LogPro Manual

Bernard Sufrin*

Version of February 7, 2002

Running LogPro

The *LogPro* interpreters are available in the Laboratory filestore as

```
/ecslab/sufrin/logpro/bin/logpro
/ecslab/sufrin/logpro/bin/x86-solaris/logpro
/ecslab/sufrin/logpro/bin/sparc-solaris/logpro
/ecslab/sufrin/logpro/bin/x86-linux/logpro
```

The first of these is a bytecode interpreter, and can be used anywhere there is an installation of ocaml 2.02. The others are binary executables which can be used on a machine with the appropriate architecture and operating system.[1]

Once the appropriate path is added to your PATH variable, the *LogPro* interpreter is invoked with the command

```
logpro [path path ... path]
```

After the commands in the specified files have been read,[2] the intepreter enters interactive mode, initially prompting for each command with `--`,[3] and responding to queries by displaying the substitutions which satisfy them[4] in sequence. After each substitution is displayed, the interpreter prompts with a "`?`": responding to this with "`.`" terminates the search for substitutions; responding with a newline causes the search to continue. At any stage during the evaluation of a query, it can be interrupted by typing the appropriate interrupt character for the operating system on which it is being run.

*Oxford University Computing Laboratory

[1] The Linux version was compiled and runs under a completely standard RedHat 6.1 distribution. I don't know of any reason why it might not run under other Linux distributions.

[2] For each *path*, *LogPro* reads the file *path*.`lp` if it exists, and otherwise reads the file *path*.

[3] If the interpreter requires more input in response to its prompt – perhaps because a query or directive has not yet been terminated properly, then it will prompt again with its secondary prompt: `>>`

[4] Or "`Yes`" if there are no variables in the query

Commands

A *LogPro* command is either a *clause* taking the form

$$formula \texttt{ :- } formula\texttt{, } formula\texttt{, } \cdots\ formula\texttt{.}$$

or a *query* taking one of the (equivalent) forms

$$\texttt{:- } formula\texttt{, } formula\texttt{, } \cdots\ formula\texttt{.}$$
$$formula\texttt{, } formula\texttt{, } \cdots\ formula\texttt{?}$$

or a *directive* taking one of the forms in Table 1.

If there are no formulæ to the right of the entailment symbol `:-` in a clause then that symbol may be omitted.

Command		Effect
`#use`	*path*	Read the file specified by the given path (no more than once per session)
`#width`	*number*	Set the output width to *number* columns
`#infix0`	"*symbol*"	Declare the symbol to be a right-associative infix operator, priority 0
`#infix1`	"*symbol*"	Declare the symbol to be a right-associative infix operator, priority 1
...		
`#infix7`	"*symbol*"	Declare the symbol to be a right-associative infix operator, priority 7
`#infix8`	"*symbol*"	Declare the symbol to be a *left-associative* infix operator, priority 8
`#prefix`	"*symbol*"	Declare the symbol to be a prefix operator with maximal priority
`#notfix`	"*symbol*"	Declare the symbol to be a prefix operator with minimal priority
`#trace`	on	Trace the invocation of each predicate
`#trace`	all	As above and show the complete current substitution at each invocation
`#trace`	off	Stop tracing
`#check`	on *or* off	Enable or disable the occurs check in unification
`#count`	on *or* off	Enable or disable the printing of inference count with answers

Table 1: Directives

Notation

Any text appearing between a `/*` and the subsequent `*/` is treated as a comment and ignored, as is any text appearing between a `--` and the subsequent newline.

A *word* is any consecutive sequence of letters, digits, underscores or primes.

A *symbol* is any consecutive sequence of one or more of the characters "`!^%$@&/\;=><`", "`:+-*/~|`"

The symbol `:-` is the *entailment* symbol. Any other symbol or word may be declared to be an infix, prefix, or notfix operator using one of the directives outlined in Table 1. Unless

declared otherwise by such a directive, all symbols behave as left associative infix operators of priority 8.

A *term* has one of the forms described below, whilst a *formula* has any of these but the first three.

1. a (possibly negated) *integer*. For example:

```
34567
-32
```

2. a *double-quoted string*.

