June 1984

Report No. STAN-CS-84-1006

EKL-An Interactive Proof Checker User's Reference Manual

by

Jussi Kctonen and Joseph S. Weening

Department of Computer Science

Stanford University
Stanford, CA 94305

EKL—An Interactive Proof Checker

User's Reference Manual

by

Jussi Ketonen and Joseph S. Weening

Research Sponsored by

Advanced Research Projects **Agency**
and
National Science Foundation

Department of Computer Science
Stanford Uni versi ty
Stanford, CA 94305

This version of the EKL rcfcrcnce manual was printed on June 1, 1984.

CONTENTS

1. Introduction.

EKL is an interactive proof checker and constructor. Its main goal is to facilitate the checking of mathematical proofs. Some of the special features of EKL are:

- The language of EKL can be extended all the way to finite-order predicate logic with typed lambda-calculus.
- Several proofs can be handled at the same time.
- Mctathcorctic reasoning allows formal extensions of the capabilities of **EKL.**
- EKL is a programmable system. The MACLISP language is available to the user, and LISP functions can be written to create input to EKL, thereby allowing expression of proofs in an arbitrary input language.

This document is a reference manual for EKL. Each of the following sections discusses a major part of the language, beginning with an overview of that area, and proceeding to a detailed discussion of available features. To gain an acquaintance with EKL, it is recommended that you read only the introductory part of each section.

EKL may be used both at the Stanford Artificial Intelligence Laboratory (SAIL) computer system, and on DEC TOPS-20 systems that support **MACLISP.**

1.1. Acknowledgements.

We are grateful to everyone who contributed to this project, and especially to John McCarthy and Patrick Suppes for their advice and encouragement, and to Richard Weyhrauch and the FOL group for many important discussions. The project benefited indirectly from experience with previous interactive theorem provers, most recently Weyhrauch's FOL, but also from Robin Milner's LCF, Whitfield Diffie's PCHECK and still earlier programs and papers by William Wciher, Paul Abrahams and John McCarthy. Wc also wish to thank Richard Gabriel for introducing us to the mysteries of MACLISP.

This project was partially supported by NSF grants MCS-79-05998, MCS-80-41560 and MCS-82-06565, and ARPA contract MDA903-80-C-0102.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University, or of any agency of the U. S. Government.

2. A first example.

Let us begin with an example of what EKL can do. Suppqsc you are given the task of proving that the least upper bound of a set, is unique. What this means first of all is that we are given an ordering relation $\leq$, which is reflexive: $x \leq x$ for all x; anti-symmetric: $x \leq y$ and $y \leq x$ imply $X = y$ for all x and y; and transitive: $x \leq y$ and $y \leq z$ irnply $x \leq z$ for all x, y, and z. We say that x is an upper bound of a set S if $y \leq x$ for all $y \in S$. And x is a least upper bound of S if x is an upper bound of S and $x \leq y$ for all y which are upper bounds of S. We want to show that if both x and y are least upper bounds of S, then x = y.

A first attempt might look something like the following.

```
Proof? (proof unique)
UNIQUE started.
1.
```

EKL asks for a name for the proof, and we narne it **UNIQUE** by typing (proof unique). EKL confirms this, and asks for the first line.

```
1. (axiom |∀x.leq(x,x)|)

2. (axiom |∀x y.leq(x,y)∧leq(y,x)⊃x=y|)

3. (axiom |∀x y z.leq(x,y)∧leq(y,z)⊃leq(x,z)|)

4. (define bound |∀x s.bound(x,s)≡∀y.in(y,s)⊃leq(y,x)|)

6. (define lub |∀x s.lub(x,s)≡bound(x,s)∧∀y.bound(y,s)⊃leq(x,y)|)

6.
```

We enter as axioms the three properties of the ordering $\leq$, and the definitions of bound and least upper bound. Notice that we say **leq(x,y)** instead of **x≤y**. Later in this manual (section 6.1) you will see how to declare "≤" as an infix name for this relation.

```
6. (derive |∀x y s.lub(x,s)∧lub(y,s)⊃x=y| (1:5))
; failed to derive
∀X Y S.LUB(X,S)∧LUB(Y,S)⊃X=Y

6.
```

Next, we type a formula cxpressin; our desired result, and ask EKL to derive it. The (1: **5)** in the command means that lines 1 through 5 are to be used in determining the validity of the formula. Howevevcr, it turns out that this is too complicated, and EKL reports that it failed to derive our formula.† Since no new line was introduced in the proof, EKL prompts for line 6 again. Now we will attack the problem in smaller steps.

† EKL spends a considerable amount of computer time before failing. As we will see in section 8, this often happens with DERIVE, and makes other commands preferable.

```
6. (assume |lub(x,s)|)
deps: (6)

7. (assume |lub(y,s)|)
deps: (7)
```

The two lines above ask EKL to "assume" that x is a least upper bound of S and that y is a least upper bound of S. The notations deps: (6) and deps: (7), which were typed by EKL, mean that any use of these assumptions will refer back to these lines.

```
8. (derive |bound(x,s)| (6 5))
BOUND(X,S)
deps : (6)

9. (derive |bound(y,s)| (7 5))
BOUND(Y,S)
deps: (7)
```

EKL is now able to make simple derivations, whenever we tell it which lines to. use. The dependencies of the derived lines include lines introduced by the ASSUME command, but not axioms.

```
IO. (derive |leq(x,y)| (6 9 5))
LEQ(X,Y)
deps: (6 7)

11. (derive |leq(y,x)| (7 8 5))
LEQ(Y,X) .
deps: (6 7)

12. (derive |x=y| (10 11 2))
X=Y
deps: (6 7)
```

At this point, we have done the important step in the proof. Now, we want to remove the dependencies, and introduce quantifiers.

```
13. (ci (6 7))
LUB(X,S)∧LUB(Y,S)⊃X=Y

14. (trw |∀x y s.lub(x,s)∧lub(y,s)⊃x=y| (use 13))
∀X Y S.LUB(X,S)∧LUB(Y,S)⊃X=Y

15.
```

The CI command performs "conditional introduction." This means that it forms a conditional formula (an implication) out of the lines specified and the line in the proof before the CI command, and at the same time, it removes dependencies on the lines used.

The TRW command does a "term rewrite" on the term given as its first argument; the rest of the command tells EKL which other lines to use in performing this operation. EKL transforms this formula until it is left with the formula TRUE, and then introduces the formula as a line in the proof. Now we are done, since this was the formula WC set out to prove.

Looking back, it turns out that the only axioms we used are lines 3 and 5. We could therefore shorten the proof by deleting the unused lines.

```
15. (delete1 1 3 4)
;Done.
```

Finally we display the resulting proof. Notice thnt EKL has renumbered all of the lines.

```
12. (show)

1. (AXIOM |∀X Y.LEQ(X,Y)∧LEQ(Y,X)⊃X=Y|)

2. (DEFINE LUB |∀X S.LUB(X,S)≡BOUND(X,S)∧(∀Y.BOUND(Y,S)⊃LEQ(X,Y))| NIL)

3. (ASSUME |LUB(X,S)|)
deps: (3)

4. (ASSUME |LUB(Y,S)|)
deps: (4)

5. (DERIVE |BOUND(X,S)| (3 2) NIL)
BOUND(X,S)
deps: (3)

6. (DERIVE |BOUND(Y,S)| (4 2) NIL)
BOUND(Y,S)
deps: (4)

7. (DERIVE |LEQ(X,Y)| (3 6 2) NIL)
LEQ(X,Y)
deps: (3 4)

8. (DERIVE |LEQ(Y,X)| (4 5 2) NIL)
LEQ(Y,X)
deps: (3 4)

9. (DERIVE |X=Y| (7 8 1) NIL)
X=Y
deps: (3 4)

10. (CI (3 4) 9 NIL)
LUB(X,S)∧LUB(Y,S)⊃X=Y

11. (TRW |∀X Y S.LUB(X,S)∧LUB(Y,S)⊃X=Y| (USE 10))
∀X Y S.LUB(X,S)∧LUB(Y,S)⊃X=Y

12.
```

3. The language of EKL.

EKL is a tool for doing symbolic reasoning. Here we will describe the class of symbols that EKL lets you manipulate. Each symbol is represented by a *name*, which is a sequence of characters that you use in input to the program and which it prints in its output. Some symbols may have more than one name, but in general we will not make much distinction between a symbol and its name.

Each symbol used in EKL has several ***attributes.*** In this section, we will discuss three of the attributes, the type, ***syntype,*** and sort.

3.1. Types.

Most symbols in a proof are either function symbols or the arguments of functions. In some formal systems, such as the pure lambda-calculus, no distinction is made between these; any symbol may be interpreted as a function, and any symbol may be the argument of a function. EKL does not share this view; it is a typed language. (It does have a λ operator, though.)

Types form a hierarchy. At the lowest level, there are ***atomic*** types, which are named by a single identifier. Thus we might have a type NUMBER, a type PERSON, a type A. You are free to create new types whenever appropriate in constructing a proof. There are two predefined atomic types in EKL, called TRUTHVAL and GROUND.

At the next level, there are ***functional*** types, often called operators. These are named by using the symbol "→" to combine other types. For example, **PERSON→NUMBER** is the name of a functional type. If an symbol has type **PERSON→NUMBER**, then it represents a function that takes a PERSON as its argument and returns a NUMBER as its value. More complex constructions are possible. A function of type **(PERSON→PERSON)→NUMBER** takes as its argument a function of type **PERSON→PERSON**, and returns as its value a NUMBER.

Quite often, a function has several arguments. To satisfy the need for expressing this, EKL has product types, which are named using the symbol "⊗". A functionoftype **GROUND⊗GROUND→GROUND** takes two arguments of type GROUND, and returns a result of type GROUND. EKL also has list types, which are used to represent functions taking a variable number of arguments. For example, the type **NUMBER*→NUMBER** represents a function that takes zero or more numbers as its arguments, and returns a number. Products and lists need not appear as the arguments of functions. They can "stand alone," as in the type **NUMBER⊗NUMBER**, which represents a pair of numbers.

There are instances in which EKL needs to verify the correctness of an expression with respect to type. Let us assume that a symbol F has been given the type **NUMBER⊗NUMBER→NUMBER**, and the symbols N and M both have type **NUMBER**. Then F (M, N) is a syntactically correct expression. If P has type **NUMBER⊗NUMBER**, then F(P) is also syntactically correct.

What EKL actually does is consider all functions as having just *one* argument, but this argument can be a *tuple*. A tuple is a sequence of terms represented by enclosing them in parentheses (or square brackets) and separating them by commas. In verifying the syntactic correctness of function applications, EKL may take a tuple appearing as the last argument of a function, and combine it with any preceding arguments into a single list. Thus **F(W,X,(Y,Z))** is the same as F (W , X , Y , Z) . EKL also identifies (X) with X (where X could be a more complicated term than a single variable).

Since list types (such as GROUND*) represent a sequence of zero or more terms, a tuple may have zero terms in it. This is represented as "()". EKL identifies a tuple such as (A, B , C , ()) with

(A,B ,C); and a function applied to the empty tuple, such as F(), can be thought of as a function with no arguments. The type of the empty tuple is represented by the symbol EMPTY.

Some symbols are used in ways that bind variables. Examples of this are the quantifiers ∀ and 3. A binding operator in an expression is followed by a list of variables which it binds, and a term (called the matrix) to which the binding applies. The type of a binding operator is written in the form <T1>⊗T2→T3. Tl is the type of the list of variables (thought of as a tuplc, although it is written without any parentheses), T2 is the type of the matrix, and T3 is the type of the result. Tl, T2, and T3 may be any types. For example, <GROUND⊗GROUND*>⊗TRUTHVAL→TRUTHVAL is a type describing a symbol that binds one or more symbols of type GROUND, has a matrix of type TRUTHVAL, and returns an expression of type TRUTHVAL.

When the type of the variables being bound may be empty, there need be no symbols between the binding operator and the "." that ends the list of variables. Thus, for example, the expression A. G(X) is legal, and is a function that, when applied to the empty tuple, returns G(X).

3.2. Syntypes.

Each symbol has an attribute called its ***syntype,*** which further describes the ways in which it may legally appear in expressions. Therearefivesyntypes: VARIABLE,CONSTANT,BINDOP,DEFINED, and SPECIAL.

Most symbols (including function symbols) have the syntype VARIABLE. This allows the most freedom in use of the symbol. A CONSTANT is like a VARIABLE in most respects, except that it may not be bound in the list of variables associated with a binding operator.

The syntype BINDOP must be declared for any symbol whose type is a binding type, as described above. Bindops are more restricted than constants; they may not appear except with their associated list of bound variables and the matrix to be bound.

The syntypes DEFINED and SPECIAL are assigned to certain symbols by EKL; you cannot declare them yourself. Their meaning will be described later in this manual.

3.3. Predeclared symbols.

As was mentioned above, the types TRUTHVAL, GROUND, and EMPTY are predefined in EKL. TRUTHVAL is a type used to represent the domain of logical values, i.e. the set {true, *false*}. There are two predeclared symbols, TRUE and FALSE, with type TRUTHVAL and syntype CONSTANT. Any term whose type is TRUTHVAL is called a formula or a wff (well-formed formula). Any function whose result is of type TRUTHVAL is called a predicate.

Table 1 shows the other predcclared symbols. The symbols with syntypo SPECIAL do not really have a type as discussed in section 3.1. They are all treated specially by EKL.

Most of these symbols have a special syntax when used in terms. The logical symbols ⊃, ≡, A, V, =, and ≠ are all infix operators, which means you say P⊃Q instead of ⊃(P, Q). The ***conditional*** operator CONDI uses the syntax IF p THEN q ELSE r instead of CONDI (p,q,r). The tuple-generator NTUPLE is accessed by typing (X , Y , Z) or [X , Y , Z]. The others all use the ordinary prefix or bindop forms.

The bindop λ is used to construct functions from terms having free variables. Its type, in any particular instance, depends on the types of its bound variables and its matrix. For example, the term AX. F (G(X)) is the function that is the composition of the functions F and G.

