AD-785 072

# LOGIC FOR COMPUTABLE FUNCTIONS DESCRIPTION OF A MACHINE IMPLEMENTATION

Robin Milner

Stanford University

Prepared for:

Aavanced Research Projects Agency
National Aeronautics and Space Administration

May 1972

STANFORD ARTIFICIAL INTELLIGENCE PROJECT
MEMO AIM-169

STAN-CS-72-288

1

# LOGIC FOR COMPUTABLE FUNCTIONS
# DESCRIPTION OF A MACHINE IMPLEMENTATION

BY

ROBIN MILNER

SUPPORTED BY
NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
AND
ADVANCED RESEARCH PROJECTS AGENCY
ARPA ORDER NO. 457

MAY 1972

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

39

STANFORD ARTIFICIAL INTELLIGENCE PROJECT
MEMO AIM-169

MAY 1972

COMPUTER SCIENCE DEPARTMENT
REPORT CS-288

# LOGIC FOR COMPUTABLE FUNCTIONS
## DESCRIPTION OF A MACHINE IMPLEMENTATION

by

Robin Milner

ABSTRACT: This paper is primarily a user's manual for LCF, a proof-checking program for a logic of computable functions proposed by Dana Scott in 1969 but unpublished by him. We use the name LCF also for the logic itself, which is presented at the start of the paper. The proof-checking program is designed to allow the user interactively to generate formal proofs about computable functions and functionals over a variety of domains, including those of interest to the computer scientist - for example integers, lists and computer programs and their semantics. The user's task is alleviated by two features: a subgoaling facility and a powerful simplification mechanism. Applications include proofs of program correctness and in particular of compiler correctness; these applications are not discussed herein, but are illustrated in the papers referenced in the introduction.

This research was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defence under Contract SD-183 and in part by the National Aeronautics and Space Administration under Contract NSR 05-020-500.

# LOGIC FOR COMPUTABLE FUNCTIONS
# DESCRIPTION OF A MACHINE IMPLEMENTATION

by

Robin Milner

## CONTENTS

---------

## 1. INTRODUCTION

LCF is based on a logic of Dana Scott, proposed by him at Oxford in the Fall of 1969, for reasoning about computable functions. In Section 2 we present this logic, essentially as Scott himself presented it, but using the typed λ-calculus instead of the typed combinators S and K, since the former is more familiar to computer scientists and is in any case easier to work with. Section 3 then describes the machine implementation of a proof-checker for the logic. We refer to both the logic and the implementation as the typed logic for computable functions, or typed LCF, or just LCF.

The logic presupposes no special domain of computation (e.g. lists or integers). However, particular domains can be axiomatized in it : Scott gave an axiomatization for arithmetic and we suggest a partial axiomatization for lists in Section 3. But many interesting results - e.g. equivalence of recursion equation schemata - are provable in the pure logic without any proper (non-logical) axioms.

It is hoped that a potential user of the system can, with the help of the example of Section 3.1 and with Section 4, get onto the machine without reading the whole of this document.

Further discussion of LCF and examples of its applications can be found in the following papers:

Milner,R., "Implementation and applications of Scott's logic for computable functions", Proc. ACM Conference on Proving Assertions about Programs, New Mexico State University, Las Cruces, New Mexico, Jan 6-7, 1972.

Weyhrauch,R. and Milner, "Program semantics and correctness in a mechanized logic", Proc. USA-Japan Computer Conference, Tokyo, Oct 1972 (to appear).

Milner and Weyhrauch, "Proving compiler correctness in a mechanized logic", Machine Intelligence 7, ed. D. Michie, Edinburgh University Press 1972 (to appear).

Newey,M., "Axioms and Theorems for integers, lists and finite sets in LCF", forthcoming AI Memo., Computer Science Dept., Stanford University, 1972.

We give no further references here; they may be found in the above papers.

## 2. THE LOGIC LCF

### Types

At bottom "tr" and "ind" are types. Further if $\beta 1$ and $\beta 2$ are types then $(\beta 1 \to \beta 2)$ is a type. We adopt the convention that $\to$ associates to the right and frequently omit parentheses; thus we write $\beta 1 \to \beta 2 \to \beta 3$ for $(\beta 1 \to (\beta 2 \to \beta 3))$. With each term of the logic there is an unambiguously associated type. For a term t we write

$$t:\beta$$

to mean that the type associated with t is $\beta$. Throughout we use $\beta, \beta 1, \beta 2, \ldots$ as metavariables for types.

### Terms (metavariables s,t,s1,t1,...)

The following are terms:

Identifiers(metavariables x,y) - sequences of upper or lower letters and digits. We assume that the type of each identifier is uniquely determined in some manner.

Applications - $s(t) : \beta 2$ , where $s:\beta 1 \to \beta 2$ and $t:\beta 1$.

Conditionals - $(s \to t1,t2) : \beta$ , where $s:tr$ and $t1,t2:\beta$.

$\lambda$-expressions - $[\lambda x.s] : \beta 1 \to \beta 2$ , where $x:\beta 1$ and $s:\beta 2$.

$\alpha$-expressions - $[\alpha x.s] : \beta$ , where $x,s:\beta$.

This strict syntax is relaxed in the machine implementation (see Section 3) to allow a saving of parentheses and brackets.

The intended interpretation of the $\alpha$-expression $[\alpha f.s]$ is the minimal fixed-point of the function or functional denoted by $[\lambda f.s]$. For example:

$$[\alpha f.[\lambda x.(p(x) \to f(a(x)),b(x))]]$$

denotes the function defined recursively as follows:

$$f(x) \Leftarrow \text{if } p(x) \text{ then } f(a(x)) \text{ else } b(x).$$

## Constants

The identifiers TT,FF denote truthvalues true and false. UU denotes the totally undefined object of any type: in particular, the undefined truthvalue.

## Atomic well-formed formulae (awffs)

The following is an awff:

$$s \sqsubseteq t$$

where s and t are of the same type. The intended interpretation of $s \sqsubseteq t$ is, roughly, that t is at least as well defined as, and consistent with, s.

## Well-formed formulae (wffs) (metavariables P,Q,P1,Q1,...)

Wffs are sets of zero or more awffs, written as lists with separating commas. They are interpreted as conjunctions. We use

$$s \equiv t$$

to abbreviate $s \sqsubseteq t,\ t \sqsubseteq s$ .

## Sentences

Sentences are implications between wffs, written

$$P \vdash Q$$

or, if P is empty, just

$$\vdash Q$$

## Proofs

A proof is a sequence of sentences, each being derived from zero or more preceding sentences by a rule of inference.