3. a *variable* – which is a word beginning with an uppercase letter. For example:

```
X
Rumpelstiltskin
State_Space
```

4. an *atom* – which is a word beginning with a lowercase letter. For example:

```
it_is_raining
david
pruneSquallor
```

5. a *prefix composite* – an atom, followed by a parenthesized, comma-separated, list of *term*s. For example:

```
grandparent(X, Y)
connected(From, To, Via)
simply(purple)
negative(2)
```

6. an *infix composite* – a pair of terms separated by an infix operator. For example, after the directive `#infix3 "divides"`

```
Var + Const
3 divides 62
```

7. a *leftfix composite* – a term followed by a bracketed term. This is regarded as syntactic sugar for a prefix composite with functor `[]`. For example:

`f[Arg]` is sugar for `[](f, Arg)`
`g[x][y]` is sugar for `[]([](g, x), y)`

8. a *rightfix composite* – a term bracketed by the "fat brackets" `[|` and `|]`. This is regarded as syntactic sugar for a prefix composite with functor `[||]`. For example:

`[|I|] S` is sugar for `[||](I, S)`
`[|I|][|J|]S` is sugar for `[||](I, [||](J, S))`

Leftfix composites have higher priority than rightfix composites. For example:

[|I|]J[S] is sugar for [||](I, [](J, S))

9. an *outfix composite* -- a curly-bracketed, comma-separated, possibly-empty, list of *term*s. This is regarded as syntactic sugar for a prefix composite with functor {}. For example:

   {} is sugar for {}()
   {foo, bar} is sugar for {}(foo, bar)

10. a *notfix operator* followed by a *term*. The only built-in notfix operator is not, but others may be declared using the #notfix directive (Table 1). *Notfix operator*s have less priority than any other forms of operator. For example

```
not male(X)
not append(X, Y, Z)
not not contradiction(X)
```

With directives #notfix "let", and #infix3 "in"

```
let x=3 in x+2
```

yields the formula whose structure is

```
let(in((=)(x, 3), (+)(x, 2)))
```

11. a *prefix operator* followed by a *term*. There are no built-in prefix operators, but they may be declared using the #prefix directive (Table 1). *Prefix operator*s have higher priority than infix operators. For example, after the directives #prefix "factor" and #infix3 "or"

```
factor Y or magic(Y, Z)
```

Built-In Relations

The relations and symbols built-in to *LogPro* are described in Table 2.

Changes

V1.44 (a) The syntax of terms has been extended to accomodate "fat brackets".
(b) Different primary and secondary prompts (-- and >> respectively)

Typeset on August 14, 2019

Relation	Arguments	Effect (Restriction)
`!`		the cut symbol
`fail`		always fails
`sum`	(A, B, C)	C is the sum of A and B. (No more than one variable)
`prod`	(A, B, C)	C is the product of A and B. (No more than one variable)
`succ`	(A, B)	B is the natural number successor to A. (No more than one variable)
`>=`	(A, B)	$A \geq B$. (No variables).
`>`	(A, B)	$A > B$. (No variables).
`<=`	(A, B)	$A \leq B$. (No variables).
`<`	(A, B)	$A < B$. (No variables).
`num`	(A)	A is an integer.
`str`	(A)	A is a string.
`atom`	(A)	A is an atom.
`var`	(A)	A is an uninstantiated variable.
`nonvar`	(A)	A is not an uninstantiated variable.
`toString`	(A, B)	B is the string representing the term A.
`toFormula`	(A, B)	B is the formula represented by the string A.
`functor`	(A, OP, B)	A is the formula with principal connective OP and list of arguments B.
`call`	$(A, B, ...)$	The formulae $A, B, ...$ are evaluated in sequence.
`len`	(A, B)	B is the length of the string A.
`cat`	(A, B, C)	C is the catenation of the strings A and B (No more than one variable).
`hd`	(A, B)	B is a string consisting of the first character of the string A (ditto).
`tl`	(A, B)	B consists of all but the first character of the string A (ditto).
`ascii`	(A, B)	B is the ascii code of the first character of the string A (ditto).
`read`	(A)	The string A is the next line read from the terminal.
`show`	$(A, B, ...)$	The terms $A, B, ...$ are printed on the terminal followed by a newline.
`write`	$(A, B, ...)$	The terms $A, B, ...$ are printed on the terminal.
The following relations manipulate the built-in updateable mapping from terms to terms		
`map_Lookup`	(A, B)	B is the current value of the mapping at key A.
`map_Add`	(A, B)	The value of the mapping at key A is overlaid by B.
`map_Enter`	(A, B)	The value of the mapping at key A becomes B; previous values removed.
`map_Remove`	(A)	Remove the topmost value of the mapping at key A.
`map_All`	(A, B)	B is the list of values of the mapping at key A.

Table 2: Built-in Relations