sin(z))≡(F(x)=7∨y=z)
```

WFFs may be built by adding a quantified individual variable (INDVAR) to another WFF. Thus to state that the above WFF concerning SOMEONEHASCALLEO is true of all telephones, we write:

```
∀u.(TELEPHONE(u)⊃(IS_RINGING(u)⊃SOMEONEHASCALLED(u)))
```

To state that *for all x, **there** exists a y such that* y=Sin(x), we write:

```
Vx. ∃y.y=Sin(x)
```

The quantified variables need not appear in the matrix of the WFF. **For** example, the WFF, ***there*** *exists an x and a y such that 3-4,* would be:

```
3x y.(3=4)
```

Consider a WFF **A=∀x.B,** where B is also a WFF. The scope the quantified variable x is B. The variable **x** is said to be *bound* in the WFF A. Any variable bound in B is also bound in A. All other variables mentioned in A are said to be free in A. The same rules apply, of course, for the **WFFs** employing the 3 quantifier rather than the V quantifier. Thus, in the WFF:

```
Vx.(∃y.(x*0=y-y)⊃(z=3))
```

the INDVAR x and y are BOUND, while the INDVAR z is free. The scope of Vx in this WFF is ∃y.(x*0=y-y)⊃(z=3). The scope of ∃y in this WFF is (x*0=y-y).**WFFs** can be transformed into equivalent WFF by correctly renaming the bound variables. Thus, if INDVAR x and w are of the same SORT, the above WFF is equivalent to:

```
Vu,(∃y.(w*0=y-y)⊃(z=3))
```

Declaring an INDVAR to be an element of a SORT assures FOL that whenever that variable is used, it will be in that SORT. Thus, if the variable u was declared to be of SORT TELEPHONE, then our first ***forall*** statement about ringing telephone would be equivalent to:

```
Vu.(IS_RINGING(u)⊃SOMEONEHASCALLED(u))
```

section 5 Axioms

One way of expressing definitions and known facts about the world to FOL is by the use of AXIOMs. An **axiom.command** consists of the word AXIOM, followed by an identifier and a colon, a list of **WFFs** separated by commas, and terminated by two semicolons. If the "list of" **WFFs** contains only one WFF, then that WFF is referred to by the identifier. If more than one WFF is included in the list of **WFFs,** then they are **refered** to collectively by the identifier, or individually by new identifier formed by appending the number of the WFF in the list to the identifier. This is perhaps better explained by the use of an-example:

```
*****AXIOM ACES:
*       Vx y. ( (ACE (x) A-ACE (y))⊃CAPTURES(x y)),
*       Vs. (ISSUIT(s)⊃∃x.(ACE(x)∧SUIT(x)=s)),
*       SUIT (AceofSpades)=Spade,
*       Vx. (ACE (x)≡RANK(x)=Ace) ;;
```

The first WFF on this list is named **ACES1**, the second, ACES2, and so forth. They can be collectively referenced as ACES.

If we gave the following statement to FOL:

```
*****AXIOM ZEROADD: Vx.(x+0=x);;
```

then this WFF would be referable only by the name **ZEROADD**, not as **ZEROADD1**.

It is also possible to nest axioms, creating a hierarchy of names to refer to them. The FOL manual contains details on this procedure.

Axioms that contain predicate parameters (PREDPAR) are AXIOM SCHEMA. Any predicate may be substituted for the PREDPAR in an axiom schema, provided it has the same number of arguments as the PREDPAR which it is replacing. Axiom schema are useful in axioms that are valid for any properly used predicate, such as induction axioms. Information on the use of axiom schema is in the FOL manual.

It is important to remember that a proof is only as valid as the axioms upon which it is based. From axioms that do not describe the "real" situation, it is possible to prove "unreal" theorems; from axioms that are mutually contradictory, it is possible to prove anything.

sect ion 6 The FOL Proof

There are three kinds of commands that can be given to FOL: Declarative commands, of which declarations and axioms are examples, administrative commands, which are used to obtain information about the state of the proof, and proof generating commands.

A FOL proof consists of a set of declarations, axioms, and other declarative commands, followed by a series of legal FOL inferences. Each of these deductions generates a line of a PROOF. Proof steps are numbered consecutively, starting with line 1, and may be refered to by their line number. These lines may or not be dependent upon other steps of the proof. The validity of any line with a dependency has not been proven to be greater than that of the steps it is dependent upon. A line with no dependencies is a theorem in the given axiom system.

Each command for generating a line of a proof is a *rule of inference*. Each such rule requires stating either the WFF that is to be infered, a list of previous proof lines and axioms from which to make the inference, or both of these.

section 6.1 Introduction to the Two by Two Colored Grid World

To demonstrate the various proof generating commands in FOL, we shall axiomatize a small section of the *two by two colored grids* ***world***. Conceptually, the two by two colored grid world consists of square ***grids*** of four ***squares***, each colored with one of a set of colors, in this case, RED, GREEN, YELLOW and BLUE. The squares are identified as **S11, S12, S21** and **S22,** where we use the numbers as row and column coordinates. This is a sample grid:

S11 RED	*S12* BLUE
S21 GREEN	s22 RED

A grid

We want each of the four colors, and each of the four squares, to be different; that is, RED is not equal to GREEN, and **S11 is** not S22. We can speak of the **COLOROF** a particular grid-square combination, and the next square clockwise, of any square, an operator we designate as "→". The FOL declarations of these facts look like:

```
*****DECLARE PREDCONST COLOR SQUARE GRID 1[PRE];
*****DECLARE INDCONST RED GREEN YELLOW BLUEεCOLOR;
*****DECLARE INDCONST S11 S12 S21 S22 ε SQUARE;
*****DECLARE OPCONST COLOROF(GRID SQUARE)=COLOR;
*****DECLARE OPCONST →(SQUARE)=SQUARE;
```

The reader with access to the FOL proof checker is urged to run the FOL program, and give it the commands in this section. Remember that the five asterisks are the FOL prompt, typed by FOL, not the user.

We **have** declared the **ARITY** one **PREDCONSTs** to be prefix operators. This permits us to use them with or without parentheses around their arguments.

We will also want variables for each of our SORTs, so the following declarations will prove handy:

```
*****DECLARE INDVAR G G1 G2 ε GRID:
*****DECLARE INDVAR S S1 S2 ε SQUARE;
*****DECLARE INDVAR C C1 C2 CA CB ε COLOR:
```

As grids are composed of their four colors, we will have use for an operator MAKEGRID, which takes four colors, and forms a grid from them. We declare it:

```
*****DECLARE OPCONST MAKEGRID(COLOR,COLOR,COLOR,COLOR)=GRID;
```

The ext ent of the SORT of colors and the SORT of squares may be axiomatized with t he following two **AXIOMs.** We will later consider a better way of telling FOL about these two finite sets.

