A Sample Session.

The following example shows a complete Logos session, annotated with explanations. The definitions used are provided in **.l** files along with the Logos application program.

```
LOGOS version 0.7 (Preliminary) of January 1996
<<<
```

We will define a simple, well-known world of actors and actions. Jerry and his nephew are mice:

```
<<< mouse(jerry).
<<< mouse(jerrysnephew).
```

Tom is a cat, Spike is a dog, and we introduce a tiger:

```
<<< cat(tom).
<<< dog(spike).
<<< tiger(timmy).
```

If we ask "who is a cat?":

```
<<< cat(X)?
```

we receive the expected answer.

```
X=tom
```

If instead we ask "who is not a cat?":

```
<<< ~cat(X)?
```

we receive a less direct answer:

```
X=X WHERE X/=tom
```

meaning "X is anything other than Tom". This begins to illustrate the way Logos works. Logos computes over an open universe of terms; it does not assume that the terms presented to it so far {timmy, spike, tom, jerry, jerrysnephew} are the only terms in the universe, but that other unknown terms are also present. Thus Logos can not perform negation by the relatively simple method of calculating the set of terms that satisfy a predicate, then subtracting them from the set of all terms.

Instead, Logos computes new rules that define negated versions of all the predicates. These new rules may be seen in the output of the "show" command:

```
<<< show(cat)?
cat(tom).
~cat(V1) where V1/=tom.

<<< show(mouse)?
mouse(jerry).
mouse(jerrysnephew).
~mouse(V2) where (V2/=jerry /\ V2/=jerrysnephew).
```

Logos performs a great deal of preprocessing to reconstruct the rules entered by the user into a more suitable form. Along with constructing negated versions of all predicates, it also converts all predicates into a purely positive form, in which the negation operation never appears (after this preprocessing, "~mouse" is a new predicate symbol; it is not the "~" operation applied to the "mouse" predicate). In addition to this, it also converts rules for a single predicate into a disjoint and complete form. This process is illustrated by the following sub-example:

```
p(f(k)) :- q(k).
p(f(X)) :- qq(X).
```

These two rules are converted into:

```
p(f(k)) ← q(k) ∨ qq(k).
p(f(X)) where X/=k ← qq(X).
p(A) where A isnt f ← false.
```

From these three, the definition of ~p is derived:

```
~p(f(k)) ← ~q(k) ∧ ~qq(k).
~p(f(X)) where X/=k ← ~qq(X).
~p(A) where A isnt f ← true.
```

Finally the two sets of rules are simplified, to produce the final form listed by show(p):

```
p(f(k)) <- q(k).
p(f(k)) <- qq(k).
p(f(X)) where X/=k <- qq(X).
~p(f(k)) <- ~q(k) /\ ~qq(k).
~p(f(X)) where X/=k <- ~qq(X).
~p(A) where A isnt f.
```

Now the rules of behaviour are defined: cats eat mice, and tigers eat everything but tigers. The symbol "/\" represents the logical **and** operation as opposed to Prolog's sequential **and** operation, which is also available (using the traditional comma symbol):

```
<<< eats(X,Y) :- cat(X) /\ mouse(Y).
<<< eats(X,Y) :- tiger(X) /\ ~tiger(Y).
```

Animals chase one-another either for food:

```
<<< chases(X,Y) :- eats(X,Y).
```

or for some other reason:

```
<<< chases(X,Y) :- dog(X) /\ cat(Y).
```

So now we can request some deductions:

```
<<< chases(tom,jerry)?      Does Tom chase Jerry?
TRUE

<<< eats(tom,X)?            Who does Tom chase?
X=jerry
X=jerrysnephew

<<< chases(X,jerry)?        Who chases Jerry?
X=tom
X=timmy

<<< chases(X,Y)?            Who chases whom?
X=spike /\ Y=tom
X=tom /\ Y=jerry
X=tom /\ Y=jerrysnephew
X=timmy /\ Y=Y where Y/=timmy
```

This final answer means that Timmy, being a tiger, chases everyone but himself. Remember that Logos can not explicitly list all of the things that Timmy chases, due to the open universe assumption.

