Distribution Category:
Mathematics and Computers
(UC-32)

ANL-82-84 Rev. 1

ANL--82-84-Rev.1

DE84 012115

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

LOGIC MACHINE ARCHITECTURE INFERENCE MECHANISMS - LAYER 2 USER REFERENCE MANUAL RELEASE 2.0*

Ewing L. Lusk and Ross A. Overbeek

Mathematics and Computer Science Division

April 1984

* This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under contract W-31-109-Eng-38.

LEGIBILITY NOTICE

A major purpose of the Technical Information Center is to provide the broadest dissemination possible of information contained in DOE's Research and Development Reports to business, industry, the academic community, and federal, state and local governments.

Although a small portion of this report is not reproducible, it is being made available to expedite the availability of information on the research discussed herein.

1

CONTENTS

Logic Machine Architecture Inference Mechanisms - Layer 2 User Reference Manual Release 2.0

Ewing L. Lusk
Ross A. Overbeek

ABSTRACT

Logic Machine Architecture (LMA) is a package of software tools for the construction of inference-based systems. This document is the reference manual for layer 2 of LMA. It contains the information necessary to write LMA-based systems at the level of layer 3. Such systems include theorem provers, reasoning components for expert systems, and customized deduction components for a variety of application systems.

1. Introduction

Logic Machine Architecture (LMA) is a layered architecture for the creation of logic "inference engines". The principles underlying its design are presented in [8] and [6]. The individual theoretical notions incorporated in LMA are the results of a long-running research effort in automated deduction and are discussed elsewhere in the literature[7, 9, 12, 13, 14, 15, 16, 21, 22, 25, 26]. This document describes layer 2 —the set of tools that can be used to create uniprocessing theorem provers, reasoning components for expert systems, or customized deduction components suitable for a wide variety of applications. The first major system built using these tools, a theorem prover incorporating all of these ideas, is fully described in [10].

There are a variety of reasons for attempting to form a standardized set of commands:

a) Most researchers are unable to commit the man-years required to develop a powerful program. The effort required to create systems can be reduced dramatically by using standardized tools.

b) With a standardized set of tools, improvements made by one research team can be easily transferred to other cooperating teams.

c) Students can be trained in the use of such tools in much the same way that they are trained in the use of higher-level languages.

d) Hardware or firmware implementations of selected commands become more feasible (because of the larger user community).

In this document we shall go through the basic layer 2 commands, illustrating their use. In addition, a few extra routines that have been coded at Argonne will be described (they are useful utility modules). User comments on either the tools or this document should be directed to Ross Overbeek or Ewing Lusk at Argonne National Laboratory.

This document has changed only minimally since the version for Release 1.0. Routines to perform weighting of clauses, literals, and terms have been added, the formats acceptable to the distributed translators have been slightly extended, the set of system-defined symbols (especially those used for simplification) has been expanded, and an extra parameter has been added to the routines that integrate objects. A number of bugs have been fixed, and performance has been substantially improved.

2. Special LMA Types

Throughout this document, we define the parameters required to invoke our service modules. For each routine we have attempted to document precisely the type and value of each parameter. In many cases one routine will construct a data item to be passed to other routines. Such data items can be of a variety of types. Strictly speaking, users need not know the "meaning" of such types, since they are not expected to (and should not) access any of the values stored in such data items. However, some users have found it convenient to be able to reference at least a minimal description of such LMA layer 2 types. Hence, we are including the following table:

LMA Layer 2 Types	
common2ptr	a pointer to the layer2 common area
csptr	a pointer to a cstring, which is used to hold a string of arbitrary length
ivecptr	a pointer to an ivec, which is used to hold an integer of arbitrary length
objectptr	a pointer to an object, which is used to represent lists, clauses, literals, terms, and attributes
stkntptr	a pointer to a stkntry, which maintains position in a set of clauses generated by an inference rule
upbptr	a pointer to a upb, which is used to maintain an updatable position in a list

3. Strings

The need to process strings of unbounded length occurs frequently. Accordingly, we have provided a set of modules that implement the abstract data type "string". The following routines define the operations that can be performed on strings:

---

alstring - allocate a string

alstring(sptr,retcd,com2)

This routine allocates a string.

sptr	-	a csptr set to reference the allocated string		
retcd	-	an integer return code set as follows:		
		0	-	success
		memfail	-	memory allocation failure
com2	-	a pointer to the layer 2 common area		

---

dealstring - deallocate a string

dealstring(sptr,com2)

This routine can be invoked to deallocate a string.

sptr	-	a csptr
com2	-	a pointer to the layer 2 common area

getstring - get the next character from a string

getstring(s,c)

This routine gets the next character from a string and updates the current position. Just as with putstring, c is passed back as an integer (ord(character-from-string)). If there are no more characters in the string, a 0 is returned.

s	- a csptr to a string
c	- a returned integer with ord(next-character); a 0 is returned at end-of-string

putstring - insert a character into a string

putstring(s,c,retcd,com2)

This routine inserts the character into s at the current position. The variable c is an integer representing the character. This clumsiness is due to an attempt to make all variables passed to layer 2 routines to be integers or pointers. In any event c should be the ord(character-to-be-inserted).

s	- a csptr to a string
c	- an integer containing the ord(character-to-be-inserted)
retcd	- an integer return code set as follows: 0 - success memfail - memory allocation failure
com2	- a pointer to the layer 2 common area

resetstring - reset current position in a string to the start

resetstring(s)

This routine resets the current position in s to the start of s.

s	- a csptr to a string

wreostring - set end-of-string

wreostring(s)

This routine sets the end-of-string. This should be issued
after all putstring calls have been made to build the string.

    s - a csptr to a string

The documentation above is directly from the code. It illustrates the style that we have chosen to comment all of our routines. To see how these routines can be used, let us consider a simple routine to read in a string from a terminal. This routine should read in characters, putting them into one of these indefinitely long strings. To make the routine useful, let us implement the UNIX outlook - the string will be ended with a <cr> (carriage return), unless the carriage return is immediately preceded by the character '\'. When a line is continued, we'll put in a blank between the two lines. This should cause something like

    first line<cr>
    second line.<cr>

to go into the string as "first line second line.". The code for this routine, l2readstr, is as follows:

```
{-------------------

l 2 r e a d s t r

-------------------}

{.

l2readstr - read a string from the terminal

l2readstr(sin,retcd,com2)

This routine reads in a string from the standard input.
The string is terminated by a <cr> that is not preceded
by a '\'.

     sin        - a csptr set to reference the string
     retcd      - an integer return code set as follows:
                       0 - success
                       1 - eof (sin has not been allocated)
```

```
                        memfail - memory allocation failure
    com2       - a pointer to the layer 2 common area

}

procedure l2readstr(var sin: csptr;
                    var retcd: integer;
                    var com2: common2ptr);
var done,bs: boolean;
    c: char;
    outch: integer;

begin
alstring(sin,retcd,com2);
if retcd = 0 then
    begin
    done := false;
    while not done do
        begin
        bs := false;
        while (retcd = 0) and (not eoln) and (not eof) do
            begin
            if bs then
                begin
                outch := ord('\');
                putstring(sin,outch,retcd,com2);
                bs := false;
                end;
            read(c);
            if c = '\' then
                bs := true
            else
                begin
                outch := ord(c);
                putstring(sin,outch,retcd,com2);
                end;
            end;
        if eof then
            begin
            dealstring(sin,com2);
            done := true;
            retcd := 1;
            end
        else
            begin
            readln;
            if c <> '\' then
                done := true
            else
                begin
                outch := ord(' ');
                putstring(sin,outch,retcd,com2);
                end;
            end;
```

```
          if retcd = memfail then
               dealstring(sin,com2);
          end;
     if retcd = 0 then
          wreostring(sin);
     end;
end; {l2readstr}
```

This code illustrates a point that may very well be puzzling at first. Every parameter to a layer 2 routine must be a pointer or an integer. Further, the parameter must be a variable, so expressions or self-defining terms cannot be used in calls. This prevents

```
putstring(sin,ord(' '),retcd,com2)
```

for example. These restrictions are to facilitate interface between different programming languages. Exactly how they make these interfaces easier to develop and maintain is beyond the scope of this discussion.

The corresponding routine to write a string out to a terminal would be as follows:

```
{------------------

l 2 w r i t e s t r

------------------}

{.

l2writestr - write a string to the standard output device

l2writestr(sout)

This routine displays the string on the standard output device.

     sout        - a csptr to a string

}

procedure l2writestr(var sout: csptr);

var chint: integer;

begin
resetstring(sout);
getstring(sout,chint);
while (chint <> 0) do
     begin
     write(chr(chint));
     getstring(sout,chint);
     end;
end; {l2writestr}
```

This routine could, of course, be fixed up to break things nicely into lines, so that long lines would not break arbitrarily, based on the terminal screen size.

At this point we are going to include a short program that just reads in a string from a terminal and echoes it back. The code shows a few points about setting up a program to invoke LMA routines:

```
program echostr(input,output);

const
#include 'l2constants.i';
type
#include 'l2types.h';

#include 'l2externals.h';

var rc: integer;
     cs: csptr;
     com2: common2ptr;

begin
initcom2(com2);
write('enter a string: ');
l2readstr(cs,rc,com2);
writeln('retcd from l2readstr = ',rc:1);
l2writestr(cs);
writeln;
dealstring(cs,com2);
writeln('success');
end.
```

The "includes" that start the program handle declaration of standard constants, types, and external declarations. At some point you might peruse the constants and types files to notice the labels that we've selected, but it probably isn't required at this stage. Notice the

```
initcom2(com2);
```

This is a required command to start things off. Since PASCAL doesn't allow static storage, we need a common area in which to maintain data. The format and contents need not concern you. However, you must initialize the "layer 2 common area" before invoking any layer 2 commands.

Perhaps a comment on overhead is in order. Many users may fear the cost implied by the use of these routines. For most uses the overhead of these routines will be greatly overshadowed by the cost of performing inference operations. In any event lower-level primitives (in layer 0) can be employed to perform block transfers to and from a string in the rare cases where efficiency is a serious issue.

The main use of strings will be to pass formulas in portable format (described below) to and from the LMA routines. This use is covered in detail in a later section. Finally, before leaving the topic of strings, we include the calling conventions for two useful routines (which you could easily code from the primitives above):

compstrings - compare two strings

compstrings(s1,s2,retcd)

This routine compares the contents of the two strings s1 and s2. The comparison proceeds a character at a time, left-to-right. The first two characters that disagree determine the comparison, which is a Pascal character compare. If two such characters are not found, but one string is shorter, then the shorter string is "less than" the other.

s1	-	a csptr to a string		
s2	-	a csptr to a second string		
retcd	-	an integer return code set as follows:		
		0	-	strings are equal
		1	-	s1 < s2
		2	-	s1 > s2

copystring - create a copy of a string

copystring(s1,s2,retcd,com2)

This routine creates a copy of the string referenced by s1 and sets s2 to reference the copy.

s1	-	a csptr to a string		
s2	-	a csptr set to reference the new copy		
retcd	-	an integer return code set as follows:		
		0	-	success
		memfail	-	memory allocation failure
com2	-	a pointer to the layer 2 common area		

4. Vectors of Integers

Vectors of integers are used, among other things, to record the events used to infer a particular formula from one or more other formulas. Sometimes the computation of a new formula requires many single steps (or "events"). Thus, the encoded information on how the result was computed can become arbitrarily long. It is extremely important that the user of layer 2 actually have access to all of the details of such a computation. Much of the information may be discarded, since a user normally does not need to know about all of the events (e.g., simplifying 1+1 to 2). However, the experience reported on expert systems would indicate that a user should be able to extract the details, if desired. This motivates the definition of another abstract data type — "a vector of integers of arbitrary length". The routines that you have available to define and manipulate such vectors are as follows:

alivec - allocate an integer vector

alivec(ivptr,retcd,com2)

This routine allocates an ivector.

ivptr	-	an ivecptr set to reference the allocated ivector		
retcd	-	an integer return code set as follows:		
		0	-	success
		memfail	-	memory allocation failure
com2	-	a pointer to the layer 2 common area		

dealivec - deallocate an integer vector

dealivec(ivptr,com2)

This routine deallocates an ivector.

ivptr	-	an ivecptr
com2	-	a pointer to the layer 2 common area

getivec - get an entry in an integer vector

getivec(v,i,val)

This routine sets val to the value of the ith element in v.
If the ith element was not set to a value by putivec, then
val will be set to -1.

v	-	an ivecptr to an integer vector
i	-	an integer subscript (from 1)
val	-	an integer set to the value of the ith element of v

iveclen - get the length of an integer vector

iveclen(v,len)

This routine sets len to the length of v. The length is actually the max subscript used on a putivec.

v	-	an ivecptr to an integer vector
len	-	an integer set to the length of v

putivec - insert an entry into an integer vector

putivec(v,i,val,retcd,com2)

This routine puts "val" into the ith entry in v.

v	-	an ivecptr to an allocated integer vector
i	-	an integer subscript (from 1)
val	-	an integer
retcd	-	an integer return code set as follows: 0 - success 1 - i is invalid memfail - memory allocation failure
com2	-	a pointer to the layer 2 common area

writeeov - reset the size of a vector to a lower value

writeeov(ivec,maxsub)

This operation is used to reduce the size of a vector. If maxsub is >= the current size, the operation has no effect.

ivec	-	an ivecptr
maxsub	-	the desired new length (an integer)

The use of these routines will be presented in detail after we've looked at inference rules and the "history" vectors that they produce. For now it is enough to realize that they are available and that, as a user of layer 2, you will probably want to develop packages using these routines to decode the inference events into readable English.

5. Portable Format for Clauses, Literals, and Terms

The current version of layer 2 implements the abstract data types of *List*, *Clause*, *Literal* and *Term*. In the future we hope to include more types defined by other users of LMA.

Clauses, literals, and terms have both an internal and a portable representation. They are all embedded in the layer 1 concept of "object"[8], which results in the obvious similarity of their formats.

5.1. Portable Format of a Term

The portable format of a term is defined as follows:

1. A ***label*** is normally any string of characters that does not include one of the following characters:

   " ;:(),&|"

   However, a string enclosed in either single or double quotes is also a legal label. Finally, if the string is enclosed in quotes (of either type), a doubled occurrence of the delimiter can be included in the label. For example,

   'I can''t make it'

   is a valid label.

2. A ***name*** is a label that does not begin with one of the characters

   "stuvwxyz%_"

   The term ***constant*** will occasionally be used in this document. Constants, function symbols, and predicate symbols are represented by names.

3. A ***variable*** is a label that does begin with one of the characters in the above list (i.e., a label that is not a name).

4. A string of the form

   (<name> <arg-1> <arg-2> ...)

   is an ***application***. Here each of the arguments can be a name, a variable, or an application. Thus,

   (F x (G a))

   is an application.

5 A ***term*** can be a name, a variable, or an application. Throughout this manual we call a term represented by an application a ***complex term***. Similarly, a literal that is not a propositional constant will be referred to as a ***complex literal***.

These descriptions of term and literal correspond (more or less) to the normal versions[2,[illegible]]. However, we have actually generalized the notion of application, allowing the first element itself to be a constant, a variable, or an application. This will allow occasional forays into higher-order logic, if desired.

5.2. Portable Format of a Literal

The portable format of a literal is defined as follows:

1. A literal has the same format as a term.
2. A literal of the form

   (NOT <atom>)

   is called a negative literal.
3. A non-negative literal is called a positive literal.

5.3. Portable Format of a Clause

The portable format of a non-unit clause is

(OR <literal-1> (OR <literal-2> ... (OR <literal-m> <literal-n>)));

The portable format of a unit clause is simply the literal followed by a ';'.

Note that our definitions are somewhat loose. That is,

(EQUAL (OR TRUE x) TRUE);

is a perfectly "acceptable" clause. Our attitude is that the portable format of a clause should be produced as the output of a translator from any format that the user desires. It is the responsibility of the user to supply such a translator (although we supply a simple one in the LMA package, and intend to include better translators in the future, they are not part of the layer 2 set of routines). It is the responsibility of the translator to detect errors.

5.4. Converting Between Portable and Internal Formats

We shall now describe the routines to convert between the internal and portable formats. It should be noted that throughout these descriptions, when a portable object is converted into an internal representation, the result is a *non-integrated* object. The difference between an integrated and a non-integrated object will be covered in detail later. For now it will suffice to note that objects that will be kept around more or less permanently are normally integrated into a "structure-sharing" representation. Such integration allows improved algorithms for computing inferred clauses and subsumption checks[14].

The input routines all rename variables to x1, x2, x3.... We sometimes refer to the "number" of a variable. By this we mean i for xi. Thus, the variable number of x4 is 4.

Once a clause, literal, or term in portable format exists in a string, you can obtain the internal representation by using one of the following routines:

clinput - convert a clause from external to object format

clinput(clext,clobj,retcd,com2)

This routine takes a cstring containing a clause in external format (terminated by a ';') and constructs a non-integrated object to represent the clause.

clext	- a csptr to a cstring containing the clause		
clobj	- an objectptr set to reference the generated clause		
retcd	- an integer return code set as follows:		
	0	-	successful conversion
	2	-	error detected in format of clause
	3	-	; was the first character
	memfail	-	memory failure
com2	- a pointer to the layer 2 common area		

litinput - convert a literal from external to object format

litinput(litext,litobj,retcd,com2)

This routine takes a cstring containing a literal in external format (terminated by a ';') and constructs a non-integrated object to represent the literal.

litext	- a csptr to a cstring containing the literal		
litobj	- an objectptr set to reference the generated literal		
retcd	- an integer return code set as follows:		
	0	-	successful conversion
	2	-	error detected in format of literal
	3	-	; was the first character
	memfail	-	memory failure
com2	- a pointer to the layer 2 common area		

trminput - convert a term from external to object format

trminput(trmext,trmobj,retcd,com2)

This routine takes a cstring containing a term in external format (terminated by a ';') and constructs a non-integrated object to represent the term.

trmext	-	a csptr to a cstring containing the term		
trmobj	-	an objectptr set to reference the generated term		
retcd	-	an integer return code set as follows:		
		0	-	successful conversion
		2	-	error detected in format of term
		3	-	; was the first character
		memfail	-	memory failure
com2	-	a pointer to the layer 2 common area		

Note that in all cases the object being converted must include a terminating semicolon. Further, the internal format is always referenced by means of an objectptr. This is because all three abstract data types are embedded in the layer 1 concept of object. The routine clinput will normally be the most widely used of the three routines. The others are included for the rare instances in which the user wishes to input specific literals or terms (e.g., to force favored use of the input objects).