## Inference rules

Let us write P(s/x) or t(s/x) for the result of substituting s for all free occurrences of x in P or t, after first changing bound variables in P or t so that no variable free in s becomes bound by the substitution. We have not stated conditions on the types of identifiers and terms with each rule; any consistent assignment of types is admissible.

```
         *****    |-   RULES    *****

INCL     ----------        (Q a subset of P)
           P  |-  Q

         P  |-  Q1     P  |-  Q2
CONJ     ----------------------
              P  |-  Q1∪Q2

         P1  |-  P2     P2  |-  P3
CUT      -----------------------
                P1  |-  P3


         *****    ⊂    RULES    *****


APPL     ----------------------------
         s1 ⊂ s2   |-   t(s1) ⊂ t(s2)


REFL     -------------
         P   |-   s ⊂ s

         P   |-   s1 ⊂ s2     P   |-   s2 ⊂ s3
TRANS    ---------------------------------------
                 P   |-   s1 ⊂ s3


         *****   UU   RULES   *****


MIN1     ------------
          |-   UU ⊂ s


MIN2     ----------------
          |-   UU(s) ⊂ UU
```

```
        *****   CONDITIONAL RULES   *****

CONDT   -----------------
          |-   TT → s,t ≡ s


CONDU   -----------------
          |-   UU → s,t ≡ UU


CONDF   -----------------
          |-   FF → s,t ≡ t



        *****   λ   RULES   *****

            P   |-   s ⊂ t
ABSTR   -----------------------  (x not free in P)
        P   |-   [λx.s] ⊂ [λx.t]


CONV    -----------------------
          |-   [λx.s](t) ≡ s(t/x)


ETACONV -----------------   (x and y distinct)
          |-   [λx.y(x)] ≡ y



        *****   TRUTH   RULE   *****

        P, s≡TT  |-  Q     P, s≡UU  |-  Q     P, s≡FF  |-  Q
CASES   ------------------------------------------------------
                              P  |-  Q



        *****   α   RULES   *****

FIXP    -----------------------
          |-   [αx.s] ≡ s([αx.s]/x)

        P  |-  Q(UU/x)     P, Q  |-  Q(t/x)
INDUCT  ----------------------------------   (x not free in P)
                 P  |-  Q([αx.t]/x)
```

## 3. THE MACHINE IMPLEMENTATION OF LCF

We now describe the machine version of the logic of Section 2, and how to use it interactively on the machine.

The user has available four groups of commands:

- Rules of Inference - to generate new sentences or steps from zero or more previous steps. (Section 3.2)
- Goal Oriented Commands - to specify and attack goals and subgoals. (Section 3.3)
- Miscellaneous - mainly to do with displaying or filing parts or all of the proof so far, and the goals. (Section 3.4)
- Commands for axioms and theorems - to enable the user to create axiom systems, to prove and file theorems in these systems, and later to recall and instantiate those theorems. (Section 3.7)

Before describing the commands in detail, and the syntax of wffs, terms, etc., it may be helpful to see an example.

### 3.1 An Example

Let us introduce the machine version of LCF by a simple example which, although short, exhibits many of the features. It is a proof of a version of recursion induction, which states that if F is defined recursively and G (another function) satisfies F's recursive definition then F⊂G. In other words, we prove that F is the minimal fixed point of its defining equation.

After initialization (see Section 4), the system types 5 asterisks as a signal to the user to start a proof. In fact, 5 asterisks are always the signal for the user to continue his proof. Thus, in what follows the user's contribution may be distinguished by being preceded by *****. We explain each user and machine contribution on the right of a vertical line.

```
*****ASSUME  F≡[αF.FUN F], G≡FUN G;
        |The user assumes a wff (a sequence of atomic wffs
        |separated by commas, where each atomic wff has ≡ or
        |⊂ infixed between two terms). Every user
        |command ends with a semicolon. Detailed syntax is
        |given later - but note in particular that application
        |may be represented (sometimes) by juxtaposition as in
        |"FUN G" to save parentheses. Note also that F occurs both
        |free and bound (by α) without confusion.
```

```
 1  F≡[αF.FUN(F)]  (1)
 2  G≡FUN(G)  (2)

          |The machine separates the assumption into two sentences,
          |giving each a stepnumber.  Every sentence which the
          |machine generates will have a stepnumber, and will consist
          |of a  wff  followed by a list of stepnumbers of assumptions
          |on which the  wff  depends.  A sentence
          |
          |       n  P  S
          |
          |where  P is a  wff  and  S  a list of stepnumbers is the
          |analogue in LCF of the sentence
          |
          |       Q  |-  P
          |
          |of pure LCF, where Q  is the conjunction of assumptions
          |designated by S.  Each of steps 1 and 2 above thus
          |represents an instance of P  |-  P, which is a special
          |case of the inclusion rule of Section 2.

*****GOAL  F⊂G;

          |The user states his goal, but does not attack it yet.
          |He might list several goals before attacking any of them;
          |in each case the machine will simple give a goal number:

NEWGOAL #1  F⊂G

          |Goal numbers are distinguished from stepnumbers by #.

*****TRY 1 INDUCT 1;

          |The user wants to attack GOAL1 using the tactic of
          |induction on Step 1 - which is (as it must be) a
          |recursive definition - i.e.  F≡[αF.FUN(F)].

NEWGOAL #1#1  UU⊂G
NEWGOAL #1#2  FUN(F1)⊂G  ASSUME F1⊂G

          |The machine says that the induction base and step
          |must be established.  For the step it picks an arbitrary
          |identifier not used previously (actually for mnemonic reasons
          |it picks something which only differs from the instantiated
          |bound variable in its numerical suffix).

          |We now have two goals generated by the machine, at
          |a lower level.  The user need not - but probably will -
          |choose to prove #1 by proving #1#1 and #1#2.

*****TRY 1;
```

```
        |User chooses to attack #1#1 first.  He need (and must)
        |only refer to the goal by the last integer in its goal
        |number.  This time he doesn't state a tactic - he knows
        |how to prove it himself - so the machine merely steps down
        |a level in the goal tree and waits. Actually, he could
        |use the SIMPL tactic (see Section 3.3), since this
        |tactic notices instances of MIN1 and would therefore
        |save the user his next two commands.

*****MIN1 G;

        |The user notes that the subgoal UU⊂G can be proved
        |by the first minimality rule (see Section 2), so calls
        |it with the appropriate term - G - as parameter.

3  UU⊂G

        |The machine obediently generates the proper instance of
        |minimality.  Notice that this sentence depends on no
        |assumptions.

*****QED;

        |The user can say QED  to tell the machine that he has
        |proved exactly the goal under attack.

GOAL  #1#1  PROVED.  BACK UP TO GOAL #1
REMAINING SUBGOALS:

2  FUN(F1)⊂G  ASSUME  F1⊂G

        |The machine agrees.  Now in general it will back up
        |the goal tree until it finds a goal some of whose subgoals
        |have not been proved.  It will remain at that level and list
        |these subgoals for the user to try.

*****TRY 2;

        |Again, user gives no tactic.

4  F1⊂G  (4)

        |The machine makes the assumption of the goal for him (note
        |that goal #1#1 had no assumption), and waits.

*****APPL FUN,4;

        |"Apply FUN to Step 4" - an instance of application (see
        |Section 2).

5  FUN(F1)⊂FUN(G)  (4)
```

```
*****SYM 2;

        |Turn step 2 around ready for an application of transitivity.

6  FUN(G)≡G  (2)

*****TRANS 5,6;

        |Note that the parameters of inference rules are always
        |stepnumbers or terms, separated by commas.

7  FUN(F1)⊂G  (2 4)

*****QED;

GOAL #1#2 PROVED.  BACK UP TO GOAL #1.  NO MORE SUBGOALS

8  F⊂G  (2 1)

        |-----The machine hasn't finished yet, but note that it
        |makes an explicit step to represent the proof of #1 (so
        |that the whole sequence of steps, with all the goal
        |structure stripped away, shall be a formal proof).  Note
        |also the assumptions of step 8.
GOAL #1 PROVED.  BACK UP TO TOP LEVEL.  NO MORE SUBGOALS.

        |(There might have been more goals listed at top level,
        |since the user can list many before attacking any).

*****SHOW PROOF RECIND;

        |The user decides to keep his proof on a file called RECIND.
        |The version kept is shown below.  Notice that not every-
        |thing which the user typed reappears; in particular, the
        |statement of a goal is not reproduced, only its trial.

        |If the user wanted instead to display his proof (at any
        |point, not just at the end) he would just type "SHOW PROOF;"

                     PROOF

          1     F ≡ [αF,FUN(F)]  (1)  ----  ASSUME.
          2     G ≡ FUN(G)  (2)  ----  ASSUME.

            ------------------------
           |TRY #1   F ⊂ G            INDUCT 1.
           |   ------------------------
           |   |TRY #1#1   UU ⊂ G
           |   |3    UU ⊂ G  ---- MIN1 G.
           |   ------------------------
           |   ------------------------
|   |TRY #1#2   FUN(F1) ⊂ G  ASSUME    F1 ⊂ G  .
|   |4     F1 ⊂ G (4) ---- ASSUME.
|   |5     FUN(F1) ⊂ FUN(G)  (4) ---- APPL 4 FUN.
|   |6     FUN(G) ≡ G  (2) ---- SYM 2.
|   |7     FUN(F1) ⊂ G  (4 2) ---- TRANS 5 6.
|    ------------------------
|8    F ⊂ G  (2 1) ---- INDUCT 3 7.
 -------------------------
```