Symbol	Type	Syntype
⊃	TRUTHVAL⊗TRUTHVAL→TRUTHVAL	CONSTANT
≡	TRUTHVAL⊗TRUTHVAL→TRUTHVAL	CONSTANT
∧	TRUTHVAL*→TRUTHVAL	CONSTANT
∨	TRUTHVAL*→TRUTHVAL	CONSTANT
¬	TRUTHVAL→TRUTHVAL	CONSTANT
=		SPECIAL
≠		SPECIAL
CONDI		SPECIAL
NTUPLE		SPECIAL
πi		SPECIAL
πLi		SPECIAL
UNIVERSAL		SPECIAL
NATNUM	GROUND→TRUTHVAL	CONSTANT
∀		SPECIAL
∃		SPECIAL
λ		SPECIAL
'		SPECIAL
↑		SPECIAL
↓		SPECIAL

Table 1. Predeclared symbols.

Accompanying the NTUPLE operator, which combines terms of types $\tau_1, \ldots, \tau_n$ into a term of type $\tau_1 \otimes \cdots \otimes \tau_n$, there are the projection operators π and πL. Each of these is followed by a positive integer indicating the type of projection you want. For example, π1(X,Y,Z)=X, π2(X,Y,Z)=Y, πL1(X,Y,Z)=(Y,Z), and πL2(X,Y,Z)=Z. In general, πi applied to a term of type $\tau_1 \otimes \cdots \otimes \tau_i \otimes \cdots \otimes \tau_n$ yields a term of type τ_i, and πLi applied to such a term yields a term of type $\tau_{i+1} \otimes \ldots \otimes \tau_n$. (A more complete description is given in sec tion 3.5.)

The symbols NATNUM, ', t, and ↓ are described in section 11.

3.4. Sorts.

Each symbol has another attribute called its sort. A sort is a predicate of type τ→TRUTHVAL, where τ is the type of the symbol. Sorts are used to make distinctions between objects in different domains, but that have the same syntactic use and hence the same type.

For instance, it is generally useful to make most low-level (i.e. non-functional) objects have type GROUND. Sorts can be used to distinguish their domains, such as NATNUM (natural number), REAL (real number), SEXP (LISP S-expression), etc.

A sort can be any term of the proper type. In particular, to describe the sorts of higher-level objects, WC could declare operators such as "→", "⊗", "*", and "∩", and then provide axioms stating, for instance, that if a function F has sort A→B, and X has sort A, then F(X) has sort B. This particular axiom would be

$$\forall F\ A\ B.(A \to B)(F) \equiv \forall X.\ A(X) \supset B(F(X)),$$

or

$$\to\ =\ (\lambda A\ B.\ XF.\ \forall X.\ A(X) \supset B(F(X))).$$

Similarly, we can write

$$\forall A\ B\ X\ Y.\ (A \otimes B)(X,Y) \equiv A(X) \wedge B(Y),$$
$$\forall A\ X\ Y.\ A^*(X,Y) \equiv A(X) \wedge A^*(Y),$$
$$\forall A\ B\ X.\ A \cap B(X) \equiv A(X) \wedge B(X).$$

The intuitive descriptions of these definitions are that F has sort $A \rightarrow B$ if F maps all objects of sort A into sort B; a pair has sort $A \otimes B$ if the first component has sort A and the second has sort B (and this can be extended in the obvious ways to 3-tuples, 4-tuples, etc.); a list has sort A* if each element in the list has sort A; and an object has sort $A \cap B$ if it has both sort A and sort B. Note, however, that all of this happens only if you provide the axioms shown above; none of it is built in to EKL.

EKL uses the sort information that you provide in order to correctly eliminate quantifiers and do X-substitutions. For example, EKL will not apply the definitions above to terms unless it can verify that these terms have the required sorts. This is necessary to show the correctness of the theorems that you are proving, and using EKL's sorts prevents the formulas from being cluttered with sort restrictions written explicitly.

If you do not give a sort to a symbol in a declaration, then EKL gives it the sort UNIVERSAL. A symbol with the sort UNIVERSAL does not encounter the restrictions just described. Thus, UNIVERSAL is the "most, general" sort. If X has sort A, and Y has sort UNIVERSAL, then $\forall Y.P(Y)$ is a stronger statement than $\forall X.P(X)$.

3.5. Types, revisited.

This section presents a complete and formal description of the type system used by EKL. The only reason that EKL has a type system at all is to prevent self-application of functions. This means that expressions like F(F) are never allowed, and through a process called type checking, EKL ensures that no formula having self-application can arise. The reason for doing this is that self-application can lead to logical contradictions.

If you are reading this manual for the first time, you should probably skip to the next section.

Before presenting the formal type system, there is one more type operator to be introduced. This is the "v" symbol, which stands for the combination of types. (Note that this is unrelated to the "v" symbol that may be used in terms.)

A function of type $(GROUND \vee TRUTHVAL) \rightarrow TRUTHVAL)$ can be used with an argument that has either type GROUND or type TRUTHVAL; its result will be of type TRUTHVAL. Types such as this are most often generated by EKL for its own use in checking the well-formedness of expressions; you will not often need to use them in declarations.

A type T_1 is said to be a subtype of a type T_2 if having type T_1 implies having type T_2. This is written as $T_1 \preceq T_2$. For example, TRUTHVAL $\preceq$ $GROUND \vee TRUTHVAL$ and $GROUND \otimes GROUND \preceq$ GROUND*. To keep terms properly typed we require that the type of any term be a subtype of the type of a term it replaces, whenever EKL makes a substitution.

The EKL type structure is an algebra $(\mathcal{T}, \otimes, \vee, \rightarrow, *, \preceq, \text{EMPTY})$ with the following set of axioms.†

1. $\otimes$ is a left-associative operation on $\mathcal{T}$ with EMPTY as a right unit.
2. $\vee$ is an associative, commutative and idempotent operation on $\mathcal{T}$. It is distributive over $\otimes$.
3. The $\preceq$ relation is a partial order with the following additional properties: For all types $T, T_1, T_2, \ldots, T', T_1', T_2', \ldots,$

† This entire discussion is describing EKL as a mathematical object; thus these axioms are not being formulated as EKL terms themselves.

a. $T \preceq T_1$ if and only if $T \vee T_1 = T_1$.
b. EMPTY $\preceq T^*$ and $T \otimes T^* \preceq T^*$. Hence, $T^* \succeq \text{EMPTY} \vee T \vee (T \otimes T) \vee (T \otimes T \otimes T) \vee \cdots$.
c. $T \preceq T_1' \rightarrow T_2'$ iff $T = T_1' \rightarrow T_2'$; also $T_1 \rightarrow T_2 = T_1' \rightarrow T_2'$ iff $T_1 = T_1'$ and $T_2 = T_2'$. Similarly for binding types.

We will call a ***P-type*** any type that is $\succeq$ a product type.

In the theory (but not in the actual type language), there arc projection operators π_i and πl_i on types, which correspond directly to the operators πi and $\pi \mathrm{L} i$ on EKL terms. Here are listed rules for the type of expressions involving π_i and πl_i; from these rules you can easily determine the result of an expression that uses πi or $\pi \mathrm{L} i$.

1. If x has type T, then the type of π_i (x) is $\pi_i(T)$, which is
   a. T, if T is not a P-type,
   b. T_i, if $T = T_1 \otimes \cdots \otimes T_n$ and i < n and EMPTY $\not\preceq T_n$,
   c. $\pi_{i-n+1}(T_n)$, if $T = T_1 \otimes \cdots \otimes T_n$ and $i \geq n$ and EMPTY $\not\preceq T_n$,
   d. $\pi_i(A_1) \vee \cdots \vee \pi_i(A_n)$, if $T = A_1 \vee \ldots \vee A_n$,
   e. $\text{EMPTY} \vee \pi_1(A) \vee A$, if $T = A^*$ and $i = 1$,
   f. $\pi_i(A) \vee \pi_{i-1}(T)$, if $T = A^*$ and $i > 1$.
2. If x has type T, then the type of πl_i (x) is $\pi l_i(T)$, which is
   a. T, if T is not a P-type,
   b. $T_{i+1} \otimes \cdots \otimes T_n$, if $T = T_1 \otimes \ldots \otimes T_n$ and i < n and EMPTY $\not\preceq T_n$,
   c. $\pi l_{i-n+1}(T_n)$, if $T = T_1 \otimes \ldots \otimes T_n$ and i $\geq$ n and EMPTY $\not\preceq T_n$,
   d. $\pi l_i(A_1) \vee \cdots \vee \pi l_i(A_n)$, if $T = A_1 \vee \cdots \vee A_n$,
   e. $\pi l_1(A) \vee A$, if $T = A^*$ and i = 1,
   f. $\pi l_i(A) \vee \pi l_{i-1}(T)$, if $T = A^*$ and i > 1.

The other type rules employed by EKL in type checking are:

1. If x_i has type T_i, for i = 1,. . . , n, then $(x_1, x_2, \ldots, x_n)$ has type $T_1 \otimes T_2 \otimes \cdots \otimes T_n$.
2. If x has type T_1 and y has type T_2 and p has type $\preceq$ TRUTHVAL, then IF p THEN x ELSE y has type $T_1 \vee T_2$.
3. For any x and y, $x = y$ has type TRUTHVAL.
4. For any x, UNIVERSAL(z) has type TRUTHVAL.
5. For any x, tx and 'x have type GROUND.
6. For any x of type GROUND, $\downarrow x$ has a variable type.
7. If x_i has type T_i for $i = 1, \ldots, n$, then $\forall x_1 \ldots x_n . p$ and $\exists x_1 \ldots x_n . p$ have type TRUTHVAL if p has type $\preceq$ TRUTHVAL, and Xxi . . . $x_n . p$ has type $T_1 \otimes \ldots \otimes T_n \rightarrow T$ if p has type T.
8. If F has type $T_1 \rightarrow T_2$ and x has type $\preceq T_1$, then $F(x)$ has type T_2.
9. If the bindop B has type $\langle T_1 \rangle \otimes T_2 \rightarrow T_3$, and *varlist* has type $\preceq T_1$ (when viewed as a tuple), and m has type $\preceq T_2$, then the expression B *varlist* . m has type T_3.

3.5.1. Variable types.

With *variable* types, EKL lets you dispense with completely describing the type of each symbol. Instead, the types of symbols arc defined implicitly by their use in expressions. Variable types are named by preceding au identifier with the symbol "?". It is easiest to see how they are used by means of an example.

Suppose WC let X and Y have type ?ABC, let F have type ?B, and let G have type ?P. EKL would accept expressions such as X=F (Y) , or G(F (X))=X, since it is able to determine that there exist assignments to the variable types that make these legal expressions. (It will not tell you what

those assignments are, unless you ask for them using the SHOW-TYPEBINDS command.) In this case, there is an infinite class of such assignments, since all the types are variable. If we made X and Y of type GROUND, though (or any non-variable type), there would be only one valid assignment of types to ?B and ?P. If we try an expression such as G(G), EKL will complain, since there is no valid assignment to the type of G that makes this expression legal.

Here is an example of the use of variable types, making use of the LISP axioms defined in section 13. (Come back to this later if you don t understand all of the things that it does.) The axiom HIGH~ORDER~DEFINITION and its context are as follows:

```
;labels:HIGH,ORDER,DEFINITION
38. (AXIOM
      |∀BIGFUN ATOM_FUN.(∃DEFINED_FUN.(∀X Y.(ATOM X⊃
                                          DEFINED_FUN(X)=ATOM_FUN(X))∧
                                         DEFINED_FUN(X.Y)=
                                         BIGFUN(X,Y,DEFINED_FUN(X),
                                               DEFINED-FUN(Y))))))

(show-context high-order-definition)

; context:
;ATOM has syntype CONSTANT, and type GROUND→TRUTHVAL
;SEXP has syntype CONSTANT, and type GROUND→TRUTHVAL
;Z has syntype VARIABLE, and type GROUND
;Y has syntype VARIABLE, and type GROUND
;X has syntype VARIABLE, and type GROUND
;CONS has syntype CONSTANT, and type (GROUND⊗GROUND)→GROUND
;ARB has syntype VARIABLE, and type ?ARBITRARY
;BIGFUN has syntype VARIABLE, and type (GROUND⊗GROUND⊗(@ARB)⊗(@ARB))→(@ARB)
;ATOM_FUN has syntype VARIABLE, and type GROUND→(@ARB)
;DEFINED_FUN has syntype VARIABLE, and type GROUND→(@ARB)
```

The definition uses the symbol ARB having the variable type ?ARBITRARY.

This axiom is used to justify the existence of various recursively defined LISP functions. For example, one can prove the existence of the function APPEND from it.

The reason why we use a variable type in HIGH~ORDER~DEFINITION is that we may want to define functions of different types using it. Thus in one application of the line we may define *a* LISP function of type GROUND→GROUND, whereas in another application the function might be of type GROUND+TRUTHVAL.

In the following we prove the existence of a LISP function FLAT with the following properties:

```
∀X Y Z.(ATOM X⊃FLAT(X,Y)=X.Y)∧FLAT(X.Y,Z)=FLAT(X,FLAT(Y,Z))
```

This definition is not in the form suggested by HIGH-ORDER-DEFINITION. However, we can define a higher type object FLATFUN(X)=λY.FLAT(X,Y) using it. This is done by applying the axiom HIGH~ORDER~DEFINITION with variables ATOM-FUN and BIGFUN instantiated in order to force EKL to do the right type coercions.

```
(proof flat)

1. (define flatfun |∀x y.(atom(x)⊃flatfun(x)=(λY.X.Y))∧
                    (flatfun(X.Y)=(λZ.(flatfun(X))((flatfun(Y))(Z))))|
```

```
(use high-order-definition
     ue: ((atom-fun . |λx.λy.x.y|)
  (bigfun . |λx y arbl arb2.λz.arb1(arb2(z))|)) ))
;FLAT is unknown.
;the symbol FLAT declared to have type (GROUND⊗GROUND)→GROUND
```

ATOM-FUN has the type GROUND→(GROUND→GROUND) above. Thus the variable type ?ARBITRARY has been coerced into GROUND→GROUND in the application of HIGH_ORDER_DEFINITION. We can now define FLAT and prove some of its properties.

```
2. (define flat |∀x y.flat(x,y)=(flatfun(x))(y)()

3. (ue (phi |λx.∀z.sexp (flatfun(x))(z)|) sexpinduction (open flatfun))
;∀X Z.SEXP (FLATFUN(X))(Z)
'(label simpinfo)

4. (trw |∀x y z.(atom(x)⊃flat(x,y)=x.y)∧(flat(x.y,z)=flat(x,flat(y,z)))|
     (open flat flatfun))
;∀X Y Z.(ATOM X⊃FLAT(X,Y)=X.Y)∧FLAT(X.Y,Z)=FLAT(X,FLAT(Y,Z)) .
```

4. The input syntax.

What you type to EKL is a sequence of characters, but inside the computer it is represented differently. To perform the translation, EKL has a *parser*. It is not necessary to know how the parser works, but it is useful to know which input strings it accepts and which it rejects, and how it resolves ambiguities.