To create the portable representations given the internal representation, the following routines can be used:

cloutput - convert a clause from object to external format

cloutput(clobj,clext,retcd,com2)

This routine converts the clause referenced by "clobj" to an external format in "clext".

clobj	-	an objectptr to a clause
clext	-	a csptr set to reference a cstring containing the clause followed by a semicolon
retcd	-	an integer return code set as follows: 0 - success 1 - clobj does not reference a clause memfail - memory allocation failure
com2	-	a pointer to the layer 2 common area

litoutput - convert a literal from object to external format

litoutput(litobj,litext,retcd,com2)

This routine converts the literal referenced by "litobj" to an external format in "litext".

litobj	-	an objectptr to a literal
litext	-	a csptr set to reference a cstring containing the literal followed by a semicolon
retcd	-	an integer return code set as follows: 0 - success 1 - litobj does not reference a literal memfail - memory allocation failure
com2	-	a pointer to the layer 2 common area

trmoutput - convert a term from object to external format

trmoutput(trmobj,trmext,retcd,com2)

This routine converts the term referenced by "trmobj" to an external format in "trmext".

trmobj	-	an objectptr to a term
trmext	-	a csptr set to reference a cstring containing the term followed by a semicolon
retcd	-	an integer return code set as follows: 0 - success memfail - memory allocation failure
com2	-	a pointer to the layer 2 common area

6. Lists

In this section we shall discuss our implementation of *lists*. Theorem provers built on the layer 2 LMA routines will normally keep lists of clauses. Algorithms that infer new clauses, check subsumption, etc. will require the ability to maintain a position in a list. We implemented the ability to maintain a position in a list that is subject to insertions and deletions. That is, if you are progressing through a list one clause at a time, and the next clause is deleted, then the "access-next" operation should be intelligent enough to return the first clause past the deleted one. If you allow an arbitrary sequence of insertions and deletions between access operations, however, the concept of *position* must be defined rather precisely. When an element of a list is accessed, you may think of the position as fixed on the occurrence of that element in the list (if the element occurs in several lists, you may think of an occurrence as the position of the element in a given list). As long as that occurrence is not deleted before the next access, the concepts of "next element" and "previous element" are straightforward. The difficulties arise when the occurrence is deleted. To analyze this case, consider the following list:

...p3,p2,p1,e*,s1,s2,s3,....

Here the p1,p2,p3... are the "predecessors" of e*, and s1,s2,s3... are the "successors" of e*. Suppose that a position is established on e*, and that e* is deleted. Any arbitrary sequence of insertions and deletions is then performed on the list. An "access-next" operation will now retrieve the first element to the right of the rightmost element that was a predecessor of e* at the point where e* was deleted. Similarly, an "access-previous" will retrieve the first element to the left of the leftmost element that was a predecessor of e* at the point where e* was deleted.

The implementation of this concept of position relies on maintaining ***updatable pointers***[6]. These pointers are used to record the fact that a position has been established on a given element in a list. Whenever any deletions or insertions occur, these updatable pointers are checked and "updated" (if required). A large number of such updatable pointers can significantly degrade performance. Hence, it is desirable to avoid maintaining a large number of positions at any one time. Normally, there will be no problem. However, if you establish a position and then decide that more accesses are not required, you should make sure that the position is "canceled" (which releases the updatable pointers used to maintain the position). When an access operation reaches the end of a list (so that no element is returned), the position is automatically canceled.

Before discussing how to input or output a list of clauses, we must introduce the operations that characterize lists.

lstcreate - create an empty list

lstcreate(newlist,retcd,com2)

This routine sets newlist to reference an empty list.

newlist	- an objectptr set to reference the created empty list		
retcd	- an integer return code set as follows:		
		0	- success
		memfail	- memory failure
com2	- a pointer to the layer2 common area		

lstinsfirst - insert an object at the head of a list

lstinsfirst(object,listobj,retcd,com2)

This routine inserts "object" as the first element in "listobj".

object	- an objectptr		
listobj	- an objectptr referencing a list		
retcd	- an integer return code set as follows:		
		0	- success
		1	- listobj does not reference a list
		memfail	- memory failure
com2	- a pointer to the layer 2 common area		

lstinslast - insert an object at the end of a list

lstinslast(object,listobj,retcd,com2)

This routine inserts "object" as the last element in "listobj".

object	-	an objectptr		
listobj	-	an objectptr referencing a list		
retcd		an integer return code set as follows:		
		0	-	success
		1	-	listobj does not reference a list
		memfail	-	memory failure
com2	-	a pointer to the layer 2 common area		

lstinsbefore - insert an object ahead of a designated position in a list

lstinsbefore(object,listobj,uobjpos,retcd,com2)

This routine assumes that "uobjpos" is an updatable position in listobj (established by one of the traversing routines). "object" is inserted ahead of the position.

object	-	an objectptr to the object to insert		
listobj	-	an objectptr to the list into which the insertion occurs		
uobjpos	-	a upbptr set by a traversing operation		
retcd	-	an integer return code set as follows:		
		0	-	success
		1	-	listobj is not a list
		memfail	-	memory failure
com2	-	a pointer to the layer 2 common area		

lstinsafter - insert an object after a designated position in a list

lstinsafter(object,listobj,uobjpos,retcd,com2)

This routine assumes that "uobjpos" is an updatable position in listobj (established by one of the traversing routines). "object" is inserted immediately after the position.

object	- an objectptr to the object to insert		
listobj	- an objectptr to the list into which the insertion occurs		
uobjpos	- an upbptr set by a traversing operation		
retcd	- an integer return code set as follows:		
		0	- success
		1	- listobj is not a list
		memfail	- memory failure
com2	- a pointer to the layer 2 common area		

lstaccfirst - access the first element in a list

lstaccfirst(listobj,object,uobjpos,retcd,com2)

This routine can be used to set "object" to reference the first element in "listobj". If the operation is successful, "uobjpos" becomes a valid updatable pointer.

listobj	- an objectptr to the list		
object	- an objectptr set to reference the first element in listobj		
uobjpos	- an upbptr set to an updatable position if retcd gets set to 0		
retcd	- an integer return code set as follows:		
		0	- success
		1	- listobj is empty
		2	- listobj does not reference a list
		memfail	- memory failure
com2	- a pointer to the layer 2 common area		

lstacclast - access the last element in a list

lstacclast(listobj,object,uobjpos,retcd,com2)

This routine can be used to set "object" to reference the last element in "listobj". If the operation is successful, "uobjpos" becomes a valid updatable pointer.

listobj	- an objectptr to the list		
object	- an objectptr set to reference the last element in listobj		
uobjpos	- an upbptr set to an updatable position if retcd gets set to 0		
retcd	- an integer return code set as follows:		
		0	- success
		1	- listobj is empty
		2	- listobj does not reference a list
		memfail	- memory failure
com2	- a pointer to the layer 2 common area		

lstaccnext - access the next element in a list

lstaccnext(listobj,object,uobjpos,retcd,com2)

This routine sets "object" to reference the next element in "listobj" past the position represented by "uobjpos". If there are no more elements in the list, the position (uobjpos) will automatically be canceled (check the retcd to see if the end was reached).

listobj	- an objectptr to the list		
object	- an object set to reference the next element		
uobjpos	- an upbptr representing the position in the list		
retcd	- an integer return code set as follows:		
		0	- success
		1	- no more elements in the list (position is automatically canceled)
		2	- listobj is not a list
com2	- a pointer to the layer 2 common area		

lstaccprev - access the previous element in a list

lstaccprev(listobj,object,uobjpos,retcd,com2)

This routine sets "object" to reference the previous element in "listobj" ahead of the position represented by "uobjpos". If there are no previous elements in the list, the position (uobjpos) will automatically be canceled (check the retcd to see if the head was reached).

listobj	-	an objectptr to the list
object	-	an object set to reference the previous element
uobjpos	-	an upbptr representing the position in the list
retcd	-	an integer return code set as follows: 0 - success 1 - no previous elements in the list (position is automatically canceled) 2 - listobj is not a list
com2	-	a pointer to the layer 2 common area

lstcancpos - cancel position in a list

lstcancpos(uobjpos,com2)

This routine can be used to cancel a position in a list.

uobjpos	-	an upbptr established by previous traversing operations
com2	-	a pointer to the layer 2 common area

lstnumel - find the number of elements in the list

lstnumel(listobj,i)

This routine sets i to the number of elements in listobj.

listobj	-	an objectptr to a list
i	-	an integer set to the # of elements in the list

lstdisconnect - disconnect an object from a list

lstdisconnect(uobjpos,com2)

This routine disconnects the object at the position given by uobjpos (the position is "on" the last object returned by a traversal routine that used uobjpos) from the list in which the position occurs.

uobjpos	-	an upbptr set by a previous traversal
com2	-	a pointer to the layer 2 common area

lstaltpos - has the object referenced by a position been disconnected?

lstaltpos(uobjpos,retcd)

This routine sets retcd to 1 if the object that used to be referenced by uobjpos was disconnected from the list. That is, uobjpos was set by a previous traversal operation. If the object that was returned by that call got disconnected, retcd will get set to 1.

- uobjpos - an upbptr set by a previous traversal operation
- retcd - an integer return code set as follows:
  - 0 - the referenced object was not disconnected
  - 1 - the position has been altered (the object was disconnected)

lstloc - locate an object in a list, if it is there

lstloc(listobj,object,uobjpos,retcd,com2)

This routine looks to see if "object" occurs in "listobj". If so, uobjpos is set to the position of the object. Note that this is considered a traversing operation, in that uobjpos must eventually be canceled.

listobj	-	an objectptr to a list		
object	-	an objectptr		
uobjpos	-	an upbptr set to the position of object in listobj		
retcd	-	an integer return code set as follows:		
		0	-	success
		1	-	failure (object doesn't occur in listobj)
		2	-	listobj is not a list
		memfail	-	memory failure
com2	-	a pointer to the layer 2 common area		

lstdelete - delete an empty list

lstdelete(listobj,com2)

This routine deletes listobj, if it is an empty list. If not, no action will take place.

listobj	-	an objectptr to a list
com2	-	a pointer to the layer 2 common area

lstcopy - copy a list

lstcopy(fromlist,tolist,retcd,com2)

This routine copies the list pointed to by fromlist and sets tolist to reference the copy. The actual elements of the list are not copied. The new list references the same subelements as the fromlist.

fromlist	- an objectptr to a list		
tolist	- an objectptr set to reference the copy		
retcd	- an integer return code set as follows:		
	0	-	success
	1	-	fromlist is not a list
	memfail	-	memory failure
com2	- a pointer to the layer 2 common area		

lstobjfloc - find the first list containing a given object

lstobjfloc(obj,listobj,listpos,retcd,com2)

This routine sets listobj to reference the first list that contains a given object.

obj	- an objectptr		
listobj	- an objectptr set to reference the first list that contains obj		
listpos	- an upbptr used to maintain position in the set of lists that contain obj		
retcd	- an integer return code set as follows:		
	0	-	success
	1	-	no list contains obj
	memfail	-	memory allocation failure
com2	- a pointer to the layer 2 common area		

lstobjnloc - locate the next list that contains a given object

lstobjnloc(obj,listobj,listpos,retcd,com2)

This routine locates the next list in the set that contains obj.

obj	-	an objectptr
listobj	-	an objectptr set to reference the next list
listpos	-	an upbptr used to maintain position in the set of containing lists
retcd	-	an integer return code set as follows: 0 - success; 1 - no more lists contain obj
com2	-	a pointer to the layer 2 common area

lstobjcanc - cancel position in the set of containing lists

lstobjcanc(listpos,com2)

This routine cancels position in the set of containing lists. It should not be used if a previous lstobjfloc or lstobjnloc (for the same listpos) returned a nonzero retcd.

listpos	-	an upbptr used to maintain position in the set
com2	-	a pointer to the layer 2 common area

These operations are for the most part exactly what you would expect. Remember, failure to cancel positions can lead to serious degradation.

6.1. Input of a List of Clauses

Now we can present the code to enter both single clauses and entire lists of clauses. It is assumed that lists are terminated by a semicolon. Thus,

```
(EQUAL x x);
(EQUAL (F x (I x)) 0);
;
```

would be a list of two clauses. The routine *cltread* reads from the standard input, but the changes required to read from any file differ only trivially (but frequently depend on the PASCAL compiler that you use).

```
{-----------------

c l t r e a d

------------------}

{.

cltread - read a clause from the terminal

cltread(clobj,retcd,com2)

This routine reads characters from the terminal up through the
next ';'.  Then it tries to convert the resulting string into
a clause in object format.  If all goes well,  clobj is set
to reference the constructed clause.

     clobj      - an objectptr set to reference the generated clause
     retcd      - an integer return code set as follows:
                       0 - success
                       2 - error detected in the format
                       3 - ';' was the first character
                       memfail - memory failure
     com2       - a pointer to the layer 2 common area

}

procedure cltread(var clobj: objectptr;
                  var retcd: integer;
                  var com2: common2ptr);

var s1: csptr;

begin
l2readstr(s1,retcd,com2);
if retcd = 0 then
     begin
     clinput(s1,clobj,retcd,com2);
     if (retcd = ?) then
          writeln('cltread - clinput failed');
     dealstring(s1,com2);
     end
else
     writeln('cltread - l2readstr failed');
end; {cltread}

{-----------------

c l l s t t r e a d

-----------------}

{.
```

cllsttread - read in a list of clauses from the terminal

cllsttread(listobj,retcd,com2)

This routine reads in a list of clauses from the terminal and sets "listobj" to reference the constructed list. The format for a list is a sequence of clause entries, followed by a semicolon. Each clause entry is a clause in external format, followed by a semicolon. If any errors are detected, "listobj" is set to contain the clauses successfully read (nil on a total bust).

```
    listobj   - the objectptr set to reference the constructed list
    retcd      - an integer return code set as follows:
                        0 - success
                        1 - format error detected
                        memfail - memory failure
    com2       - a pointer to the layer 2 common area

}

procedure cllsttread(var listobj: objectptr;
                     var retcd: integer;
                     var com2: common2ptr);

var clobj: objectptr;
    dummyret:integer;

begin
listobj := nil;
retcd := 0;
lstcreate(listobj,retcd,com2);
if retcd = 0 then
    begin
    cltread(clobj,retcd,com2);
    while retcd = 0 do
        begin
        lstinslast(clobj,listobj,retcd,com2);
        if retcd = 0 then
            cltread(clobj,retcd,com2);
        end;
    if retcd = 3 then
        retcd := 0;
    end
else
    writeln('cllsttread - lstcreate failed');
end; {cllsttread}
```

6.2. Output of a List of Clauses

The code to write out a list of clauses is straightforward:

```
{------------------

c l t w r i t e

------------------}

{.

cltwrite - write a clause to the terminal

cltwrite(clobj,retcd,com2)

This routine can be invoked to write the clause referenced by
"clobj" to the terminal.

     clobj      - an objectptr referencing a clause
     retcd       - an integer return code set as follows:
                         0 - success
                         memfail - memory failure
     com2        - a pointer to the layer 2 common area

}

procedure cltwrite(var clobj: objectptr;
                   var retcd: integer;
                   var com2: common2ptr);

var clext: csptr;

begin
cloutput(clobj,clext,retcd,com2);
if retcd = 0 then
     begin
     l2writestr(clext);
     writeln;
     end;
end; {cltwrite}

{------------------

l s t t w r i t e

------------------}

{.

lsttwrite - write a list of clauses to the terminal
```

```
lsttwrite(listobj,retcd,com2)

This routine writes out the clauses from the list "listobj".
After the whole list a semicolon is written.

     listobj   - an objectptr referencing a list
     retcd      - an integer return code set as follows:
                           0 - success
                           memfail - memory allocation failure
     com2       - a pointer to the layer 2 common area

}

procedure lsttwrite(var listobj: objectptr;
                           var retcd: integer;
                           var com2: common2ptr);

var clobj: objectptr;
     pos: upbptr;
     pretcd: integer;

begin
lstaccfirst(listobj,clobj,pos,pretcd,com2);
retcd := 0;
while (retcd = 0) and (pretcd = 0) do
     begin
     cltwrite(clobj,retcd,com2);
     lstaccnext(listobj,clobj,pos,pretcd,com2);
     end;
if retcd = 0 then
     writeln(';');
end; {lsttwrite}
```

The code for *lsttwrite* does not check to make sure every element in the list is actually a clause. The routine *cllsttwrite*, which is included in the layer 2 package, writes out only the clauses in the list (as well as verifying that listobj is actually a list).

7. Input and Output Through a Translator

Before continuing, we should emphasize that our input and output routines are included only as illustrations. Proper input and output will go through a translation package. We include in our package a simple translation package: *ifthentran* is a routine that converts from an "if-then" format (we give a few examples after the routines) to portable format, and *doutcl* converts a clause in internal format into a readable format. For example, the following routines can be used to replace *cltread* and *cltwrite:*

```
{------------------

h c l t r e a d

-------------------}

{.

hcltread - read a clause from the input terminal

hcltread(clobj,retcd,com2)

This routine reads characters from the terminal up through the
next ';'.  Then it tries to convert the resulting string into
a clause in object format  If all goes well,  clobj is set
to reference the constructed clause.  This routine is like cltread
except that the external format of the clause is assumed to be
the "if-then" format.

     clobj      - an objectptr set to reference the generated clause
     retcd      - an integer return code set as follows:
                        0 - success
                        2 - error detected in the format
                        3 - ';' was the first character
                        memfail - memory failure
     com2       - a pointer to the layer 2 common area

}

procedure hcltread(var clobj: objectptr;
                   var retcd: integer;
                   var com2: common2ptr);

var s1,s2: csptr;

begin{hcltread}
l2readstr(s1,retcd,com2);
if retcd = 0 then
     begin
     ifthentran(s1,s2,retcd,com2);
     if retcd <> 0 then
          writeln('nonzero retcd from ifthentran')
     else
          begin
          clinput(s2,clobj,retcd,com2);
          if (retcd = 2) then
               writeln('hcltread - clinput failed');
          dealstring(s2,com2);
          end
     dealstring(s1,com2);
     end
else
     writeln('hcltread - l2readstr failed');
end; {hcltread}
```

```
{------------------

h c l t w r i t e

------------------}

{.

hcltwrite - write a clause to the terminal

hcltwrite(clobj,retcd,com2)

This routine can be invoked to write the clause referenced by
"clobj" to the terminal.  It is like cltwrite except that the
clause is written in disjunctive format (as if it were the
conclusion part of an "if-then" format).

     clobj      - an objectptr referencing a clause
     retcd      - an integer return code set as follows:
                        0 - success
                        memfail - memory failure
     com2       - a pointer to the layer 2 common area

}

procedure hcltwrite(var clobj: objectptr;
                        var retcd: integer;
                        var com2: common2ptr);

var clext: csptr;
     s1: csptr;

begin {hcltwrite}
alstring(clext,retcd,com2);
if retcd = 0 then
     doutcl(clobj,clext,retcd,com2);

if retcd = 0 then
     begin
     l2writestr(clext);
     dealstring(clext,com2);
     end;
end; {hcltwrite}
```

In this case the rather straightforward translator included in our package gets invoked. This will allow the use of clauses like the following:

If x = y & y = z then x = z;

x <= y | y <= x;

If -Mad(John) then Happy(Mary) | Absent(Mary);

if member(x,[John,Joe,Dick]) & (y = [x|z]) then Reject(y);

The printable versions of these clauses produced by *doutcl* would be as follows:

If (x1 = x2) & (x2 = x3) then (x1 = x3);

(x1 <= x2) | (x2 <= x1);

Mad(John) | Happy(Mary) | Absent(Mary);

If member(x,[John,Joe,Dick]) & (y = [x|z]) then Reject(y);

It is only a crude example of what is really needed. We include it to indicate the point at which translators would be coupled into a system based on the layer 2 operations.