## 3.2 Rules of Inference

Let us assume for the moment the syntax classes <wff>, <awff> (atomic wff), <term>. Details of these are in Section 3.6, but for now look only at the conventions given for syntax definitions at the start of that Section.

We need for the present

```
<stepname> ::= <integer>| ---_--- | . <identifier> ?( (+|-) <integer> )

<termname> ::= ?( :G|:<stepname> ) ?( :<integer> ) (:L|:R)

<range> ::= <stepname> | ?<stepname> : ?<stepname>
```

In a <stepname> "-" means "the last step", "--" means the last step but one, etc., and for example ".DD-1" means the step preceding that labelled DD. See Section 3.4, the LABEL command, for how to label steps.

A <termname> may appear anywhere that a term can appear - for example as a subterm of a term - and frequently saves typing long formulae. We explain termnames by a few examples (suppose the last step was numbered 15) :

```
:15:1:R    )
:-:1:R     )
:15:R      )   all designate the term which occurs as
-:R        )   right hand side in the first <awff> of Step 15.
:R         )

:.DD:2:L       designates the lhs of the second <awff>
               of the step labelled DD.

:G:2:R     )   designate the rhs of the second <awff> of
               the current goal - THISGOAL (See Section 3.3)
```

The <range>s 12, 20:30, :40, 50: denote respectively the single step 12, the steps 20 to 30 inclusively, the steps up to and including 40, and the steps from 50 onwards.

We now list the rules, with some examples. Note that in the machine implementation there is no type-checking whatsoever. We rely on the user to use types consistently.

ASSUME <wff>;

Each <awff> Ai in the <wff> is given a new stepnumber ni, and the steps

```
n1  A1(n1)
n2  A2(n2)
. . . . .
```

are generated. Each one is a tautology, since a step P(n) means Q |- P, where Q is the <awff> at step number n. Thus the purpose of ASSUME is only to introduce references for <awff>s. See Section 3.1 for examples of ASSUME.

SASSUME <wff>;

Like ASSUME, but every <awff> of the <wff> is henceforward treated as a simplification rule (see Section 3.5).

INCL <stepname>, <integer>;

Picks out an <awff>. Example:

```
---------------------------
|15  Z≡F(X,Y), A≡B, [λX.X](Y)⊂14 (13 7)
|*****INCL 15,2;
|16  A≡B (13 7)
---------------------------
```

CONJ ___,<range>,___ ;

Forms conjunction of all steps in the <range>s. Example:

```
----------------------------
|15  P⊂Q,R≡S (12)
|    -----
|17  F≡G (12 4)
|*****CONJ ---,-;
|18  P⊂Q, R≡S, F≡G (12 4)
----------------------------
```

CUT <stepname>, <stepname>;

If the steps referred to are P(m1,m2,..) and Q(n1,n2,..) respectively, where the m's and n's are stepnumbers, and if every <awff> referenced by the n's occurs as an <awff> in P, then the step Q(m1,m2,..) is generated. Example:

```
------------------------------
|7  F≡G  (7)
|  -----
|12  P⊂Q  (7)
|   -----
|15  F≡G, G⊂H  (14 2)
|*****CUT 15,12;
|16  P⊂Q  (14 2)
-------------------------------
```

HALF <stepname>;
Replaces "≡" by "⊂" in the first <awff>, and throws the rest away. Example:

```
-------------------------------
|6  X≡G(X), Y≡H(Y) (1 3)
|*****HALF 6;
|7  X⊂G(X)  (1 3)
-------------------------------
```

SYM <stepname>;
Interchanges the terms in the first <awff> (provided "≡" occurs) and throws the rest away. Example (continuing the previous):

```
-------------------------------
|*****SYM 6;
|8  G(X)≡X (1 3)
-------------------------------
```

TRANS <stepname>, <stepname>;
Looks at the first <awff> in each <wff>. If these are s1(≡|⊂)s2, s2(≡|⊂)s3 respectively, then s1⊂s3 or s1≡s3 is generated, the assumptions being "unioned". Example:

```
-------------------------------
|12  X≡Y(Z), P⊂Q (11 4)
|   -----
|13  Y(Z)⊂Y(X)  (4 9 8)
|*****TRANS 12,13;
|14  X⊂Y(X)  (11 4 9 8)
-------------------------------
```

APPL (<stepname>, ___,<term>,___ |<term>,<stepname>);
In the first case, applies both sides of the first <awff> of <stepname> to the <term>s in sequence.
In the second case, applies the <term> to both sides of the first <awff> of <stepname>. Examples;

```
-------------------------------
|10  X≡Y(Z), P⊂Q  (9 4)
|*****APPL F,10;
```

```
|11  F(X)≡F(Y(X))  (9 4)
|    -----
|22  F≡[λX,X],P⊂Q  (11 4)
|*****APPL 22,:-:2:R;
|23  F(Q)≡[λX.X](Q)  (11 4)
------------------------------
```

ABSTR <stepname>, ___,<identifier>,___ ;
Does λ-abstraction on 1st <awff>. The identifiers must not occur free in any of the assumptions of the step. Example(continuing the previous):

```
------------------------------
|*****ABSTR  22,F;
|24 [λF,F]≡[λF,[λX,X]]  (11 4)
------------------------------
```

CASES ) These are not present as inference rules, since it is
) less tedious to use their goal oriented versions (see
INDUCTION ) Section 3.3).

CONV (<stepname>|<term>);
Does all λ-conversions in the <term> or <stepname>. Example:

```
------------------------------
|    -----
|14  B≡[λX,X(X)][λX,X(Y)]
|*****CONV -;
|15  B≡Y(Y)
------------------------------
```

Remark: the term in 14 violates the type structure, but the system does not check this.