As a first step, we view the input stream as a sequence of tokens. Tokens may be formed out of letters and digits, or out, of special characters like "A" and "+". Thus "A", "DOG5", "c", and "∃!" are all valid tokens, while "F*" and "TREE-TOP" would be subdivided into several tokens.† The charac tcr "_" is treated as a letter. Spaces may be used to delimit tokens, as in the list of variables in "VX Y.P(X,Y)".

Some tokens may be declared to be infix names, postfix names, or *bindop* names. All other tokens, except for

( ) [ ] { 3 . , : IF THEN ELSE

are known as ***identifiers.*** Identifiers may be used either as the arguments to functions, or as functions themselves in prefix form. Parentheses may be dropped in many cases when there is only one argument to a function. Thus "CAR(X)" and "CAR X" are equivalent.

The main work of the parser is to form a ***term*** out of the sequence of tokens. **A** term can take any of the following forms.

term term	(function application)
term infixop term	
term postfixop	
bindop identifier . . . identifier . term	
(***term***)	
(***term , . . . , term***)	
[***term , . . . , term***]	
{ ***term , . . . , term*** 3	
{ ***identifier*** : ***term*** 3	
IF term THEN ***term*** ELSE ***term***	

Not all expressions formed by the above rules are meaningful terms, though. The previous section discussed restrictions involving the type and syntype of symbols.

Parentheses can be used to delimit a term, or to form a tuple from a list of terms. Square brackets may be used in the same way.

When using a binding operator, the "·" following the list of bound variables is optional if the next token is not an identifier. Thus, you may write things like ∀Y(X=F(Y)) or ∀X∃Y.P(X,Y).

The two constructions using braces allow a special syntax to bc used for the symbols FINSET and CLASS. The first case is equivalent to the function FINSET with the arguments in the termlist. The second is equivalent to the bindop CLASS, binding the single variable before the ":", and having the term after the ":" as the matrix. Note, however, that FINSET and CLASS are not predeclared; thus their type must be given before using these special forms. See section 6.1 for an example of how this is done.

---

† A sequence of special characters such as "3!" must be declared before it will be recognized as a single token. Digits may be included in such a string.

The description above leaves many ambiguities unresolved. For example, the term "a+b*c", where "+" and "*" are infix operators, can be interpreted as equivalent to either "(a+b) *c" or "a+(b*c)". The choice that is made depends on the relation between an attribute of "+" and "*" called the *binding power.* This is a number between 0 and 1000. If "+" has a higher binding power, the first alternative will be chosen, while if "*" has a higher binding power, EKL will use the second. (If they are the same, EKL associates to the left, so the first alternative will be used.)

The predefincd operators have the following binding powers:

↑, ↓	900
=, ≠	800
1	700
∧	600
V	500
≡	400
⊃	300
∀, ∃, λ	200
CONDI	100

Another ambiguity involves the order of function application, when parentheses are dropped. For example, "A B C" could mean "(A(B))(C)" or "A(B(C))". To resolve this, EKL looks at the binding powers of A and B; if A's is higher then "(A(B))(C)" is chosen; if B's is higher or they are the same, "A(B(C))" is used. The default binding power is 1000, so most of the time the second form, which is more common in mathematics, is used.

4.1. Special characters.

Unless you are using the SAIL system, your computer may not allow you to use all of the special characters printed in this manual. EKL on such a system will take input and print output in a different form. This is done by using a different special character, or a sequence of characters, for each of the non-standard symbols. The correspondence is as follows.

∀	ALL
3	EX
λ	LAMBDA
≡	IFF
⊃	IMP
A	&
V	OR
1	NOT
t	^ or t
↓	UNQT
π	PROJ
πL	PRO JL
⊗	'
→	:

Some terminals have a t key, in which case that is used. Otherwise, the ^ character, which has the same internal code, is used instead.

Using some of these means that there must be extra spaces in formulas. For example, A≡B becomes A IFF B, and π2 (X) becomes PROJ 2(X).

5. EKL proofs.

A proof in EKL is a sequence of *lines*. Each line either is a ***declaration*** or contains a command and a formula. At any time, there may exist several proofs. To distinguish them, each proof has a proofname. One proof is always the currently active proof. To refer to a line within a proof, we give its name and number. Thus, "FOO#4" refers to the fourth line in the proof named FOO.† A line in the currently active proof may be referred to by just its number.

A negative number tells EKL to count lines from the end of the proof. Thus -1 means the last line in the current proof, and "FOO#-2" is the next-to-last line in the proof FOO. The symbol "*" may also be used to refer to the last line in the current proof.

A *linerange* is a set of lines. They need not all be in the same proof. An expression like "FOO#4:7" refers to a linerange of consecutive lines in a proof. Other lineranges are represented by lists of lines and lineranges, e.g.

(FOO#4 FOO#6:7 3)

refers to lines 4, 6, and 7 in the proof FOO, and line 3 in the currently active proof. Using a command described later on, you can give a name to a linerange, and thus refer to it symbolically.

The purpose of declaration lines is to assign attributes to symbols. When you do this, you create a context, which is the set of attributes of all currently defined symbols. The importance of this concept is that each line in a proof may have its own context. Thus, in one line X could be an ordinary symbol of type GROUND, while in another it could be an operator. No problems will arise unless you try to combine these two lines in some way; then EKL will complain that the types are incompatible.

Lines containing formulas assert that those formulas are valid statements in the proof. Each such line may have ***dependencies,*** which are other lines on whose truth the formula depends. Thus, we may have, the formula "VX. P (X)" on line 5, and have "P(A)" on line 9 with a dependency on line 5. Dependencies may only contain preceding lines in the current proof, or lines in other proofs. (In some cases, a line may also depend on itself.) The command associated with each line describes how to prove the formula on that line.

The goal of a proof is to generate a line with some desired formula on it, having dependencies only on ***axioms*** that you are willing to accept as true. To facilitate this, EKL allows you to declare certain lines as axioms; dependencies on these lines are not shown, making your goal to get the desired formula on a line with no dependencies.

† Usually MACLISP defines "#" to be a special character on input. EKL has this feature explicitly turned off.

6. Basic commands.

You create a proof in EKL by giving the program a sequence of commands. The system is interactive, in that there are commands to find out the state of any proof, and commands to undo the action of previous commands, in addition to the commands that cause a change to the current proof.

The first thing to do is start the EKL program. The way to do this depends on the system that you are using. On a TOPS-20 system, if EKL resides in the directory **<EKL>**, type

```
@<ekl>ekl
```

where "@" is the monitor's prompt. At SAIL, type

```
.r ekl
```

to the monitor's "." prompt. EKL will first say

```
Proof?
```

Right now, there is no currently active proof, and you can do almost nothing until you define one. So, make up a proofname, say TEST, and type

```
Proof? (proof test)
TEST started.
1.
```

EKL commands use the syntax of LISP: the form **(func arg1 arg2 . . . argn)** means "apply the function FUNC to the arguments **ARG1** through **ARGn**." Because EKL commands are in fact LISP functions, the terms "function" and "command" will be used interchangeably throughout this manual.

Most of the commands that you type to EKL, which we will call **"top-level"** commands to distinguish them from LISP functions used in other contexts, are of a special form that does not evaluate its arguments. If they did evaluate arguments, you would have to insert QUOTEs in front of almost everything.

Occasionally, when one of the arguments is itself the result of evaluating a LISP function, you will want an EKL command that does evaluate its arguments. These functions are described in section 12.1.

In addition to not evaluating any arguments, top-level commands allow some arguments to be optional. ***Any*** of these may be given its ***default*** value either by omitting it, if there are no arguments following it, or by giving it the value NIL. In this manual, the symbol "***&optional***" will indicate that all function arguments following it are optional. **As** an example, given the function description

(COM line ***&optional*** number mode),

the following are all legal as input to EKL:

(corn 5) uses 6 for line, and the default values for number and mode.

(corn 5 2) specifies line and number, and uses the default for mode.

(corn 5 nil insert) specifies **line** and mode, and uses the default for number.

(corn **5** 2 insert) specifies all three arguments.

When the symbol ***"&rest"*** is used in a function description, it means that all arguments from that point on (if any) are collected into a list and assigned to the parameter named after the ***&rest.*** For example, using

(TRW term ***&rest*** rewriter),

we have

(trw |p(x)| 3) uses (3) for rewriter.

(trw |p(x)| 3 4 7) uses (3 4 7) for rewriter.

(trw |p(x)|) uses NIL for rewriter.

The symbols ***"&optional"*** and "*&rest*" are never typed by you as input to EKL; they just serve as a notation in this manual for describing the arguments to commands.

Note the use of vertical bars to delimit "p(x)" in the examples above. These are necessary since otherwise the parentheses would be interpreted as defining a list in LISP. Lower-case input is converted by EKL to upper-case, even within vertical bars.

EKL commands may be abbreviated as long as the abbreviation is unambiguous. For example, the PROOF command may be abbreviated to PRO, since it is the only command beginning with the letters PRO, but not to P or PR, since other commands begin with those letters. In this manual, however, we will only show unabbreviated forms of the commands.

6.1. Declarations.

A *declaration* is used to supply attributes to a symbol. Each symbol you use in an EKL proof has a declaration, either one that you give for it, or one that is generated by EKL. The latter kind is called a ***default*** declaration. You can give explicit declarations using the DECL command.

The syntax of DECL is

(DECL identifiers &rest attributes)

Identifiers is either a single identifier, or a list of identifiers enclosed in parentheses. Attributes is a list of attributes for the symbol or symbols being declared.

The following are the attributes that can be assigned. Each attribute that is not assigned is given a default value.

TYPE: specifies the type of the symbols being declared. Types are described in section 3.1. The default (if TYPE: is not specified) is type GROUND.

SYNTYPE: specifies the syntype of the symbols being declared, one of CONSTANT, VARIABLE, or BINDOP. The default is VARIABLE. If the syntype is BINDOP, then the type must be a binding type.

SORT: specifies the sort of the symbols being declared. Sorts are described in section 3.4.

BINDINGPOWER: specifies the binding power of the symbols being declared, affecting parenthesization of expressions. It may be any integer between 0 and 1000. The default is 1000.

SPECIALFORM: is used to select one of the special forms recognized by the parser. It may be FINSET or CLASS.

INFIXNAME: specifies the token that the parser will use to recognize the symbol as an infix operator.

PREFIXNAME : , POSTFIXNAME : , and BINDNAME: similarly specify the tokens used to recognize the symbol as a prefix operator, postfix operator, or bindop.

UNARYNAME : is like PREFIXNAME : , except that it indicates that you prefer to see the argument of this Function printed without parentheses. This can only be done for one-argument functions.

ASSOCIATIVITY: can be LEFT, RIGHT, BOTH, or NIL, the default being NIL. This provides information that the rewriter (see section 7.1) will use to simplify expressions involving this operator. Associativity is only meaningful for operators of certain types. For example, if "+" is the infixname for a symbol of type GROUND⊗GROUND*→GROUND and it is declared left associative, then (A+B)+C will be rewritten as A+B+C, but A+(B+C) will not be rewritten. Right associativity docs just the opposite, and BOTH says that both types of rewriting are desired. NIL means that the operator is not associative. Bindops may be associative too; an example is "∀", for which the term ∀X . ∀Y. F (X , Y) will be rewritten as ∀X Y. F(X,Y). Most of the predeclared symbols (section 3.3) are associative; the binding operators are right associative.

Here are some examples of declarations:

1. (decl (a b c))

This just tells EKL that you are going to use A, B, and C as variables. Their type is not given, so EKL will give them the type GROUND. If, instead, you use A, B, and C without giving a declaration, two things will be different: first, EKL will print messages like

```
;A is unknown.
;B is unknown.
;C is unknown.
;the symbol C declared to have type GROUND
;the symbol B declared to have type GROUND
;the symbol A declared to have type GROUND
```

and second, the types may be other than GROUND, if the symbols are used in a place where another type is required.

2. (decl x (type: ground) (sort: sexp))

This says that Xis of type GROUND, syntype VARIABLE, and sort SEXP. If SEXP has not already been declared (an appropriate type for it would be GROUND→TRUTHVAL), it will be given a default declaration (see section 6.1.1).

3. (decl append (type: | ground⊗ground*→ground |) (syntype : constant)
(associativity: both) (infixname: *) (bindingpower: 840))

This shows many of the options of DECL, and is typical of the declaration of an infix operator. The syntype is CONSTANT because we never envision ourselves saying things like ∀APPEND.(...). By declaring APPEND associative, you will never have to prove formulas like A*(B*C)*D=(A*B)*(C*D); in fact, EKL will quickly change both sides of this equation into A*B*C*D for you. Internally, this is the same as APPEND(A,B,C,D). Using the name APPEND instead of * is especially useful if you want to say something like

APPEND=λU V.IF NULL U THEN V ELSE CAR(U).(CDR(U)*V),

where ". " has been defined as an infixname for CONS. The way to declare CONS for this would be

```
(decl cons (type: |(ground⊗ground)→ground|) (syntype: constant)
 (inf ixname : |. |) (prefixname: cons) (bindingpower: 850))
```

No ASSOCIATIVITY: is given, because it is not meaningful for functions that take only two arguments. If the input contains a term like A. B. C, without parentheses, it will be parsed as **(A.B).C.**

To use the special forms **{** ***term*** , . . . , ***term*** **}** and **{** ***identifier*** : ***term 3*** described in section 4, you must make a declaration for the operators **FINSET** and CLASS. **A** typical case is as follows.