8. IDs

Each layer 1 object may be assigned an *id*. An id is an integer that uniquely identifies an object. All lists, clauses, literals, and terms are objects, so they can all be assigned ids. The following routines can be used to assign and reference ids:

l2assignid - assign an id to an object

l2assignid(objptr,retcd,com2)

This routine assigns an id to the object referenced by objptr. If the object already has an id, a new id will not be assigned.

objptr	-	an objectptr
retcd	-	an integer return code set as follows:
		0 - success
		1 - failure (object already has an id)
com2	-	a pointer to the layer 2 common area

l2refid - access the id of an object

l2refid(objptr,id)

This routine returns the id of an object (0 if the object has not been assigned an id).

objptr	-	an objectptr
id	-	an integer set to contain the id of the object

l2idref - access the object with a given id

l2idref(id,objptr,retcd,com2)

This routine sets objptr to reference the object with the given id.

id	-	an integer id
objptr	-	an objectptr set to reference the desired object
retcd	-	an integer return code set as follows: 0 - success 1 - no such object exists
com2	-	a pointer to the layer 2 common area

Note that, while ids do uniquely identify clauses, they are inconvenient in the sense that they are not guaranteed to be consecutive (and usually clause ids are not, bvecause of ids assigned to new literals and terms). This situation can be remedied by implementing a simple mapping from the set of clause ids to consecutive integers (and think of the integers as "clause numbers").

9. Integration

Objects that are not strictly temporary and may participate in inference steps are normally *integrated*. When a term is integrated, it is kept in a "structure shared" data structure. No object occurs more than once in the integrated structure. Rather, there is a single copy that points to each occurrence. This allows one to locate a desired object (e.g., a literal that unifies with a given literal) and then follow pointers to each occurrence of the object (e.g., to all clauses that contain the literal). This structure sharing significantly improves performance on many inference rules, subsumption, and demodulation. When an object is integrated, it will be assigned an id (if it does not already have one).

Note that our use of the term "structure sharing" is quite distinct from other forms, such as that used by Boyer and Moore[1] and by David Warren[20]. The version that we use is described in [14]. It is not important for the user of layer 2 to be familiar with the details of the structure sharing or exactly what is meant by integrating an object. Those details are important only to those implementing more layer 2 routines (such as new abstract data types or inference rules) based on the layer 1 primitives.

10. Basic Clause Processing Primitives

The following operations can be used to manipulate clauses. They are certainly not the complete set, since inference rules, subsumption, etc. are not included. We shall cover those commands later.

clacclit - access a literal in a clause by means of a subscript

clacclit(clobj,litobj,i,retcd,com2)

This routine sets "litobj" to reference the ith literal in "clobj".

clobj	- an objectptr to a clause		
litobj	- an objectptr set to reference the desired literal		
i	- an integer identifying which literal is desired		
retcd	- an integer return code set as follows:		
		0	- success
		1	- invalid subscript
		2	- clobj is not a clause
com2	- a pointer to the layer 2 common area		

clcopy - copy a clause

clcopy(fromcl,tocl,retcd,com2)

This routine can be invoked to copy a clause.

fromcl	- an objectptr to a clause		
tocl	- an objectptr set to reference the copy		
retcd	- an integer return code set as follows:		
		0	- success
		1	- fromcl is not a clause
		memfail	- memory allocation failure
com2	- a pointer to the layer 2 common area		

clcreate - create a null clause

clcreate(clobj,retcd,com2)

This procedure is used to create an empty clause. If no literals are inserted into the clause, it will evaluate to the propositional constant FALSE.

clobj	- an objectptr set to reference the generated clause		
retcd	- an integer return code set as follows:		
	0	-	success
	memfail	-	memory allocation failure
com2	- a pointer to the layer 2 common area		

cldelint - delete an integrated clause

cldelint(clobj,retcd,com2)

This routine removes the clause from any lists that contain it. If nothing else contains it (such as a longer clause), it is deleted itself. If the clause is a substructure in an existing object (such as another clause), it will not be physically deleted; however, it will no longer be considered a clause (and cannot be referenced by clause manipulation routines).

clobj	- an objectptr to a clause		
retcd	- an integer return code set as follows:		
	0	-	success
	1	-	something besides a list contained the clause (this is ok, as well)
	2	-	clobj is not a clause
com2	- a pointer to the layer 2 common area		

cldelnon - delete a nonintegrated clause

cldelnon(clobj,retcd,com2)

This routine can be called to delete the nonintegrated clause referenced by "clobj". In this case a retcd value of 1 indicates a probable error.

clobj	-	an objectptr to a nonintegrated clause		
retcd	-	an integer return code set as follows:		
		0	-	success
		1	-	something besides a list contained the clause
		2	-	clobj does not reference a clause
com2	-	a pointer to the layer 2 common area		

cldisconnect - delete a literal from a nonintegrated clause

cldisconnect(clobj,subscr,retcd,com2)

This procedure removes the indicated literal from the clause. The literal itself is not deallocated.

clobj	-	an objectptr to a clause		
subscr	-	an integer subscript of the literal to disconnect		
retcd	-	an integer return code set as follows:		
		0	-	success
		1	-	invalid subscript
		2	-	clobj does not reference a clause
		memfail	-	memory failure
com2	-	a pointer to the layer 2 common area		

clinslit - insert a literal into a clause

clinslit(clobj,litptr,subscr,retcd,com2)

This procedure inserts the literal given by litptr into the clause designated by clobj.

```
clobj          - an objectptr to a clause
litptr         - an objectptr to the literal to insert
subscr         - an integer subscript giving the subscript
                 value that the inserted literal will have after
                 it is inserted. This variable must be set correctly
                 before invoking clinslit.
retcd          - an integer return code set as follows:
                   0        - success
                   1        - invalid subscript
                   memfail  - memory failure
com2           - a pointer to the layer 2 common area
```

clintegrate - integrate a clause

clintegrate(clobj,unifopt,retcd,com2);

This routine can be used to integrate a clause. The routine will delete the nonintegrated clause, replacing it in every list with a reference to the integrated clause.

```
clobj          - an objectptr to a nonintegrated clause
 unifopt -    option indicating whether unification properties
                   should be set on terms other than literals
                        0     - set unification properties on
                          all terms
                        1     - set unification properties on
                          literals only
retcd          - an integer return code set as follows:
                   0        - success
                   1        - clobj is not a clause
                   2        - clobj was already integrated
                              (this means a bug in the calling
                   program  - do not ignore it!!!)
                   3        - clobj contains a subobject that
                              should be a literal, but is not
                   memfail  - memory allocation failure
com2           - a pointer to the layer 2 common area
```

clnumlit - find the number of literals in a clause

clnumlit(clobj,i,com2)

This routine sets i to the number of literals in clobj (i is set to -1, if clobj is not a clause).

clobj	-	an objectptr to a clause
i	-	an integer set to the number of literals in the clause
com2	-	a pointer to the layer 2 common area

11. Basic Literal and Term Processing Primitives

The primitives for basic manipulation of literals and terms are used far less frequently than those for clauses. They are included to allow a level of control that will seldom be required.

litaccarg - access argument by means of subscript

litaccarg(litobj,i,argobj,retcd,com2)

This routine sets argobj to reference the ith argument of the literal litobj.

litobj	-	an objectptr to a literal
i	-	an integer subscript
argobj	-	an objectptr set to reference the ith argument
retcd	-	an integer return code set as follows: 0 - success 1 - litobj is not a literal 2 - i is invalid
com2	-	a pointer to the layer 2 common area

litaccatom - access the atom of a literal

litaccatom(litobj,atomobj,retcd,com2)

This routine sets atomobj to reference the atom of the literal referenced by litobj.

litobj	-	an objectptr to a literal	
atomobj	-	an objectptr set to reference the atom	
retcd	-	an integer return code set as follows:	
		0	- success
		1	- litobj does not reference a literal
com2	-	a pointer to the layer 2 common area	

litcopy - copy a literal

litcopy(oldlitobj,newlitobj,retcd,com2)

This routine creates a copy of a literal. Attributes and properties are not copied. The oldlitobj may be integrated or nonintegrated. The copy is nonintegrated.

oldlitobj	-	an objectptr to a literal	
newlitobj	-	an objectptr set to reference the generated copy	
retcd	-	an integer return code set as follows:	
		0	- success
		1	- oldlitobj is not a literal
		memfail	- memory allocation failure
com2	-	a pointer to the layer 2 common area	

litercon - create a literal (constant)

litercon(conobj,symbol,retcd,com2)

This routine creates a "constant" node to represent the given symbol.

conobj	- an objectptr set to reference the generated object		
symbol	- the symbol representing the constant		
retcd	- an integer return code set as follows:		
		0	- success
		1	- symbol would represent a variable (starts with s-z, _, or %)
		memfail	- memory allocation failure
com2	- a pointer to the layer 2 common area		

litervar - create a literal (variable)

litervar(varobj,i,retcd,com2)

This routine creates a "variable" xi and sets varobj to reference
the created object.

varobj	- an objectptr set to reference the generated object		
i	- the variable number (an integer > 0)		
retcd	- an integer return code set as follows:		
		0	- success
		1	- i is invalid
		memfail	- memory allocation failure
com2	- a pointer to the layer 2 common area		

litcrcomplex - create an "empty" complex literal

litcrcomplex(comobj,predsymbol,retcd,com2)

This routine creates an empty "complex literal". comobj is set to reference the created object. If the predicate symbol begins with s-z, _, or %, it will become a variable).

comobj	- an objectptr set to reference the generated object		
predsymbol-	- the predicate symbol for the created literal		
retcd	- an integer return code set as follows:		
	0	-	success
	memfail	-	memory allocation failure
com2	- a pointer to the layer 2 common area		

litdelint - delete an integrated literal

litdelint(litobj,retcd,com2)

This routine removes the literal from any lists that contain it. If nothing else contains it (such as a clause), it is itself deleted.

litobj	- an objectptr to a literal		
retcd	- an integer return code set as follows:		
	0	-	success
	1	-	failure, literal is contained in at least one object
	2	-	litobj does not reference a literal
com2	- a pointer to the layer 2 common area		

litdelnon - delete a nonintegrated literal

litdelnon(litobj,retcd,com2)

This routine can be called to delete the nonintegrated literal referenced by "litobj".

litobj	-	an objectptr to a nonintegrated literal		
retcd	-	an integer return code set as follows:		
		0	-	success
		1	-	something besides a list contained the literal
		2	-	litobj does not reference a literal
com2	-	a pointer to the layer 2 common area		

litdisconnect - remove argument (by subscript) from a literal

litdisconnect(litobj,i,retcd,com2)

This routine disconnects the ith argument of the literal. The literal MUST be nonintegrated. The argument is not deleted. Thus, if the user wishes it discarded, the routine trmdelnon should be used after disconnecting it.

litobj	-	an objectptr to a literal		
i	-	the subscript of the argument to disconnect		
retcd	-	an integer return code set as follows:		
		0	-	success
		1	-	litobj is not a literal
		2	-	i is invalid
com2	-	a pointer to the layer 2 common area		

litinsarg - insert an argument (by subscript)

litinsarg(litobj,i,argobj,retcd,com2)

This routine inserts the given argument as the ith argument of the given literal. litobj must reference a nonintegrated complex literal.

litobj	- an objectptr to a literal		
i	- the subscript for the argument to be inserted		
argobj	- an objectptr to the argument to be inserted		
retcd	- an integer return code set as follows:		
		0	- success
		1	- litobj does not reference a complex literal
		2	- i is an invalid subscript
		memfail	- memory allocation failure
com2	- a pointer to the layer 2 common area		

litintegrate - integrate a literal

litintegrate(litobj,unifopt,retcd,com2)

This routine integrates the literal pointed to by litobj (and alters litobj to reference the integrated literal).

litobj	- an objectptr to a literal		
unifopt	- option indicating whether unification properties should be set on terms other than literals		
		0	- set unification properties on all terms
		1	- set unification properties on literals only
retcd	- an integer return code set as follows:		
		0	- success, an integrated version did not previously exist
		1	- success, litobj references a previously existing integrated literal
		2	- the literal was previously integrated
		3	- litobj does not reference a literal
		memfail	- memory allocation failure
com2	- a pointer to the layer 2 common area		

litnumarg - access the number of arguments in a literal

litnumarg(litobj,i,retcd,com2)

This routine sets i to the number of arguments in the literal referenced by litobj.

litobj	-	an objectptr to a literal	
i	-	an integer set to the number of arguments in the literal	
retcd	-	an integer return code set as follows:	
		0	- success
		1	- litobj does not reference a literal
com2	-	a pointer to the layer 2 common area	

litpred - access the predicate for a literal

litpred(litobj,predobj,retcd,com2)

This routine accesses the predicate for a given literal.

litobj	-	an objectptr to a literal	
predobj	-	an objectptr set to reference the predicate of the literal	
retcd	-	an integer return code set as follows:	
		0	- success
		1	- litobj does not referenc a literal
com2	-	a pointer to the layer 2 common area	

litsign - determine the sign of a literal

litsign(litobj,sign,com2)

This routine sets "sign" to reflect the sign of the literal referenced by litobj.

litobj	-	an objectptr to a literal		
sign	-	an integer return code set as follows:		
		0	-	positive
		1	-	negative
		2	-	litobj does not reference a literal
com2	-	a pointer to the layer 2 common area		

trmaccarg - access argument by means of a subscript

trmaccarg(comobj,i,argobj,retcd)

This routine sets argobj to reference the ith argument of the complex term comobj.

comobj	-	an objectptr to a complex term		
i	-	an integer subscript		
argobj	-	an objectptr set to reference the ith argument		
retcd	-	an integer return code set as follows:		
		0	-	success
		1	-	comobj is not a complex term
		2	-	i is invalid

trmacccon - access constant symbol

trmacccon(conobj,symbol,retcd,com2)

This routine sets symbol to reference a cstring containing the symbol represented by the object pointed to by conobj.

conobj	-	an objectptr to a constant		
symbol	-	a csptr set to reference the symbol		
retcd	-	an integer return code set as follows:		
		0	-	success
		1	-	conobj does not reference a constant
		memfail	-	memory allocation failure
com2	-			

trmaccvar - access variable number for a variable

trmaccvar(varobj,i,retcd)

This routine sets i to the variable number of the variable represented by varobj.

varobj	-	an objectptr to a variable		
i	-	an integer set to the variable number		
retcd	-	an integer return code set as follows:		
		0	-	success
		1	-	varobj does not represent a variable

trmcopy - copy a term

trmcopy(oldtrmobj,newtrmobj,retcd,com2)

This routine creates a copy of a term. Attributes and properties are not copied. The oldtrmobj may be integrated or nonintegrated. The copy is nonintegrated.

oldtrmobj	- an objectptr to a term		
newtrmobj	- an objectptr set to reference the generated copy		
retcd	- an integer return code set as follows:		
		0	- success
		memfail	- memory allocation failure
com2	- a pointer to the layer 2 common area		

trmcrcon - create a constant

trmcrcon(conobj,symbol,retcd,com2)

This routine creates a "constant" node to represent the given symbol.

conobj	- an objectptr set to reference the generated object		
symbol	- the symbol representing the constant		
retcd	an integer return code set as follows:		
		0	- success
		1	- an invalid symbol was specified (it began with s-z, _, or %)
		memfail	- memory allocation failure
com2	- a pointer to the layer 2 common area		

trmcrvar - create a variable

trmcrvar(varobj,i,retcd,com2)

This routine creates a "variable" xi and sets varobj to reference the created object.

varobj	-	an objectptr set to reference the generated object		
i	-	the variable number (an integer > 0)		
retcd	-	an integer return code set as follows:		
		0	-	success
		1	-	i is invalid (i < 1)
		memfail	-	memory allocation failure
com2	-	a pointer to the layer 2 common area		

trmcrcomplex - create an "empty" complex term

trmcrcomplex(comobj,funcsymbol,retcd,com2)

This routine creates an empty "complex term". comobj is set to reference the created object. The function symbol should not begin with s-z, _, or % (or it will become a variable). This routine does allow the use of a variable as the function symbol.

comobj	-	an objectptr set to reference the generated object		
funcsymbol-	-	the function symbol for the created term		
retcd	-	an integer return code set as follows:		
		0	-	success
		memfail	-	memory allocation failure
com2	-	a pointer to the layer 2 common area		

trmdelint - delete an integrated term

trmdelint(trmobj,retcd,com2)

This routine removes the term from any lists that contain it. If nothing else contains it (such as a literal), it is itself deleted.

trmobj	-	an objectptr to a term
retcd	-	an integer return code set as follows:
		0 - success
		1 - failure, term is contained in at least one object
com2	-	a pointer to the layer 2 common area

trmdelnon - delete a nonintegrated term

trmdelnon(trmobj,retcd,com2)

This routine can be called to delete the nonintegrated term referenced by "trmobj".

trmobj	-	an objectptr to a nonintegrated term
retcd	-	an integer return code set as follows:
		0 - success
		1 - something besides a list contained the term
com2	-	a pointer to the layer 2 common area

trmdisconnect - remove argument (by subscript) from a complex term

trmdisconnect(comobj,i,retcd,com2)

This routine disconnects the ith argument of the complex term. The complex term MUST be nonintegrated. The argument is not deleted. Thus, if the user wishes it discarded, the routine trmdelnon should be used after disconnecting it.

comobj	- an objectptr to a complex term		
i	- the subscript of the argument to disconnect		
retcd	- an integer return code set as follows:		
	0	-	success
	1	-	comobj is not a complex term
	2	-	i is invalid
com2	- a pointer to the layer 2 common area		

trmfunc - access the function for a complex term

trmfunc(comobj,funcobj,retcd,com2)

This routine accesses the function for a given complex term.

comobj	- an objectptr to a complex term		
funcobj	- an objectptr set to reference the object representing the function		
retcd	- an integer return code set as follows:		
	0	-	success
	1	-	comobj does not reference a complex term
com2	- a pointer to the layer 2 common area		

trminsarg - insert an argument (by subscript) into a complex term

trminsarg(comobj,i,argobj,retcd,com2)

This routine can be used to insert an argument (the ith) into a complex term (referenced by comobj). Neither comobj nor argobj may be integrated.

comobj	- an objectptr to a complex term		
i	- an integer subscript for the argument		
argobj	- an objectptr to the inserted argument		
retcd	- an integer return code set as follows:		
		0	- success
		1	- comobj does not reference a complex term
		2	- i is invalid
		memfail	- memory allocation failure
com2	- a pointer to the layer 2 common area		

trmintegrate - integrate a term

trmintegrate(trmobj,unifopt,retcd,com2)

This routine integrates the term pointed to by trmobj (and alters trmobj to reference the integrated term).

trmobj	- an objectptr to a term
unifopt	- option indicating whether unification properties should be set 0 - set unification properties 1 - do not set unification properties
retcd	- an integer return code set as follows: 0 - success, an integrated version did not previously exist 1 - success, trmobj references a previously existing integrated term 2 - the term was previously integrated memfail - memory allocation failure
com2	- a pointer to the layer 2 common area

trmnumarg - get the number of arguments in a complex term

trmnumarg(comobj,i,retcd)

This routine can be used to find the number of arguments in a complex term.

comobj	-	an objectptr to a complex term		
i	-	an integer set to the number of arguments in the term		
retcd	-	an integer return code set as follows:		
		0	-	success
		1	-	comobj does not reference a complex term

12. Properties

We now come to one of the more interesting features of LMA layers 1 and 2. Objects can be assigned *properties*. A property is an integer. Thus, each object may have a set of associated properties. Similarly, a property may be thought of as referencing a set of objects that have the property. There are two classes of properties, those that the user explicitly associates with objects and those that are automatically assigned (during integration) to objects. Automatically assigned properties are used to facilitate searches for objects that will unify with a specified object. The user properties are in the range 1 to (propfirst-1), where ***propfirst*** is a defined constant (8000000 on most machines, 8000 on machines that support only 16-bit integers). Note that the routines to assign properties do not verify that the given property is in a specific range.