There is no “list of all things”, so a more expressive logical formula must be produced instead.

We can design a small program to generate all interesting forms of activity, and write them out.

```
<<< action :- eats(X,Y), write(X,eats,Y,nl)  \/
  <           chases(X,Y), write(X,chases,Y,nl).
```

In this definition, the sequential and operation (comma) must be used instead of the logical and, so that the predicates `eats` and `chases` will be sure to have had an opportunity to instantiate the variables X and Y before they are printed.

```
<<< action?
spike chases tom
timmy eats V3 WHERE V3/=timmy
tom eats jerrysnephew
tom eats jerry
timmy chases V4 WHERE V4/=timmy
tom chases jerry
tom chases jerrysnephew
TRUE
```

Now for some more complex rules.

An animal is edible if there is some other animal that is interested in eating it:

```
<<< edible(X) :- (exist Y eats(Y,X)).
```

The parentheses around the whole existential formula are required so that the scope of any quantifier is always obvious. Prolog can handle existential quantifiers; they are used implicitly whenever a new variable is introduced after the “:-”.

```
<<< edible(X)?
X=jerrysnephew
X=jerry
X=X where X/=timmy
```

After discovering that Jerry and Jerrysnephew are both edible, the system finds a more general rule that everything other than Timmy is edible. It would be possible to implement Logos in such a way that only the most general answers are printed, but that would require waiting until all answers have been found before printing any. In an interactive system, or for a program that produces an unending stream of answers, this is not suitable.

Any animal that eats everything but itself is considered to be fierce:

```
<<< fierce(X) :- (all Y  X/=Y -> eats(Y,X)).
```

Universal quantification is beyond the scope of Prolog. Logos performs the operation through logical manipulations of the defining formulae; due to the “open universe” assumption it is not possible to simply test all known terms.

```
<<< fierce(X)?
X=timmy
```

Logos is capable of complex computations involving the quantifiers. For instance, All animals are either fierce or edible:

```
<<< (all X  fierce(X) \/ edible(X))?
TRUE
```

No animals are both fierce and edible:

```
<<< ~(exist X  fierce(X) /\ edible(X))?
TRUE
```

List processing provides another demonstration.

The syntax for lists follows Prolog's exactly: individual elements at the head of a list are separated by commas; the one term after the vertical bar represents the whole of the rest of the list. This is illustrated by asking Logos to unify an explicit list with one containing variables:

```
<<< [1,2,3,4,5,6]=[A,B,C|D]?
A=1 ∧ B=2 ∧ C=3 ∧ D=[4,5,6]
```

An item is a member of a list if it is the same as the first element of the list, or if it is a member of the rest of the list:

```
<<< member(X, [X|Rest]).
<<< member(X, [Y|Rest]) :- member(X, Rest).
<<< member(2, [1,2,3])?
TRUE
<<< member(f(X), [f(a),g(b),f(c)]?
X=a
X=c
<<< member(X, [1,2,3])?
X=1
X=2
X=3
```

`append(A,B,C)` should be true if the list `C` results from appending the two lists `A` and `B`. Naturally, appending an empty list to something has no effect:

```
<<< append([ ], L, L).
```

and if L3 is obtained by appending L1 and L2, then L3-with-E-added can be obtained by appending L1-with-E-added and L2:

```
<<< append([E|L1], L2, [E|L3]) :- append(L1, L2, L3).
```

Now some queries illustrate some of the possibilities:

```
<<< append([1,2], [3,4], [1,2,3,4])?
TRUE
<<< append([1,2], [3,4], X)?
X=[1,2,3,4]
<<< append([1,X], [Y,4], [1,2,3,Z])?
X=2 ∧ Y=3 ∧ Z=4
<<< append(X, Y, [1,2,3])?
X=[ ] ∧ Y=[1,2,3]
X=[1] ∧ Y=[2,3]
X=[1,2] ∧ Y=[3]
```

```
X=[1,2,3] ∧ Y=[]
```