1. **(decl finset** (type: **|ground*→ground|)** (specialform: **finset))**

2. (assume **|x={a,b,c}∧y={d,e}∧z={f}|)**
deps: (2)

3. **(decl** class (type: **|<ground>⊗truthval→ground** I) (syntype : bindop)
(specialf orm: class))

4. (assume **|x={y:p(y)}|)** .
deps: **(4)**

If, after you declare a symbol, you want to change some it its attributes, you can use the function REDECLARE. This has the form

(REDECLARE symbol attributes),

where attributes may include the following:

(INFIXNAME: name)
(POSTFIXNAME: name)
(PREFIXNAME: name)
(UNARYNAME: name)
(BINDNAME: name)
(BINDINGPOWER: number)

These new attributes take effect immediately, and will be used in printing all formulas in lines connected to the current declaration of symbol.

6.1.1. Default declarations.

If you use a symbol without first declaring it, EKL will provide a declaration for you. Some default declarations came up in the previous set of examples.

If the type of the symbol is unknown, EKL will compute a type which is valid with respect to your use of the symbol. If, for example, you type (derive **|p(x,f(y))|**), and F, P, X, and Y are all undeclared, then EKL will assign the type GROUND to X and Y, **GROUND→GROUND** to F, and **GROUND⊗GROUND→TRUTHVAL** to P. It knows that P must result in a TRUTHVAL because the DERIVE command always operates on a formula.

The other attributes assigned to a new symbol may depend on symbols that are already declared. If there is already a symbol named, say, XYZ, and you define a new symbol with XYZ as its initial characters, say XYZA or XYZ23, and if the type computed for your new symbol is the same as the type of XYZ. then EKL will give the new symbol the same attributes as the existing symbol, unless XYZ has an infixname, prefixname, unaryname, or bindname different from XYZ. (Sometimes EKL generates new symbols on its own, and uses this rule.) If you use a new symbol which has no prefix that is already a declared symbol, then EKL will give it all of the default attributes described above. If there are several prefixes that match (for example, if there is also an XY), then the attributes of the one which matches in the largest number of characters are used.

Whenever a default declaration is made, EKL prints a message. To supress these messages, you can give the command

(SETQ DECTALK NIL).

6.2.. Introducing facts.

In addition to describing the attributes of symbols, a proof usually begins with the statement of some initial facts, from which a desired conclusion is derived. EKL provides four commands to introduce facts.

The first two, ASSUME and AXIOM, are closely related.. The syntax for these commands is

(ASSUME formula),
(AXIOM formula).

Both commands introduce formula as the next line in the current proof. The difference between ASSUME and AXIOM is that ASSUME generates a dependency on its own line, while AXIOM does not.

If an axiom is meant to serve as the definition of a particular symbol in it, then it is better to use the DEFAX command, which has the form

(DEFAX symbol formula).

This will enable EKL to automatically use the definition in matches involving symbol. Often such matching goes on "behind the scenes," and giving this information frees you from having to refer to the DEFAX line explicitly. For example:

1. (defax a |∀x.a(x)=c|)

2. (trw |a(1)=c|)
A(1)=C

In all other respects, DEFAX is just like AXIOM.

The fourth command for introducing facts is DEFINE. It has the form

(DEFINE symbol formula &test rewriter).

The idea of DEFINE is that formula expresses a definition of symbol. The definition is accepted when EKL can verify that the formula

∃symbol.formula

simplifies to TRUE, using the rewriter (see section 7). The rewriter. parameter may be provided to give EKL information needed to do this. If it is able to do so, EKL introduces formula as a line in the proof.

After finishing a DEFINE command, **symbol** is given the syntype DEFINED in the current context.† This means that lines using symbol cannot be combined with lines where it has been introduced by other means. Here is an example.

1. (axiom |¬∀x(x=zero)|)

2. (define one |one≠zero| (use 1))

EKL accepts this line, and there are no dependencies. If we had said ASSUME rather than AXIOM in line 1, then line 2 would have a dependency on line 1. (Of course, the usual definition of the number "one" contains more information than this.) Note that if we say

3. (assume |one≠zero|)
deps: (3)

we get the same formula, but with a dependency, which will carry through to any line that depends on this one.

You should use **AXIOM** or DEFAX only for those formulas that really are axioms in the theory that you arc working in. No such restrictions apply to DEFINE, even though it does not cause a dependency on its own line, as ASSUME does. (DEFINE may cause dependencies on lines other than its own, through lines given explictly in the rewriter parameter, or lines in SIMPINFO, described in section 7.2.) .

DEFAX and DEFINE look almost the same, but there is an important difference: when you say DEFAX, EKL does not check that ∃**symbol**. formula is valid; it simply accepts it as an axiom. DEFINE causes EKL to check the validity of the formula, so you do not need to worry about changing your theory (possibly making it inconsistent!) when you introduce a new symbol.

† Its other attributes, such as type, sort, etc. remain the same.

7. Rewriting.

The most frequent way of generating new formula lines in **EKL** is by using a process known as rewriting. This means using information that is already part of the proof to simplify formulas, while preserving their validity, or to substitute equals for equals. EKL's rewriting commands also have access to a decision procedure, which tries to verify the validity of formulas directly. Whenever the decision procedure is applied to a formula, and it succeeds, the formula is replaced by TRUE.

The basic rewriting command is RW, which has the syntax

(RW ***&optional*** line &rest rewriter).

Line specifies the line to be rewritten. (See section 5 for the ways to name a line.) The default for line is "*" , the previous line in the proof. Rewriter tells EKL what kind of rewriting to do. When RW is done, the formula that results from rewriting line is introduced as a new line in the proof.

The function TRW does the same operations as RW, but on an arbitrary term instead of on the formula of a line. Its syntax is

(TRW term ***&rest*** rewriter).

TRW changes term to a new expression, say **term1**, using the rewriting rules requested by rewriter; then if **term1** is the formula **TRUE**, term is introduced as a line in the proof. Otherwise the formula **term=term1**, or **term≡term1** if the terms are of type TRUTHVAL, **is** the proof line created.

7.1. Standard rewriting.

Whenever rewriting is done, certain standard simplifications are made. They arc the following, in which an expression of the form $\alpha \rightarrow \beta$ means that wherever α occurs., it is replaced by β.

¬TRUE → FALSE	¬FALSE → TRUE	¬¬P → P	
TRUE∧P → P	FALSE∧P → FALSE	P∧TRUE → P	P∧FALSE → FALSE
TRUE∨P → TRUE	FALSE∨P → P	P∨TRUE → TRUE	P∨FALSE → P
TRUE⊃P → P	FALSE⊃P → TRUE	P⊃TRUE → TRUE	P⊃FALSE → ¬P
TRUE≡P → P	P≡TRUE → P	FALSE≡P → ¬P	P≡FALSE → ¬P
TRuE=P → P	P=TRUE → P	FALSE=P → ¬P	P=FALSE → ¬P

IF TRUE THEN P ELSE Q → P
IF FALSE THEN P ELSE Q → Q
IF P THEN TRUE ELSE Q → P∨Q
IF P THEN FALSE ELSE Q → ¬P∧Q
IF P THEN Q ELSE TRUE → P⊃Q
IF P THEN Q ELSE FALSE → P∧Q
X≠Y → ¬(X=Y)
UNIVERSAL(X) → TRUE
XX. F(X) → F if the sort of X is UNIVERSAL
$\pi i(x_1, x_2, \ldots, x_n) \rightarrow x_i$ if $1 < n$ and $i < n$
$\pi i(x_1, x_2, \ldots, x_n) \rightarrow \pi i - n + 1(x_n)$ if $1 < n$ and $i \geq n$
$\pi i(x) \rightarrow x$ if the type of x is not a P-type

. $\pi Li(x_1,x_2,\ldots,x_n) \rightarrow (x_{i+1},\ldots,x_n)$ if 1 < n and $i < n$
$\pi Li(x_1,x_2,\ldots,x_n) \rightarrow \pi Li-n+1(x_n)$ if $1 < n$ and $i \geq n$
$\pi Li\,(x) \rightarrow$ x if the type of x is not a P-type
$(x_1,x_2,\ldots,x_i,(x_{i+1},\ldots,x_n)) \rightarrow (x_1,x_2,\ldots,x_i,x_{i+1},\ldots,x_n)$
$(x_1,x_2,\ldots,x_n,()) \rightarrow (x_1,x_2,\ldots,x_n)$

The last substitution above can be considered a special case of the one before it.

If A and **B are** "the same," meaning that they differ only in the names of bound variables, and the corresponding names have the same sort, then the following simplifications are made.

A=B → TRUE **A≡B** → TRUE **A⊃B** → TRUE
IF P THEN A ELSE B → A

Also, unnecessary quantified variables and quantifiers are dropped, X-eliminations are made if the sorts match, the declared associativity of operators is used to simplify terms, and conditional expressions are pushed out (an expression like **F(** IF P THEN A ELSE **B)** is converted to IF P THEN F(A) ELSE F(B)), but not past binding operators or the operators **∧**, ∨, and **⊃**.

Conditional replacements are done in the standard rewriting procedure. In an expression such as **P⊃Q**, an instance of P within **Q** is replaced by TRUE; similar rules are used in more complex situations. When rewriting the right-hand side of a conditional formula, EKL keeps a list of facts that may be assumed. For example, when rewriting D in **(A⊃B)⊃ (C⊃D)**, the facts that may be assumed are **A⊃B** and C.

These facts are used to verify whether the conditions of application of a rewriting rule are valid, and to rewrite subparts to TRUE in conditional reduction. In the above situation, if D unifies with C, then EKL will rewrite D to TRUE. This can also be done if D unifies with B and A unifies with C. Thus the rewriting system can deal in a rather powerful way with conditional statements. Also, in a term like **P∧Q**, EKL may assume that **Q** is TRUE in rewriting P, and in **P∨Q**, it may assume that **Q is** FALSE in rewriting P.

Another type of condition of application is sort restrictions on unifiable variables. If X has sort S, then **∀X .P(X)** may be applied to P(A) only if EKL can verify that A has sort S.

The standard rewriting procedure will also replace a term of the form 3X. P **(X)** with TRUE if there is a statement of the form **∃Y . Q** (Y) or Q(Y) in the current **sct** of facts such that P(T) is equal to **Q(Y)** for some term T.

Here are some examples of formulas that can be completely proved (i.e., rewritten to **TRUE)** with the above rules.

1. (trw |a=b⊃b=a|)
A=B⊃B=A

2. (trw |a=b∧b=c⊃c=a|)
A=B∧B=C⊃C=A

3. **(trw |a=f (b)∧b=f (a)⊃f** (f **(a))=a|)**
A=F(B)∧B=F(A)⊃F(F(A))=A

Conditional replacements allow fairly complex formulas to be **rewritten:**

4. **(decl (<)** (infixname: **<)** (type: **|ground⊗ground→truthval|)**
(bindingpower : 650))

5. (trw **| (∀x** y. **(x<y)⊃inc(seq(y)** ,**seq(x)))∧(∀x.a1(x)⊃(x<succ(z1)))⊃**
(∀x.a1(x)⊃inc(seq(succ(z1)) ,seq(x))) **|)**

```
(∀X Y.X<Y⊃INC(SEQ(Y),SEQ(X)))∧(∀X.A1(X)⊃X<SUCC(Z1))⊃
(∀X.A1(X)⊃INC(SEQ(SUCC(Z1)),SEQ(X)))
```

7.2. Specifying rewriters.

Rewriters are specified as LISP lists. The following are all of the forms for rewriters.

rewriter . . . rewriter
@name
(USE linerange &rest options)
(OPEN &test symbols)
(PART subpart &rest rewriter)
(FORM term &rest rewriter)
(NUSE *&rest* linerange)
(DER &rest linerange)
(NDER *&rest* linerange)

The first form above indicates that a rewriter may be a list of rewriters. (Usually the rewriter parameter of a command is defined as a *&rest* argument. Lists of rewriters are flattened before being applied.) The form @name allows you to associate a logical name with a rewriter, and then use the name to refer to it. See section 10.3 for a description of the NAME-REWRITER function, which defines such a name.

By default, EKL implicitly includes (USE SIMPINFO) as part of every rewriter. SIMPINFO is the name of a linerange to which you can add lines using the LABEL command; see section 10.3. You can prevent a rewriter from using these lines by including (NUSE SIMPINFO) in it.

The next few sections describe each of the above rewriter forms in turn.

7.2.1. Using lines as rewriters.

The form (USE linerange) tells EKL that all of the lines in linerange are to be applied to the term being rewritten. The lines will be applied in the order given in linerange.

What does it mean for a line to be "applied"? If the line consists of an equality, say A=B, and the term being rewritten is A, then it is changed to B. This also works when the line contains a universally quantified formula such as ∀X Y . A=B, as long as EKL can verify that the quantifiers are being applied correctly. †

If the line is of the form ¬A, and the term being rewritten is A, then it is rewritten to FALSE.

If the line contains a conjunction of formulas, EKL applies the conjuncts one at a time to the term being rewritten.

If the line contains an implication or a conditional (possibly preceded by a universal quantifier), this causes "conditions of application." Suppose the rewriting line is A⊃B. Then EKL will replace B with TRUE in places where A must hold, such as in A∧C⊃P(B, D) or IF A THEN B∨C ELSE Q. This

† EKL won't allow the rewriting of absolute constants like TRUE, FALSE, numerals or quoted expressions or m&a-theoretic expressions. Thus "0=1" cannot be used to replace 0 by 1.

is done by "discharging" the condition associating A with B when it enters the consequent of an implication or the true branch of a conditional cxprcssion.

If the line contains a universally quantified formula such as ∀X Y . α, and the formula being rewritten matches α with instances of the hound variables replaced by valid terms, then the formula is replaced by TRUE. If the line contains a formula such as F=λX . α, EKL treats this in the same way as if it were VX. F(X)=α.