An object to which properties are assigned must have an id. It is not necessary to integrate the object, though. If it is integrated, it will automatically have an id. Otherwise, you can assign an id with *l2assignid*.

The routines to set and delete properties for an object are as follows:

l2setprop - set a property for an object

l2setprop(objptr,prop,retcd,com2)

This routine associates the property (an integer) with the designated object.

objptr	-	an objectptr
prop	-	an integer property
retcd	-	an integer return code set as follows: 0 - success 1 - fail: object has no id 2 - fail: object already has the property memfail - memory allocation failure
com2	-	a pointer to the layer 2 common area

l2delprop - delete a property

l2delprop(objptr,prop,retcd,com2)

This routine deletes a search property.

objptr	-	an objectptr to an object with the property
prop	-	a property (integer)
retcd	-	an integer return code set as follows: 0 - success 1 - failure (object does not have that property)
com2	-	a pointer to the layer 2 common area

To access the set of objects that have one or more properties, you must create a ***property request vector***. This integer vector is simply an encoded Boolean condition in prefix notation. The operators are the following defined constants:

notcond - the complement operator

orcond - the union operator

andcond - the intersection operator

Thus, the vector (andcond,2,orcond,notcond,1,3) represents a request for all objects that

1. have the property 2 and
2. either have the property 3 or do not have the property 1.

The commands for locating the set of objects that satisfy a given property request vector are as follows:

l2locfp - locate the first of a set of objects by property

l2locfp(cond,obj,pos,retcd,com2)

An encoded Boolean condition of properties is in an integer vector (a property request vector). Objects are returned by l2locfp/l2locnp in order of increasing id.

cond	-	an ivecptr to the property request vector	
obj	-	an objectptr set to reference the returned object	
pos	-	an integer that represents a position in the set of objects that satisfy the request	
retcd	-	the return code is set to:	
		0	- ok, an object is returned
		1	- no object could be found
		memfail	- memory failure
com2	-	common area for layer 2	

l2locnp - locate the next object that satisfies a condition

l2locnp(pos,obj,retcd,com2)

pos	-	an integer that maintains position in the set of objects that satisfy the request	
obj	-	an objectptr set to reference the returned object (if retcd gets set to 0)	
retcd	-	the return code is set to:	
		0	- an object is returned
		1	- no more objects could be found
com2	-	common area for layer 2	

l2cancploc - cancel a property location position

l2cancploc(pos,com1)

Cancel position in a search by means of a property request.

pos	-	an integer giving the position
com2	-	the common area for layer 2

These commands reference the elements of the set through first/next operations. A *position* is maintained in the set. The user must either process the entire set or cancel the position.

13. Inference Rules

This section describes the clause-based inference mechanisms that are currently implemented in layer 2. The reader should note that we do intend to frequently supplement this set with both clause-based and non-clause based inference mechanisms.

13.1. Meanings of the Inference Rules

In this section we describe the routines that can be called to deduce new facts from existing ones. We assume that the reader knows what the expressions clause, literal, and term mean[2, 5]. Inference rules are processes for producing new clauses from existing clauses. LMA supports a wide variety of inference rules. A key to effective use of the system is knowing which inference rules to apply in a given situation.

Hyper-resolution

The most straightforward type of logical deduction is the following:

if P then Q
  P

---

therefore Q

In clause form this becomes

-P or Q
 P

---

therefore Q

The new clause, Q, is formed from the clauses (-P or Q) and P by *clashing* the literal -P against the literal P. A more general form of this pattern occurs when there are more hypotheses in the "if-then" statement. A sentence like if P and Q and R, then S

becomes, when rendered into clausal form,

-P or -Q or -R or S

We can deduce S if *all* of P, Q, and R are known to be true. Therefore from the four clauses

-P or -Q or -R or S
P
Q
R

we can deduce S. This is the pattern of deduction used in production systems and many of the systems described as "rule-based" systems.

Of course the literals in the above clauses may contain variables which may require instantiation in order for clashes to occur. For example, the sentence

"All men are mortal"

becomes

"Either x is not a man or x is mortal."

if we also know that

"Socrates is a man"

then we can deduce

"Socrates is mortal."

This is an example of the pattern

-P(x) or Q(x)
P(a)
―――――――

therefore Q(a).

Hyper-resolution is an inference rule that encompasses the above cases and more[14, 17, 19, 21]. It generalizes them in two ways. First, the "if-then" clause may have more than one conclusion literal. The clause

if P and Q then R or S

becomes

-P or -Q or R or S.

Secondly, the clauses containing literals that clash against the hypothesis literals in the "if-then" clause can have more than one literal, as long as all their literals are positive. A typical pattern might be as follows:

-P or -Q or -R or S
P or T
Q or W
R
――――――――――

T or W or S.

Note that hyper-resolution requires that all of the negative literals in the "if-then" clause be clashed against corresponding literals in other clauses. For example, from

-P or -Q or -R

and

P or S

hyper-resolution would *not* deduce

S or -Q or R

(although binary resolution, described below, would do so). When variables are present, their instantiations must be consistent. For example, from

-P(x,y) or -Q(x) or R(x,y)
P(z,b)
Q(a)

hyper-resolution deduces

R(a,b).

Hyper-resolution is perhaps the most commonly used inference rule in situations where equality substitutions do not play a major role. It corresponds to a natural mode of human reasoning. Its restriction that all negative literals must be clashed corresponds to the rule: "Don't draw any conclusions until all of the hypotheses are satisfied."

For a wide class of reasoning problems, hyper-resolution is sufficient. It is the rule that most resembles the inference mechanism used in production systems.

UR-Resolution

It is not hard to see that the use of hyper-resolution by itself will lead to the derivation of clauses with only positive literals in them. While this is sufficient for a large class of problems, a number of reasoning tasks require the derivation of clauses containing negative literals.

Rather than abandon all restrictions on what kinds of clauses are allowed to be derived, we now focus on the desirability of clauses containing only one literal. Such clauses are called *unit clauses* or *units*. A unit clause can be regarded as a statement of fact, whereas multi-literal clauses represent conditional statements (if they contain both positive and negative literals) or statements of alternatives. Unit clauses are therefore more desirable in many situations. *UR-resolution* (Unit-Resulting resolution) [12] removes the restriction that derived clauses must have only positive literals, but imposes the restriction that derived clauses must be units. For example, from

-P or -Q or R
P
-R

UR-resolution would derive -Q, whereas hyper-resolution would be unable to derive anything. UR-resolution emphasizes units in another way as well: all but one of the clauses that participate in the deduction must be unit clauses, although they can be either positive or negative. One might say that UR-resolution emphasizes unit clauses in exactly the same way that hyper-resolution emphasizes positive clauses. With variables present, another example might be

P(x,y) or -Q(a) or R(x,z)
Q(x)
-R(b,c)

P(b,y).

Binary Resolution

Both hyper-resolution and UR-resolution derive much of their power from the fact that many clauses can participate in the clash, which corresponds to taking several reasoning steps at once. Very occasionally it is necessary to employ resolution in very small steps. The form of resolution used in this case is called *binary* resolution; it corresponds to the smallest possible deductive step.

The only "restriction" on binary resolution is that exactly two clauses may participate in the clash[18]. Since both hyper-resolution and UR-resolution can be thought of as sequences of binary resolutions, this is really not a restriction. An example might be

-P or Q or -R
-Q or S

-P or S or -R

Notice that this result could not have been obtained by hyper-resolution (since it is not positive) nor by UR-resolution (since it is not a unit). However, any hyper-resolvent or UR-resolvent can be obtained (eventually) by binary resolution. For example, the hyper-resolution

-P or -Q or R
P or S
Q

R or S

can be carried out by a sequence of binary resolutions:

-P or -Q or R
P or S

-Q or R or S
Q

R or S.

A disadvantage of binary resolution is that clauses are likely to be created which are longer than existing clauses, for example,

-P or -Q or R or S
-S or T or -U or V.

-P or -Q or R or T or -U or V

It is easy to see how unrestricted use of binary resolution can lead to a very large collection of very weak clauses. (A clause having many literals can be thought of as making a weaker statement than one with few literals.)

Unit Resolution

One restriction that is sometimes placed on binary resolution is the requirement that one of the two clauses involved in the clash be a unit[3, 24]. The motivation for this restriction is that if one clause is a unit, then the resulting resolvent will consist of the other participating clause with one of its literals removed (and perhaps some of its variables instantiated). Thus, derived clauses will be shorter than the clauses that produced them, for example,

-P or -Q or R or S
-R

-P or -Q or S

or, with variables present,

-P(x,y) or Q(f(x),b) or -R(x,c)
R(a,z)

-P(a,y) or Q(f(a),b)

These are not the only resolution-based inference rules supported by LMA, but they do represent the ones most often used.

Factoring

There is one inference rule that derives new clauses from a single clause rather than from pairs of clauses. It is called *factoring* and involves the unification of literals within the same clause[2, 5], for example,

P(a,x) or P(y,b)

P(a,b).

The new clause is said to be a *factor* of the original one.

Factoring is important because without it the resolution rules described above are incomplete, which means that given a set of contradictory clauses, a contradiction may not be derived. The classical example is as follows:

P(x) or P(x)
-P(x) or -P(x)

This set of clauses is contradictory, since P(x) is a factor of the first clause and -P(x) is a factor of the second clause. But without factoring, a rule like binary resolution will only derive the tautology P(x) or -P(x).

Paramodulation

The next inference rule we consider is not based on resolution at all. Instead, it is based on the substitution properties of the equality relation. For example, if we know that John's wife is sick, and that John's wife is Sue, then we know that Sue is sick. This is an instance of the pattern

P(a)
Equal(a,b)

P(b).

In this example, the result P(b) is called a ***paramodulant*** rather than a resolvent[4, 15, 16]. The clause P(b) is said to be obtained by paramodulating *into* the clause P(a) ***from*** the equality clause Equal(a,b). The terms in the "from" clause and in the "into" clause are identical in the above example, but in general are required only to be unifiable. Here is an example in which a substitution must be made in the "into" clause:

P(f(x),x)
Equal(f(a),b)

P(b,a)

and here is one in which the substitution must be made in the "from" clause:

P(g(a),b)
Equal(g(x),x)

P(a,b).

Sometimes, substitutions are made in both terms:

P(f(a,x),x)
Equal(f(y,b),y)

P(a,b).

In the previous examples, both the "into" and "from" clauses are units, but this is not a requirement for paramodulation, for example,

P(f(x,g(y))) or Q(x,y)
Equal(f(a,g(b)),c)

P(c) or Q(a,b).

Note that, as usual, when a substitutions is make for a variable, it must be made for all occurrences of the variable in the clause. The "from" clause can also have extra literals:

P(f(a,x))
Equal(f(y,b),c) or Q(y)

P(c) or Q(a).

The expressions *into* and ***from*** can also refer to the terms being matched as well as the clauses in which they occur. In the above example, one would say that paramodulation occurred from the term f(y,b) into the term f(a,x).

The terms paramodulated into or from may even be variables, although this is sometimes considered undesirable. An example of paramodulation into a variable would be

P(f(x),x)
Equal(g(b),h(a))

P(f(h(a)),h(a)),

and an example of paramodulating from a variable would be

P(a)
Equal(x,f(x,x))

P(f(a,a)).

Various kinds of restrictions are sometimes imposed on paramodulation. These include blocking paramodulation into variables or from variables, and restricting the "from" term to be either the lefthand or righthand side of the equality literal. In the previous examples, the lefthand side was always used as the "from" term, but this is not necessary. In the following example, we are paramodulating not from the variable, but rather from the righthand side of the equality:

P(f(a,x)) or Q(x)
Equal(y,f(y,y))

P(a) or Q(a)

Another type of restriction limits the kinds of substitutions that are allowed. For example, one might require that the "into" term be an instance of the "from" term, or that the "from" term be an instance of the "into" term. In *non-complexifying* paramodulation, variables in the "into" term can be replaced only by other variables or constants, unless they occur nowhere else in the into clause.

13.2. Routines that Implement the Inference Rules

For each inference rule there are normally three routines:

1. Each inference rule includes a routine which initiates an operation that can generate one or more new clauses. This first routine returns only the first clause in the set of clauses that could be generated, along with a "position" in the set. This position has the type *stkntptr*.
2. A second routine is passed the position and returns the next clause in the set. This routine can be called repeatedly until all of the clauses in the set are returned.
3. If all of the clauses in the set are not desired, the user can cancel the position at any point. It must be stressed that failure to cancel such positions in sets can lead to severe degradation.

Perhaps the simplest inference rule that is currently implemented in layer 2 is factoring:

ffactor - generate the first factor of the given clause

ffactor(givcl,retcl,history,pos,retcd,com2)

This routine is used to generate the first of a set of factors from the given clause.

givcl	-	an objectptr to a clause
retcl	-	an objectptr set to reference the generated clause
history	-	an ivecptr set to return details on how the factor was produced (nil if no factor is returned)
pos	-	a stkntptr used to maintain position in the set of factors
retcd	-	an integer return code set as follows: 0 - success (retcl references the new factor) 1 - no factor could be produced memfail - memory allocation failure
com2	-	a pointer to the layer 2 common area

nfactor - generate the next factor of a clause

nfactor(pos,retcl,history,retcd,com2)

This routine generates the next factor in the set that can be derived from the given clause.

pos	-	a stkntptr that maintains position in the set
retcl	-	an objectptr set to reference the new factor
history	-	an ivecptr set to reference an ivector giving derivation information
retcd	-	an integer return code set as follows: 0 - success (retcl references the new factor) 1 - no more factors can be generated memfail - memory allocation failure
com2	-	a pointer to the layer 2 common area

cfactor - cancel position in a set of factors

cfactor(pos,com2)

This routine cancels position in the set of factors of a given clause.

pos	-	a stkntptr used to maintain position in the set
com2	-	a pointer to the layer 2 common area

The ffactor and nfactor commands return nonintegrated clauses. The given clause is not altered in any way. For the moment we shall ignore the ***history vector***. We will cover it in detail in the next section.

The other inference rules involve accessing parent clauses other than the given clause. Each such parent (other than the given clause) must be integrated. Frequently, the set of acceptable parents must be restricted (e.g., to implement a set-of-support strategy). To do this, the user forms a list called the ***clashobj***. The clashobj may contain other lists or clauses. Parents other than the given clause must occur in either clashobj or a list that occurs in clashobj. If the given clause is allowed to be used more than once in forming an inference (e.g., in forming a hyper-resolvent), it should also be included in the clashobj. If a nil clashobj is used, any clause in the integrated structure is acceptable.