```
*****AXIOM EXTCOLOR:∀C.(C=RED∨C=GREEN∨C=YELLOW∨C=BLUE),
*                      ¬RED=GREEN,
*                      ¬RED=YELLOW,
*                      ¬RED=BLUE,
*                      ¬GREEN=YELLOW,
*                      ¬GREEN=BLUE,
*                      - YELLOW- BLUE; ;

*****AXIOM EXTSQUARE:∀S.(S=S11∨S=S12∨S=S21∨S=S22),
*                      ¬S11=S12,
*                      ¬S11=S21,
*                      ¬S11=S22,
*                      ¬S12=S21,
*                      ¬S12=S22,
*                      ¬S21 =S22; ;
```

The fact that any grid is the s um of the colors of its squares is expressed by the axiom GRIDSIZE.

```
*****AXIOM GRIOSIZE:
*       ∀G. (G=MAKEGRID(COLOROF(G S11)
*                       COLOROF (G S12),
*                       COLOROF (G S21),
*                       COLOROF(G S22)));;
```

And the particular clockwise ordering of the squares by the axiom NEXTSQUARE.

```
*****AXIOM NEXTSQUARE: '  →(S11)=S12,
*                          → (S12) =S22,
*                        . →(S22)=S21,
*                          →(S21)=S11;;
```

We are now free to explore t he concepts involved in the two by two grid world. Let us begin by declaring four predicates that express facts about grids that interest us.

We wish the predicate HAS, on grids and colors, to be true whenever one of the squares of the given grid is the stated color. Similarly, FREEOF shall be true if no square of the given grid is of t he given color. ALLTHESAHE states that every square of its grid argument is the s ame color, ALLDI FFERENT, that each square is colored differently. A grid is PLAID if the squares alternate in color. Thus, we get t he following declarations and **axioms:**

```
*****DECLARE PREDCONST ALLTHESAME (GRID) [PRE];
*****DECLARE PREDCONST ALLOI FFERENT (GRID) [PRE];
*****DECLARE PREDCONST HAS(GRID COLOR):
*****DECLARE PREOCONST PLAID(GRID) [PRE];
*****DECLARE PREDCONST FREEOF(GRID COLOR);
```

```
*****AXIOM  DEFINITIONS:
*           ∀G.(ALLTHESAME G≡∃C. ∀S. COLOROF(G S)=C),
*           ∀G.(ALLDIFFERENT G≡
*                 ∀S1 S2. (COLOROF(G S1)=COLOROF(G S2)⊃S1=S2)),
*           ∀G C.(HAS(G C)≡∃S.COLOROF(G S)=C),
*           ∀G. (PLAID G≡∀S1.∃C1 C2. (¬C1=C2∧(COLOROF(G S1)=C1∧
*                                       COLOROF(G →(S1))=C2∧
*                                       COLOROF(G →(→(S1)))=C1))),
*           ∀G C. (FREEOF(G C)≡∀S.¬COLOROF(G S)=C);;
```

sectiori 6.2 Sample Proof in the Grids World

For our first grids world proof, we wish to prove that for any grid, G, **if** all the squares of G are colored identically (ALLTHESAME), and the color of (COLOROF) square S21 on G is RED, then G has no square that **is** GREEN (it is FREEOF GREEN).

The first proof step generating command is the ASSUME command, described in our proof of the mortality of Socrates. FOL will remember the dependence of those sections of the proof that rely on **an** assumption. Remember that **a theorem** is **not** proven until it **is free of dependencies.**

We **are** trying to prove the WFF:

```
∀G.((ALLTHESAME G∧COLOROF(G,S21)=RED)⊃FREEOF(G,GREEN))
```

arid find it useful to assume the first half of the implication, for some general variable, G. If we can prove this **true** for any G, we will be able to universally generalize, and assert its truth for **all G.**

To repeat, commands to FOL begin after the five star prompt, and continue until the semicolon. The numbered lines are the proof steps that FOL prints. **Statements in this font are** commands typed to **FOL;** Statements in this font are **FOL's** responses. In this paper, some of **FOL's** responses **have** been reformatted for easier reading.

```
*****ASSUME ALLTHESAME G∧COLOROF(G,S21)=RED;

1 ALLTHESAME G∧COLOROF(G,S21)=RED (1)
```

The parenthesized one at the end of the line signifies that this proof step is dependent upon line one. As line one **was** an assumption, this is only fair. Any deduction that uses line one will also be dependent upon line one. Proof steps may be dependent upon several other lines.

Many axioms and proof steps are of the form "for all x , y . . . " where x, y, . . . is a list of variables. For instance, all of the axiom DEFINITIONS is of this form. The FOL command for specializing an axiom or proof step whose major connective is a ∀ to a particular individual is ∀E. ∀E stands for *forall elimination.* For each of the sentential connectives and quantifiers (∧, ∨, ≡, ⊃, ¬, ∀, and ∃) FOL has **a** rule for the introduction of that connective, and another for its elimination. These are abbreviated with the two character command formed by joining the connective or quantifier in question, with an ***I*** or an ***E*,** depending upon whether that connective (or quantifier) is to be introduced into or eliminated from the given WFF. As most of the functions of these rules can be done by the TAUT and TAUTEQ rules, we shall consider only those among these commands that cannot be done any other way.

We assumed that the grid G is an ALLTHESAME grid. What is an ALLTHESAME grid? The axiom DEFINITIONS1 tells us. To use it, we must specialize its variable to G. To eliminate the ∀ character, we use the ∀E rule. The syntax of the ∀E rule is the word ∀E, followed by either the name of a WFF that is an axiom (that is, DEFINITIONS1, not DEFINITIONS) or a line number, followed by a list of TERMs. If the SORT of a TERM is not the same as (or *less general* than), the SORT of the variable it wishes to instantiate, FOL will insert that as a condition of the proof step. The ∀E command is terminated by a semicolon (of course). This rule can only be applied to proof steps or axioms that have as their *main connective a* ∀, followed by at least as many variables as we wish to eliminate.

In the, future, we will refer to the concept of *proof step or axiom that is a* WFF as a VL. Each VL has its own dependencies. The origin of the name VL is obscure.