Finally, if the line being applied matches the formula being rewritten (allowing different names for bound variables), the formula is replaced by TRUE.

Any variable occurring free in a line which does not occur free in any of the dependencies of the line is considered unifiable. EKL also tries to use definitions (given by DEFINE or DEFAX, section 6.2) in matching.

In associative situations EKL attempts one-to-one matches on the lists of operands. For example, if "+" is declared associative, then applying X+Y=U to A+X+Y+B yields A+U+B. Similarly, if "+" is right-associative, then the application of X+Y=U to A+X+Y yields A+U.

There may be many ways for a rewriter to change a formula, and not all of them arc desirable. EKL usually *accepts* the new term only in cases where it is "simpler" than the old term, For example, X is a simpler term than F(X), and F (X , F(X)) is simpler than F (X , Y), since it eliminates the symbol Y, replacing it with symbols aready in the term. The term A is simpler than B, because it is lexicographically lower, and A+B is simpler than B+A. The latter reduction could be made by using an axiom for the commutativity of "+" in NIL mode (see below). In this way, the rewriter tries to normalize terms.

For more control over the rewriting, you can say

(USE linerange option . . . option),

. where each option is one of the following.

DIRECTION: REVERSE
DIRECTION: SIMPLER
COND : sexpr
MODE: EXACT
MODE: ALWAYS
UE: ((var . term) . . . (var . term))

Note that the ":" is part of the option name, and that several options may be "strung together" in a single USE rewriter, without parentheses. For example,

(TRW|FOO(A,B)|(USE FO MODE: EXACT COND: QUUX)).

A non-atomic S-expression used as the object of COND: will, of course, be surrounded with its own parentheses. The parentheses used in the object of UE: are to make the entire object be one S-expression.

The DIRECTION: option tells EKL to apply equalities in the reverse of the normal direction, or to apply them in whichever direction will make the formula simpler.

The COND: option allows you to provide conditions of application. The sexpr is evaluated as a LISP S-expression, and the linerange is used only if the result is not NIL.

The MODE: option controls what happens after a line is applied. If the mode is not specified, the application is accepted only if it results in a "simpler" term.

If the mode is EXACT, the application will be accepted whether or not the result is simpler. However, the line will not be applied again to any terms produced by its original application. This is used to prevent infinite loops when the line being applied is something like ∀X.X=F(X).

If the mode is ALWAYS, the line will be applied as many times as possible, without preventing repeated applications as in EXACT mode. This is dangerous, since it may lead to infinite looping. See section 10.8 for how to get out of such a loop.

The UE: option allows rewriting with a modified form of the lines in linerange. UE: asks EKL to apply the formula obtained by applying the UE function (see section 9) using the given variables and terms, and apply the resulting formula to the term being rewritten.

The rewriter (OPEN &rest symbols) is equivalent to (USE linerange), where linerange consists of all of the lines involved in the definition of the symbols in the list symbols. The rewriting is done with MODE: EXACT set; no other options are possible.

Here are examples of some of the options described above.

```
1. (assume |a=b|)
deps: (1)

2. (assume |∀x.p(x)⊃q(x)|)
deps: (2)

3. (assume |p(a)∧p(b)∧¬q(a)|)
deps: (3)

4. (rw 3)
P(A)∧P(B)∧¬Q(A)
deps: (3)
```

Standard rewriting does nothing to the formula on line 3.

```
6. (rw 3 (use 1))
P(A)∧P(B)∧¬Q(A)
deps: (1 3)
```

Applying line 1 does not immediately result in a formula that is any simpler than the original, so EKL decides not to make any change.

```
6. (rw 3 (use 1 mode: exact))
P(B)∧¬Q(B)
deps: (1 3)
```

With MODE: EXACT, we force EKL to use line 1, causing P(A) to bc replaced by P(B). Then standard rewriting, which is always performed at the end, removes the duplicate occurence of P(B).

```
7. (rw 3 (use 2))
FALSE
deps: (2 3)
```

Using line 2 makes the formula contradictory, and the rewriter is able to reduce it to FALSE.

8. (rw 3 (use 1 direction: reverse>)
P(A)∧¬Q(A)
deps: (1 3)

9. (setq foo nil)

9. (rw 3 (use 1 cond: foo mode: exact))
P(A)∧P(B)∧¬Q(A)
deps: (1 3)

10. (setq foo t)

IO. (rw 3 (use 1 cond: foo mode: exact))
P(B)∧¬Q(B)
deps: (1 3)

The assignments to FOO preceding lines 9 and 10 do not become part of the proof. This could lead to some mystification if you see the result without seeing the input. However, in each case EKL preserves the correctness of the formulas; the COND : option only causes it to use or not use certain lines in rewriting.

Here is an example that shows the power of EKL's list types.

1. (decl pare (type: |ground*|))

2. (decl (x y z))

3. (decl distinct (type: |ground*→truthval|))

4. (decl dist (type: |ground⊗ground*→truthval|))

6. (defax dist |∀x y pars.dist(x, ())∧dist(x,y,pars)=(x≠y∧dist(x,pars)) |)
(label distdef)

6. (defax distinct
|∀x pars.distinct()∧distinct(x,pars)=(dist(x,pars)∧distinct(pars))|)
(label distinctdef)

This defines DISTINCT so that it can be applied- to any number of arguments. For example:

7. (trw |distinct(x,y,z)| (use (distdef distinctdef) mode : always))
;DISTINCT(X,Y,Z)≡¬X=Y∧¬X=Z∧¬Y=Z

7.2.2. Rewriting subparts.

Frequently, you will want to apply a rewriter to part of a term, instead of the entire term itself. This can be done with the PART form shown above. The subpart selects a portion of the current term, and the rewriters listed after it are applied to that subpart.

The NUSE form allows you to cancel the use of a linerange. Thus you can allow a linerange to apply everywhere in a formula or subformula, except in selected subparts where you don't want to use it.

1. (assume |a=b|)
deps: (1)

2. (assume |p(a)∧(q(a)∨(r(a)⊃s(a))∨u(a))|>
deps: (2)

Watch where B is substituted for A in the following lines, based on the subpart selected.

3. (rw 2 (part 1 (use 1 mode: exact)))
P(B)∧(Q(A)∨(R(A)⊃S(A))∨U(A))
deps : (1 2)

4. (rw 2 (part 2 (use 1 mode: exact)))
P(A)∧(Q(B)∨(R(B)⊃S(B))∨U(B))
deps : (1 2)

6. (rw 2 (part 2 (part 1 (nuse 1)) (use 1 mode: exact)))
P(A)∧(Q(A)∨(R(B)⊃S(B))∨U(B))
deps: (1 2)

6. (rw 2 (part 2 (part 2 (part 1 (use 1 mode: exact)))))
P(A)∧(Q(A)∨(R(B)⊃S(A))∨U(A))
deps: (1 2)

EKL allows nested levels of PART forms to be written more compactly using **the "#" symbol in** subpart descriptors. The previous example could instead be:

6. (rw 2 (part 2#2#1 (use 1 mode: exact)))
P(A)∧(Q(A)∨(R(B)⊃S(A))∨U(A))
deps: (1 2)

The way in which a part number specifies a **subterm** is as follows. In the general case, the term is a function j applied to terms $t_1, \ldots, t_n$. **Then** PART 0 specifies **the operator j (which might** itself be a complex term, so you might want, to rewrite it); and PART 1 through PART n name the operands. In the case of a bindop applied to a list of variables and a matrix, PAR! 1 gives the matrix. There is no way to rewrite the operator or variables of such **a term.**

(EKL also allows subpart descriptors to be used in combination with names of lines wherever terms are allowed. For instance, (trw |f oo#3#1#2 I) means rewrite the formula that is subpart 1#2 of line 3 of proof FOO. The proofname is required when you use this form, even for lines in the current proof.)

If figuring out the subpart number of a particular term is difficult, you can specify the term itself instead, using the FORM rewriter. The above examples could instead be

3. (rw 2 (form |p(a)| (use 1 mode: exact)))
P(B)∧(Q(A)∨(R(A)⊃S(A))∨U(A))
deps: (1 2)

4. (rw 2 (form |q(a)∨(r(a)⊃s(a)∨u(a))| (use 1 mode: exact)))
P(A)∧(Q(B)∨(R(B)⊃S(B))∨U(B))
deps : (1 2)

5. (rw 2 (form |q(a)∨(r(a)⊃s(a)∨u(a) |

```
                    (form |q(a)| (nuse 1)) (use 1 mode: exact)))
P(A)∧(Q(A)∨(R(B)⊃S(B))∨U(B))
deps: (1 2)

6. (rw 2 (form |r(a)| (use 1 mode: exact>>)
P(A)∧(Q(A)∨(R(B)⊃S(A)))
deps: (1 2)
```

The matching of the term given in FORM to the term being rewritten is "static," done before rewriting starts, and must specify exactly the term to be rewritten.

It is generally easiest to use FORM for short subterms and PART for longer ones that would take more time to type in to EKL.

7.2.3. The decision procedure.

In rewriting, you can use EKL's decision procedure to replace a formula or subformula of a line by TRUE. You call this decision procedure by using the rewriter (DER &rest linerange).

The general problem of deciding the validity of formulas in EKL's language is undecidable, however, so DER is actually a semi-decision procedure. This means that if it returns TRUE, then the formula certainly is valid, but it may fail to prove the truth of some valid formulas. For these, you must provide assistance, either by using other forms of rewriting or by breaking the proof into smaller steps.

Another thing to be aware of is that DER, in order to work on as large a class of formulas as possible, uses a lot of computer time. Often, the other forms of rewriting will prove the same formula valid in much less time. Thus, you should try DER only for simple deductions that the rewriter itself cannot do.

The lines in linerange are used as additional help in the decision procedure. EKL first forms a conditional expression of the form $f_1 \ldots f_n \supset$ term, where $f_1, \ldots, f_n$ are the formulas of the lines in linerange, and term is the term being rewritten. (Thus, saying just (DER) asks EKL to derive the validity of the term as it stands.) Then, EKL tries to prove that the resulting formula is valid. If it succeeds, the original term is replaced by TRUE, and dependencies on the lines in linerange are attached to the final formula.

The rewriter form (NDER &rest linerange) is the same, except that it tries to derive the negation of the term being rewritten, and replaces the term by FALSE if successful.

8. Deriving new lines.

The DERIVE command, first used in the example in section 2, is a way of asking EKL to verify that a given formula follows from others. It is essentially the same as the DER rewriter, but in the form of a top-level EKL command. Its general form is

(DERIVE formula *&optional* linerange &rest rewriter)

The **linerange** specifies which formulas you want, EKL to use in deriving your formula. It is optional, but if you leave it out you are asking EKL to make a proof with no assumptions, i.e., to prove that formula is valid. If linerange is non-empty, then what EKL actually does is construct the formula $f_1 \wedge f_2 \wedge \bullet$ *nAj,1formuIa, where $f_1, f_2, \ldots, f_n$ arc all of the formulas of the **lines in** linerange, and try to prove that this is valid. So you can see that putting unnecessary lines in linerange will cause the program to do more work than it has to, and therefore it may run longer before giving an answer.

Unfortunately, DERIVE will not always succeed. This may be because you tried to derive **a** formula which is not valid, or because the formula is not in the class that DERIVE is able to decide. In that case, you will have to determine a strategy for proving your formula, and guide the proof-checker through a sequence of smaller steps. As with DER, you should try other things before using DERIVE, since it often consumes much of the computer's resources.

The optional **rewriter** parameter specifies what form of rewriting is to be done on the formula before the decision procedure is used.

Here are some examples of DERIVE, using a well-known set of formulas.

1. (axiom |∀x.man(x)⊃mortal(x)|)

2. (assume |man(socrates)|)
deps : (2)

3. (derive |mortal(socrates)| (1 2))
MORTAL(SOCRATES)
deps: (2)

4. (assume |¬mortal(zeus)|)
deps: (4)

6. (derive |¬man(zeus)| (1 *))
¬MAN (ZEUS)
deps: (4)

6. (derive |∃x.¬man(x)| *)
∃X.¬MAN(X)
deps : (4)

Some of these could have been done just as well with the rewriter. Line 3, for example, would work as (trw |mortal(socrates)| (use 1 2)), and this form should be preferred. However, EKL's rewriter† is not powerful enough to do line 5, while DERIVE is.

† At the time of this writing

The formula that DERIVE failed to handle in section 2 turns out to actually be derivable, though it isn't easy. Instead of

6. (derive |∀x y s.lub(x,s)∧lub(y,s)⊃x=y| (1:5))

you should say

6. (derive |∀x y s.lub(x,s)∧lub(y,s)⊃x=y| (1:3) (open lub bound))

Then have patience, because **EKL** took $5\frac{1}{2}$ minutes to derive this formula when testing it for this reference manual!

9. Other functions.

Although rewriting is the most common way of generating new lines, there are situations where you will need other tools. This section describes EKL commands that handle these special cases. Of the commands in this section, the most commonly used is **UE**, which replaces a universally quantified variable in a formula by an expression.

9.1. Elimination and introduction of quantifiers.

The **UE** command is used to eliminate universal quantifiers in formulas. It takes the two forms

(**UE** (var term) ***&optional*** line &rest rewriter)
(UE **pairlist** ***&optional*** line ***&rest*** rewriter)

In the first form, var should be a universally quantified variable in the formula of line. It is replaced by term wherever it appears in the formula, and the quantifier is removed. The formula is then rewritten using rewriter. For this to succeed, though, the sort of term must be the same as the sort of var (see section 3.4). The term is optional; if unspecified, var is left in the formula and the quantifier is simply removed.

Another requirement for **UE** is that term must be substitutable for var; i.e., either the type of term is a subtype of type of var; or else var does not occur in an applied-to position in term and the types of var and term, respectively, are **A→B** and **C→D** with **B** $\succeq$ **D;** and the substituted formula is well-typed. (See section 3.5.)