With these points in mind, the user should now be able to understand the following inference commands:

fbinary - generate the first of a set of binary resolvents

fbinary(givcl,clashobj,retcl,history,pos,retcd,com2)

This routine is used to generate the first of a set of resolvents from the given clause (givcl) and clauses that occur in clashobj. Thus, it is intended that clashobj be a list of the lists from which other clauses are selected to complete the clash.

givcl	- objectptr to the given clause
clashobj	- all clauses (other than the given clause) that make up a clash must be in this object, unless clashobj is nil (in which case any clause can participate in a clash).
retcl	- an objectptr set to reference the first generated resolvent (or nil, if there are none)
history	- an ivecptr that is set to return the details of how the clash was formed (see documentation of history vector formats)
pos	- a stkntptr that must be passed to nbinary to get the rest of the resolvents
retcd	- an integer return code set as follows: 0 - a resolvent was successfully calculated 1 - no resolvents were calculated memfail - memory failure
com2	- a pointer to the layer 2 common area

nbinary - generate the next resolvent

nbinary(pos,retcl,history,retcd,com2)

This routine generates the next in a set of resolvents.

pos	- a stkntptr that maintains position in the set
retcl	- an objectptr set to reference the next resolvent (or nil, if no clause is returned)
history	- an ivecptr that returns the details of how the resolvent was generated
retcd	- an integer return code set as follows: 0 - a resolvent was successfully calculated 1 - no resolvents were calculated memfail - memory failure
com2	- a pointer to the layer 2 common area

cbinary - cancel position in a set of resolvents

cbinary(pos,com2)

This routine must be called to stop generating resolvents before getting a non-zero return code from fbinary or nbinary.

pos	-	a stkntptr used to maintain position in the set
com2	-	a pointer to the layer 2 common area

fp1 - generate the first of a set of p1 resolvents

fp1(givcl,clashobj,retcl,history,pos,retcd,com2)

This routine is used to generate the first of a set of p1 resolvents from the given clause (givcl) and clauses that occur in clashobj. Thus, clashobj is intended to be a list of the lists from which other clauses are selected to complete the clash.

givcl	-	objectptr to the given clause
clashobj	-	all clauses (other than the given clause) that make up a clash must be in this object, unless clashobj is nil (in which case any clause can participate in a clash).
retcl	-	an objectptr set to reference the first generated resolvent (or nil, if there are none)
history	-	an ivecptr that is set to return the details of how the clash was formed (see documentation of history vector formats)
pos	-	a stkntptr that must be passed to np1 to get the rest of the resolvents
retcd	-	an integer return code set as follows: 0 - a resolvent was successfully calculated 1 - no resolvents were calculated memfail - memory failure
com2	-	a pointer to the layer 2 common area

np1 - generate the next p1 resolvent

np1(pos,retcl,history,retcd,com2)

This routine generates the next in a set of p1 resolvents.

pos	-	a stkntptr that maintains position in the set
retcl	-	an objectptr set to reference the next resolvent (or nil, if no clause is returned)
history	-	an ivecptr that returns the details of how the resolvent was generated
retcd	-	an integer return code set as follows: 0 - a resolvent was successfully calculated 1 - no resolvents were calculated memfail - memory failure
com2	-	a pointer to the layer 2 common area

cp1 - cancel position in a set of p1 resolvents

cp1(pos,com2)

This routine must be called to stop generating resolvents before getting a non-zero return code from fp1 or np1.

pos	-	a stkntptr used to maintain position in the set
com2	-	a pointer to the layer 2 common area

funit - generate the first of a set of unit resolvents

funit(givcl,clashobj,retcl,history,pos,retcd,com2)

This routine is used to generate the first of a set of unit resolvents from the given clause (givcl) and clauses that occur in clashobj. Thus, clashobj is intended to be a list of the lists from which other clauses are selected to complete the clash.

givcl	-	objectptr to the given clause
clashobj	-	all clauses (other than the given clause) that make up a clash must be in this object, unless clashobj is nil (in which case any clause can participate in a clash).
retcl	-	an objectptr set to reference the first generated resolvent (or nil, if there are none)
history	-	an ivecptr that is set to return the details of how the clash was formed (see documentation of history vector formats)
pos	-	a stkntptr that must be passed to nunit to get the rest of the resolvents
retcd	-	an integer return code set as follows: 0 - a resolvent was successfully calculated 1 - no resolvents were calculated memfail - memory failure
com2	-	a pointer to the layer 2 common area

nunit - generate the next unit resolvent

nunit(pos,retcl,history,retcd,com2)

This routine generates the next in a set of unit resolvents

pos	-	a stkntptr that maintains position in the set
retcl	-	an objectptr set to reference the next resolvent (or nil, if no clause is returned)
history	-	an ivecptr that returns the details of how the resolvent was generated
retcd	-	an integer return code set as follows: 0 - a resolvent was successfully calculated 1 - no resolvents were calculated memfail - memory failure
com2	-	a pointer to the layer 2 common area

cunit - cancel position in a set of resolvents

cunit(pos,com2)

This routine must be called to stop generating resolvents before getting a non-zero return code from funit or nunit.

pos	-	a stkntptr used to maintain position in the set
com2	-	a pointer to the layer 2 common area

funitconflict - test for unit conflict (first)

funitconflict(givcl,clashobj,retcl,history,pos,retcd,com2)

This routine is used to generate the first of a set of null clauses from the given clause (givcl) and clauses that occur in clashobj. Thus, clashobj is intended to be a list of the lists from which other clauses are selected to complete the clash. The given clause must be a unit, and similarly for the clashed against clause.

givcl	-	objectptr to the given clause
clashobj	-	all clauses (other than the given clause) that make up the clash must be in this object, unless clashobj is nil (in which case any clause can participate in the clash).
retcl	-	an objectptr set to reference the first generated null clause (or nil, if there are none)
history	-	an ivecptr that is set to return the details of how the clash was formed (see documentation of history vector formats)
pos	-	a stkntptr that must be passed to nunitconflict to get the rest of the null clauses
retcd	-	an integer return code set as follows: 0 - a null clause was successfully calculated 1 - no null clauses were calculated memfail - memory failure
com2	-	a pointer to the layer 2 common area

nunitconflict - generate the next null clause (using unit conflict)

nunitconflict(pos,retcl,history,retcd,com2)

This routine generates the next in a set of null clauses that are generated using unit conflict.

pos	-	a stkntptr that maintains position in the set
retcl	-	an objectptr set to reference the next null clause (or nil, if no clause is returned)
history	-	an ivecptr that returns the details of how the null clause was generated
retcd	-	an integer return code set as follows: 0 - a null clause was successfully calculated 1 - no null clauses were calculated memfail - memory failure
com2	-	a pointer to the layer 2 common area

cunitconflict - cancel position in a set of null clauses

cunitconflict(pos,com2)

This routine must be called to stop generating null clauses before getting a non-zero return code from funitconflict or nunitconflict.

pos	-	a stkntptr used to maintain position in the set
com2	-	a pointer to the layer 2 common area

hyperf - generate hyper-resolvents (first)

hyperf(givcl,clashobj,retcl,history,pos,retcd,com2)

This routine is used to generate the first of a set of hyper-resolvents from the given clause (givcl) and clauses that occur in clashobj. Thus, clashobj is intended to be a list of the lists from which other clauses are selected to complete the clash.

givcl	-	objectptr to the given clause
clashobj	-	all clauses (other than the given clause) that make up the clash must be in this object, unless clashobj is nil (in which case any clause can participate in the clash).
retcl	-	an objectptr set to reference the first generated hype-rresolvent (or nil, if there are none)
history	-	an ivecptr that is set to return the details of how the clash was formed (see documentation of history vector formats)
pos	-	a stkntptr that must be passed to hypern to get the rest of the hyperresolvents
retcd	-	an integer return code set as follows: 0 - a hyperresolvent was successfully calculated 1 - no hyperresolvents were calculated memfail - memory failure
com2	-	a pointer to the layer 2 common area

hypern - generate the next hyper-resolvent

hypern(pos,retcl,history,retcd,com2)

This routine generates the next in a set of hyper-resolvents.

pos	-	a stkntptr that maintains position in the set
retcl	-	an objectptr set to reference the next hyper-resolvent (or nil, if no clause is returned)
history	-	an ivecptr that returns the details of how the hyper-resolvent was generated
retcd	-	an integer return code set as follows: 0 - a hyper-resolvent was successfully calculated 1 - no hyper-resolvents were calculated memfail - memory failure
com2	-	a pointer to the layer 2 common area

hypercanc - cancel position in a set of hyper-resolvents

hypercanc(pos,com2)

This routine must be called to stop generating hyper-resolvents before getting a non-zero return code from hyperf or hypern.

pos	-	a stkntptr used to maintain position in the set
com2	-	a pointer to the layer 2 common area

urf - generate UR-resolvents (first)

urf(givcl,clashobj,retcl,history,pos,retcd,com2)

This routine is used to generate the first of a set of UR-resolvents from the given clause (givcl) and clauses that occur in clashobj. Thus, clashobj is intended to be a list of the lists from which other clauses are selected to complete the clash.

givcl	-	objectptr to the given clause
clashobj	-	all clauses (other than the given clause) that make up the clash must be in this object, unless clashobj is nil (in which case any clause can participate in the clash).
retcl	-	an objectptr set to reference the first generated UR-resolvent (or nil, if there are none)
history	-	an ivecptr that is set to return the details of how the clash was formed (see documentation of history vector formats)
pos	-	a stkntptr that must be passed to urn to get the rest of the UR-resolvents
retcd	-	an integer return code set as follows: 0 - a UR-resolvent was successfully calculated 1 - no UR-resolvents were calculated memfail - memory failure
com2	-	a pointer to the layer 2 common area

urn - generate the next UR-resolvent

urn(pos,retcl,history,retcd,com2)

This routine generates the next in a set of UR-resolvents.

pos	-	a stkntptr that maintains position in the set
retcl	-	an objectptr set to reference the next UR-resolvent (or nil, if no clause is returned)
history	-	an ivecptr that returns the details of how the UR-resolvent was generated
retcd	-	an integer return code set as follows:
		0 - a UR-resolvent was successfully calculated
		1 - no UR-resolvents were calculated
		memfail - memory failure
com2	-	a pointer to the layer 2 common area

urcanc - cancel position in a set of UR-resolvents

urcanc(pos,com2)

This routine must be called to stop generating UR-resolvents before getting a non-zero return code from urf or urn.

pos	-	a stkntptr used to maintain position in the set
com2	-	a pointer to the layer 2 common area

funitdel - generate the first of a set of unitdel resolvents

funitdel(givcl,clashobj,retcl,history,pos,retcd,com2)

This routine is used to generate the first of a set of unitdel resolvents from the given clause (givcl) and clauses that occur in clashobj. Thus, clashobj is intended to be a list of the lists from which other clauses are selected to complete the clash.

givcl	-	objectptr to the given clause
clashobj	-	all clauses (other than the given clause) that make up a clash must be in this object, unless clashobj is nil (in which case any clause can participate in a clash).
retcl	-	an objectptr set to reference the first generated resolvent (or nil, if there are none)
history	-	an ivecptr that is set to return the details of how the clash was formed (see documentation of history vector formats)
pos	-	a stkntptr that must be passed to nunitdel to get the rest of the resolvents
retcd	-	an integer return code set as follows: 0 - a resolvent was successfully calculated 1 - no resolvents were calculated memfail - memory failure
com2	-	a pointer to the layer 2 common area

nunitdel - generate the next unitdel resolvent

nunitdel(pos,retcl,history,retcd,com2)

This routine generates the next in a set of unitdel resolvents.

pos	-	a stkntptr that maintains position in the set
retcl	-	an objectptr set to reference the next resolvent (or nil, if no clause is returned)
history	-	an ivecptr that returns the details of how the resolvent was generated
retcd	-	an integer return code set as follows: 0 - a resolvent was successfully calculated 1 - no resolvents were calculated memfail - memory failure
com2	-	a pointer to the layer 2 common area

cunitdel - cancel position in a set of resolvents

cunitdel(pos,com2)

This routine must be called to stop generating resolvents before getting a non-zero return code from funitdel or nunitdel.

pos	-	a stkntptr used to maintain position in the set
com2	-	a pointer to the layer 2 common area

paraff - get the first paramodulant from the given clause

paraff(givcl,retcl,clashobj,instopt,intoopt,fromopt,hist,pos,retcd,com2)

This procedure can be invoked to generate the first of a set of paramodulants using the given clause as the from clause.

givcl	- an objectptr to the given clause		
retcl	- an objectptr set to reference the generated clause		
clashobj	- an objectptr such that all into clauses must be contained in this object, unless it is nil (in which case any into clause is ok).		
instopt	- an integer giving the instantiation options:		
		0	- both into and from can be instantiated
		1	- "into" term must be instance of equality arg
		2	- equality arg must be instance of into term
		3	- noncomplexifying paramodulation (into variables can be instantiated only to constants or variables, unless they occur nowhere else in the "into" clause)
intoopt	- options governing "into" terms:		
		0	- any term is ok
		1	- variables are not ok
		2	- neither variables nor constants are ok
fromopt	- options governing "from" terms		
		0	- either arg of equality, no restr
		1	- only left arg, no restr
		2	- either arg, no var
		3	- only left arg, no var
		4	- either arg, no var or constant
		5	- only left arg, no var or constant
hist	- an ivecptr set to the derivation data		
pos	- a stkntptr used to maintain position in the set		
retcd	- an integer return code:		
		0	- returned clause successfully
		1	- no paramodulant could be generated
		memfail	- memory failure
com2	- a pointer to the layer 2 common area		

paranf - get next paramodulant from the given clause

paranf(pos,retcl,hist,retcd,com2)

This procedure generates the next paramodulant coming from the given clause.

pos	- a stkntptr used to maintain position in the set	
retcl	- an objectptr set to reference the generated clause	
hist	- an ivecptr set to the derivation info	
retcd	- an integer return code:	
	0	- success
	1	- no more paramodulants could be formed
	memfail	- memory failure
com2	- a pointer to the layer 2 common area	

parafcanc - cancel position in a set of "from" paramodulants

parafcanc(pos,com2)

This procedure is used to cancel position in a set of paramodulants.

pos	- the stkntptr used to maintain position in the set
com2	- a pointer to the layer 2 common area

parafi - get the first paramodulant into the given clause

parafi(givcl,retcl,clashobj,instopt,intoopt,fromopt,hist,pos,retcd,com2)

This procedure can be invoked to generate the first of a set of paramodulants.

givcl	-	an objectptr to the given clause		
retcl	-	an objectptr set to reference the generated clause		
clashobj	-	all from clauses must be contained in this object, unless it is nil (in which case any from clause is ok)		
instopt	-	an integer giving the instantiation options:		
		0	-	both "into" and "from" can be instantiated
		1	-	"into" term must be instance of equality arg
		2	-	equality arg must be instance of "into" term
		3	-	noncomplexifying paramodulation ("into" variables can be instantiated only to constants or variables, unless they occur nowhere else in the "into" clause)
intoopt	-	options governing "into" terms:		
		0	-	any term is ok
		1	-	variables are not ok
		2	-	neither variables nor constants are ok
fromopt	-	options governing "from" terms		
		0	-	either arg of equality, no restr
		1	-	only left arg, no restr
		2	-	either arg, no var
		3	-	only left arg, no var
		4	-	either arg, no var or constant
		5	-	only left arg, no var or constant
hist	-	the ivecptr set to the derivation data		
pos	-	a stkntptr used to maintain position in the set		
retcd	-	an integer return code:		
		0	-	returned clause successfully
		1	-	no paramodulant could be generated
		memfail	-	memory failure
com2	-	a pointer to the layer 2 common area		

parani - get next paramodulant into the given clause

parani(pos,retcl,hist,retcd,com2)

This procedure generates the next paramodulant going into the given clause.

pos	-	a stkntptr used to maintain position in the set		
retcl	-	an objectptr set to reference the generated clause		
hist	-	an ivecptr set to the derivation info		
retcd	-	an integer return code:		
		0	-	success
		1	-	no more paramodulants could be formed
		memfail	-	memory failure
com2	-	a pointer to the layer 2 common area		

paraicanc - cancel position in a set of "into" paramodulants

paraicanc(pos,com2)

This procedure is used to cancel position in a set of paramodulants.

pos	-	the stkntptr used to maintain position in the set
com2	-	a pointer to the layer 2 common area

13.3. Inference Rule History Vectors

Each inference rule returns an integer vector (referenced via an ivecptr) that describes the sequence of actions used to infer the returned clause. In this section we give the format of these history vectors. We include the formats produced by simplification and demodulation, operations that are described in later sections. The format of history vectors is as follows:

operations
a sequence of operations

An operation is one of the following:

a) factoring

1 - factor operation code
l1sub - subscript of 1 literal
l2sub - subscript of the second literal

b) resolve

2 - resolution operation code
l1sub - subscript of literal in "main" clause
Here "main" means the given cl or the result to this point of operating on the given clause.
p2id - id of clashed clause
l2sub - subscript of literal in p2id

c) paramodulation into

3 - paramodulation-into operation code
<into-position vector>
p2id - id of from-clause
<from-position vector>

d) paramodulation from

4 - paramodulation-from operation code
<from-position vector>
p2id - id of the into clause
<into-position vector>

e) special symbol reduction

5 - special symbol reduction operation code
<position-vector of the simplified term>

f) tautology reduction

6 - tautology reduction (a clause contains L and -L) operation code.
l1sub
l2sub

g) duplicate literal removal

7 - duplicate literal removal operation code
l1sub
l2sub

h) tautology reduction (a literal is TRUE)

8 - tautology reduction (TRUE literal) operation code
l1sub

9 - FALSE removal (FALSE literal) operation code
l1sub

Here a position vector has the following format:

n - number of elements in the position vector
v1
v2
.
.
.
vn

The user of an inference rule may wish to discard this information, display it, or save it in a "log file". If he decides to save it, we recommend using the portable format of an object. This would lead to the following formats for externally logged inference history:

an axiom (A <id> <object>);
Here <id> is the numeric id, and <object> is the object (which will be a clause for most of our purposes).

an inference - (I <id> <object> <parent1> <alter-sequence>);
Here the <id> is of the generated clause. If this is the same as <parent1>, all future references to the <id> in the log file pertain to the generated clause.
<alter-sequence> is of the following form:

(C <mod1> (C <mod2> ... (C <modn> NIL)))...

Here <modi> is one of the following forms:

into-paramodulant: (INTO <into-pos> <from-id> <from-pos>)

<into-pos> and <from-pos> are position vectors of the form:

(C <num> (C <num> ... (C <num> NIL)))...

from-paramodulant: (FROM <from-pos> <into-id> <into-pos>)

resolvent: (R <lit1-sub> <parent2> <lit2-sub>)

factor: (F <lit1-sub> <lit2-sub>)

special-symbol reduction: (SPEC <sym-position>)

tautology-1: (TAUT1 <lit1-sub> <lit2-sub>)
(clause contains L and -L)

duplicate literal removal:

(DUP <lit1-sub> <lit2-sub>)

tautology-2: (TAUT2 <lit-sub>)
(a literal is TRUE)

FALSE removal: (FALREM <lit-sub>)

To help prepare such a file, we include the following two routines:

logclause - prepare a log entry for a clause (axiom)

logclause(clobj,s,retcd,com2)

This routine creates a log entry for the given clause and points s at the resulting string (it does not write the string to a file).

clobj	-	an objectptr to a clause		
s	-	a csptr set to reference the created string		
retcd	-	an integer return code set as follows:		
		0	-	success
		1	-	clobj does not reference a clause
		memfail	-	memory allocation failure
com2	-	a pointer to the layer 2 common area		

loginference - create a string with a log entry for an inference

loginference(histvec,newcl,parentid,s,retcd,com2)

This routine can be used to create a string with the correct log entry to represent an inference.

histvec	-	an ivecptr to a history vector created by the inference
newcl	-	the newly derived clause
parentid	-	id of the given clause
s	-	a csptr set to reference the generated string
retcd	-	an integer return code set as follows: 0 - success 1 - histvec does not contain a valid history vector memfail - memory allocation failure
com2	-	a pointer to the layer 2 common area

14. Subsumption

14.1. Definition of Subsumption

Subsumption is the mechanism by which unnecessary clauses are discarded [2, 5, 18]. The simplest situation occurs when a clause is derived that is already present in the clause space. In this case we want to discard the newly derived clause.

More generally, the newly derived clause may be recognizably less general than some existing clause without being identical to it. There are two basic ways this can happen.

The first is that the literals of the new clause may form a subset of the literals of the existing clause. For example, if we already know

P or Q

and derive

Q or S or P

then we may discard the new clause, since it is logically weaker than the original one.

The second is that the new clause may be an instance of the original clause, for example,

Old clause: P(a,x)
New clause: P(a,b)

Since any resolution in which the new clause might participate will occur with the old clause anyway, we discard the new clause.

These two ways in which a new clause may be less general than an existing clause may, of course, be combined, for example,

Old clause: P(a,x) or Q(y) or P(y,b)
New clause: Q(a) or P(a,b)

So in general, clause A ***subsumes*** clause B if there is a substitution for the variables in clause A such that after the substitution, the literals of clause B form a subset of the literals of clause A.

This process of discarding new clauses that are subsumed by existing clauses is called ***forward subsumption.*** The subsumption process also can occur in the opposite direction. That is, a newly derived clause may subsume one or more existing clauses, in which case we probably want to keep the new clause and discard the subsumed clauses. This process is called ***backward subsumption.***

14.2. The Routines that Implement Subsumption

Two versions of subsumption checks are supplied:

1. Forward subsumption allows you to determine which clause or clauses subsume a given clause.
2. Backward subsumption allows you to determine which clauses are subsumed by a given clause.