All of the above examples are well within the scope of Prolog. Logos can go further, providing some simple theorem proving:

```
<<< (all X,Y append([X], [Y], [X,Y]))?
TRUE
<<< (exist X,Y append([X], [Y], [Y,X]))?
TRUE
<<< (exist X,Y append([X], [Y], [Y,X]) ∧ X/=Y)?
FALSE
```

Basic Syntax

A **name** consists of a lower-case letter followed by any combination of letters and digits; underlines may be used between other characters.

A **variable** consists of a capital letter followed by any combination of letters and digits; underlines may be used between other characters.

Names of predicates and functions (the constants that appear in terms) are indistinguishable except by their context. Names may include an explicit "arity" as in the example `p/2`. In any context where the "arity" can not be deduced, it must be stated explicitly.

A **term** consists of a variable, an integer, or a function name optionally followed by a list of arguments separated by commas, in parentheses. The arguments are terms. Examples: `k, cat, f(k), X, f(X)`. A term may also consist of a list of terms in square brackets, separated by commas. Example: `[1,k,f(X),cat]`.

An **atomic formula** consists of a predicate name optionally followed by a list of arguments separated by commas, in parentheses. The arguments are terms. Examples: `p, p(k), p(X,Y)`.

A **formula** is constructed from atomic formulae by applying operators. The common operators are, in order or precedence:

~	Negation
, and /\	Sequential and parallel conjunction
; and \/	Sequential and parallel disjunction
-> and <-	Implications
<->	Equivalence

Parentheses may be used in the normal way, to over-ride operator precedences.

Formulae may also be constructed by applying special operators to terms. The most important operators are:

=	The two operands are unified
:=	The second operand should be an expression; it is evaluated and unified with the first operand, normally a variable.
/=	The two operands are unified, and the result negated.
< =< >= >	The standard arithmetic comparisons
is	Second operand should be a function name; result is true if

first operand is that function applied to some arguments.

`isnt` The result of `is`, but negated.

A formula may also be constructed from one of the **quantifiers**, `all` and `exist`. The syntax for a quantified formula is `(` *quantifier variable-list formula* `)`, Examples; `(all X p(X))`, `(exist X,Y p(X,Y) /\ q(X))`.

An **expression** is a special kind of term, consisting of integers and variables combined with the arithmetic operators +, –, *, /, and parentheses.

A **rule**, contributing to the definition of a predicate, has the syntax:

head `where` *guard* `:-` *body*`.`

the head is in the syntax of an atomic formula, the guard and body are both general formulae. If the guard is true, the "`where` *guard*" portion may be omitted; if the body is true, the "`:-` *body*" portion may be omitted. The guard provides a constraint on the applicability of a rule, so the rule "`p(X) where X/=k:- q(X).`" is not the same as the rule "`p(X):- X/=k /\ q(X).`"; the difference becomes important when negative rules are constructed.

A **query** consists of any formula followed by a question-mark.

Commands

The commands available are listed below; they may be typed at any time to the **<<<** prompt, and should be terminated by a "**.**".

exit.

Exit from Logos.

show.

List all user-defined predicate names.

show(predicate-name**).**

Display the definition of a user-defined predicate.

help.

Displays a list of all commands and built-in predicates, with a brief description.

help(name**).**

Provides a more complete description (if available) of a command or built-in predicate.

clear.

Erases the definitions of all user-defined predicates.

load(name**).**

Load a Logos program file called **name.l** from the directory (or folder) from which the Logos application was launched.

selection(sequential).