```
*****∀E DEFINITIONS1 G ;

2 ALLTHESAME G≡∃C.∀S.COLOROF(G,S)=C
```

The line now refers to a specific grid, G, rather than to all grids, as the axiom did. Note that since this fact is obtained exclusively through the use of an axiom, this line has no dependencies.

The TAUT command, introduced in the first proof, will decide all propositional tautologies. The command consists of the word TAUT, followed by a WFF, followed by a list of line numbers and axiom names, separated by commas. This list is called the *reason list* for this inference. If the WFF is a tautological consequence of the given lines and axioms, then a new proof step with that WFF is created; if not, an error message is printed. The second half of line two is a tautological consequence of lines one and two; we obtain it in one step.

```
*****TAUT  ∃C.∀S.COLOROF(G,S)=C  1,2;

3 ∃C.∀S.COLOROF(G,S)=C (1)
```

Since a line with a dependency of line one was used in the reason list of this TAUT, this line is also dependent upon 1.

Note that the tautology decider can determine the equivalence of WFFs identical except for the renaming of bound variables, but cannot look *inside* the matrix of a quantified WFF any further than that. That is the reason for the heavy emphasis on the ∀E and the ∃E rules.

Line three states that there exists some color, C, such that for all squares, S, the COLOROF(G,S) is C. We want to have a name for this color, so we use the ∃E command used in the second proof. We apply it to the WFF on line three.

```
*****∃E 3 c;

4 ∀S.COLOROF(G,S)=C (4)
```

This line is dependent upon itself (4), not on line one. This is because by using the ∃E command, we have said, in effect, "Assume that this fact is true of the individual C". Any statement we can prove which is not dependent upon the fact that we called the variable C will not have four as a dependency. Instead, FOL will make that proof step dependent upon the same lines as step three depends upon.

We now have the fact that COLDROF (G,S) =C for all squares S. We wish to apply this to two different squares. First, we know what the COLOROF(G S21) is (red) from our original assumption. This will tell us that C is RED. Secondly, we are going to want to make a statement about all squares S (that the COLOROF (G S) is not GREEN), so we are going to want to instantiate line four to a general S. Lines five and six are the result of these commands.

```
*****∀E 4 S21 ;

5 COLOROF(G,S21)=C (4).

*****∀E 4 S;

6 COLOROF(G,S)=C (4)
```

Recall the second command for deciding tautologies, **TAUTEQ**. Whereas TAUT knows only about propositional tautologies, **TAUTEQ knows** the meaning of the predicate =, and can substitute equals for equals in predicates, though not in operators. While TAUTEQ is more powerful than TAUT, it is also much slower, and its use, when not required, is not recommended.

```
*****TAUTEQ ¬COLOROF(G,S)=GREEN 1,5,6,EXTCOLOR;

7 ¬(COLOROF(G,S)=GREEN) (1)
```

Here the use of the axiom name EXTCOLOR refers to all of the subparts of EXTCOLOR. Note that the dependency on line 4 has disappeared. This dependency was introduced by an **∃E**, and, like the similar dependencies in the second proof, has been removed when the named variable (C) is out of the proof.

Having established this fact for the square S, we wish to generalize it to all S. This process, which is often called universal generalization, is accomplished through the use of the ∀I command (for *for-all introduction*).

```
*****∀I 7 s;

8 ∀S.¬(COLOROF(G,S)=GREEN) ( 1 )
```

The axiom DEFINITIONS5 states that a grid is free of a color if for all squares S, the color of that grid on that square is not the given color. Just like in step two, we use the instantiation rule for **forall** statements with this axiom, getting

```
*****∀E DEFINITIONS5 G GREEN;

9 FREEOF(G,GREEN)≡∀S.¬(COLOROF(G,S)=GREEN)
```

Our tautology decider can now tell us that grid G is free of GREEN. We give the TAUT rule the WFF we wish to establish, and the VLs which imply it.

```
*****TAUT FREEOF(G GREEN) 8 9:

10 FREEOF(G,GREEN) (1)
```

This was the conclusion we desired. To remove the dependency upon line one, we introduce an implication, with line one as the premise, and line ten as the conclusion. This is done by the use of the ⊃I command. Of all of the commands we have considered so far, only the ⊃I command removes dependencies. (Dependencies caused by the use of ∃E commands are removed when the variable or variables instantiated by that command are no longer in the proof). Later, we shall also consider the ¬I and ¬E rules, which also eliminate dependencies.

```
‘*****⊃I 1⊃10;

11 (ALLTHESAME G∧COLOROF(G,S21)=RED)⊃FREEOF(G,GREEN)
```

We note that line eleven is not dependent upon any other line. It is true, to the extent that our axiom system is valid. Being free of dependencies, we can generalize its variables to all individuals of their SORTs. (FOL will not let us generalize proof steps that contain variables which are *free* in their dependencies -- to do so would be equivalent to letting us conclude that because some single apple x is red, that all apples are red.

The command for generalizing a proof step to all elements of a SORT is, once again, **forall** introduction, ∀I.