ETACONV <term>;
Eta-converts the <term>, provided it has the form [λx.s(x)], with x not free in the term s. Example (remember that F(X,Y) abbreviates (F(X))(Y) ):

```
-------------------------
|*****ETACONV [λY. F(X,Y)];
|49  [λY. F(X,Y)]≡F(X)
-------------------------
```

EQUIV <stepname>,<stepname>;
Looks at the first <awff> in each <wff>. If these are s1⊂s2, s2⊂s1 respectively, then s1≡s2 is generated. Example:

```
------------------------------
|    -----
|16  X⊂Y, P≡Q (12)
|    -----
```

```
|17  Y⊂X, H⊂G (1 2)
|*****EQUIV 16,17;
|18  X≡Y  (12 1 2)
-------------------------------
```

REFL1 <term>;
Gives t≡t where t is designated by the mterm.  Example:

```
-------------------------------
|*****REFL X(XX);
|19  X(XX) ≡ X(XX)
-------------------------------
```

REFL2 <term>;
Like REFL1, but gives t⊂t.

MIN1 <term>;
Gives UU⊂t.  Example: see Section 3.1

MIN2  <term>;
Gives UU(t)≡UU. Example (continuing the previous):

```
-------------------------------
|*****MIN2 ;L;
|20 UU(X(XX)) ≡ UU
-------------------------------
```

CONDT <term>;
Checks that the <term> t has form TT→s1,s2 and if so generates t≡s1.  Example:

```
-------------------------------
|  -----
|21 F(X) ≡ TT→X,F(G(Y,X)) (10)
|*****CONDT :R;
|22 TT→X, F(G(Y,X)) ≡ X
-------------------------------
```

CONDF <term>;
Checks that the <term>  t  has form  FF→s1,s2 and if so generates  t≡s2.

CONDU <term>;
Checks that the mterm  t  has form  UU→s1,s2 and if so generates  t ≡ UU.

FIXP <stepname>;
Checks that the first <awff> is a recursive definition e.g.  s≡[αG.t], and generates  s≡t(s/G). Example:

```
-------------------------------
|- - - - - - - - -
|23  F ≡ [αG,H([λF,G(F)])]
|*****F1XP 23;
|24  F ≡ H ([λF1,F(F1)])
-------------------------------
```

SUBST <stepname> ?( OCC ___,<integer>,___ ) IN (<stepname>|<term>);
Let the first <stepname> have t1 $ t2 as its first <awff>, where $ stands for ≡ in case (1), and for ≡ or ⊂ in case (2),

Case (i), If there is an <stepname> following "IN" , then t2 is substituted for all occurrences designated by the <integer>-list (or all occurrences, if no list) of t1 in the <wff>.

Case (ii), If there is a <term> s following "IN" then s $ s' is generated, where s' is the result of substituting t2 for the appropriate occurrences (as in case (i)) of t1 in s'.

Note that for t1 to occur in a term s any occurrence of a free variable in t1 must not be bound in s. Also see the caution on occurrence numbers in Section 3.6.

Example:

```
--------------------------------
|25  [λX.F(X)] ⊂ G(F(X),F(X)) (2 3)
|    -----
|26  F(X) ≡ X  (5 1)
|*****SUBST 26 OCC 1 IN 25;
|27  [λX.F(X)] ⊂ G(X,F(X))  (2 3 5 1)
|*****SUBST 26 IN !25:R;
|28  G(F(X),F(X)) ≡ G(X,X)  (5 1)
--------------------------------
```

SIMPL  (<stepname>|<term>) ?___( (BY|WO) ___,<range>,___ )___ ;
In the case of an <stepname>, its <wff> is simplified (see Section 3.5) using as simplification rules those in SIMPSET together with those designated by the <range>-list following each "BY", and without those designated by the <range>-list following each "WO". A <term> t is similarly simplified, to t1 say, and t ≡ t1 is generated, The SIMPSET remains unchanged,

Example, continuing the previous (Section 3.5 gives more detail):

```
--------------------------------
|    -----
|29 [λP,P→F(X),Y](TT) ⊂ UU(X)  (10)
|*****SIMPL - BY 26;
|30  X⊂UU  (10 5 1)
--------------------------------
```

This happens because CONV, CONDT, MIN2 are among the simplification rules.

## 3.3 Goal-Oriented Commands

Anything provable with the goal oriented commands is provable in PURE LCF, but most proofs would then be tedious (that's why we only describe the INDUCTION and CASES rules in goal-oriented form). Experience shows that with the goal-oriented commands the user has only to type a small fraction of what he would otherwise have to type.

The user may generate a subgoal structure of arbitrary depth. This structure is represented by three entities; GOALTREE, GOALLIST and THISGOAL. THISGOAL is always the goal currently under trial; all its ancestors in GOALTREE are (indirectly) also under trial; the subgoals of THISGOAL are listed in GOALLIST. Each goal has a goal number - e.g. #1#2#3 - which indicates its ancestors and (by the number of parts) its level in the tree. Here is a sample goal structure;

```
LEVEL 0                     •                 )
                    --------|--------         )
                    |       |       |         )
LEVEL 1           #1•     #2•     #3•         )
                            |                 )
LEVEL 2                     •#2#1             )  GOALTREE
                    --------|--------         )
                    |               |         )
LEVEL 3             •#2#1#1         •#2#1#2 <---THISGOAL
                                    |
                            --------|--------
                            |       |       |
                            •       •       •   GOALLIST
                     #2#1#2#1  #2#1#2#2  #2#1#2#3
```

FIGURE 1

Each goal has a status (not shown in diagram) which is either "UNDER TRIAL" (only THISGOAL and its ancestors have this status), or "NOT TRIED" or "PROVED".

The user has five goal oriented commands available: we give first their syntax, then detailed descriptions.

```
GOAL <wff> ?(ASSUME|SASSUME) <wff> ;

TRY ?<integer> ?<tactic> ;

QED ?<stepname> ;

ABANDON ;

SCRATCH <integer> ;

        <tactic> ::=  CONJ   |
                      CASES <term>   |
                      ABSTR  |
                      SIMPL ?___( (BY|WO) ___,<stepname>,___ )___ |
                      SUBST <stepname> ?(OCC ___,<integer>,___ )  |
                      INDUCT <stepname> ?(OCC ___,<integer>,___ )  |
                      USE <identifier> ?___,<instantiation>,___

        <instantiation> ::= <identifier> ← <term>
```

The GOAL command.
-----------------

GOAL specifies a new goal to be added to GOALLIST. Its effect on the goal structure of Figure 1 is as follows (Figure 2):

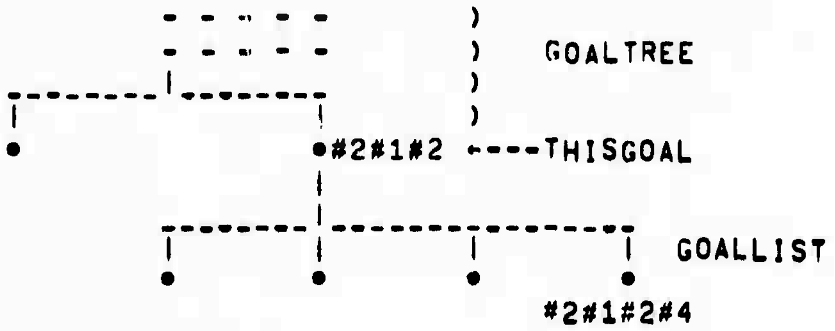

FIGURE 2

(Notice that the new goal isn't yet under trial)

A goal may or may not be given assumptions. The only difference between ASSUME AND SASSUME is that in the latter case, when the goal is tried, the assumption wff will be added to the set of simplification rules (See Section 3.5) for the duration of this goal's trial. Examples:

```
------------------------------
|*****GOAL F⊂G;
|NEWGOAL #1 F⊂G
|*****GOAL F(X)≡G(Y) SASSUME F≡G, X≡Y;
|NEWGOAL #2 F(X)≡G(Y) SASSUME F≡G, X≡Y
------------------------------
```

The only purpose of the system's reply is to allot the goal a number.