As with the inference rules, clashobj is used to restrict the set of clauses to check. The routines to perform the subsumption checks are as follows:

fsubfirst - get first clause that subsumes given clause

fsubfirst(givcl,clashobj,lenopt,retcl,pos,retcd,com2)

This routine returns the first clause in a set of clauses (all contained in clashobj) that subsume the given clause. The lenopt parameter can be used to suppress checks of a longer clause subsuming a shorter clause. This can save time, and it is needed when doing subsumption checks on factors.

- givcl - an objectptr to the given clause
- clashobj - an objectptr; if nil, all clauses are checked; else, only clauses contained in clashobj are checked
- lenopt - an integer specifying whether or not a longer clause may subsume a shorter clause:
  - 0 - a longer clause may subsume a shorter clause
  - 1 - a longer clause may not subsume a shorter clause
- retcl - an objectptr set to reference the first subsuming clause
- pos - a stkntptr used to maintain position in the set of subsuming clauses
- retcd - an integer return code set as follows:
  - 0 - found a subsumer successfully
  - 1 - no subsumer could be found
  - memfail - memory failure
- com2 - a pointer to the layer 2 common area

fsubnext - get next clause that subsumes the given clause

fsubnext(pos,retcl,retcd,com2)

This routine locates the next clause in the set of clauses that subsume the given clause.

- pos - a stkntptr used to maintain position in the set
- retcl - an objectptr set to reference the returned clause
- retcd - an integer return code set as follows:
  - 0 - success
  - 1 - no clause could be found
  - memfail - memory failure
- com2 - pointer to the layer 2 common area

cancfsub - cancel position in set of clauses that subsume given clause

cancfsub(pos,com2)

This routine cancels the position in the set of clauses that subsume the given clause.

pos	-	a stkntptr maintaining position in the set
com2	-	a pointer to the layer 2 common area

bsubfirst - get first clause subsumed by given clause

bsubfirst(givcl,clashobj,lenopt,retcl,pos,retcd,com2)

This routine returns the first clause in a set of clauses (all contained in clashobj) that are subsumed by the given clause. The lenopt parameter can be used to suppress checks of a longer clause subsuming a shorter clause. This can save time, and it is needed when doing subsumption checks on factors.

givcl	-	an objectptr to the given clause
clashobj	-	an objectptr; if nil, all clauses are checked; else, only clauses contained in clashobj are checked
lenopt	-	an integer specifying whether or not a longer clause may subsume a shorter clause: 0 - a longer clause may subsume a shorter clause 1 - a longer clause may not subsume a shorter clause
retcl	-	an objectptr set to reference the first subsumed clause
pos	-	a stkntptr used to maintain position in the set of subsumed clauses
retcd	-	an integer return code set as follows: 0 - found a subsumed clause successfully 1 - no clause could be found memfail - memory failure
com2	-	a pointer to the layer 2 common area

bsubnext - get next clause subsumed by the given clause

bsubnext(pos,retcl,retcd,com2)

This routine locates the next clause in the set of clauses subsumed by the given clause.

pos	- a stkntptr used to maintain position in the set		
retcl	- an objectptr set to reference the returned clause		
retcd	- an integer return code set as follows:		
	0	-	success
	1	-	no clause could be found
	memfail	-	memory failure
com2	- pointer to the layer 2 common area		

cancbsub - cancel position in set of clauses subsumed by given clause

cancbsub(pos,com2)

This routine cancels the position in the set of clauses subsumed by the given clause.

pos	- a stkntptr maintaining position in the set
com2	- a pointer to the layer 2 common area

15. A Simple Theorem Prover

Now we have all the tools required to put together a simple theorem prover. This program uses just hyper-resolution as the inference rule, but does perform complete subsumption checks. It is a "toy" program, but it is still fairly powerful. In fact, it is better than many of the programs that have been reported in the literature. A more extensive theorem prover built with the LMA tools is described in [10].

```
program tp(input,output);

const
#include 'l2constants.i';
type
#include 'l2types.h';

#include 'l2externals.h';
```

```
var
     axlist: objectptr;          {the list of axioms}
     soslist: objectptr;         {the set of support list}
     hbglist: objectptr;         {the have-been-given list}
     clashlists: objectptr;      {list of lists to clash against}
     allclauses: objectptr;      {list of list to subsume from}
     givencl: objectptr;         {the given clause, chosen from soslist}
     resolvent: objectptr;       {newly generated clause}
     subsumer: objectptr;        {subsuming clause in forward subsumption}
     subsumed: objectptr;        {clause subsumed in back subsumption}
     inclause: objectptr;        {input clause while being integrated}
     histvec: ivecptr;           {derivation history vector for new clause}
     retcd: integer;             {general-purpose return code}
     listretcd: integer;         {return code for list processing}
     numlits: integer;           {number of literals of new clause}
     lenopt: integer;            {subsumption option}
     unifopt: integer;     {option for integration routine -
                                  set unification properties on all
                                  terms (not just literals)}
     hyperpos: stkntptr;         {position in set of resolvents}
     subsumerpos: stkntptr;      {position in set of subsumers}
     subsumedpos: stkntptr;      {position in set of subsumed clauses}
     sospos: upbptr;             {position in set of support}
     listpos: upbptr;            {general-purpose list position}
     done: boolean;              {flag to indicate end of main loop}
     com2: common2ptr;           {the layer 2 common area}

begin{tp};

{acquire the common area for layer 2 services}
initcom2(com2);

{read in the list of axioms}
writeln('enter axioms');
cllsttread(axlist,retcd,com2);
if (retcd = 0) then
     begin
     writeln('axioms are as follows:');
     cllsttwrite(axlist,retcd,com2);
     end
else
     writeln('input of axioms list failed');

{now integrate the axioms - that is add them to the formulae database}
lstaccfirst(axlist,inclause,listpos,listretcd,com2);
while (listretcd = 0) do
     begin
     unifopt := 0;
     clintegrate(inclause,unifopt,retcd,com2);
     lstaccnext(axlist,inclause,listpos,listretcd,com2);
     end;

{now read in the set-of-support list}
writeln('enter set of support');
```

```
cllsttread(soslist,retcd,com2);
if (retcd = 0) then
     begin
     writeln('set of support clauses are as follows:');
     cllsttwrite(soslist,retcd,com2);
     end
else
     writeln('input of set of support list failed');

{integrate the set-of-support clauses}
lstaccfirst(soslist,inclause,listpos,listretcd,com2);
while (listretcd = 0) do
     begin
     unifopt := 0;
     clintegrate(inclause,unifopt,retcd,com2);
     lstaccnext(soslist,inclause,listpos,listretcd,com2);
     end;

{make clashlists a list containing axlist and hbglist.
 make allclauses a list containing axlist, soslist, and hbglist}
lstcreate(hbglist,retcd,com2);
lstcreate(clashlists,retcd,com2);
lstcreate(allclauses,retcd,com2);

lstinslast(axlist,clashlists,retcd,com2);
lstinslast(hbglist,clashlists,retcd,com2);

lstinslast(axlist,allclauses,retcd,com2);
lstinslast(soslist,allclauses,retcd,com2);
lstinslast(hbglist,allclauses,retcd,com2);

{
This is the main loop.  Select a clause from the set-of-support,
generate all hyper-resolvents between it, axioms, and clauses on the
hbglist.  Put the generated hyper-resolvents that are not subsumed onto
the soslist.  When that is all done, move the given clause from the
soslist to the hbglist and start over - until no more clauses exist in the
soslist or the null clause is generated.
}
done := false;
while not done do
     begin
     {select a "given clause}
     lstaccfirst(soslist,givencl,sospos,retcd,com2);
     if (retcd <> 0) then
          begin
          done := true;
          writeln('no more clauses in set of support');
          end
     else
          begin
          write('given clause is: ');
          cltwrite(givencl,retcd,com2);
          {generate the first hyper-resolvent}
```

```
hyperf(givencl,clashlists,resolvent,histvec,
       hyperpos,retcd,com2);
{This loop processes generated hyper-resolvents}
while (retcd = 0) and (not done) do
    begin
    write('resolvent: ');
    cltwrite(resolvent,retcd,com2);
    {throw away the derivation information}
    dealivec(histvec,com2);
    {now check for the null clause}
    clnumlit(resolvent,numlits,com2);
    if (numlits = 0) then
        begin
        writeln('null clause found');
        done := true;
        end
    else
        {forward subsumption check}
        begin
        lenopt := 0;  {allow clauses to subsume shorter ones}
        fsubfirst(resolvent,allclauses,lenopt,subsumer,
                  subsumerpos,retcd,com2);
        if (retcd = 0) then
            begin
            writeln('resolvent subsumed');
            {cancel position in the set of clauses that subsume
             the generated hyper-resolvent}
            cancfsub(subsumerpos,com2);
            {delete the nonintegrated hyper-resolvent}
            cldelnon(resolvent,retcd,com2);
            end
        else
            {back subsumption check}
            begin
            lenopt := 0;  {allow subsumption by a longer clause}
            bsubfirst(resolvent,allclauses,lenopt,subsumed,
                      subsumedpos,retcd,com2);
            {This loop deletes clauses subsumed by the new
             hyper-resolvent}
            while (retcd = 0) do
                begin
                write('resolvent subsumes existing clause: ');
                cltwrite(subsumed,retcd,com2);
                cldelint(subsumed,retcd,com2);
                bsubnext(subsumedpos,subsumed,retcd,
                         com2);
                end;
            {add the hyper-resolvent to the integrated formula
             database and to the set of support}
                      unifopt := 0;
            clintegrate(resolvent,unifopt,retcd,com2);
            lstinslast(resolvent,soslist,retcd,com2);
            end;
        end;
```

```
            hypern(hyperpos,resolvent,histvec,retcd,com2);
            end{while};
        end;
    {If the given clause was not deleted (due to subsumption), move it
     to the hbglist}
    if not done then
        begin
        lstaltpos(sospos,retcd);
        if retcd = 0 then
            begin
            lstdisconnect(sospos,com2);
            lstinslast(givencl,hbglist,retcd,com2);
            end;
        end;
    lstcancpos(sospos,com2);
    end;{while}
end.{tp}
```

16. Demodulation/Simplification

16.1. Meaning of Demodulation

Demodulation is the process of rewriting a clause in place using an equality substitution[25]. The rewriting is controlled by unit equality clauses called *demodulators*, for example,

P(f(a),b)
Equal(f(a),c)
____

P(c,b)

The clause P(c,b), called a *demodulant*, replaces the existing clause P(f(a),b), which is deleted. (The clause P(c,b) could also be derived by paramodulation, but the parent clause would not be deleted.)

Variables may be present in the demodulators, and in the clauses they demodulate, but instantiation of variables can occur only in the term in the equality, for example,

P(f(g(a)),g(a))
Equal(f(g(x)),h(x))
____

P(h(a),g(a)).

The demodulated clause need not be a ground clause (that is, it may contain variables):

Q(f(x),x)
Equal(f(x),g(x))
____

Q(g(x),x).

In general, one can specify that a demodulator apply left-to-right, right-to-left, or either way. In LMA, a user variable in the demodulator controls the direction of demodulation.

In the presence of multiple demodulators, many may apply, and each may apply more than once, for example,

P(f(g(a)),f(a))
Equal(g(x),h(x)) (left-to-right)
Equal(a,h(a)) (right-to-left)
Equal(f(a),b) (left-to-right)

P(b,b).

Since a demodulator may apply more than once, looping may occur[13]. This possibility occurs naturally in demodulators that express commutativity, such as

Equal(f(x,y),f(y,x))

In the presence of this demodulator, a clause like P(f(a,b)) would demodulate to P(f(b,a)), then to P(f(a,b)), then P(f(b,a)), etc. This is prevented in the following way.

When a clause is designated as an "either-way" demodulator, then whether it is applied or not depends on the lexical ordering of the instantiations of its variables. Lexical ordering of symbols can be allowed to default or can be specified by use of the LEX predicate. Depending of the lexical ordering of a and b, the demodulator

Equal(f(x,y),f(y,x))

will demodulate P(f(a,b)) to P(f(b,a)) or leave it unchanged. In this way canonical forms for expressions can be maintained. This is discussed in more detail at the end of the next section.

When existing demodulators are applied to a newly derived clause, the process is called ***forward demodulation***. It is also possible for new demodulators to be added to the clause space, in which case one may want to apply them to some or all of the existing clauses in the clause space. This process is called ***back demodulation***. An example would be the following situation. Suppose the set of existing clauses contains

P(f(h(a)))
Equal(f(b),c)

and the new demodulator

Equal(h(a),b)

is derived. Then by back demodulation the clause

P(f(b))

is derived, which immediately demodulates to

P(c).

The clause P(f(h(a))) is replaced by P(c).

16.2. Implementation of Demodulation and Simplification

Demodulation has been found to have a variety of uses[13, 23, 25]. Our implementation differs from the original conception somewhat:

1. We produce a single demodulant from any given clause. However, the routines *fdemodf* and *fdemodn* could be rewritten to produce any number of possible demodulants (we recommend the use of a single demodulant).
2. In forming the demodulant of a clause, we not only apply equality transformations but we also perform "function evaluations". For example, ($SUM 1 1) would be rewritten as 2, even though no demodulator existed to cause the reduction.

To understand the behavior of the demodulation-simplification routine, one must understand the meanings attached to the following system-defined symbols:

$SUM(n1,n2)	if n1 and n2 are self-defining numeric values, this simplifies to the value n1+n2
$NEG(n1)	if n1 is a self-defining numeric value, this simplifies to -n1
$PROD(n1,n2)	if n1 and n2 are self-defining numeric values, this simplifies to n1*n2.
$DIV(n1,n2)	if n1 and n2 are self-defining numeric values, and if n2 <> 0, then this evaluates to n1/n2
$MOD(n1,n2)	if n1 and n2 are self-defining integers, then this evaluates to n1 modulo n2
$POWER(n1,n2)	if n1 and n2 are self-defining integers, then this evaluates to n1 raised to the power n2
$COMP(n1,n2)	if n1 and n2 are ground values, then this evaluates to 0 if n1 = n2 -1 if n1 < n2 1 if n1 > n2
$AND(x1,x2)	evaluates to the logical and of x1 and x2. The arguments may be either 0's and 1's or TRUE's and FALSE's.
$OR(x1,x2)	evaluates to the logical or of x1 and x2
$NOT(x)	evaluates to the logical negation of x

$OUT(t)	if this occurs in a unit clause, and t is ground (contains no variables), t is written to the terminal and this evaluates to NIL. The term t may be a list of terms, enclosed by "[" and "]".
$IN	if this occurs in a unit clause, this evaluates to an object entered from the terminal, terminated by a ";"
$OUTIN(t)	if this occurs in a unit clause, and if t is ground, then t is written to the terminal and the whole term is replaced with an object entered from the terminal, terminated by a ";".
$CHR(n)	this symbol is only evaluated when a $OUT or a $OUTIN causes something to be written to the terminal. In that case this expression evaluates to "chr(n)", the ASCII character represented by the value n.
$GT(t1,t2)	This expression evaluates only if it is ground. In that case it evaluates to TRUE if t1 > t2. Else, it evaluates to FALSE.
$GE(t1,t2)	This expression evaluates only if it is ground. In that case it evaluates to TRUE if t1 >= t2. Else, it evaluates to FALSE.
$LT(t1,t2)	This expression evaluates only if it is ground. In that case it evaluates to TRUE if t1 < t2. Else, it evaluates to FALSE.
$LE(t1,t2)	This expression evaluates only if it is ground. In that case it evaluates to TRUE if t1 <= t2. Else, it evaluates to FALSE.
$EQ(t1,t2)	This expression evaluates only if it is ground. In that case it evaluates to TRUE if t1 = t2. Else, it evaluates to FALSE.
$NE(t1,t2)	This expression evaluates only if it is ground. In that case it evaluates to TRUE if t1 <> t2. Else, it evaluates to FALSE.
NOT(TRUE)	evaluates to FALSE
NOT(FALSE)	evaluates to TRUE

Besides the above, the following special symbols have been defined:

NIL	used to mark the end of lists
$C	used as "concatenate", a binary operator to form lists (that is, $C(a,$C(b,NIL)) is equivalent to [a,b])
$JUNK	any clause containing this symbol will evaluate to TRUE, if simplified
TRUE	any clause containing this literal will be simplified to TRUE
FALSE	will be removed from any clause by simplification
AND	currently not used in simplification
OR	used (along with NOT) in the representation of clauses

We intend to extend this list significantly, since the existence of such primitives can have an enormous impact on the ease of performing many operations.

There are three types of routines now included in layer 2 of LMA for demodulation/simplification:

1. Just after a clause has been generated (but before it has been integrated), the routine ***simplify*** can be used to apply demodulators and function evaluation to the clause. The clause itself is altered, and the history information is added to the end of the history vector produced by the inference rule.
2. The demodulants of a clause can be obtained by using ***fdemodf*** and ***fdemodn***. The given clause is not altered, and a new history vector is produced. In this sense forward demodulation behaves like an inference rule. ***fdemodn*** always fails under the current implementation (since only a single demodulant is produced).
3. When a new equality becomes a demodulator, clauses that are already integrated can be back demodulated. The ***bdemodf*** and ***bdemodn*** commands return demodulants o' existing clauses. They do not delete the parent.