```
*****∀I 11 G:

12 ∀G.((ALLTHESAME G∧COLOROF(G,S21)=RED)⊃FREEOF(G,GREEN))
```

Line twelve is our desired theorem. Note that it is free of dependencies. Any line with a dependency is not a valid theorem.

section 7 Simplification mechanisms

One of the motivations for the use of predicate calculus in artificial intelligence research is as a vehicle for the expression of reasoning in a well understood machine manipulative form. Not all intelligent action is based purely on -deduction; in fact, most of human intelligence relies more upon observation than reasoning. 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 and the axioms of number theory.

FOL has a method of doing purely computational tasks. SIMPLIFY permits the *attachment* of computational functions and predicates in the programming language LISP to the operators and predicates of the FOL proof structure. When a LISP function is attached to a FOL operator, we are assuring FOL that the value of that function can be computed by evaluating the associated LISP function.

This paper will not attempt to explain the use of the LISP language. The reader unfamiliar with LISP is refered to McCarthy[1962] or Weissmann [1967].

In FOL, the mapping between the FOL proof structure and LISP is generated by the ATTACH command. Attachment generates no proof steps; rather, it is a declaration, and like the other declarative commands, may contain axiomatic information.

FOL permits mapping between PREDCONSTs and LISP predicate functions, OPCONSTs and LISP functions, and INDCONSTs and LISP atoms and s-expressions. The specified map can be either one *way,* from FOL to LISP, or, in the case of INDCONSTs, two *way,* in both directions.

Let us return to the two by two colored grid world. We wish our proof checker to be capable of manipulating these grids, so we need an internal representation for them. FOL allows a user to have more than one representation of each object in LISP. The details are beyond the scope of this primer, but may be found in the FOL manual. In cases where only one representation is being used the following command must precede all attachment commands.

```
*****REPRESENT * AS UNIVERSE:
```

A suitable internal format is to represent a grid as a two element list, with each element being a two element list of colors. The FOL command to declare this attachment is:

```
*****ATTACH GRID (DE GRID (L) (AND
*                   (EQ (LENGTH L) 2)
*                   (EQ (LENGTH (CAR L)) 2)
*                   (EQ (LENGTH (CADR L)) 2)
*                   (COLOR (CAAR L))
*                   (COLOR (CADAR L))
*                   (COLOR (CAADR L))
*                   (COLOR (CADADR L))));
```

Here *LENGTH* is the standard LISP LENGTH function that returns the integral length of the top level of a list. But the predicate COLOR is unknown to LISP. Hence, we must also define an attachment to COLOR. The LISP function MEMQ provides a convenient method.

```
*****ATTACH COLOR (DE COLOR (X)
*             (MEMQ X (QUOTE (RED GREEN YELLOW BLUE)))I;
```

Note the form of an attachment to a function or predicate: the word ATTACH, followed by the predicate's name, a LISP function, and a semicolon. The LISP function can be a standard, predefined function, like CAR or LENGTH, a DEFPROP or DE expression (which also puts the function of that name on the property list of that atom), or a LISP LAMBDA expression.

The given attachments are still not enough, as we must tell FOL that the LISP atoms RED, GREEN, YELLOW and BLUE are to correspond to the FOL INDCONSTs RED, GREEN, YELLOW and BLUE. This leads us to the other kind of attachment statement, where an INDCONST is attached to a LISP atom or s-expression. The appropriate FOL commands would be:

```
*****ATTACH RED ↔ RED;
*****ATTACH GREEN ↔ GREEN;
*****ATTACH YELLOW ↔ YELLOW;
*****ATTACH BLUE ↔ BLUE;
```

The ↔ in this command tells FOL that the stated map is two way; that if the atom YELLOW is the result of the evaluation of a LISP function that is attached to a FOL OPCONST, then that atom YELLOW is meant to correspond to the FOL INDCONST YELLOW. If we had wanted to represent YELLOW internal to LISP as the atom Y, we would have said:

```
*****ATTACH YELLOW ↔ Y;
```

The simplification structure knows of yet another type of declaration, the EXTENSION command. Many real problems, and especially those of a non-mathematical nature, deal with small finite sets. For example, the set of *playing cards in the deck, the people in the room where the body was found,* and colors *in* the *grids world* are all examples of small finite sets. Declaring the extension of such a set enables the SIMPLIFY command of FOL to do two things: to search the items in that set when seeking to satisfy a *forall* or *there exists* statement, and implicitly to differentiate between the items of the extension set. That is, if a SORT P has an extension of A B C and D, then the simplification mechanism may use the fact that A≠B, A≠C, B≠D, and so forth. Either of these properties can be simulated without the use of the extension statement; the first, through the use of an axiom listing the elements of the set, the second through another axiom explicitly stating the inequalities involved. But both of these methods are long and clumsy, and are best avoided.

The FOL extension command consists of the word EXTENSION, followed by a SORT, and a set expression (a set, or the union or intersection of sets) which constitutes the extension of that SORT. Other SORTs whose extension has already been defined may be used as sets in an extension command. In our two by two colored grids world example, the extension of COLOR can be declared by the statement:

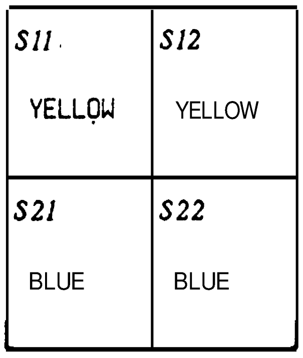
```
*****EXTENSION COLOR {YELLOW BLUE GREEN RED};
```

After this declaration, the simplification mechanism can conclude the axiom EXTCOLOR. Similarly the axiom EXTSQUARE can be replaced by the declaration:

```
*****EXTENSION SQUARE {S11 S12 S21 S22};
```

The same INDCONST may belong to more than one extension; however, the only things FOL will permit to be declared elements of an extension are INDCONSTs. (remembering, of course, that LISP S-expressions and the natural numbers are also INDCONSTs.)

Suppose we have a particular grid in mind. We might want to call the grid whose top *row is* YELLOW and whose bottom row, BLUE, by the name, MYGRID.



HYGRID

We would make the declaration:

```
*****DECLARE INDCDNST MYGRID ∊ GRID:
```

Following ‘the structure for grids mentioned above, we’d want to include the attachment:

```
*****ATTACH MYGRIO ↔ ( (YELLOW YELLOW) (BLUE BLUE) ) ;
```

Let us make the following attachments to the PREDCONSTs HAS and FREEOF. Note that the function FREEOF is defined in terms of the function’ HAS.

```
*****ATTACH HAS (DE HAS (G C)
*                (AND (GRID G)
*                   (OR
*                      (MEMQ C (CAR G))
*                      (MEMQ C (CADR G))))) ;
*****ATTACH FREEOF (LAMBDA (G C) (AND (GRID G) (NOT (HAS G C)))) ;
```

We can then- give FOL the following commands, obtaining the stated proof steps.