The TRY command.
----------------

TRY specifies one of the goals of GOALLIST to be tried (if the <integer> is absent, the last goal specified is assumed). If the user gives no tactic, the new GOALLIST will be null (Figure 3).

```
                - - - - -                        )
                - - - - -                        )
        --------|--------                        )
        |               |                        )
        •               •#2#1#2                  )  GOALTREE
                        |                        )
                --------|----------------        )
                |       |       |        |       )
                •       •       •        •
                            THISGOAL
                               |
                      (GOALLIST initially null)
```

FIGURE 3

But if the user gives a tactic, the system will set up a new GOALLIST for him, whose number of members depends on the tactic. Tactics are described later in this section, but look at the Example following QED's description below to see what happens without them.

The QED command.
----------------

QED indicates that the <stepname> - or previous step if no <stepname> - proves THISGOAL; the user will normally say QED when he TRIED this goal with no tactic. Sometimes the user has been able to prove a contradiction, i.e. any of the <awff>s <tv>≡<tv> or <tv>⊂<tv> where the <tv>s are distinct members of (TT,UU,FF) and in the case of ⊂ the first <tv> is not UU. QED will accept a contradiction, since it proves anything. The effect of QED is to restore Figure 3 to Figure 2, with the difference that the status of #2#1#2#3 will become "PROVED"; further, if THISGOAL (of figure 2) was TRIED with a tactic, and all subgoals generated by this tactic are now "PROVED", the system will back further up the tree. This may continue for many steps: eventually the system will stop and tell the user which goal has now become THISGOAL, and which members of its GOALLIST remain to be proved.

The following example continues the one above, and illustrates TRY and QED:

```
 ------------------------------------
|*****TRY 2;
|13 F ≡ G   (13)           )  The system makes the assumptions.
|14 X ≡ Y   (14)           )
|
|*****APPL 13,X;           )
|15  F(X)≡G(X)  (13)       )
|                          )
|*****APPL G,14;           )
|16  G(X)≡G(Y)  (14)       )  The user proves the goal.
|                          )
|*****TRANS 15,16          )
|17  F(X)≡G(Y)   (13 14)   )
|                          )
|*****QED;                 )
|GOAL #2 PROVED.  BACK UP TO TOP LEVEL. )  The system
|REMAINING SUBGOALS:                    )   backs up.
|1  F⊂G
 ------------------------------------
```

## The ABANDON command.

ABANDON indicates that the user doesn't like his current trial of THISGOAL. The effect will be to restore Figure 3 to Figure 2 - but the status of #2#1#2#3 becomes again "NOT TRIED". Thus no further backing up can happen.

## The SCRATCH command.

SCRATCH removes the indicated goal from GOALLIST. However, the system will refuse to scratch goals generated by tactics.

Tactics ,
---------

We now describe the tactics available. There are six basic ones, each based on a particular inference rule; in addition the user may employ any THEOREM (see section 3.7) as a tactic.

For CONJ, the system generates a separate subgoal for each <awff> in the goal.

For CASES, if s is the <term> and P is the <wff> of the goal, the system generates the 3 subgoals P SASSUME s≡TT, P SASSUME s≡UU, P SASSUME s≡FF.

For ABSTR, the system instantiates in each <awff> in the goal for as many bound variables as are bound by the outermost λ in its left-hand side, thus generating a single new subgoal. New variables are chosen which are not free in the proof so far. For example, if the goal is [λX Y.F(Y,X)] ≡ [λZ.G(Z,Z)] , and X is already free in the proof, the new goal will be F(Y,X1) ≡ G(X1,X1,Y).

For SIMPL, the system generates a new subgoal by simplifying the goal as far as possible, using a modified SIMPSET (if any "BY" or "WO" is present) as explained in Section 3.2 under the SIMPL rule. The modified SIMPSET remains in force, but the old one will be reinstated when the new goal is either proved or ABANDONed (see section 3.5). If the system discovers that all <awff>s of the new subgoal are identically true - i.e. they are all of the form s⊑s or s≡s or UU⊑s - it initiates the backing up process described under QED above instead of generating the subgoal. If some but not all of the <awff>s are identically true they are simply omitted from the new subgoal.

For SUBST, the system generates a new subgoal by substituting the rhs of <stepname> for the lhs of <stepname> in the goal - either throughout, or at the designated occurrences when an <integer>-list is given. (see the caution on occurrence numbers in section 3.6).

For INDUCT, let P be the <wff> of the goal. The system checks that <stepname> has the form s≡[αy.t] - i.e. that it is a recursive definition. In that case, it generates two new subgoals. The first is

P{UU/s}

and the second is

P{t{y'/y}/s} ASSUME P{y'/s}

where y' is a variable not previously used free, and where the substitution in P takes place at appropriate occurrences, exactly as for SUBST above.

For USE, the <identifier> is a THEOREM name. The system will instantiate the THEOREM by matching its consequent to the goal, taking into account any instantiations supplied explicitly by the user, and will generate the appropriate instance of its antecedent as a new goal. See section 3.7 for a fuller discussion of THEOREMS.

We now give examples of each tactic (except CONJ, which is easy to understand). Some are realistically combined.

```
 -----------------------------------------------
→|*****GOAL P→X,P→Y,Z ≡ P→X,Z;
 |NEWGOAL #1 P→X,P→Y,Z ≡ P→X,Z
 |
→|*****TRY CASES P;
 |NEWGOAL #1#1 P→X,P→Y,Z ≡ P→X,Z SASSUME P≡TT
 |NEWGOAL #1#2 P→X,P→Y,Z ≡ P→X,Z SASSUME P≡UU
 |NEWGOAL #1#3 P→X,P→Y,Z ≡ P→X,Z SASSUME P≡FF
 |
→|*****TRY 1 SIMPL;
 |25 P≡TT (25)                              ) Here SIMPL reduces goal
 |26 P→X,P→Y,Z ≡ P→X,Z (25)                 ) #1#1 to identity, using
 |GOAL #1#1 PROVED.  BACK UP TO GOAL #1     ) 25 and also an instance
 |REMAINING SUBGOALS:                       ) of CONDT as simp. rules.
 |2 P→ - - - - - - -Z  SASSUME P ≡ UU
 |3 P→ - - - - - - -Z  SASSUME P ≡ FF
 |
→|*****TRY 2 SIMPL;
 |(etc.)
 -----------------------------------------------
```

The example looks long, but the users contribution (shown by "→") is short. (The system keeps reminding the user of what subgoals remain.) The "hard copy" proof produced by the SHOW command will be comparatively short.