The second form of the **UE** command allows simultaneous elimination of universally quantified variables; **pairlist** is a list of pairs of the form **((var .** term) **. . . (var . term)).** In both forms, the default for line is the last line in the current proof, and the default for rewriter is **NIL** (causing standard rewriting to be done). The dependencies of the result are those of line, plus any introduced by **SIMPINFO** in rewriting.

```
1. (assume |∀x y.p(x,y)=q(y,x)|)
deps: (1)

2. (ue (x |f(a)|))
∀Y.P(F(A),Y)=Q(Y,F(A))
deps: (1)

3. (ue ((x . |f(a)|) (y . x)) 1)
P(F(A),X)=Q(X,F(A))
deps: (1)
```

Actually, the variables do not always need to be universally quantified in the formula. They may be free variables, provided that they do not occur free in **any** of the dependencies of the line containing the formula. These dependencies include axioms (for which the dependency is not shown by EKL) as well as assumptions or other lines. Another way to express this is that EKL considers the formula implicitly quantified over these variables.

There are no commands to introduce universal quantifiers? or to eliminate or introduce existential quantifiers. If you want to do these, you can use the rewriter, or **DERIVE** or **DEFINE. For**

example, if you have the formula 3X. P (X) on line 12 and you want to introduce a name Y for some particular X for which P(X) is true, then

(define y |p(y)| (use 12))

should work.

9.2. Conditional introduction.

There are two commands used to derive conditional formulas. They are

(CI linerange ***&optional*** **line** ***&rest*** **rewriter)**
(TCI linerange term &rest rewriter)

CI makes an implication whose antecedent (the left side) is the conjunction of the lines in linerange, and whose consequent (the right side) is the formula of line, and then rewrites the result using rewriter.

TCI makes an implication whose antecedent is the conjunction of the lines in linerange, **and** whose consequent is the formula that would result from the command **(TRW** term rewriter) .

Usually, **CI** and **TCI** are used to eliminate dependencies; since lines in linerange are premises in the resulting formula, they are removed from the list of dependencies.

```
1. (axiom |∀x.p(x)⊃q(x)|)

2. (assume |p(a)|)

3. (trw |q(a)| (use 1 2))
Q(A)
deps: (2)

4. (ci (2) 3)
P(A)⊃Q(A)
```

The default for the line parameter in **CI is** the last line in the current proof.

9.3. Proof by cases.

There is a special EKL command to handle "proof by cases" arguments. In a typical proof by cases, you start with a line of the form A1∨A2∨ . . .∨An. Then, in a series of deductions, you show, for some formula $\mathcal{F}$, that assuming A1 derives $\mathcal{F}$, that assuming A2 derives $\mathcal{F}$, . . ., and that assuming An derives $\mathcal{F}$. Now, the command

(CASES line ***&rest*** **linerange)**

ties this all together: line is the line containing the formula A1∨A2∨ . . . ∨An, and linerange is the set of lines whose formula is $\mathcal{F}$. EKL will introduce a new line whose formula is $\mathcal{F}$, and which has the dependencies of line plus the dependencies of lines in linerange, minus those dependencies of lines in linerange that are the formulas Ai. For example,

```
1. (assume |∀x.a(x)|)
deps: (1)

2. (trw |∃y.∀x.a(y)⊃a(x)| (use 1))
∃Y.(∀X.A(Y)⊃A(X))
deps: (1)

3: (assume |∃y.¬a(y)|)
deps: (3)

4. (derive |∃y.∀x.a(y)⊃a(x)| (3))
∃Y.(∀X.A(Y)⊃A(X))
deps: (3)

5. (trw |(∀x.a(x))∨(∃y.¬a(y))|)
(∀X.A(X))∨(∃Y.¬A(Y))

6. (cases 5 2 4)
∃Y.(∀X.A(Y)⊃A(X))
```

The final line in this example has no dependencies.

The order of the lines in linerange may be important for EKL to determine that all of the conditions are satisfied.

10. Utilities.

Often during the construction of a proof, you will want to change something that you have done, either because of a typing error or because you decide to prove something differently. EKL has commands that allow you to examine proofs, and make changes as long as they preserve the validity of the proofs.

10.1. Multiple proofs.

The PROOF command, which WC earlier saw is used to give a name to the proof you are working on, can also be used to create new proofs and switch between proofs. Whenever you say

(PROOF proof name),

the proof named proofname becomes the current proof; if there is no proof named proofname, then one is created.

When printing out line numbers, the proofname is included (followed by the "#" character) only if the line is not in the current proof. There are examples of this in section 5.

10.2. Comments.

Documenting a proof is as important as documenting a program, if you want other people to understand it. EKL allows you to add comments to a proof for this purpose. Each comment is considered to be attached to a line in a proof. This way, when lines move, their comments move with them. The commands that display lines (section 10.5) will print comments that are attached to those lines.

The COM command is used to manipulate comments. It takes the form

(COM line *&optional* number mode),

where line is the line whose comments you want to manipulate. Each line in a proof may have several lines of comments, and number tells EKL which one to use. The default for number is 1. Mode may be DELETE, INSERT,or EDIT (the default).

The DELETE mode tells EKL to delete the comment line given by number. The INSERT mode tells EKL to add comments in front of line number. The EDIT mode prints the line given by number, then deletes it, and then enters the INSERT mode. When inserting, every line of text that you type to EKL becomes a comment, until you type a blank line (two successive <return>s), which ends the command.

After the command is done, the line, together with its comments, is shown on the terminal.

10.3. Naming lines and rewriters.

It is often quite helpful to give symbolic names to lines or groups of lines. In EKL, you can do this with the LABEL command, which takes the form

(LABEL name &rest linerange).

This associates the label name with the lines in linerange (the default being the last line of the current proof). You can use the same name again in a later LABEL command, and this will add to the list of lines with the label name. The labels that each line has are shown as comments whenever that line is displayed.

If you say (LABEL SIMPINFO &rest linerange), then the rewriter will make implicit use of the lines in linerange, as explained in section 7.2.

To remove a label from one or more lines, use the command

(UNLABEL name &rest linerange).

You can use the name of a linerange wherever it is legal to give a line number or linerange. EKL will complain if the linerange contains more than one line, and only one line was called for.

In printing a line number or linerange, EKL tries to use linenames wherever possible. If a named linerange contains lines that are not referred to by the line being printed, they are ignored in deciding whether to use that linerange. Thus, you can keep using the same label for new lines after referring to that label, and it will not affect the way in which previous lines are printed.

You can assign symbolic names to rewriters with the command

(NAME-REWRITER name rewriter),

where rewriter is any valid rewriter (see section 7.2). Saying @name in a rewriting command then specifies the use of this rewriter.

10.4. Context Manipulation.

Usually, symbols are added to the current context when they are declared, and remain part of the context until they are redeclared. If you want to delete certain symbols from the context, you can do this with the command

(DCONTEXT &rest symbols).

This has the effect of removing the symbols mentioned from the context, and also removes any other symbols defined in the same lines as those symbols.

To add symbols that are not part of the context but are already in use, give the command

(CONTEXT &rest linerange).

This adds the context of linerange (the definitions of all symbols used in linerange) to the current context.

10.5. Displaying things.

There is a set of functions for displaying parts of a proof. The simplest of these is

(SHOW &rest linerange),

which prints the lines in the specified range. If linerange is not specified, the entire currently active proof is printed.

The commands

(SHOW-ASSUMPTIONS *&rest* linerange)
(SHOW-AXIOMS &rest linerange)
(SHOW-DECLARATIONS *&rest* linerange)
(SHOW-DEFINITIONS &rest linerange)

are similar, except they show only the assumptions, axioms, etc., in linerange. There are also the following commands:

(SHOW-COMMANDS &rest linerange) shows all of the commands used to generate the lines in linerange.

(SHOW-PROOF-FORM *&optional* line) displays line together with all lines that it depends on. The default for line is the last line of the currently active proof.

(SHOW-CONTEXT *&optional* line) displays the context of line; if line is omitted, it shows the current context of the proof.

(SHOW-TYPEBINDS line) shows the current type bindings associated with line (see section 3.5.1):

(SHOW-LABELS *&optional* line *&rest* labels) displays the lines which have a given label or labels. If neither line nor labels is given, the command shows all labels together with the lines they refer to. If line is given, the lists are relativized to the context of that line. If labels is given, it names the labels whose lines are displayed.

(SHOW-REWRITENAMES &rest rewritenames) shows the rewritenames asked for, or all rewritenames if you give no argument to the function.

(SHOW-PROOFS) prints the list of names of proofs currently in core, and the name of the currently active proof.

(SHOW-STACK) displays the current command stack (see section 10.9).

There are several global variables which give you control over what information EKL prints out. They are the following:

TALK-if T, print lines that are generated; if NIL print lines only thru SHOW-commands. Initialized to T.

DECTALK-if T, print messages about default declarations, else not. Initialized to T.

REWRITEMESSAGES-if T, the rewriter will print a message every time it succeeds in replacing a subexpression with another. Initialized to NIL.

REWRITEFAILMESSAGES-if T, the rewriter will give some information explaining why it failed to verify certain conditions of rewriting. This is useful for debugging. Initialized to NIL.

ERRORMESSAGES-if T, print EKL error messages, else not. Initialized to T.

CAREFUL-if T, prints a warning when DELETEL would cause deletion of extra lines. Initialized to T.

SHOW-CONTEXT-if T, prints the context, if nonempty, whenever printing a line. Initialized to NIL.

SHOW -DEPENDENCIES if T, prints the dependencies, if any, whenever printing a line. Initialized to T.

You can change any of these flags by typing, for example,

(SETQ DECTALK NIL).

If you want this done every time you run EKL, put the commands in an EKL. IN1 file, as described in section 10.7.

10.6. Editing proofs.

To delete a line or lines already in a proof, use the function

(DELETEL *&rest* linerange).

If linerange is omitted, the last step of the current- proof will be dcletcd. When the line or lines being deleted are used in the context or dependencies of another line, EKL will warn you, since it will have to delete these other lines as well. To suppress these warnings (and have EKL automatically do all necessary deletions),

(SETQ CAREFUL NIL).

Setting CAREFUL back to T turns the feature back on. Note that doing a DELETEL may cause lines in the rest of the proof to be renumbered.

The command

(COPYL linerange *&optional* line)

makes a copy of the lines in linerange, and puts it immediately after line. To put a copy at the beginning of a proof, use the form FOO#0, where FOO is the name of the proof, for line.

(TRANSFER linerange *&optional* line)

moves the lines in linerange to the point following line; it is like COPYL followed by DELETEL, but more efficient. The default for line is the end of the currently active proof for both of these functions. To transfer lines to the beginning of a proof, use 0 (or proofname#0) for line.

The (AGAIN) command saves you from completely retyping a line that contains an error. It causes the last command given to EKL's top level to be reloaded into SAIL's line editor. Then you can fix it and try again. (If you are not using the SAIL system, AGAIN just prints out the line for you.)

The CHANGE function allows many forms of editing. Its purpose is to replace an existing line in a proof by a new line, while retaining as much of the subsequent proof as possible. The command

(CHANGE line)

causes the following to happen:

(1) The line referenced by line is loaded into the line editor (unless it is too long, in which case CHANGE cannot be used). If you are not using the SAIL system, the line is just printed on your terminal.

(2) You now have a chance to edit the line, and type the <return> key to enter the revised version. This will temporarily create a new line at the end of the current proof.
(3) The newly created line is used to replace line.
(4) All lines with line as a dependency are re-evaluated, using the new line in place of line. The results are temporarily placed at the end of the proof(s) in which each of these lines resides. Any LABEL commands associated with these lines are also re-executed.
(5) If all goes well, the re-evaluated lines replace the existing lines that have dependencies on line. Otherwise, EKL reports an error, and you may delete all lines depending on line, or you may abort the CHANGE command. CHANGE may be used on any line, including declarations as well as lines that generate proof steps.

More commands: (CANCEL-PROOF *&optional* proof name) deletes one existing proof (defaulting to the current proof). (WIPE-OUT) deletes all the proofs in core. This is like starting EKL over again from the beginning. (RENAME-PROOF oldname newname) renames a proof.

10.7. Input and output.

A collection of EKL proofs can be saved on a disk file, so that you can quit using EKL and resume at some later point. It is also a good idea to save the current state periodically to guard against system crashes.

The internal form of all of the proofs currently in core can be stored with the command

(SAVE-PROOFS filename).

If you give just a name for filename, then the file will be given the extension . PRF, and placed on (or read from) your current area. The full form, allowing you to specify any filename, is

(SAVE-PROOFS name ext area dev)

for a file named dev: <area>name. ext in TOPS-20, and

(SAVE-PROOFS name ext prj prg)

for a file named name. ext [pr j ,prg] at SAIL. If any of the parameters are omitted, the defaults are used. SAVE-PROOFS destroys the previous version of the file, if any, so watch out!

To read in the proof from a file, start up EKL and say

(GET-PROOFS filename).

The filename rules discussed above apply here also. You can also use GET-PROOFS in the middle of an EKL session, but then there is the possibility of trying to read in a proof whose name matches an already-existing proof. In this case, EKL will read in any lines that are new, but leave alone the old lines.

Since the internal form of EKL proofs is not easily readable, there is a function

(PRETTY-PROOF proofname filename),

which formats a proof in the form seen throughout most of this manual. Filename works the same as above, except that the default extension is .PPR. Use this for output once a proof is done to

your satisfaction. If the file already exists, PRETTY-PROOF appends the proof to the text in the file (at SAIL, it creates a new page). The functions

(PRETTY-LABELS filename)
(PRETTY-REWRITENAMES filename)

print out named lineranges and rewriters (section 10.3) on the file specified, appending at the end of the file.

If you write LISP functions that interact with EKL, it is convenient to store them on separate files. The functions QDSKIN, DSKIN, and DSKOUT perform this I/O.

(DSKIN filename)

loads the file specified by filename; the way of naming the file is the same as above except that the default extension is ". LSP" instead of ". PRF". DSKIN prints the names of the functions loaded as they are read in, which is useful in checking for conflicts with the names of EKL functions. QDSKIN is a "quiet,, version of DSKIN that supresses this output.

(DSKOUT function-name-list filename)

writes the functions in the list to the file filename, using GRINDEF. The default extension for filename is ".LSP" for DSKOUT also.