```
*****SIMPLIFY ¬YELLOW=BLUE;

1 ¬(YELLOW=BLUE)

*****SIMPLIFY HAS(MYGRID YELLOW) ;

2 HAS(MYGRID,YELLOW)

*****SIMPLIFY HAS(MYGRID RED) ;

3 ¬HAS(MYGRID,RED)

*****SIMPLIFY FREEOF(MYGRID GREEN) ;

4 FREEOF(MYGRID,GREEN)
```

```
*****SIMPLIFY ∀C.(HAS(MYGRID C)⊃((C=YELLOW∨C=BLUE)∧¬C=RED));

5 ∀C.(HAS(MYGRID,C)⊃((C=YELLOW∨C=BLUE)∧¬(C=RED)))
```

The simplify mechanism has allowed us to deduce these facts by simple commands. Each of them would require either a complicated derivation, or a specific axiom, to do without simplification.

By use of the FUNCTION command, it is possible to declare auxiliary functions to FOL. Functions so declared may be used, like the standard LISP functions, in attachments and other function statements. Only **EXPRs** (not **FEXPRs** or MACROs) may be declared by the FUNCTION command. If we had declared the auxiliary function **MAKEPAIR** by

```
*****FUNCTION (DE MAKEPAIR (X Y)(CONS X (CONS Y NIL)));
```

then an *attachment to **MAKEGRID** would be:

```
*****ATTACH MAKEGRID (LAMBDA (A B C D)
*               (MAKEPAIR (MAKEPAIR A B) (MAKEPAIR C D)));
```

The value of this command is not immediately apparent in this simple example. However, as attachments increase in complexity its worth becomes more obvious.

section 7.1 The Grids World Revisited

As a final sample proof, we present a proof by contradiction, intended to illustrate the ¬I and substitution proof generating commands, and a few of the proof administrative commands.

In this proof, we seek to show that if a grid is plaid, and square *S12* is *RED*, then square *S11* is not *RED*. Expressed in predicate calculus, this looks like:

$$\forall G.((PLAID\ G \wedge COLOROF(G\ S12)=RED) \supset \neg COLOROF(G\ S11)=RED)$$

For this proof it will be useful to have an attachment for the operator →; the auxiliary function NEXT is defined for the use of the attachment function. Similarly, we must tell FOL of the mapping between the external, FOL interpretation of the individual squares, and the internal, LISP representation. For this, we have chosen the obvious mapping of the square names into themselves. The following declarations are therefore presented to FOL.

```
*****FUNCTION (DE NEXT (S L)
*      (COND ((EQ (CAR L) S)(CADR L)) (T (NEXT S (CDR L)))));

*****ATTACH → (DE → (S) (NEXT S (QUOTE (S11 S12 S22 S21 S11))));

*****ATTACH S11 ↔ S11;
*****ATTACH S12 ↔ S12;
*****ATTACH S21 ↔ S21;
*****ATTACH S22 ↔ S22:
```

It is often convenient to refer to lines in a proof not by their line number (which is apt to change if a few lines are added or removed from the proof), but rather by some symbolic or relative tag. FOL provides several mechanisms for doing this. One of these is the LABEL command. The basic form of this command is the word LABEL, an identifier, and a line number. If the line number is omitted, the next line is presumed. Once an identifier has been declared a label, it may be used wherever that line number could be used. In this example, the next command given is

```
        *****LABEL THISLINE;
```

As no implicit line number is given, THISLINE is presumed to apply to the next proof step (line 1). THISLINE can now be used as a synonym for the VL 1 in any FOL command. Note that the label is not permanently attached to the named line; a label may be redeclared for a different line.

As the theorem we wish to prove is a universal generalization of A⊃B, we find it useful, in this proof, to start by assuming the "A" part, for some general grid G.

```
        *****ASSUME PLAID G∧COLOROF(G,S12)=RED;

        1 PLAID G∧COLOROF(G,S12)=RED (1)
```

We want a proof by contradiction, so we assume the contrary of the desired result. If our assumption in line two leads to a contradiction (a conclusion of FALSE) then we will have proven our theorem.

```
        *****ASSUME COLOROF(G,S11)=RED;

        2 COLOROF(G,S11)=RED (2)
```

We now consult our definition of a plaid grid. The axiom DEFINITIONS4 defines plaid grids; we use VE to instantiate it to our grid G.

```
        *****∀E DEFINITIONS4 G;

        3 PLAID G≡∀S1.∃C1 C2.(¬(C1=C2)∧(COLOROF(G,S1)=C1∧(COLOROF(G,→(S1))=C2∧
                                         COLOROF(G,→(→(S1)))=C1)))
```

We will wish to apply the second part of this VL to two different squares; to do so, we must first isolate the quantified part. The TAUT command will get it for us. Note the introduction of the subpart designator, the :#2 in the taut command. FOL provides a method for refering to part of a previously mentioned WFF by use of a *subpart designator*. The WFF that is *on* any VL may be accessed by appending a : to that VLs name; successive first operands and second operands of the *main connective* of a WFF are accessed by appending #1's and #2's to the colon. For instance, the main connective of line two is =, the first operand, the TERM COLOROF(G,S11). Thus, in this proof, FOL will treat 2:#1 as synonymous with COLOROF(G,S11). Similarly, 1:#2#1#1 is the INDVAR G. The reader is urged to experiment with subpart designators; the relief from the recopying of long WFFs is certainly a good incentive for their use. Subpart designators may be used wherever a WFF or TERM is required. Care must be exercised, however, to insure that the expression designated is the desired one.

Let us call the resulting VL ALLSQ.

```
*****LABEL ALLSQ;
*****TAUT 3: #2 1,3;

4 ∀S1.∃C1 C2.(¬(C1=C2)∧(COLOROF(G,S1)=C1∧(COLOROF(G,→(S1))=C2∧
                                COLOROF(G,→(→(S1)))=C1))) (1)
```

This fact is to be applied first to the square S12. Here we use another method of refering to other lines in FOL, the ↑. When used in place of a line number, a single ↑ refers to the previous line; each additional ↑ moves the referent back a line. Thus, the string ↑↑ refers to the proof step two lines ago; the line five back would be ↑↑↑↑↑.