The routines that perform these three operations are as follows:

clsetdemod - designate a given clause as a demodulator

clsetdemod(cl,dcode,com2)

This routine establishes the given clause as a demodulator. The dcode indicates whether left-to-right, right-to-left, or either type of demodulation is desired. The clause must be a positive unit (not pseudo-unit with more than 1 literal) of the form

EQxxxxx(t1,t2)

The EQ can be upper or lower case. The xxxxx can be any string (including null). t1 and t2 are arbitrary terms.

cl	-	an objectptr to a clause
dcode	-	an integer code (dleft, dright, and deither are defined constants) dleft - right dright - left deither - right with lex pref check)
com2	-	a pointer to the layer 2 common area

clenddemod - stop use of a clause as a demodulator

clenddemod(clobj)

This routine makes the clause referenced by clobj stop being used as a demodulator.

clobj	-	an objectptr to a clause being used as a demodulator

simplify - simplify a clause

simplify(clobj,clashobj,hist,count,retcd,com2)

This routine simplifies clobj. It may use any technique that seems to work. For now we use

- demodulation
- special symbol evaluation (simplify arithmetic exp & .o to terminal)
- duplicate literal removal
- tautology reduction (to TRUE)

The old value of clobj is destroyed, so if you need it, copy it. It is assumed that hist is open and that the first integer contains the number of "modification elements" in the vector (0 is quite acceptable).

obj	-	an objectptr
clashobj	-	list restricting the set of other clauses that can be used in the simplification
hist	-	an ivecptr to an open ivector.
count	-	maximum number of modifications that should be made to the object (this blocks loops)
retcd	-	an integer return code set as follows: 0 - no simplification could be made 1 - clobj was successfully simplified 2 - simplified to TRUE 3 - clobj does not reference a clause 4 - count cut off simpl. 5 - simplified to null cl. memfail - memory allocation failure
com2	-	a pointer to the layer 2 common area

fdemodf - forward demodulation, first

fdemodf(givcl,retcl,clashobj,hist,pos,count,retcd,com2)

This routine returns the first demodulant of the given clause.
The current implementation results in a unique demodulant and includes the complete "simplification" logic (i.e., special symbol simplification is used).

givcl	- an objectptr to the given clause
retcl	- an objectptr set to reference the demodulant
clashobj	- all clauses except the given clause must be contained in this object (nil means any clause is ok)
hist	- the ivecptr returned with the derivation data
pos	- a stkntptr used to maintain position in the set
count	- an upper limit on the allowed number of demodulations
retcd	- an integer return code
	0 - no simplification could be made
	1 - givcl was successfully simplified
	2 - simplified to TRUE
	3 - givcl does not reference a clause
	4 - count cut off simpl.
	5 - simplified to null cl.
	memfail - memory allocation failure
com2	- a pointer to the layer 2 common area

fdemodn - forward demodulation, next demodulant

fdemodn(pos,retcl,hist,retcd,com2)

This routine returns the next demodulant of the given clause. The current implementation results in a unique demodulant of a clause, so that this routine now always sends back a return code of 0.

pos	- the stkntptr used to maintain position in the set of demodulants		
retcl	- an objectptr set to reference the demodulant		
hist	- an ivecptr set to contain the derivation data		
retcd	- an integer return code		
	0	-	no more simpifications could be made
	1	-	givcl was successfully simplified
	2	-	simplified to TRUE
	4	-	count cut off simpl.
	5	-	simplified to null cl.
	memfail	-	memory allocation failure
com2	- a pointer to the layer 2 common area		

fdemodcanc - cancel position in a set of "forward" demodulants

fdemodcanc(pos,com2)

This procedure is used to cancel position in a set of demodulants.

pos	- the stkntptr used to maintain position in the set
com2	- a pointer to the layer 2 common area

bdemodf - back demodulation, first

bdemodf(givcl,retcl,retid,clashobj,hist,pos,count,retcd,com2)

This routine returns the first back demodulant of the given clause.

givcl	-	an objectptr to the given clause
retcl	-	an objectptr set to reference the demodulant
retid	-	an integer set to the id of the "into" parent
clashobj	-	all clauses except the given clause must be contained in this object (nil means any clause is ok)
hist	-	the ivecptr returned with the derivation data
pos	-	a stkntptr used to maintain position in the set
count	-	an integer giving the upper limit on the number of simplifications that can be performed on a back demodulated clause
retcd	-	an integer return code
		0 - no demodulants could be made
		1 - an existing clause was successfully simplified
		2 - demodulated and simplified to TRUE
		3 - givcl does not reference a clause
		4 - count cut off simpl.
		5 - demodulated and simplified to null cl.
		memfail - memory allocation failure
com2	-	a pointer to the layer 2 common area

bdemodn - back demodulation, next demodulant

bdemodn(pos,retcl,retid,hist,retcd,com2)

This routine returns the next back demodulant from the given clause.

pos	-	the stkntptr used to maintain position in the set of demodulants		
retcl	-	an objectptr set to reference the demodulant		
retid	-	an integer set to the id of the first "into" parent		
hist	-	an ivecptr set to contain the derivation data		
retcd	-	an integer return code		
		0	-	no more back demodulants could be made
		1	-	a back demodulant was successfully formed
		2	-	the back demodulant simplified to TRUE
		4	-	count cut off simplification
		5	-	the back demodulant simplified to null clause
		memfail	-	memory allocation failure
com2	-	a pointer to the layer 2 common area		

bdemodcanc - cancel position in a set of "back" demodulants

bdemodcanc(pos,com2)

This procedure is used to cancel position in a set of demodulants.

pos	-	the stkntptr used to maintain position in the set
com2	-	a pointer to the layer 2 common area

Note that when you make a positive unit equality clause a demodulator, you can cause rewrites to go from left to right, from right to left, or in either direction. For example,

(EQUAL (F x e) x);

would normally be left to right,

(EQUAL x (F e x));

would normally be right to left, and

```
(EQUAL (F x y) (F y x));
```

would normally be allowed to rewrite in either direction. An "either" demodulator causes both sides of the instantiated equality to be compared. A rewrite occurs only if the resulting term is "less than" the original. For example, suppose that

```
(P (F a1 b1) e (F e c1));
```

were to be simplified using the three above demodulators. Demodulation will progress (in effect) from the rightmost term, continuing to the left until no more terms can be simplified. Thus, (F e c1) first simplifies to c1. Then we progress on until (F a1 b1) is reached. This will be rewritten as

```
(F b1 a1)
```

if (F b1 a1) < (F a1 b1), where "<" represents a "lexical comparison". This comparison proceeds by finding the first symbols in which the terms differ. Then the indices into the symbol table are examined. The rule is that s1 < s2 (where s1 and s2 are symbols) if s1 occurs later in the symbol table than s2. This causes newly generated symbols to compare less than previously existing symbols. The user can force a given lexical ordering (of all but system-defined symbols) by using an initial input clause of the form

```
(LEX s1 s2 s3 s4 ...sn);
```

Here s1-sn are the symbols given in decreasing order.

17. Immediate Evaluation Rules

Demodulation is normally performed either upon newly generated clauses or (when new equalities become demodulators) upon previously existing clauses (using back demodulation). However, when an inference rule such as hyper-resolution or UR-resolution is being used, there are times when one would like to demodulate the nucleus between steps in forming the final resolvent. For example, consider the nucleus (written in the if-then format):

```
If Person(_x) &
  Person(_y) &
  $LT(_x,_y) &
  Compat(_x,_y)
then
  PossiblePair(_x,_y);
```

Here one would like the first two literals to be removed. Then either the third literal should simplify to FALSE (and be removed), or backtracking should begin. In fact ground literals with predicates of $LT, $LE, $GT, $GE, $NE, and $EQ are evaluated in the middle of calculating hyper-resolvents and UR-resolvents.

18. User Variables and Attributes

Some users will find it necessary to attach information to specific objects. This can be done using either of two mechanisms — user variables or attributes. User variables are just an array of integers kept in each object. They can be accessed or altered rapidly. Attributes are themselves non-integrated objects. The operations that are provided for processing user variables and attributes are as follows:

---

l2accuvar - access the value of a user variable

l2accuvar(objptr,i,value)

This routine accesses the value of the ith user variable.
Note that "maxl2uvar" defines the maximum subscript (1 is the minimum).

objptr	-	an objectptr;
i	-	an integer subscript in the range 1-maxl2uvar
value	-	an integer set to the value of the ith user variable

---

l2setuvar - set a user variable in an object

l2setuvar(objptr,i,value)

This routine sets the ith user value in the object.
Note that the constant "maxl2uvar" contains the maximum legal value of i (1 is the first value).

objptr	-	an objectptr
i	-	subscript of the user variable to be set
value	-	an integer value to put in the user variable

l2setattr - set an attribute on an object

l2setattr(objptr,attrcd,attrobj,retcd,com2)

This routine adds an attribute to the object referenced by "object". The attribute is the object referenced by "attrobj" and will have the attribute code given by attrcd.

objptr	-	an objectptr		
attrcd	-	an integer giving the attribute code		
attrobj	-	an objectptr to the attribute (must be nonintegrated)		
retcd	-	an integer return code set as follows:		
		0	-	new attribute set
		1	-	attribute replaces old attribute
		memfail	-	memory allocation failure
com2	-	a pointer to the layer 2 common area		

l2delattr - delete an attribute from an object

l2delattr(objptr,attrcd,retcd,com2)

This routine is used to delete the attribute with a code equal to the specified value.

objptr	-	an objectptr		
attrcd	-	an integer identifying the attribute to delete		
retcd	-	an integer return code set as follows:		
		0	-	success
		1	-	no such attribute on the object
com2	-	a pointer to the layer 2 common area		

l2getattr - get the attribute for a given code

l2getattr(objptr,attrcd,attrobj,retcd)

This routine sets attrobj to reference the attribute of objptr that has the specified attribute code.

objptr	- an objectptr		
attrcd	- an integer identifying the desired attribute		
attrobj	- an objectptr set to reference the desired attribute		
retcd	- an integer return code set as follows:		
		0	- success
		1	- no such attribute

l2getfattr - get the first attribute on a given object

l2getfattr(objptr,attrcd,attrobj,attrpos,retcd)

This routine sets attrobj to reference the first attribute on the object referenced by objptr.

objptr	- an objectptr		
attrcd	- an integer set to the code of the first attribute		
attrobj	- an objectptr set to reference the first attribute		
attrpos	- an attrptr used to maintain position in the set of attributes		
retcd	- an integer return code set as follows:		
		0	- success
		1	- no attributes on the object

l2getnattr - get the next attribute on an object

l2getnattr(attrcd,attrobj,attrpos,retcd)

This routine returns the attribute code and value for the next attribute on an object. The attrpos parameter maintains position in the set of attributes.

attrcd	- an integer set to the code for the next attribute		
attrobj	- an objectptr set to the value for the attribute		
attrpos	- an attrptr used to maintain position in the set of attributes		
retcd	- an integer return code set as follows:		
		0	- success
		1	- no more attributes on the object

19. Qualification and Locking

There are three ways a literal in a clause can be made *nonclashable*.

1. The literal may be determined to be a qualifier[21].
2. The occurrence of the literal may be locked[9].
3. All occurrences of the literal may be locked[9].

Qualification amounts to specifying that a function or predicate requires "conditions of definition". The whole topic is discussed in Winker's paper. We have found qualification useful on a surprisingly wide variety of problems. To make it work, you use *setqual* to specify which literals qualify a given predicate/function symbol. Then *qualcl* is invoked to mark the qualifying literals as nonclashable. The inference rules ignore unclashable literals (they are copied into the inferred clause), unless *setiglock* is used to cause clashability tests to be ignored.

You can make an occurrence of a literal nonclashable by invoking *setcllock*. It can later be made clashable by using *delcllock*.

Finally, you can make all occurrences of a variable nonclashable by assigning it a positive lock value using *setlitlock*. The lock can be removed with *dellitlock* or tested with *getlitlock*.

The detailed definitions of all of the routines that relate to the topics of qualification and locking are as follows:

setqual - add a qualification template

setqual(clobj,retcd,com2)

This routine uses clobj to establish a qualification template.
clobj should be a clause of the form

TEMPLATE(t1) or L2 or L3 ...

This indicates that any instance of t1 must be qualified with the corresponding instances of L2, L3,....
clobj is "lost" to the calling routine. Therefore, if you wish to keep it, copy it before calling setqual.

clobj	- an objectptr to a clause		
retcd	- an integer return code set as follows:		
	0	-	success
	1	-	clobj is not in the correct format
	memfail	-	memory allocation failure
com2	- a pointer to the layer 2 common area		

qualcl - mark qualifiers on a clause

qualcl(clobj,retcd,com2)

This routine marks the qualifiers on a clause.
The clause should probably be integrated, since integrating a clause loses its attributes (which are used to record qualifiers).

clobj	- an objectptr to the clause		
retcd	- an integer return code set as follows:		
	0	-	success
	1	-	clobj is not a clause
	memfail	-	memory allocation failure
com2	- a pointer to the layer 2 common area		

setqwopt - set qualification warning message option

setqwopt(val,com2)

This routine sets the flag that determines whether or not warning messages for incompletely qualified clauses should be written out.

val	- an integer code:		
	0	-	no warning messages
	1	-	warnings are written
com2	- a pointer to the layer 2 common area		

setcllock - lock an occurrence of a literal in a given clause

setcllock(clobj,i,retcd,com2)

This routine makes the ith literal of clobj unclashable.

clobj	- an objectptr to a clause		
i	- an integer giving the literal to lock		
retcd	- an integer return code set as follows:		
	0	-	success
	1	-	clobj is not a clause
	2	-	i is invalid
	memfail	-	memory allocation failure
com2	- a pointer to the layer 2 common area		

delcllock - unlock an occurrence of a literal in a given clause

delcllock(clobj,i,retcd,com2)

This routine makes the ith literal of clobj clashable.

clobj	- an objectptr to a clause		
i	- an integer giving the literal to unlock		
retcd	- an integer return code set as follows:		
	0	-	success
	1	-	clobj is not a clause
	2	-	i is invalid
	memfail	-	memory allocation failure
com2	- a pointer to the layer 2 common area		

getcllock - get the lock character for an occurrence of a literal

getcllock(clobj,i,val,retcd,com2)

This routine sets val to 0 if the ith literal of obj is unlocked.
Else, val is set to 1.

clobj	- an objectptr to a clause		
i	- an integer designating the literal		
val	- an integer set to reflect the lock value		
	0	-	unlocked
	1	-	locked
retcd	- an integer return code set as follows:		
	0	-	success
	1	-	clobj is not a clause
	2	-	i is invalid
com2	- a pointer to the layer 2 common area		

setlitlock - set a lock value on a literal

setlitlock(litobj,n,retcd,com2)

This routine sets the lock value n on the literal litobj. Lock values must be greater than 0. Literals with a lock are not clashable, unless "setiglock" has been called to suppress clashability checks.

litobj	-	an objectptr to a literal		
n	-	an integer giving the desired lock value (>0)		
retcd	-	an integer return code set as follows:		
		0	-	success
		1	-	litobj does not reference a literal
		2	-	n is invalid
		memfail	-	memory allocation failure
com2	-	a pointer to the layer 2 common area		

getlitlock - access the literal lock on a given literal

getlitlock(litobj,n,com2)

This routine sets n to the literal lock on the literal referenced by litobj. If litobj does not reference a literal, or if no lock has been set, the value will be 0.

litobj	-	an objectptr to a literal
n	-	an integer set to the lock value
com2	-	a pointer to the layer 2 common area

litclashable - is literal clashable?

litclashable(clobj,i,retcd,com2)

This routine checks to see whether or not an occurrence of a literal is clashable.

clobj	-	an objectptr to a clause		
i	-	an integer designating the literal		
retcd	-	an integer return code set as follows:		
		0	-	literal is clashable
		1	-	literal is not clashable
com2	-	a pointer to the layer 2 common area		

setiglock - set the flag that determines whether or not locks are ignored

setiglock(val,com2)

This routine can be used to indicate whether literal and clause (literal occurrence) locks are observed or ignored by inference rules.

val	-	an integer: 0->observe locks; 1->ignore locks
com2	-	a pointer to the layer 2 common area

Before leaving the topic of qualification and locking, one extra point is worth noting. Several inference rules use the concept of "unit clause" to restrict the set of generated clauses. For example, unit resolution requires that one of the two parents be a unit clause. We have introduced the notion of ***pseudo-unit clause***. A pseudo-unit clause has exactly one clashable literal (i.e., it can have more than one literal, but only one can be clashable). Inference rules such as unit resolution and UR-resolution have been implemented in a way that allows pseudo-unit clauses to be treated as unit clauses. This generalization does not apply to demodulation, however; a demodulator must contain only one literal.

20. Weighting

Weighting[13] is a mechanism for assigning a number to a clause, literal, or term. This number can then be used for such things as determining whether to keep a newly derived clause, picking the next given clause, or deciding whether a newly derived equality should become a demodulator. The use of weighting in an LMA-based theorem prover is deiscussed in detail in[11]

20.1. Weighting Parameter Sets

A collection of options called a weighting parameter set can be used to determine the weight of a clause, literal, or term. Each weighting parameter set consists of sixteen real numbers and a list of patterns. The numbers are structured into three families of five plus one other number. These numbers describe how weights of clauses are built up from the weights of their component literals, weights of literals from the weights of their component predicates and arguments, and the weights of terms from the weights of their function symbols and subterms. The patterns describe how this weighting algorithm is to be bypassed to give special weights to certain classes of clauses, literals, or terms. We will discuss the algorithmic mechanism first and patterns later.

20.2. Weighting Without Patterns

Let us assume that the pattern list for the weighting parameter set we are interested in is empty (this is the default). Then the weight of a clause is calculated from the sixteen numbers in the weighting parameter set in the following way.

Constants and Variables

The weight of a constant is 1. The weight of a variable is the number entered and displayed as "variable weight." The default variable weight is 1.

Complex Terms

For each of clauses, literals, and terms, there is a set of five numbers that controls the way in which their weights are calculated from the weights of their components. The names of these numbers are #ARG, MAXARGWT, SUMARGWT, SYMCT, and BASE. They have slightly different meanings for clauses, literals, and terms. We begin with terms. Simple terms (constants and variables) were covered above. The weight of a complex term (one containing subterms) is calculated as follows:

```
weight of term =   BASE +
                   SYMCT * (number of symbols in term) +
                   #ARG * (number of immediate subterms of term) +
                   MAXARGWT * (weight of heaviest immediate subterm) +
                   SUMARGWT * (sum of weights of all immediate subterms)
```

Note that BASE does not apply to simple terms.

For the purposes of weighting, the major function symbol of a term is considered one of its subterms. The number of symbols is the total number of names of constants, variables, and function symbols appearing in the term. Thus the term

g(a,f(x1,maxlock))

is considered to have three subterms and to contain five symbols.

Suppose, for example, that the variable weight is set to 1 and that the weighting coefficients for terms are as follows, which is the default setting:

#ARG=0 MAXARGWT=0 SUMARGWT=1 SYMCT=0 BASE=0

Then the weights for some sample terms are as follows:

a	1
x	1
f(a)	2

f(a,b)	3
f(a,g(a,b))	5

On the other hand, if the term weighting coefficients are

#ARG=1 MAXARGWT=0 SUMARGWT=0 SYMCT=0 BASE=100

then the weights of these same terms are as follows:

a	1
x	1
f(a)	102
f(a,b)	103
f(a,g(a,b))	103

Literals

There are separate values of #ARG, MAXARGWT, SUMARGWT, SYMCT, and BASE for literals. With these values, the weight of a literal is calculated as follows:

weight of literal = BASE +
SYMCT * (number of symbols in literal) +
#ARG * (number of arguments of literal) +
MAXARGWT * (weight of heaviest argument) +
SUMARGWT * (sum of weights of all arguments)

For the purposes of weighting, the predicate symbol of a complex literal is counted as one of its arguments. Negative literals can have their weights adjusted, but this is done with patterns, discussed below. The negation symbol is not included in the symbol count.

Suppose that the weighting coefficients for terms are set to the defaults described above and that the weighting coefficients for literals are as follows, which is the default setting:

#ARG=0 MAXARGWT=1 SUMARGWT=0 SYMCT=0 BASE=0

Then the weights from some sample literals are

P	1
-P	1
P(a,b)	1
P(f(a))	2
P(f(a,b),a)	3

If, instead, the literal weighting coefficients are

#ARG=1 MAXARGWT=5 SUMARGWT=0 SYMCT=0 BASE=0

then the weights of these same literals are

P	1
-P	1
P(a,b)	8
P(f(a))	12
P(f(a,b),a)	18

Note that the first two literals are weighed as constants, not as literals with one argument.