If the same initialization procedure is necessary each time that EKL is started, it can be done automatically by placing commands in a file called EKL . INI. This file will be read and executed by EKL (using QDSKIN) before giving you its first prompt.

10.8. Handling errors.

In addition to errors reported by EKL, you may encounter errors caught by the MACLISP system that supports EKL. Usually, these cause an error message, followed by a return to EKL's top level. In the case of severe errors that do not land you back in EKL, typing control-E will force a return to top level. The control-G key may be used at any time to get to the top level of MACLISP, and subsequently the command (EKL) will return to EKL.

It is also possible to give commands that cause EKL to run very long before responding, or even go into an infinite loop. If you want to abort such a loop, on TOPS-20 versions of MACLISP type control-E to get back to the top level of EKL, and at SAIL first type ESC I to interrupt LISP and then, after it prints "?", type E.

If you are in the middle of a CHANGE command, it is a bit harder to abort. This is because EKL may have created new lines at the end of the proof, which are intended to replace other lines when the CHANGE command completes. The procedure to use is as follows.

- If you have said CHANGE, but have not yet given EKL the changed text of the line, control-E will bring you back to thc EKL top level.
- If EKL has started re-executing lines, and you want it to stop, type control-B. (ESC I followed by B at SAIL.) This puts you in a MACLISP break loop, and you will see the message ; BKPT ^B. Now type (CEXIT) . This is a function that will clean up any proof lines that are temporarily at the end of the current proof. It may ask you questions about whether to keep these lines. When CEXIT is done, you will be back at the top level of EKL.

If a LISP function that you have written (see section. 12) causes an erroneous input to EKL, it may be desirable to recover from the error. This can be done using MACLISP's CATCH and THROW functions. The tag thrown for failures like non-existent lines or failure of DER is DERIVATIONERROR; thus a (*CATCH 'DERIVATIONERROR . . .) will catch such errors.

10.9. The command stack.

Sometimes EKL will reject one of your commands because it uses an undeclared symbol, or is in some way not yet a legal command. So that you do not need to retype the command after fixing your error, you can put it on a command stack with the function (DEFER). Then, when ready to reenter the command, say (RESUME), or say (REDO) to edit it in the line editor first (if you are using the SAIL system). Note that DEFER puts on the stack the last command given to the top level of EKL, even if this is some command such as an editing function.

Any number of commands can be DEFERed, and RESUME will always give you the most recent one, and "pop" the stack. The command (SHOW-STACK) shows the current state of the command stack, and (CLEAR-STACK) completely erases the stack.

11. Meta theory.

Meta theory lets you extend the capabilities of EKL by adding additional rules to the rewriter. These rules are expressed as axioms, and the validity of a proof depends as much on these axioms as on normal axioms expressing statements in the language of EKL.

If term is an EKL term, then the expression tt erm is a name for term. This name is a symbol of type GROUND. And, if term is a term of type GROUND, **↓term** is an EKL term of a new variable type. (Section 3.5.1 discusses variable types. By "new," we mean that EKL arbitrarily chooses a symbol that has not yet been used as the name of the variable type to give to **↓term**.) The rewriter will transform **↓** tterm to term if term has the right type, and has no free variables.

Another way to generate names is with the special operator " ' " (backquote). In input to EKL, this must be followed by a LISP S-expression, and the result is a symbol of type GROUND, which is a name for that S-expression. The "t" and " ' " operators correspond (roughly) to the LISP QUOTE function, and "I" corresponds to LISP's EVAL.

Numbers in your input formulas, as well as the identifiers T and NIL, are automatically transformed into quoted atoms. They cannot be declared. The predeclared predicate NATNUM is TRUE if its argument is a natural number.

One use of meta theory involves certain functions that are said to be attached. These are the LISP functions CAR, **CDR**, CONS, APPEND, MEMBER, EQUAL, ATOM, NULL, LESSP, PLUS, TIMES, **MINUS**, GREATERP, ZEROP, PLUSP, MINUSP, **ADD1**, and **SUB1**, as well as some others named below. To use an attached function, it should first be declared, so that it has the correct type. If a function F is attached, and the rewriter is given a term such as F **(↑X1**, . . . , **↑Xn)**, then it will evaluate the LISP form **(F** Xl . . . **Xn)**, resulting in some expression S. If the result of F, from its declaration, has type GROUND, then the result of rewriting F(**↑X1** , . . . , **↑Xn)** will be **↑S**. If the result of F is of type TRUTHVAL, then the new term will be TRUE if S is non-NIL, and FALSE if S is NIL. An attachment is also made between the EKL symbol "=" and the LISP function EQUAL; no declaration is required to make this work.

Attached functions which do not correspond directly to MACLISP functions are NATNUM, which returns TRUE if its argument is a non-negative integer, SEXP, which returns TRUE whenever its argument is an S-expression, and LISTP, which returns TRUE if its argument is a list, i.e. if the argument is NIL or an S-expression which ends in NIL if you keep applying **CDR** until reaching an atom. The PRED (predecessor) function subtracts one from natural numbers, always returning **a** natural number, so PRED **(0)=0**.

Here are some examples of rewriting using meta theory.

1. **(decl** z (type: ground) **(syntype:** constant))

2. (trw **|↓↑z|)**
↓↑Z≡Z

3. **(decl** plus (type : **|ground⊗ground*→ground|)** (infixname: **+))**

4. (trw **|1+2|)**
1+2=3

5. **(decl** times (type : **|ground⊗ground*→ground|)** (inf ixname : ***))**

6. (trw **|2*4|)**

```
2*4=8

7. (trw |natnum(t)|)
NATNUM(T)≡FALSE

8. (trw |natnum(1)|)
NATNUM(l)

9. (trw |cons(1,2)|)
;CONS is unknown.

;the symbol CONS declared to have type (GROUND⊗GROUND)→GROUND
CONS(1,2)='(1 . 2)
```

When a symbol is unknown, it is given a default declaration even if it is one of the attached functions. Here, we are fortunate in that EKL came up with the correct type for CONS.

```
10. (trw |car '(I . 2)|)
;CAR is unknown.
;the symbol CAR declared to have type GROUND→GROUND
CAR('(1 . 2))=1

11. (trw |append('(1 3),'(2 3))|)
;APPEND is unknown.
;the symbol APPEND declared to have type (GROUND⊗GROUND)→GROUND
APPEND('(1 3),'(2 3))='(1 3 2 3)

12. (trw |equal('a,'a)|>
;EQUAL is unknown.
;the symbol EQUAL declared to have type (GROUND⊗GROUND)→GROUND
EQUAL('A,'A)=T

13. (trw |equal(1,2) I)
EQUAL(1,2)=NIL
```

Since EQUAL is used as a predicate, we need its result to be of type TRUTHVAL. EKL won't do this automatically for the TRW command (it would for the DERIVE command), so we supply a declaration:

```
14. (decl equal (type: |ground⊗ground→truthval|))

15. (trw |equal('a,'a)|)
EQUAL('A, 'A)

16. (trw |equal(1,2)|)
EQUAL(1,2)≡FALSE
```

To extend the rewriter, you can write rules in meta theory that are applied to quoted atoms. For example, suppose you describe a new decision procedure DEC, which is a function whose value is VALID for the formulas that it decides are true. Then, the axiom

```
(AXIOM |∀P.DEC(P)=VALID⊃↓P|)
```

can be used as a way of deciding the validity of EKL formulas.

12. Programming EKL.

This section assumes that you have a previous knowledge of **MACLISP.** Since EKL is implemented in MACLISP, you can define LISP functions that provide input to EKL. These can be either command-level functions that provide a "high-level" or customized interface, translating some other form of input into EKL commands, or lower-level functions that compute the arguments for EKL functions (of the form that evaluate their arguments). Usually a system will have some combination of both of these kinds.

You may want to turn off EKL's normal output of lines as they are generated. The variable **TALK,** if set (or LAMBDA-bound) to T, causes each line to be printed as it is generated. This is the normal setting. If TALK is NIL, lines are only printed using SHOW-commands.

12.1. The evaluating forms of EKL functions.

In section 6, it was mentioned that most of the **EKL** functions have a form in which the arguments to the function are evaluated in the usual LISP manner. These functions find their main use in programs that **create** input to EKL. Occasionally, however, you may find that this form is needed in expressing a command directly to EKL in the most natural way.

The evaluating forms of EKL commands are listed below. Generally the evaluating form of a command has the same name as the standard form of that command, but preceded by an "E".

(EASSUME formula)
(EAXIOM formula)
(ECASES line linerange)
(ECI linerange line rewriter)
(EDEFAX symbol formula)
(EDEFINE symbol formula rewriter)
(EDERIVE formula linerange rewriter)
(ERW line rewriter)
(ETCI linerange formula rewriter)
(ETRW term rewriter)
(EUE pairlist line rewriter)

None of these forms use ***&optional*** or ***&rest*** arguments. If a linerange or rewriter is to be a list, you must explicitly construct that list.

12.2. Internal-form EKL expressions.

When typing commands to EKL, it is most convenient to enter formulas in the notation of logic, and let EKL's parser translate them to internal form. In programming, though, you will often want to construct new terms without using the parser. For example, given two terms α and β, you might want to form their conjunction $\alpha \wedge \beta$, but to do so through the parser would mean constructing a LISP atom with the characters of α, followed by "A", followed by the characters of β.

A set of functions and macros, present in the MACLISP environment of EKL, allows you to construct EKL terms without knowing the details of the internal form. (This also allows the internal form to change in future implementations of EKL.) There are also functions to extract parts of terms in internal form, and to get an external representation, e.g. the name of an EKL symbol. Unlike the other EKL functions, which are executed as commands for their effect, and whose returned values are unimportant, the functions described here do not (usually) have side effects, and do return a value.

The function (PARSE-TERM term) converts **term,** which is a LISP atom representing the external form of a term, into internal form. It may have some side effects, namely making default declarations for symbols found in term. The context and parsing information (such as operator binding power) is whatever is currently active. The function (UNPARSE-TERM term) does the opposite: it converts an internal form into a LISP atom that may then be printed.

As mentioned in section 3, EKL symbols are different from the LISP atoms that name them. (MK-ATOM name) converts a LISP atom into the corresponding EKL symbol. There is no checking to sec whether the symbol has been declared. (ATOM-NAME-OF atom) returns the name of an EKL symbol.

In addition to symbols, you can construct the internal form of a metatheoretic EKL term (see section 11) with the function (MK-META sexp), which corresponds to the "'" operator in the parser, and you can get the LISP S-expression form of a metatheoretic term with the function (META-SEXP-OF metaterm).

Given an EKL internal form, there are several predicates that let you find out what the form is. They are IS-META-EXP, IS-SYMBOL-EXP, IS-ATOMIC-EXP, IS-BOUND-EXP, and IS-APPL-EXP. Each of these functions takes an EKL internal form as an argument, and returns T or NIL. The function IS-TERM-FORM takes any S-expression as an argument, and returns T if that S-expression is a legal EKL internal form.

If you know that a term is of the proper form, you can apply a ***selector*** function to extract a part of it. If expr is the application of a function to a list of arguments, then (OP-OF expr) returns the function (in internal form), and (OPERANDS-OF expr) returns the list of arguments, (SUBEXPS-OF expr) returns a list of all the sub-expressions of expr. If expr is the application **of a** binding operator to a list of variables and a matrix (see section 3.1), then (OP-OF expr) returns the *external* name of the binding operator, (BNDVARS-OF expr) returns a list of the external names of the variables, and (MATRIX-OF expr) returns the internal form of the matrix. For either kind of application, (OP-NAME-OF expr) can be used to get the external form of the operator, if it is an EKL symbol.

And finally, we come to the functions that construct internal forms. (MK-APPL op explist) makes a function application, with OP as the function and explist as the operands. Note that explist is a list; MK-APPL always takes exactly two arguments. When explist has only one member, it is easier to use the function (MK-UNAPPL op exp), and when there are two operands, you can say (MK-BINAPPL op exp1 exp2). To construct an internal form with a binding operator, the function to use is (MK-BOUND-APPL opname varnamelist exp). Note that opname **and** varnamelist are in external form, while exp is in internal form.

12.3. **Predefined functions and macros.**

EKL includes a number of prcdefined functions and macro definitions to simplify the writing of programs.

The following functions return information about any line, given a line designator line. None of these functions prints out its returned value; thus at the top level of EKL, you must say (PRINT (PAST linerange)) to find out the value of (PAST linerange). All of them evaluate their arguments.

(PROOF-WFF line) returns a copy of the formula of line in the currently active proof. If line is not a formula line, returns NIL. Error if no such line exists.

(PROOF-CMD line) returns the command of line.

(PROOF-CONTEXT line) returns a list giving the context of line. The list contains pairs of the form (symbol . **lineno**), where **lineno** is the line number in which symbol was declared.

(PROOF-DEPENDENCIES line) returns the list of line numbers that line depends on.

(PROOF-FREEVARS line) returns a list of all the variables in line that may be universally generalized.

(PROOF-DEFINED line) returns all the defined atoms in line and the lines where they are defined.

(PROOF-MENTIONS line) returns the list of line numbers that are mentioned in line.

(PAST &rest linerange) returns the list of those lines that are logical predecessors of the lines in linerange.

(FUTURE &rest linerange) returns the list of those lines that are logical successors of the lines in linerange.

(CONNECTED &rest linerange) returns the list of all lines that are connected to one of the lines in linerange, i.e., the closure of linerange under PAST and FUTURE.

(WHERE-USED symbol) returns the list of lines in which symbol appears.

(PROOF-LENGTH) returns the length of the current proof.

(PROOFNAMES) returns the list of names of proofs in core.

Another set of functions returns information about symbols:

(GET-TYPE symbol &optional line) returns the type of symbol together with its expansion.

(GET-SORT symbol ***&optional*** line) returns the sort of symbol.

(GET-SYNTYPE symbol ***&optional*** line) returns the syntype of symbol.

(GET-PARSEINFO symbol ***&optional*** line) returns a list used by EKL in parsing and unparsing symbol. This includes the binding power of symbol, if declared, and its infixnames, postfixnames, and unarynames, if any exist.