The next example illustrates the remaining tactics, and also application to a particular subject matter - lists. The first four steps are the result of SASSUME by the user. Note also the abbreviations ∀X Y. etc., as explained in section 3.6.

```
  ---------------------------------------------------
  |1 ∀X Y. HD(CONS(X,Y)) ≡ X  (1)
  |2 ∀X Y. TL(CONS(X,Y)) ≡ Y  (2)
  |3 ∀X Y.NULL(CONS(X,Y)) ≡ FF  (3)
  |4  NULL(UU) ≡ UU  (4)
  |
→ |*****ASSUME AP ≡ αF.λX Y.NULL X→Y,CONS(HD X,F(TL X,Y));
  |5  AP ≡ [αF.[λX Y.NULL(X)→Y,CONS(HD(X),F(TL(X),Y))]]  (5)
  |
→ |*****FIXP 5;
  |6  AP ≡ [λX Y.NULL(X)→Y,CONS(HD(X),AP(TL(X),Y))]  (5)
  |
→ |*****GOAL ∀X.AP(X,AP(Y,Z)) ≡ AP(AP(X,Y),Z);
  |  NEWGOAL #1 ∀X.AP(X,AP(Y,Z)) ≡ AP(AP(X,Y),Z)
  |
→ |*****TRY INDUCT 5 OCC 1,4;
  |NEWGOAL #1#1 ∀X.UU(X,AP(Y,Z)) ≡ AP(UU(X,Y),Z)
  |NEWGOAL #1#2 ∀X.[λX Y.NULL(X)→Y,CONS(HD(X).F1(TL(X),Y))]
  |(X,AP(Y,Z))
  |≡ AP([λX Y.NULL(X)→Y,CONS(HD(X),F1(TL(X),Y))](X,Y),Z)
  |ASSUME ∀X.F1(X,AP(Y,Z)) ≡ AP(F1(X,Y),Z)
  |
→ |*****TRY 1 ABSTR;
  |NEWGOAL #1#1#1 UU(X,AP(Y,Z)) ≡ AP(UU(X,Y),Z
  |
→ |*****TRY SUBST 6 OCC 2;
  |NEWGOAL #1#1#1#1 UU(X,AP(Y,Z)) ≡
  |        [λX Y.NULL(X)→Y,CONS(HD(X),AP(TL(X),Y))](UU(X,Y),Z)
  |
→ |*****TRY SIMPL;
  |7  UU(X,AP(Y,Z)) ≡ [λX Y.NULL(X)→Y,CONS(HD(X),AP(TL(X),Y))]
  |        (UU(X,Y),Z)  (4)
  |GOAL #1#1#1#1 PROVED.  BACKUP TO GOAL #1#1#1.  NO MORE SUBGOALS
  |8  UU(X,AP(Y,Z)) ≡ AP(UU(X,Y),Z)  (4 5)
  |GOAL #1#1#1 PROVED.  BACKUP TO GOAL #1#1.  NO MORE SUBGOALS
  |9  ∀X.UU(X,AP(Y,Z)) ≡ AP(UU(X,Y),Z)  (4 5)
  |GOAL #1#1 PROVED.  BACKUP TO GOAL #1.
  |REMAINING SUBGOALS:
  |2  (Here follows a restatement of goal #1#2)
  |(etc.)
  ---------------------------------------------------
```

Note that simplification (using the built-in simplification rules CONV and MIN2 and CONDU as well as Step 4) reduced goal #1#1#1#1 to identity, and the system generated step 7 on these grounds. In backing up, it generates an explicit final step, identical to the goal statement in its wff, to tie up the proof of each goal proved.

Note also that the user's contribution (indicated by "→") is short in the above example.

Finally, here is an example of a THEOREM used as a tactic (read section 3.7 first!). It also shows how the user can make many of the inference rules into tactics - even using the same names. Of course, THEOREMS used as tactics will at least as often be substantial results previously proved and filed (consider the frequent occurrence in informal proofs of "to prove XXX it is sufficient, by Theorem AAA, to prove YYY and ZZZ").

First, to make a THEOREM out of the TRANS rule:

```
-------------------------
|*****ASSUME X≡Y, Y≡Z;
|51 X≡Y  (51)
|52 Y≡Z  (52)
|
|*****TRANS -,-;
|53 X≡Z  (51 52)
|
|*****THEOREM TRANS: 53
|THEOREM TRANS: X≡Z  ASSUME  X≡Y,Y≡Z;
-------------------------
```

Now to use TRANS as a tactic:

```
-------------------------
|*****GOAL F(A,X)≡G(X);
|NEWGOAL #1 F(A,X)≡G(X)
|TRY USE TRANS  Y←H(X,A);
|NEWGOAL #1#1 F(A,X)≡H(X,A)
|NEWGOAL #1#2 H(X,A)≡G(X)
-------------------------
```

Note that the X,Y,Z of the THEOREM are metavariables which do not conflict with the variables of the proof.

## 3.4 Miscellaneous Commands

### The SIMPSET command.

```
SIMPSET ___( (+|-) ___,<range>,___ )___ ;
```

The steps designated are added to or removed from the set of simplification rules (See section 3.5).

The SHOW command.
-----------------

```
        SHOW
            (   AXIOMS ?( ( ___,<identifier>,___ ) ) |
                THEOREMS ?( ( ___,<identifier>,___ ) ) |
                GOALTREE ?___,<range>,___ |
                THISGOAL |
                GOALLIST |
                PROOF ?___,<range>,___ |
                STEPS ?___,<range>,___ |
                SIMPSET ?___,<range>,___
                LABELS ?___,<range>,___    )
                        ?( <identifier> ?<integer> ) ;
```

If the final <identifier> is present the material is sent to the file named, otherwise it is displayed on the console. The final <integer> if present denotes the line-width.

If a <range>- or <identifier>-list is not present, the whole is shown. The <identifier>-list for AXIOMS or THEOREMS denotes the particular axioms or theorems required. The <range>-list for GOALTREE refers to levels (Ø is top level), and for PROOF, STEPS, SIMPSET and LABELS refers to stepnumbers. Thus

```
        SHOW STEPS :3, 8, 2Ø:23, 3Ø, 55: ;
```

will show steps 1,2,3,8,2Ø,21,22,23,3Ø and 55 onwards of the proof, with no goal structure; SHOW PROOF will show steps with goal structure, so is normally used with a single <range>, or a whole proof. Only the stepnumbers bound to LABELS are shown.

The FETCH command.
------------------

```
        FETCH ___,<identifier>,___ ;
```

The <identifier>-list names files. Axioms and theorems on those files will be brought in. In fact any admissible commands on these files will be treated exactly as if typed at the console - e.g. ASSUMptions may be made - so the user may prepare such files other than by SHOWING axioms or theorems. Much of what a user types is dependent on the stepnumbers that the system is generating, so the use of files prepared offline is limited. However, this difficulty is somewhat alleviated by the LABEL command (see below). The files are expected to be simply sequences of commands, so several files may easily be concatenated without editing.