Clauses

There is a third set of #ARG, etc., for clauses. Using these values, the weight of a clause is calculated as follows:

weight of clause = BASE +
SYMCT * (number of symbols in clause) +
#ARG * (number of literals of clause) +
MAXARGWT * (weight of heaviest literal) +
SUMARGWT * (sum of weights of all literals)

For weighting purposes, the number of symbols in the clause includes the implicit OR symbols between the literals, and any negation symbols in front of negative literals. Thus the clause

P | Q

is considered to contain three symbols, and

if P then Q

is considered to have four symbols, since it translates into -P | Q.

Now suppose that the weighting coefficients for terms and literals have their default settings described above, and that variable weight has its default value of 1. Suppose further that the clause weighting coefficients are

#ARG=1 MAXARGWT=0 SUMARGWT=1 SYMCT=0 BASE=-1,

which is the default. Then the weights of some sample clauses are as follows:

P;	1
P | Q;	3
P | Q | R;	5
-P;	1
if P then Q;	3
P(f(a)) | Q(x);	4

20.3. Weighting with Patterns

Weighting patterns are a mechanism for overriding the previous weighting algorithm to assign particular weights to specific terms, literals, and clauses, as well as to terms, literals, or clauses that are characterized by their matching a particular pattern. Some simple patterns and their meanings are the following:

a:+10	the term a has weight 10
NOT:+6	negative literals should have 6 added to their weight
f(2):+3	the weight of any term of the form f(<term>) should be 3 plus twice the weight of <term>.

There is a list of patterns in each weighting parameter set. If a given term, literal, or clause matches more that one pattern in the list, then the first one it matches has priority. For example, if the term f(a,b) is weighed according to the pattern list

f(a,2):+5 f(a,b):+15;

then it is given a weight of seven (assuming b has its default weight of 1).

The exact format of a weighting pattern is

<basic-pattern>:<increment>

where <increment> is a signed floating-point number, and <basic-pattern> can be any one of the following:

1. A constant. This matches only an occurrence of the constant.
2. x<int> where <int> is a positive integer (e.g., x4). This matches only a variable with the given number.
3. *x<int> where <int> is a positive integer. This matches any variable, except that multiple occurrences of *x<int> in the same pattern must match the same variable. For example, the pattern f(*x1,*x1):+2 would match the term f(x2,x2), but not the term f(x1,x2).
4. *t<int> where <int> is a positive integer. This matches any term, except that multiple occurrences of *t<int> in the same pattern must match the same term.
5. <multiplier>, which is a real number. This matches any term. The effect of a match is to multiply the weight of the subterm by the multiplier. The result is added into the weight of the current term.
6. <name>(<arg-1> <arg-2>,...<arg-n>) where <arg-i> is a <basic-pattern>. This matches a complex term in which <name> is the predicate/function symbol, and <arg-i> matches the ith subterm (for all i from 1 to n).

The weight of the term matched by the pattern is computed by adding the <increment> to the weights generated from having <multiplier>s in the pattern. Thus, if f(a,g(1.5,-.5)):+2.5 matches a term, the final weight is 2.5 (the increment) plus 1.5 times the weight of the first argument of g plus -.5 times the weight of the second argument of g.

20.4. Routines to Implement Weighting Calculations

A weighting parameter set is defined by the following type declaration (from the layer 2 type declaration file):

```
wtparm   = record
           clarray: coefarray;     {weight coefficients for clauses}
           litarray:coefarray;     {weight coefficients for literals}
           trmarray:coefarray;     {weight coefficients for terms}
           patlist:wtcalcptr;      {header to weight pattern list}
           pattree:dtreehptr;      {root of pattern search tree}
           nextpatnum:integer;     {id of next pattern inserted in tree}
           varweight:real;         {weight of variables}
         end;
```

LMA provides routine for altering the weighting coefficients, adding patterns, weighing clauses, weighing literals, and weighing terms. The routines for altering the weighting information in a parameter set (i.e., the first six fields of the parameter set) are as follows:

recwtkeys - recognize a string of weighting keyword assignments

recwtkeys(str,wtcoef,retcd,com2)

This procedure proceeds from the current position in str. It assigns values to the weighting coefficients in wtcoeff, which is an array of maxwtcoef real values. Currently, the recognized keywords and the positions of the corresponding values in wtcoef are as follows:

keyword	array position
<number><thing>	1
Here <thing> can be either <argument> or <literal>	
<maximum><thing><weight>	2
<sum><thing><weight>	3
<number><symbol>	4
<symbol><count>	4
<base>	5

For example,

numarguments = 1.4 base=8;

would cause the first array position to be set to 1.4, and the fifth to 8. The string is terminated by a semicolon. Any unrecognized keywords will result in error messages.

- str - a csptr of where to start the parse
- wtcoeff - a coefarray that gets altered when keywords are successfully recognized
- retcd - an integer return code set as follows:
  - 0 - no errors detected
  - 1 - errors detected
  - memfail - memory allocation failure
- com2 - a pointer to the layer 2 common area

recwtpats - recognize a list of weight patterns

recwtpats(str,wtparms,retcd,com2)

This routine parses a string of weight patterns, adding the successfully parsed patterns to wtlist.

str	-	a csptr to the string being parsed (ends with a semicolon or end-of-string)
wtparms	-	a wtparmptr to parameters for weighting set
retcd	-	an integer return code set as follows: 0 - success 1 - at least one invalid wt template was detected memfail - memory allocation failure
com2	-	a pointer to the layer 2 common area

The routines that can be used to convert weighting parameters to a printable format are as follows:

wtcoefsout - convert an array of weighting coef to keyword form

wtcoefsout(wtcoef,str,retcd,com2)

This routine creates a string containing the character form of the weighting coefficients in wtcoef.

wtcoef	-	a coefarray containing a set of weighting coefficients (clause, literal, or term)
str	-	a csptr that is set by the routine to reference the created string. str should not reference an allocated string when the routine is called.
retcd	-	an integer return code set as follows: 0 - success memfail - memory allocation failure
com2	-	a pointer to the layer 2 common area

wtcalcstout - convert a weight calculation list to portable format

wtcalcstout(wtlist,str,retcd,com2)

This routine converts the weight calculation information in wtlist into portable format, returning it in str.

wtlist	-	a wtcalcptr to a weight calculation list
str	-	a csptr that gets set to reference the generated string
retcd	-	an integer return code that gets set as follows: 0 - success memfail - memory allocation failure
com2	-	a pointer to the layer 2 common area

The routines that can be used to weigh a clause, literal, or term are as follows:

wtcl - calculate the weight of a clause

wtcl(clause,wtparms,weight,retcd,com2)

This routine calculates the weight of clause and returns it in weight.

clause	-	an objectptr to the clause to weigh
wtparms	-	a wtparmptr to the weighting parameters
weight	-	a real number set to the weight of the clause
retcd	-	a return code set as follows: 0 - success memfail - memory allocation failure
com2	-	a pointer to the layer 2 common area

wtlt - calculate the weight of a literal

wtlt(literal,ltcoef,trmcoef,wtroot,varwt,weight,retcd,com2)

This routine calculates the weight of literal and returns it in weight.

literal	-	an objectptr to the literal to weigh
ltcoef	-	a coefarray giving the coefficients for weighting a literal
trmcoef	-	a coefarray giving the coefficients for weighting a term
wtroot	-	a dtreehptr to root of weighting pattern tree
varwt	-	a real giving the weight assigned to variables that do not match a pattern
weight	-	a real number set to the weight of the literal
retcd	-	a return code set as follows: 0 - success memfail - memory allocation failure
com2	-	a pointer to the layer 2 common area

wttrm - calculate the weight of a term

wttrm(term,trmcoef,wtroot,varwt,weight,retcd,com2)

This routine calculates the weight of term and returns it in weight.

term	-	an objectptr to the term to weigh
trmcoef	-	a coefarray giving the coefficients for weighting a term
wtroot	-	a dtreehptr to root of weighting pattern tree
varwt	-	a real giving the weight assigned to variables that do not match a pattern
weight	-	a real number set to the weight of the term
retcd	-	a return code set as follows: 0 - success memfail - memory allocation failure
com2	-	a pointer to the layer 2 common area

21. The LISP Interface

The LISP interface is not actually part of layer 2, since it is not portable. Clearly, interfaces between languages depend upon vagaries of specific compilers. Our current LISP interface works under Berkeley UNIX and interfaces the layer 2 routines to Franz Lisp. We do not include the details of the interface here, since it will be included only for users under Berkeley UNIX. We hope to offer interfaces in other LISP environments, or to offer aid and encouragement to others who wish to develop such interfaces. As an example of what can be done, here is the simple theorem prover described before in its LISP incarnation:

```
(defun lisptp ()
  (prog (axlist soslist hbglist clashlists allclauses
         givencl resolvent subsumer subsumed clause
         com2
         histvec
         retcd listretcd numlits lenopt unifopt
         hyperpos subsumerpos subsumedpos
         sospos listpos
        )

    (initvar axlist)
    (initvar soslist)
    (initvar hbglist)
    (initvar clashlists)
    (initvar allclauses)
    (initvar givencl)
    (initvar resolvent)
    (initvar subsumer)
    (initvar subsumed)
    (initvar clause)
    (initvar com2)
    (initvar histvec)
    (initvar retcd)
    (initvar listretcd)
    (initvar numlits)
    (initvar lenopt)
    (initvar unifopt)
    (initvar hyperpos)
    (initvar subsumerpos)
    (initvar subsumedpos)
    (initvar sospos)
    (initvar listpos)

    (call initcom2 com2)

    (call cllsttread axlist retcd com2)
    (cond
        ((zerop (valueof retcd))
         (print  "axioms are as follows:")
         (terpr)
         (call cllsttwrite axlist retcd com2)
        )
        (t
         (print "input of axioms list failed")
         (terpr)
        )
    )

    (call lstaccfirst axlist clause listpos listretcd com2)
```

```
(do ()
    ((not (zerop (valueof listretcd))) nil)
    (setintvar unifopt 0)
    (call clintegrate clause unifopt retcd com2)
    (call lstaccnext axlist clause listpos listretcd com2)
)

(call clisttread soslist retcd com2)
(cond
    ((zerop (valueof retcd))
     (print "set of support clauses are as follows:")
     (terpr)
     (call cllsttwrite soslist retcd com2)
    )
    (t
     (print "input of set of support list failed")
     (terpr)
    )
)

(call lstaccfirst soslist clause listpos listretcd com2)
(do ()
    ((not (zerop (valueof listretcd))) nil)
    (setintvar unifopt 0)
    (call clintegrate clause unifopt retcd com2)
    (call lstaccnext soslist clause listpos listretcd com2)
)

(call lstcreate hbglist retcd com2)
(call lstcreate clashlists retcd com2)
(call lstcreate allclauses retcd com2)

(call lstinslast axlist clashlists retcd com2)
(call lstinslast soslist clashlists retcd com2)

(call lstinslast axlist allclauses retcd com2)
(call lstinslast soslist allclauses retcd com2)
(call lstinslast hbglist allclauses retcd com2)

(setq done nil)
(do ()
    (done nil)
    (call lstaccfirst soslist givencl sospos retcd com2)
    (cond
        ((not (zerop (valueof retcd)))
         (setq done t)
         (print "no more clauses in set of support")
         (terpr)
        )
        (t
         (print "given clause is: ")
         (terpr)
         (call cltwrite givencl retcd com2)
         (call hyperf givencl clashlists resolvent histvec
```

```
        hyperpos retcd com2)

(do ()
    ((or (not (zerop (valueof retcd))) done) nil)
    (print  "resolvent: ")
    (terpr)
    (call cltwrite resolvent retcd com2)
    (call dealivec histvec com2)
    (call clnumlit resolvent numlits com2)
    (cond
        ((zerop (valueof numlits))
         (print  "null clause found")
         (terpr)
         (setq done t)
        )
        (t
         (setintvar lenopt 0)
         (call fsubfirst resolvent allclauses lenopt
                    subsumer subsumerpos retcd com2)
         (cond
             ((zerop (valueof retcd))
              (print "resolvent subsumed")
              (terpr)
              (call cancfsub subsumerpos com2)
              (call cldelnon resolvent retcd com2)
             )
             (t
              (print "checking back subsumption")
              (terpr)
              (setintvar lenopt 0)
              (call bsubfirst resolvent allclauses lenopt
                         subsumed subsumedpos retcd com2)
              (do ()
                  ((not (zerop (valueof retcd))) nil)
                  (print "resolvent subsumes existing clause:")
                  (terpr)
                  (call cltwrite subsumed retcd com2)
                  (call cldelint subsumed retcd com2)
                  (call bsubnext subsumedpos subsumed retcd
                             com2)
              )

              (setintvar unifopt 0)
              (call clintegrate unifopt resolvent retcd com2)
              (call lstinslast resolvent soslist retcd com2)
             )
         )
        )
    )
    (call hypern hyperpos resolvent histvec retcd com2)
)
(cond
    ((not done)
     (call lstaltpos sospos retcd)
```

```
                        (cond
                            ((zerop (valueof retcd))
                             (call lstdisconnect sospos com2)
                             (call lstinslast givencl hbglist retcd com2)
                            )
                            (t nil)
                        )
                       )
                       (t nil)
                      )
                 )
                 (call lstcancpos sospos com2)
             )
         )
      )
    )
```

22. Conclusion

We are putting this set of tools into the public domain. In their current form they can be (and will be) dramatically improved. We view this project as very long-term, and we plan on reworking, upgrading, and expanding the set of tools for many years. We are inviting you to participate in this project. The advantages of coordinating development between many users appear to us to be extremely significant. We sincerely wish to integrate and distribute any improvements that anyone can make to these tools.

References

1. R. S. Boyer and J. S. Moore, "The sharing of structure in theorem proving programs," in *Machine Intelligence 7*, ed. B. Meltzer and D. Michie,American Elsevier, New York (1972).

2. Chin-Liang Chang and Richard Char-Tung Lee, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York (1973).

3. C. L. Chang, "The unit proof and the input proof in theorem proving," *Journal of the ACM* 17(4) pp. 698-707 (1970).

4. Lawrence Henschen, R. Overbeek, and Lawrence Wos, "Hyperparamodulation: a refinement of paramodulation," in *Proceedings of the Fifth Conference on Automated Deduction, Springer-Verlag Lecture Notes in Computer Science, Vol. 87*, ed. Robert Kowalski and Wolfgang Bibel,Springer-Verlag, New York ().

5. Robert Kowalski, *Logic for Problem Solving*, Elsevier North Holland, New York (1979).

6. E. Lusk, William McCune, and R. Overbeek, "Logic Machine Architecture: inference mechanisms," pp. 85-108 in *Proceedings of the Sixth Conference on Automated Deduction, Springer-Verlag Lecture Notes in Computer Science, Vol. 138*, ed. D. W. Loveland,Springer-Verlag, New York ().

7. E. Lusk and R. Overbeek, "Data structures and control architecture for the implementation of theorem-proving programs," in *Proceedings of the Fifth Conference on Automated Deduction, Springer-Verlag Lecture Notes in Computer Science, Vol. 87*, ed. Robert Kowalski and Wolfgang Bibel, ().

8. E. Lusk, William McCune, and R. Overbeek, "Logic machine architecture: kernel functions," pp. 70-84 in *Proceedings of the Sixth Conference on Automated Deduction, Springer-Verlag Lecture Notes in Computer Science, Vol. 138*, ed. D. W. Loveland,Springer-Verlag, New York (1982).

9. E. Lusk and R. Overbeek, "Experiments with resolution-based theorem-proving algorithms," *Computers and Mathematics with Applications* **8**(3) pp. 141-152 (1982).

10. Ewing L. Lusk and Ross A. Overbeek, "An LMA-based theorem prover," ANL-82-75, Argonne National Laboratory (December, 1982).

11. Ewing L. Lusk and Ross A. Overbeek, *The automated reasoning system ITP*, Argonne National Laboratory (March, 1984). preprint

12. J. McCharen, R. Overbeek, and L. Wos, "Problems and experiments for and with automated theorem-proving programs," *IEEE Transactions on Computers* **C-25**(8) pp. 773-782 (1976).

13. J. McCharen, R. Overbeek, and L. Wos, "Complexity and related enhancements for automated theorem-proving programs," *Computers and Mathematics with Applications* **2** pp. 1-16 (1976).

14. R. Overbeek, "An implementation of hyper-resolution," *Computers and Mathematics with Applications* **1** pp. 201-214 (1975).

15. G. Robinson and L. Wos, "Paramodulation and theorem proving in first-order theories with equality," pp. 135-150 in *Machine Intelligence 4*, ed. B. Meltzer and D. Michie,Edinburgh University Press (1969).

16. G. Robinson and L. Wos, "Completeness of paramodulation," *Spring 1968 meeting of the Association of Symbolic Logic 34*, p. 160 (1969).

17. J. Robinson, "Automatic deduction with hyper-resolution," *International Journal of Computer Mathematics* **1** pp. 227-234 (1965).

18. J. Robinson, "A machine-oriented logic based on the resolution principle," *Journal of the ACM* **12** pp. 23-41 (1965).

19. J. Slagle, "Automatic theorem proving with renamable and semantic resolution," *Journal of the ACM* **14** pp. 687-697 (1967).

20. D. H. D. Warren, "Implementing Prolog - compiling predicate logic programs," DAI Research Reports 39 and 40, University of Edinburgh (May 1977).

21. S. Winker, "An evaluation of an implementation of qualified hyperresolution," *IEEE Transactions on Computers* **C-25**(8) pp. 835-843 (August 1976).

22. S. Winker, L. Wos, and E. Lusk, "Semigroups, antiautomorphisms, and involutions: a computer solution to an open problem, I," *Mathematics of Computation* **37**(156) pp. 533-545 (October 1981).

23. S. Winker and L. Wos, "Procedure implementation through demodulation and related tricks," pp. 109-131 in *Proceedings of the Sixth Conference on Automated Deduction, Springer-Verlag Lecture Notes in Computer Science, Vol. 138*, ed. D. W. Loveland,Springer-Verlag, New York (1982).

24. L. Wos, D. Carson, and G. Robinson, "The unit preference strategy in theorem proving," pp. 615-621 in *Proceedings of the Fall Joint Computer Conference*, Thompson Book Company, New York (1964).

25. L. Wos, G. Robinson, D. Carson and L. Shalla, "The concept of demodulation in theorem proving," *Journal of the ACM* **14** pp. 698-704 (1967).

26. L. Wos, S. Winker, and E. Lusk, "An automated reasoning system," *Proceedings of the AFIPS National Computer Conference*, pp. 697-702 (1981).