These are also evaluating forms, and return NIL if symbol is not defined. If line is specified, the information about symbol in the context of line is returned, together with its expansion. Otherwise the information returned is from the currently active context.

(UNPARSE-SEXP sexp)

creates a new S-expression from sexp, in which all terms are replaced by their unparsed forms, and all infinite loops are replaced by the atom #.

13. Axioms for LISP.

This section contains an EKL proof that may be used as a basis for proving facts about LISP programs. First, we declare a set of variables and constants. The type GROUND is used for all LISP objects, and sorts are used to distinguish between S-expressions, lists, and atoms. We use LISTP as the sort for lists, because the name LIST will be used later on as a function that constructs lists.

the proof LISPAX:

```
1. (DECL CAR (UNARYNAME: CAR) (TYPE: |GROUND→GROUND|)
     (SYNTYPE: CONSTANT) (BINDINGPOWER: 960))

2. (DECL CDR (UNARYNAME: CDR) (TYPE: |GROUND→GROUND|)
     (SYNTYPE: CONSTANT) (BINDINGPOWER: 950))

3. (DECL ATOM (UNARYNAME: ATOM) (TYPE: |GROUND→TRUTHVAL|)
     (SYNTYPE: CONSTANT) (BINDINGPOWER: 750))

4. (DECL NULL (UNARYNAME: NULL) (TYPE: |GROUND→TRUTHVAL|)
     (SYNTYPE: CONSTANT)(BINDINGPOWER: 760))

5. (DECL LISTP (UNARYNAME: LISTP) (TYPE: |GROUND→TRUTHVAL|)
     (SYNTYPE: CONSTANT) (BINDINGPOWER: 750))

6. (DECL ALISTP (UNARYNAME: ALISTP) (TYPE: |GROUND→TRUTHVAL|)
     (SYNTYPE: CONSTANT) (BINDINGPOWER: 750))

7. (DECL SEXP (UNARYNAME: SEXP) (TYPE: |GROUND→TRUTHVAL|)
     (SYNTYPE: CONSTANT) (BINDINGPOWER: 760))

8. (DECL (U V W) (TYPE: |GROUND|) (SORT: |LISTP|))

9. (DECL (X Y Z) (TYPE: |GROUND|) (SORT: |SEXP|))

10. (DECL (XA YA ZA) (TYPE: |GROUND|) (SORT: |ATOM|))

11. (DECL (PHI) (TYPE: |GROUND→TRUTHVAL|))

12. (DECL CONS (TYPE: |(GROUND⊗GROUND)→GROUND|) (SYNTYPE: CONSTANT)
      (INFIXNAME: .) (PREFIXNAME: CONS) (BINDINGPOWER: 850))
```

Now we state some very simple axioms describing the relations between sorts and defining the meaning of the basic functions. These lines are labeled SIMPINFO so that the rewriter will USC them automatically in simplifying other formulas.

```
;labels:SIMPINFO
13. (AXIOM |∀XA.SEXP XA|)

;labels:SIMPINFO
14. (AXIOM |∀U.SEXP U|)

;labels:SIMPINFO
```

15. (AXIOM |∀X U.LISTP X.U|)

;labels:SIMPINFO
16. (AXIOM |∀U.¬NULL U⊃LISTP CDR U|)

;labels:SIMPINFO
17. (AXIOM |∀U.¬NULL U⊃SEXP CAR U|)

;labels:SIMPINFO
18. (AXIOM |∀X.¬ATOM X⊃SEXP CAR X|)

;labels:SIMPINFO
19. (AXIOM |∀X.¬ATOM X⊃SEXP CDR X|)

;labels: SIMPINFO
20. (AXIOM |∀X Y.SEXP X.Y|)

;labels:SIMPINFO
21. (AXIOM |∀X Y.-ATOM X.Y|)

;labels:SIMPINFO
22. (AXIOM |∀X U.¬NULL X.U|)

;labels:SIMPINFO
23. (AXIOM |∀U.NULL U⊃U=NIL|)

;labels:SIMPINFO
24. (AXIOM |∀X Y.CAR (X.Y)=X|)

;labels:SIMPINFO
25. (AXIOM |∀X Y.CDR (X.Y)=Y|)

;labels: SIMPINFO CONS,CAR,CDR
26. (AXIOM |∀U.¬NULL U⊃CAR U.CDR U=U|)

;labels: SIMPINFO CONS-CAR,CDR
27. (AXIOM |∀X.¬ATOM X⊃CAR X.CDR X=X|)

Note that we do not need to declare T and NIL, or provide facts about them like NULL(NIL), etc., since EKL already knows about these things as discussed in section 11.

At this point, we will state the axioms of induction for lists and S-expressions.

;labels:LISTINDUCTION
28. (AXIOM |∀PHI.PHI(NIL)∧(∀X U.PHI(U)⊃PHI(X.U))⊃(∀U.PHI(U))|)

29. (DECL PARS (TYPE: |GROUND*|))

30. (DECL (DF DF1 DF2) (TYPE: |(GROUND⊗(GROUND*))→(GROUND*)|))

3 1 . (DECL NILCASE (TYPE: |(GROUND*)→(GROUND*)|))

;labels:LISTINDUCTIONDEF
32. (AXIOM
|∀DF NILCASE DEF.(∃FUN.(∀PARS x U.FUN(NIL,PARS)=NILCASE(PARS)∧

```
                                        FUN(X.U,PARS)=
                                        DEF(X,U,FUN(U,DF(X,PARS)),PARS)))|)

;labels:SEXPINDUCTION
33. (AXIOM |∀PHI.(∀X.ATOM X⊃PHI(X))∧(∀X Y.PHI(X)∧PHI(Y)⊃PHI(X.Y))⊃
                                                                (∀X.PHI(X))|)

;labels: SEXPIMDUCTIONDEF
34. (AXIOM |∀ATOMCASE DEFSEXP DF1 DF2.
            (∃FUN.(∀PARS X Y Z.(ATOM Z⊃FUN(Z,PARS)=ATOMCASE(Z,PARS))∧
                        FUN(X.Y,PARS)=DEFSEXP(X,Y,
                                                FUN(X,DFl(X,Y,PARS)),
                                                FUN(Y,DF2(X,Y,PARS)),
                                                PARS)))|)
```

The axioms LISTINDUCTIONDEF and SEXPINDUCTIONDEF are very general ways of defining functions by induction. They state that if certain well-foundedness properties are satisfied, then there exists a function satisfying a given recursive definition. By supplying the particular instances of DF, NILCASE and DEF (for LISTINDUCTIONDEF), we will get a line of the form

```
∃FUN.(∀PARS X U.FUN(NIL,PARS)= . . . ),
```

and then by using the DEFINE command we can give a name to this function.

For the times when even these general forms are insufficient, we provide the following axiom for high-order definition:

```
35. (DECL (ARB ARB1 ARB2) (TYPE: |?ARBITRARY|))

36. (DECL BIGFUN (TYPE: |(GROUND⊗GROUND⊗(@ARB)⊗(@ARB))→(@ARB)|))

37. (DECL (DEFINED_FUN ATOM_FUN) (TYPE: |GROUND→(@ARB) |))

;labels: HIGH_ORDER_DEFINITION
38. (AXIOM
      |∀BIGFUN ATOM_FUN.(∃DEFINED_FUN.(∀X Y.(ATOM X⊃
                                          DEFINED_FUN(X)=ATOM_FUN(X))∧
                                         DEFINED-FUN(X.Y)=
                                         BIGFUN(X,Y,DEFINED_FUN(X),
                                                DEFINED-FUN(Y))))|)
```

This is the primitive recursive schema for definition on all higher type functionals. Note the USC of the variable type in do&rations; in this way we can specialize to any type.

Next we define the LIST function, which takes a variable number of arguments, and state its basic properties.

```
39. (DECL LIST (TYPE: |(GROUND*)→GROUND|) (SYNTYPE: CONSTANT))

40. (DECL LST (TYPE:|GROUND*|))

;labels: SIMPINFO
41. (AXIOM |LIST(())=NIL|)
```

```
;labels: SIMPINFO .
42. (AXIOM |∀LST.LISTP LIST(LST)|)

;labels:SIMPINFO LISTDEF
43. (AXIOM |∀X LST.LIST(X,LST)=X.LIST(LST)|)
```

The function APPEND is declared associative, but we could prove that fact if we wanted to. Puking it in the declaration is more useful in the long run, because then we won't have to refer to a line stating APPEND's associativity whenever we use it.

```
44. (DECL APPEND (TYPE: |(GROUND⊗GROUND⊗(GROUND*))→GROUND|) .
      (SYNTYPE: CONSTANT) (ASSOCIATIVITY: BOTH) (INFIXNAME: *)
      (BINDINGPOWER: 840))

;labels:SIMPINFO APPENDEF
45. (DEFAX APPEND |∀X U V.NIL*V=V∧X.U*V=X.(U*V)|)

;labels:SIMPINFO LISTAPPEND
46. (AXIOM |∀U V.LISTP U*V|)

;labels:SIMPINFO
47. (AXIOM |∀U.U*NIL=U|)

;labels:SIMPINFO
48. (AXIOM |∀X V.X.NIL*V=X.V|)
```

Here are some more functions that operate on lists, including the basic operations on alists (association lists).

```
49. (DECL (ALLP SOMEP) (SYNTYPE: CONSTANT)
                        (TYPE: |((@PHI)⊗GROUND)→TRUTHVAL|))

;labels:ALLPDEF
50. (DEFAX ALLP
      |∀PHI X U.ALLP(PHI,NIL)∧
                ALLP(PHI,X.U)=(IF PHI(X) THEN ALLP(PHI,U) ELSE FALSE)|)

;labels:SOMEPDEF
51. (DEFAX SOMEP
      |∀PHI X U.¬SOMEP(PHI,NIL)∧
                SOMEP(PHI,X.U)=(IF PHI(X) THEN TRUE ELSE SOMEP(PHI,U))|)

;labels: MAPCARDEF
52. (DEFAX MAPCAR
      |∀FN X U.MAPCAR(FN,NIL)=NIL∧MAPCAR(FN,X.U)=FN(X).MAPCAR(FN,U)|)

53. (DECL (ALIST) (TYPE: |GROUND|) (SORT: |ALISTP|))

;labels:SIMPINFO
54. (AXIOM |∀ALIST.LISTP ALIST|)

55. (AXIOM |∀ALIST.¬NULL ALIST⊃¬ATOM CAR ALIST∧ATOM CAR (CAR ALIST) I)

;labels:MKALIST
```

56. (AXIOM |∀XA Y ALIST.ALISTP (XA.Y).ALIST|)

57. (DECL ASSOC (TYPE: | (GROUND⊗GROUND)→GROUND|) (SYNTYPE: CONSTANT))

;labels:ASSOCDEF
58. (DEFAX ASSOC
|∀X XA Y ALIST.ASSOC(X,NIL)=NIL∧
ASSOC(X,(XA.Y).ALIST)=
(IF X=XA THEN XA.Y ELSE ASSOC(X,ALIST))|)

;labels:SIMPINFO
59. (AXIOM |∀X ALIST.SEXP ASSOC(X,ALIST)|)

60. (DECL MEMBER (TYPE:|(GROUND⊗GROUND)→TRUTHVAL|)(SYNTYPE: CONSTANT))

;labels:MEMBERDEF
61. (DEFAX MEMBER|∀X Y U.¬MEMBER(X,NIL)∧MEMBER(X,Y.U)=(X=Y∨MEMBER(X,U))|)

14. Examples of EKL proofs.

Here we show an example of the use of EKL, with the axioms defined in the previous section.

Our example is a proof of the associativity of APPEND. As was mentioned in the discussion of the LISP axioms, it is more useful in the long run to tell EKL explicitly that APPEND is associative; but just to show that it can be done, we will define a function with all of the other properties **of APPEND,** and then prove that it is associative.

```
Proof? (get-proofs lispax)
;file read in
;switched to LISPAX
;the proof LISPAX read in.

58. (proof append)
;APPEND started.

1. (decl newappend (type: |ground⊗ground*→ground|) (syntype: constant)
      (infixname: **) (bindingpower: 840))
```

NEWAPPEND is given an infixname similar to the one given APPEND. Next **we use DEFINE** to define NEWAPPEND; EKL will check the validity of the definition. (This takes a few seconds of computer time.)

```
2. (define newappend |∀v x u.nil**v=v∧(x.u)**v=x.(u**v)|
      (use listinductiondef))
```

First we prove termination, i.e., that for **all** arguments which are lists, NEWAPPEND returns **a** list. Since the variables U and V have sort LISTP, this is very **easy** to express.

```
3. (ue (phi |λu.∀v.listp (u**v)|) listinduction (open newappend))
W V.LISTP U ** V

4. (label simpinf o)
;Labeled.

4.
```

By adding line 3 to SIMPINFO, it will be used automatically in subsequent rewriting. We now prove associativity, and ask EKL to describe its rewriting in detail.

```
4. (setq rewritemessages t)

4. (ue (phi |λu.((u**v)**w)=(u**(v**w))|) listinduction (open newappend))
;the term NIL ** V is replaced by:
;V
;the term NIL ** (V ** W) is replaced by:
;V ** w
;the term V ** W=V ** W is replaced by:
;TRUE
;the term X.U ** V is replaced by:
;X.(U ** V)
;the term X.(U ** V) ** W is replaced by:
```

```
;X.((U ** V) ** W)
;the term (U ** V) ** W is replaced by:
;U ** (V ** W)
;the term X.U ** (V ** W) is replaced by:
;X.(U ** (V ** W))
;the term X.(U ** (V ** W))=X.(U ** (V ** W)) is replaced by:
;TRUE
;the term (U ** V) ** W=U ** (V ** W)⊃TRUE is replaced by:
;TRUE
;the term ∀X U.TRUE is replaced by:
;TRUE
;the term TRUE∧TRUE is replaced by:
;TRUE
;the term TRUE⊃(∀U.(U ** V) ** W=U ** (V ** W)) is replaced by:
;∀U. (U ** v) ** w=u ** (V ** W)
w. (u ** V) ** w=u ** (V ** W)
```