## The CANCEL command.

```
        CANCEL ?<stepname> ;
```

This steps back through the <stepname> given, otherwise just the last step. Cancelled steps are removed from the SIMPSET. Goal trials encountered will be ABANDONed. It is not possible to cancel back past any step which proves a goal.

## The INFIX command.

```
        INFIX ---,<identifier>,--- ;
```

This causes all the <identifier>s named to be treated exactly as <infix>es (see section 3.6). In particular, the user must henceforward "!" them in non-infix contexts.

## The PREFIX command.

```
        PREFIX ---,<identifier>,--- ;
```

This revokes the infix status of all <identifier>s named. Standard <infix>es are immune from this, however.

## The LABEL command.

```
        LABEL ---,<identifier> ?<stepname>,--- ;
```

Each <identifier> is attached as a label to the step indicated by the <stepname> if present, otherwise to the next step to be generated. Thus after "LABEL DD - ;" the previous step and its predecessors and successors may be later referenced by the <stepname>s ".DD", ".DD-1", ".DD+1" etc.

## 3.5 Simplification Rules.

At any stage in a proof, there is a current set of simplification rules. Steps may be added to or removed from the simplification rule set (SIMPSET) in five ways:

- By SASSUME (See Section 3.2)
- By the SIMPSET command (see section 3.4).
- By the goal tactic SIMPL (See Section 3.3).
- If the SIMPSET was modified by attacking a goal with a SASSUMption (see section 3.3) or by using the SIMPL tactic, then it will be automatically reinstated when the goal is proved or ABANDONed.
- By CANCEL (see section 3.4).

Simplification is invoked only by the SIMPL rule, (3.2) and by the SIMPL tactic (3.3). The rules are then applied repeatedly to all subterms of the appropriate awff or term until they can be applied no further.

An application of a simplification rule s ≡ t consists in finding all occurrences of s and replacing them by t (so the user must be careful not to make something like F(X)≡ G(F(X)) a simplification rule, or he will cause indefinite expansion!). In addition, in the case of a simplification rule ∀x y ... , s ≡ t , all instances of s, gained by replacing x,y,... by arbitrary terms in s, will be replaced by the appropriate instances of t.

There are five built in rules: CONV (λ-CONVERSION), MIN2 (UU(s) ≡ UU) and CONDT, CONDU, CONDF (simplification of conditionals) (see these rules of inference in 3.2). Together with the previously mentioned feature, this will allow the assumption

```
            ∀X Y.HD(CONS(X,Y)) ≡ X ,
```

when used as a simplification rule, to reduce

```
            HD(CONS(s1,s2))
via         [λX Y.X](s1,s2)
to          s1 .
```

Such formulae may usually be kept permanently in the SIMPSET. Others, notably the SASSUMptions of the CASES tactic, will come and go under system control. Still others the user will need to handle himself; a good example is the result of FIXP on a recursive definition of form s ≡ [αx.t] - the result has form s ≡ t(s/x) and so can lead to indefinite expansion as a simplification rule, but will not do so in the case that the recursive computation, which it will carry out, terminates as a consequence of other members of SIMPSET.

## 3.6 Syntax

As well as the usual BNF conventions we use the following:

```
        ( ) are for grouping syntax patterns,
        ? before a pattern means optional,
        ___P___  means one or more instances of the pattern P,
        ___,P,___ means one or more instances of P separated
                by commas.

        <wff> ::= ___,<awff>,___

        <awff> ::= ?___( ∀ ___,<identifier>,___ | <term>:: )___
                                   <term> (≡|⊂) <term>

        <term> ::= <infixterm>|<conditionalterm>

        <conditionalterm> ::= <infixterm> → <term> , <term>

        <infixterm> ::= <simpleterm> ?___(<infix><simpleterm>)___

        <simpleterm> ::= <closedterm> ?___( <closedterm>|
                                            ( ___,<term>,___ ) )___

        <closedterm> ::= <identifier>|<λterm>|<αterm>|<termname>|
                         (<term>)

        <termname> ::=  ?( :G|:<stepname> ) ?( :<integer> ) (:L|:R)

        <λterm> ::= [ λ ___<identifier>___ . <term> ]

        <αterm> ::= [ α <identifier> . <term> ]

        <identifier> ::= <word> | !<infix> | ¬ | ∂

        <word> ::= ___(<letter>|<digit>| _ )___

        <infix> ::= any of the single characters
                         ∩∪$|+-∂*∧∨/\@←≤≥<>≠=↔↑↓∈
                or any <word> with current INFIX status (3.4)
```

Spaces may occur anywhere except within a <word>, but are only necessary to separate <word>s or to separate "." from a digit (e.g. in "∀x. 0≤x ≡ TT" ). The latter is because the MLISP2 parser takes ".0" as a single element or token.

The brackets round <λterm>s and <αterm>s may be omitted when no ambiguity arises.

Examples follow, with intended interpretation:

- P→Q→X,Y,R→Y,Z  is a <conditionalterm>, abbreviating

  P→(Q→X,Y),(R→Y,Z)

- AP(AP X Y,Z)  is a <simpleterm>, abbreviating

  AP(AP(X,Y),Z) or AP((AP(X))Y,Z)
  or (AP((AP(X))Y))Z

  (Thus the type which we should associate with AP is (β→(β→β)), where β is the type of individuals.)

- λX Y.NULL X→Y,TL Xp  is a <λterm>, abbreviating

  [λX.[λY.(NULL(X)→Y,TL(X))]]

- P :: X ≡ Y  is an <awff>, abbreviating

  P→X,UU ≡ P→Y,UU

- ∀X. F(X,X) ≡ Y  is an <awff>, abbreviating

  λX.F(X,X) ≡ λX.Y

- ∀X Y. X=Y :: X ≡ Y  is an <awff>, abbreviating

  λX Y.X=Y→X,UU ≡ λX Y.X=Y→Y,UU

- !∈ ≡ λX L. X=HD(L)→TT. X∈TL(L)

  illustrates the "!"-ing (which may pronounced "shrieking" or perhaps "howling") of <infix>es, which is necessary whenever they are mentioned in a non-infixed context.

Many examples of <wff>s and <awff>s occur throughout this paper.

Caution!! Some commands refer to occurrences of a <term> in a <wff>. Occurrences are counted from left to right after all occurrences of "::" (which is an abbreviation for legibility reasons only) have been expanded as indicated in the examples, and with <infix>es considered as prefixed.

## 3.7 Commands for Axioms and Theorems

We now describe how the user may create, store away, and fetch axioms and theorems, so that he can build up a file of results over several sessions on the computer, and does not have to start from scratch each time.