```
*****∀E ↑ S12;

5 ∃C1 C2.(¬(C1=C2)∧(COLOROF(G,S12)=C1∧(COLOROF(G,→(S12))=C2∧
                                COLOROF(G,→(→(S12)))=C1))) (1)
```

The definition so produced refers to colors C1 and C2; to get to the matrix of this expression, we need to remove the quantifier.

```
*****∃E ↑ C1 C2;

6 ¬(C1=C2)∧(COLOROF(G,S12)=C1∧(COLOROF(G,→(S12))=C2∧
                                COLOROF(G,→(→(S12)))=C1)) (6)
```

Note that this line was produced by an existential elimination, and is therefore dependent only upon itself.

We can consult the simplification mechanism for the value of →(→(S21)) (the square two away, clockwise, from the upper right hand corner)

```
*****SIMPLIFY →(→(S12));

7 →(→(S12))=S21
```

The last remaining important class of proof step generating commands are the substitution commands, SUBST and SUBSTR. A substitution command is of the form SUBST <*vl*> IN <*vl*>;. The first <*vl*> is presumed to be of the form A-B or A≡B; if SUBST is used, every occurrence of B in the second <*vl*> is replaced by A, and the resulting WFF becomes the next proof step, its dependencies the union of the dependencies of the first and second <*vl*>s. SUBSTR is used for replacing A by B.

Substitutions may occasionally result in the automatic renaming of a bound variable. This is rare, but its occurrence should not unduly alarm the user.

```
*****SUBSTR ↑ IN ↑↑;

8 ¬(C1=C2)∧(COLOROF(G,S12)=C1∧(COLOROF(G,→(S12))=C2∧
                                COLOROF(G,S21)=C1)) (6)
```

Thus, we have established the color of square S21 in grid G. Another instantiation to line four will allow us to say something about the desired square, S11.

```
*****∀E ALLSQ S21;

9 ∃C1 C2.(¬(C1=C2)∧(COLOROF(G,S21)=C1∧(COLOROF(G,→(S21))=C2∧
                                        COLOROF(G,→(→(S21)))=C1))) (1)
```

Once again we need names for the two colors in line nine. Calling them Cl and C2 will lead to trouble; we have not established that they are the same as the colors referred to in line six (even though they can be proven to be the same). So let us call them CA and CB. Note the renaming by the ∃E command.

```
*****∃E ↑ CA CB;

10 ¬(CA=CB)∧(COLOROF(G,S21)=CA∧(COLOROF(G,→(S21))=CB∧
                                COLOROF(G,→(→(S21)))=CA)) (10)
```

The successor square to S21 is S11. Notice that SIMPLIFY can evaluate both operators and predicates.

```
*****SIMPLIFY S11=→(S21);

11 S11=→(S21)
```

And another substitution, this time with the SUBST command.

```
*****SUBST ↑ IN ↑↑ OCC 1;

12 ¬(CA=CB)∧(COLOROF(G,S21)=CA∧(COLOROF(G,S11)=CB∧
                               COLOROF(G,→(S11))=CA)) (10)
```

We see here another modification of the substitution commands, the *occurrence list*. If the substitution command has, before the semicolon, the word OCC, followed by an ordered list of integers, then the substitution is made only for those instances specified by the occurrence list. That is, if the command were SUBST *thisline* IN *thatline OGC* 1 3 5;, then the substitution would occur only in the first, third, and fifth occurrences in *thatline*.

This proof has produced a contradiction; lines 1, 8, and 12 imply that square S11 cannot be RED; the assumed line 2 demands it. We can use TAUTEQ to infer the WFF FALSE. In this command, we see yet another way of giving a series of line numbers to a FOL command, the RANGELIST. The lines from line M to line N, inclusive of M and N, can be shortened to M:N. If the M is omitted, the first line of the proof is presumed; if the N is absent, the last line of the proof (↑) is assumed. Note that we have given TAUTEQ not only lines 1,2,8 and 12, needed for the contradiction, but also the superfluous lines 9, 10 and 11.

```
*****TAUTEQ FALSE THISLINE,2,8:↑;

13 FALSE (1 2)
```

The variables Cl, C2, CA and CB are no longer mentioned in this line, (or in any line which this line depends upon), hence this line no longer depends on line 5.

We come to the other major FOL command for removing dependencies, ¬I. If a contradiction (FALSE) has been generated as a proof step, then we can conclude, dependent upon the other lines in the dependency of the FALSE line, the negation of one of the assumptions that caused that line, If the FALSE line has no dependencies, there is a contradiction in the axiom system.

There also exists as ¬E command, parallel in use to the ¬I command, for the elimination of a negation, rather than its introduction.

We illustrate here another type of relative line numbers. A label can have appended an additive or subtractive constant, which then refers to that many lines before or after the line associated with that label. Thus, LINEX+3 is the third line after the line LINEX; LINEX-1 is the line before LINEX.

```
*****¬I ↑, THISLINE+1;

14 ¬(COLOROF(G,S11)=RED) (1)
```

All that now remains is the insertion of the implication symbol, and the generalization of this proof to all grids.

```
*****⊃I THISLINE⊃↑;

15 (PLAID G∧COLOROF(G,S12)=RED)⊃¬(COLOROF(G,S11)=RED)

*****∀I ↑ G←G1;

16 ∀G1.((PLAID G1∧COLOROF(G1,S12)=RED)⊃¬(COLOROF(G1,S11)=RED))
```

Notice the renaming (from G to G1) accomplished in this use of ∀I. The WFF on line 16 is, of course, equivalent to the same WFF with G1 replaced by G (or, by G2, for that matter).

We have proven the theorem.

The astute reader may have noticed several shortcomings and have several questions about our proof. It is certainly longer than it needs be, and a much more powerful theorem relating the difference of colors of squares on plaid grids could certainly have been proven with the same effort. But the purpose of this exercise was to introduce the use of the line referents and the substitution commands, not to prove powerful grids world theorems. The reader may also reasonably inquire about the necessity of the substitution commands. They are needed because the TAUTEQ command is capable of substituting equals for equals in predicates, but not in functions. That is, if a and b are individuals, P a predicate, and F a function, TAUTEQ can conclude from a=b that P(a)≡P(b), but cannot conclude that F(a)=F(b). One can establish the equality of F(A) and F(B) using the SUBST and SUBSTR commands. .

section 8 Administrative commmands

section 8.1 The show commmand

Several administrative commands are worth mentioning. The SHOW command displays details of the present proof structure on your console. There are several varieties of SHOW. One can have the current proof listed by responding to the five star prompt by typing SHOW PROOF; . SHOW PROOF 42, LINEX, ↑, LINEX+2:↑↑↑↑; would list, in order, line 42, the line labeled LINEX, the previous line, and all lines from two lines after LINEX to four lines ago.

The other show commands are:

```
*****SHOW AXIOMS;
          lists all axioms;
*****SHOW AXIOMS DEFINITIONS,EXTCOLOR1;
          lists both axioms DEFINITIONS and EXTCOLOR1.
*****SHOW DECLARATIONS <syntype>;
          lists all things declared to be of that syntype;
*****SHOW DECLARATIONS <list of identifiers,;
          lists the declaration information for each identifier in the list.
*****SHOW GENERALITY <SORT list>;
          lists the SORTS both more and less general than the requested SORTs.
*****SHOW LABELS <range>;
          lists the labels and associated line numbers in that range.
*****SHOW LABELS <identifier list>;
          lists the line number for each identifier in the list.
```

.For example, the various show commands, in the context of the last proof:

```
    *****SHOW PROOF THISLINE,7,↑↑↑:↑;

    **&ASSUME PLAID G∧COLOROF(G,S12)=RED;

    1 PLAID G∧COLOROF(G,S12)=RED (1)

    *****SIMPLIFY ;

    7 →(→(S12))=S21

*****¬I    ↑,COLOROF(G,S11)=RED;

    14 ¬(COLOROF(G,S11)=RED) (1)

    *****⊃I 1⊃↑ ;

    15 (PLAID G∧COLOROF(G,S12)=RED)⊃¬(COLOROF(G,S11)=RED)
```

```
     *****∀I ↑ G←G1;

     16 ∀G1.((PLAID G1∧COLOROF(G1,S12)=RED)⊃¬(COLOROF(G1,S11)=RED))

  *****SHOW AXIOMS DEFINITIONS3,NEXTSQUARE;

     DEFINITIONS3: ∀G C.(HAS(G,C)≡∃S.COLOROF(G,S)=C)
     NEXTSQUARE:  NEXTSQUARE1: →(S11)=S12
                  NEXTSQUARE2: →(S12)=S22
                  NEXTSQUARE3: →(S22)=S21
                  NEXTSQUARE4: →(S21)=S11

     *****SHOW DECLARATIONS INDCONST:

     INDCONST
      MYGRID S22 S21 S12 S11 BLUE YELLOW GREEN RED

     *****SHOW DECLARATIONS G,MAKEGRID,COLOR,YELLOW,FOOBAZ;

     G is INDVAR of SORT GRID

     MAKEGRID is OPCONST ,
     The domain is COLOR ⊗ COLOR ⊗ COLOR ⊗ COLOR, and the range is GRID

     COLOR is PREDCONST
     The domain is UNIVERSAL[R←1000]
     COLOR is a SORT with:
             INDCONSTs BLUE YELLOW GREEN RED
             INDVARs CB CA C2 C1 C

     YELLOW is INDCONST of sort COLOR

     No declaration for FOOBAZ


     *****SHOW LABELS 1:12;
     label ALLSQ 6
     label THISLINE 1

     *****SHOW LABELS THISLINE;

     THISLINE:  1 PLAID G∧COLOROF(G,S12)=RED
```

The output of any show command can be directed to a file by inserting a right arrow and a file name before the closing semicolon. Thus, the command:

```
     *****SHOW PROOF 9:475 → THIS.PRF[FOL,REF];
```

will list the proof steps nine through 475 onto the file THIS.PRF[FOL,REF].

section 8.2 The Backup File

FOL keeps a transcription of everything typed to it on the file BACKUP.TMP. If the system crashes during an execution of FOL, this file may be edited to recover the lost input.

section 8.3 Erasing Proof Steps

Lines may be deleted from the proof by the use of the CANCEL command. CANCEL <linenumber>; removes all lines of that number or greater from the proof; CANCEL: removes just the last line. As the dependence of each proof step on the other lines of the proof is not easily determined, FOL does not allow for removal of lines from the middle of proofs.

sect ion 8.4 Reading in FOL Command Files

Input files of FOL commands may be read into FOL by the use of the FETCH command. FETCH *<filename>*; opens the file *<filename>*, and reads and executes the commands on that file; FETCH *<filename>* FROM *<identifier>;* searches the *file <filename>* for a command MARK *<identifier>:* and begins reading from that point; FETCH *<filename>* TO *<identifier>:* reads the file until MARK *<identifier>* ; is encountered. The FETCH command may contain both a FROM and a TO marker. For example,

```
*****FETCH INPUT.FOL FROM MARK1 TO MARKEND;
```

would begin reading the file INPUT.FOL, and search for the command

```
MARK MARK1;
```

FOL would begin processing commands from that point, until the command

```
MARK MARKEND;
```

was read. Control would then be returned to the previous level.

Fetches may be nested (to a depth of ten) ; that is, a fetched file may use the fetch command. Remember, the five asterisks so frequently repeated in this paper are FOL's prompt, and not part of the command. If you intlude them in a fetched file, an error message results.

Comments may be included in the fetched file through the use of the comment command. If FOL reads a statement of the form:

```
COMMENT   * . . . this is a
 - two l ine comment . . . *
```

then the string between the asterisks is ignored by the command parser. Note that *there is no semicolon after the comment command.* The use of asterisks here is arbitrary; any matching pair of delimiters (except %) would do. As the COMMENT command ignores semicolons, it is useful for removing large sections of input commands from the FOL input stream, without actually deleting them from the input file.

If FOL encounters an error in reading a fetched file, the fetch is terminated, and command returns to the user.

section 8.5 Using FOL from non-Stanford terminals

There are certain difficulties in trying to use FOL from non-Stanford terminals. The most important of these is the lack of the logical connectives and quantifers (∧,∨,¬,⊃,≡,∀,∃) on most non-Stanford keyboards. Even from a non-local terminal with the full character set, a minor difficulty arises when the monitor interprets the exists character (∃) as a control U, and ignores the line on which it' was typed. It is possible to overcome these problems, and use FOL from a non-Stanford terminal or over the ARPA net by the use of the TTY command. The command

```
*****TTY;
```

will rename the sentential connectives and quantifiers to be &, OR, NOT, IMP, IFF, FA and EX. FOL will now print those characters with their new names, and will accept these new names in WFFs. The various quantifier and connective introduction and elimitiation commands mentioned in this text also have synonyms. One may use DED for ⊃I, NI for ¬I, NE for ¬E, UC for ∀I, US for ∀E, EC for ∃I, and ES for ∃E. The derivation of these names may be found in the FOL manual.

The command

```
*****UNTTY;
```

undoes the action of the TTY command. After the UNTTY command, the connectives and quantifiers will print using their usual names, and the FOL parser will cease to recognize the TTY mode names. One can switch arbitrarily between TTY and UNTTY modes.

section 8.6 Saving the state of the proof

The our last administrative command is the EXIT command. EXIT; returns the user to the monitor; the monitor SAVE program will then save the FOL core image for later restarting. Thus, a typical sequence for saving the FOL core image and listing a proof on the lineprinter is:

```
*****SHOW PROOF →MY.PRF;

*****EXIT;

The EXIT command does a garbage collection and takes a while

Exit
↑C
..SA MYFOL    (we're talking to the monitor here)
Job saved in 130 pages. (Upper not saved)
↑C
```

```
.SPOOL MY. PRF    (Request a listing on the lineprinter)

Exit
↑C
. RU MYFOL  (Get back into FOL)

Saving input on: BACKUP. TMP

*****          (We can continue with the proof here)
```

section 9 Pointers to additional information

The source file for this paper is **FOLPRM.REF[AIM,DOC]**. The FOL commands used in this primer, **and practice** exercises for FOL can be found **on the** file **EXERCI.FOL[UP,DOC]**.

We wish to thank John McCarthy and Bill Classmire for their many helpful suggestions on and corrections to the drafts of this primer.

Bibliography.

Gardner, M. (1959) *The Scientific American book of Mathematical Puzzles and Diversions,* Simon and Schuster, New York

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

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

McCarthy, **J. (1962)** *et. al. LISP 1.5 Programmer's Manual* MIT Press, Cambridge, Massachusetts

Mendelson, E. (1964) *Introduction to Mathematical Logio* D. Van Nostrand Co. Inc., New York

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

Weissman, *C.* (196'7) *LISP 1.5 Programmer? Manual,* MIT Press, Cambridge, Massachusetts

Weyhrauch, R. and **W. Glassrnire** (to appear) *A Users Manual for FOL*

Weyhrauch, R. and **A. J. Thomas** (1974) *FOL: a Proof Checker* for ***First-order*** *Logic,* Stanford A.I. **Memo 235**

Index

→, in show command 28
→, operator 13
¬I, inference rule 26
ϵ 8
Q 1
QE, inference rule 2, 15
∀I, inference rule 17
↔ 20
= 9
administrative commands 27
A LLDIFFERENT, predicate 14
A LLTHESAME, predicate 14
ARITY 8
ARPA net, using FOL over the 30
ASSUME, inference rule 2, 15
ATTACH 19
attachment 18
AWFF 10
AXIOM SCHEMA 12
axioms 4, 11
backup file 29
BACKUP.TMP 29
boxes, puzzle about 3
CANCEL 29
colored grids world 13, 22
Comments 29
declarations 1, 7
declarations, individual constants 7
declarations, individual variable 10
declarations, operator constant 9
declarations, predicate constant 7
declarations, predicate parameter 10
DECLARE 7
dependencies 2, 3, 12, 15
dependencies, created by 3E command 6, 7, 17
errors, syntax 7
exercises 3 1
EXIT 30
EXTENSION 20
FETCH 29
FOL 1
FOL, commands 12
FOL, how to run 1
FOL, proof 12
fonts 2
FREEOF, predicate 14
FROM, in FETCH command 29
FUNCTION 22
HAS, predconst 4
HAS, predicate 14
INDCONST 1
INDCONST, declarations 7
individual constants, declarations 7
individual variable, declarations 10
INDVAR 1
INDVAR, declarations 10
inference rule, ¬I 26
inference rule, QE 2, 15
inference rule, ∀I 17
inference rule, ASSUME 2, 15
inference rule, SIMPLIFY 18, 20
inference rule, SWBST 24
inference rule, SUBSTR 24
inference rule, TAUT 2, 16
inference rule, TAUTEQ 6, 17
inference rule, ∃E 5
inference rule, ⊃I 3, 18
infix 8
IS, predconst 4
LABEL 23, 26
LABELED, predconst 4
line number 23, 24, 25, 26
LISP 7, 19
main connective 2, 16
MAKEGRID, operator 13
marbles, puzzle about 3
MARK, in FETCH command 29
MC 8
MOREGENERAL 8
MYCRID, indconst 20
NATNUM 9
natural deduction 1
OCC 25
occurrence list 25
operator constant, declarations 9
PLAID, predicate 14
PREDCONST 1
PREDCONST, declarations 8
PREDCONST, predeclared 9
predeclared, PREDCONST 9
predicate constant, declarations 7
predicate parameter 12
predicate parameter, declarations 10
PREDPAR 12
PREDPA R, declarations 10
prefix 8, 13

RANGELIST 25
reason list 16
REPRESENT 19
representation 19
rules of inference 2, 12
SEXPR 9
SHOW 27
simplification 18
SIMPLIFY, inference rule 18, 20
Socrates, proof about 1
SORT 7, 8
source file for this paper 31
subpart designator 23
SUBST, inference rule 24
substitution 24
SUBSTR, inference rule 24
SYNTYPE 7, 10
TAUT, inference rule 2, 16
TAUTEQ, inference rule 6, 17
TAUTEQ, substitution by 26
TERM 10
TO, in FETCH command 29
TTY 30
UNTTY 30
variable, bound 11
variable, free 11, 18
variable, scope 11
VL 16, 23
WFF 1, 10
↑ 24
∃ 5
∃E, inference rule 5
⊃I, inference rule 3, 18