We start with a simple example, and then describe the new commands in detail.

```
*****AXIOM LISTS:.......,∀X.NULL X :: X ≡ NIL,...;

          |The user creates an axiom consisting of several
          |<awff>s:  the example uses only one, so the others
          |are represented by ---.  The system lists them
          |for him - as new steps - and will remember the
          |collection by its name: - LISTS.

AXIOM LISTS
1 - - -
2 - - -
3 ∀X.NULL(X) :: X ≡ NIL
4 - - -

*****SASSUME NULL Y≡TT;
5 NULL(Y)≡TT  (5)
*****APPL 3,Y;
6 [λX.NULL(X)→X,UU](Y) ≡ [λX.NULL(X)→NIL,UU](Y)
*****SIMPL 6;
7 Y≡NIL  (5)

          |Note that the SASSUMption 5 has been used, so
          |it appears as a condition for 7.

*****THEOREM UNIQUENULL: 7;

          |The user wants to keep the result 7 - he will be
          |be able to instantiate for Y in later use, so the
          |system really treats it as a metatheorem.  The
          |system writes it in full for him, reminding him
          |that it depends on LISTS:-

THEOREM(LISTS) UNIQUENULL: Y≡NIL ASSUME NULL(Y)≡TT

- - - -
- - - -
- - - -

          |Suppose that the user proves some more theorems,
          |and then wants to keep his axioms (there may be
          |others besides LISTS) and theorems.  He says:
```

```
*****SHOW AXIOMS AXFILE;
*****SHOW THEOREMS THFILE;

        |He can actually select just some to be kept (3,4). Also
        |if he omits the filename, they will not be kept
        |but displayed.



---  NOW, ON SOME LATER OCCASION: ---

- - - -
- - - -
- - - -

        |The user decides he now wants to talk about lists,
        |and would like the theorems that he previously proved.

*****FETCH AXFILE, THFILE;
AXIOM LISTS
15 - - -
16 - - -
17 ∀X,NULL(X) :: X ≡ NIL
18 - - -
THEOREM (LISTS) UNIQUENULL: Y≡NIL ASSUME NULL(Y)≡TT

        |Remember there may have been other axioms and
        |theorems on these files (they should have been
        |at least represented by ---, but we didn't
        |bother).
        |
        |The crucial point is that all variables which
        |are free in the theorem, but not free in the axioms
        |on which it depends, may be instantiated, and the
        |user can force an instantiation by using the theorem
        |as an inference rule.  Suppose later he proves (step 23):
- - -
- - -
23 NULL(HD(Z))≡TT  (15 18)

        |He applies the theorem, as follows (and in this
        |case the only free instantiable variable is Y):

*****USE UNIQUENULL 23;
24 HD(Z)≡NIL  (15 18)

        |It is possible that not all the instantiable variables
        |occur in the hypothesis of the theorem:  the full
        |definition of the USE command shows how they may
        |be instantiated.
```

We now give the new commands which concern axioms and theorems.

## The AXIOM command.

```
        AXIOM <identifier> : ___,(<stepname>|<awff>),___ ;
```

The system will remember all the <awff>s, mentioned explicitly or designated by an <stepname>, by the name <identifier>; it also lists them - each with a new stepnumber. Thereafter, any THEOREMs created, and saved by the SHOW command, will be tagged as dependent on this axiom.

## The THEOREM command.

```
        THEOREM ( <identifier> : <stepname> |
                ?(  (  ___,<identifier>,___  ) )
                <identifier> : <wff> ?( ASSUME <wff> ) ) ;
```

The first option is for naming a proved result - designated by <stepname> - as a theorem. The second option is for naming an explicit sentence - i.e. <wff> ?( ASSUME <wff> ) - as a theorem, and saying what axioms it depends on (the lists of <identifier>s is a list of axiom names).

In the first option, the system will remember the theorem by name, and tag it as dependent on all axioms present in the system.

In the second option, the system will check that the axioms mentioned are present (if not it will warn you) and in any case will remember the theorem by name, and tag it as dependent on the axioms mentioned. This option is used by the system as follows: when the user saves a THEOREM on a file using the SHOW command, what the system writes on the file is precisely an instance of the second option, so that when the user FETCHes the theorem on a later occasion he will be warned of any appropriate axioms that are not present so that he can FETCH them, too.

## The USE command.

```
USE <identifier> ?___,<stepname>,___ ?( , ___,<instantiation>,___ ) ;

        <instantiation> ::= <identifier> ← <term>
```

The first <identifier> must be a THEOREM name, and the system checks that all axioms on which it depends are present. The system treats the theorem as a metatheorem in that all its free variables, except those which are free in axioms on which it depends, are treated as metavariables to be instantiated. The user supplies the instantiation in part in two ways. First, the list of <stepname>s designates a list of <awff>s, and some or all of the metavariables are bound by matching this list to the antecedent list of the theorem.

Second (since there may be metavariables which occur only in the consequent of the theorem) the user may give a list of instantiations each of which binds a term to a metavariable.

Any metavariables not thus instantiated will just be left as they stand. After matching, the USE command will generate a new step which is simply the appropriate instantiation of the consequent of the theorem. Example:

```
------------------------------------------------
|*****AXIOM AX1;  X≡Y;
|AXIOM AX1
|1 X≡Y
|
|*****THEOREM (AX1) TH1: P≡Z ASSUME Z≡R;
|- - -
|- - -
|15 F(Y)≡G(X,Y) (2 6)
|
|*****USE TH1 15, P←H(X);
|16 H(X)≡F(Y) (2 6)
------------------------------------------------
```

# 4. HOW TO USE THE SYSTEM LCF

## 4.1 Initialization and Termination

```
        R LCF
```

The system returns with an asterisk: you are now talking to LISP.

```
        (INIT)
```

This will initialize the system,which returns with 5 asterisks: you are ready to generate a proof by the commands of Section 3. 5 asterisks is always the signal for a command. Remember, all commands end with a semicolon.

To finish a proof (after maybe preserving it on a file using SHOW) type

```
        $;
```

The system will type ENDPROOF and you are then ready to start another proof with

```
        (INIT),
```

It is possible to save your core image so as to resume the proof at a later time. To do this type

```
        ↑C
        SAVE <filename>
```

and you can then either continue immediately by

```
        START
        (RESUME)
```

or at a later time by

```
        RUN <filename>
        (RESUME)
```

## 4.2 Errors and Recovery

There are three types of error message:

• If you commit a syntax error in a command, the system says

```
SYNTAX ERROR; TRY AGAIN
*****
```

• If your command is semantically suspect - for example, you try to apply TRANS (transitivity) to two steps for which it is inappropriate - you will get something like

```
NASTYTRANS; TRY AGAIN
*****
```

• If you break the system somehow and get a LISP error, usually something like

```
3246 ILL MEM REF FROM ATOM
*
*****
```

then you can try something different (your first command may yield a syntax error, in which case just repeat it) ; however, this should not occur and Malcolm Newey or I would like to know how it occurred.

If the system gets into a loop (the only known cause is if your SIMPSET allows indefinite expansion) then

```
↑C
START
(RESUME)
```

will restore you. If you thereby abort a (long or looping) simplification invoked by the SIMPL tactic you will also need to ABANDON.

## 5. ACKNOWLEDGEMENTS

The system is entirely based on the logic proposed by Dana Scott at Oxford in 1969 but unpublished by him.

I am grateful to Richard Weyhrauch for designing a better simplification algorithm which has proved indispensable, to Malcolm Newey for undertaking the necessary programming for corrections and improvements to the system - including the simplification algorithm - and to both of them for constructive criticisms and discussions which have led to many improvements. I also thank John McCarthy for encouraging me to undertake this work.

The programming of the system was eased enormously by the MLISP2 extendible parser due to Horace Enea and David Smith, and by the help they gave me in using it. In fact, extensions to the system will be simple for the same reason.