Sets the rule-selection semantics for subsequently entered definitions to "sequential". A typical predicate's definition is split into a number of lines, for example:

```
p(X) :- q(a,X).
p(X) :- q(b,X), q(c,X).
```

Such a definition is considered to be a disjunction of the individual parts. As Logos works with both sequential (Prolog-like) disjunction and parallel (logical) connectives, the **selection(sequential)** command is used to specify the sequential disjunctions are to be assumed for any definitions being entered.

selection(parallel).

Sets the rule-selection semantics for subsequently entered definitions to "parallel" (see the above description of sequential selection). Example:

Given this input:
```
selection(sequential).
p(a).
p(b).
p(c).
selection(parallel).
q(a).
q(b).
q(c).
```
The query:
```
p(X)?
```
always produces the solutions `X=a`, `X=b`, and `X=c` in that order, whereas the query:
```
q(X)?
```
will produce those same solutions, but in an unpredictable order.

simplify.

Extra simplification transformations are applied to all output produced by the next query. These transformations could be computationally expensive.

echolines.

All subsequent lines will be echoed exactly as they are read. **echolines** is cancelled when and end-of-file is reached, or when another **echolines** command is entered.

silent.

The single immediately following query will be executed silently: the final answer is not printed, and non-essential preprocessor information is suppressed. **write** predicates still produce output as normal.

new.

All rules defining predicates are marked as "old". This happens automatically whenever a **load** command starts or finishes.

When a new rule is entered, for a predicate that has already been defined by earlier rules, the effect depends upon the age of the older rules. If they are old, they are discarded, and the new rule begins a redefinition of the predicate. If they are new, the new rule as simply added to them. This allows a file to be re-read after editing, to redefine predicates, but does not interfere with normal experimental rule construction.

positiveonly.

The one next predicate to be defined, is marked as "positive only".

Before execution of a query or command, all rules (and the query itself) are subject to a complex preprocessing procedure. During preprocessing, the rules that defines a predicate are modified to make them disjoint, for example:
```
p(a,Y) :- q(Y).
p(X,b) :- r(X).
```
are converted to the following

```
p(a,Y) where Y/=b :- q(Y).
p(X,b) where X/=a :- r(X).
p(a,b) :- r(a) /\ q(b).
```

and an extra rule is added to complete the definition

```
p(X,Y) where X/=a /\ Y/=b :- false.
```

Rules in this form allow the negative version of the predicate to be correctly constructed; they also make execution of queries more efficient in some situations. As a final step in preprocessing, the rules are somewhat simplified, and those ending with "`:- false`" are discarded.

For predicates with many defining rules, preprocessing can be very time consuming (especially in this experimental implementation). For this reason, those predicates that never appear in their negative forms may be marked as **positiveonly.**

Example:

```
positiveonly.
parent_child(alf,ben).          parent_child(cal,hen).
parent_child(alf,cal).          parent_child(dan,ida).
parent_child(ben,dan).          parent_child(elf,jane).
parent_child(ben,elf).          parent_child(elf,ken).
parent_child(cal,fran).         parent_child(fran,log).
parent_child(cal,gab).          parent_child(gab,mab).
ancestor_descendant(A,D):-parent_child(A,D).
ancestor_descendant(A,D):-(exist Mid
                           parent_child(A,Mid),
                           ancestor_descendant(Mid,D))
.
ancestor_descendant(X,ken)?
    X=alf
    X=elf
    X=ben
```

The preprocessing for the `parent_child` rules would be prohibitively time-consuming without the **positiveonly** declaration.

Built-in Predicates.

The following predicates may be used in any context in a Logos program. They behave logically as any predicate should, in that they either succeed or fail, and may instantiate any variables in their arguments. They are provided because they perform functions that would be difficult or impossible to program otherwise.

read(arguments**)**

Logos provides a pure logical form of input. When a **read** predicate is executed, the input prompt ("? ") is displayed, and program execution pauses until some input is entered; the input may be any list of terms, separated by commas, and terminated by a period. The list of input terms is unified with the list of arguments to produce the result. Both the arguments and the typed input may contain variables (either already known variables, or new ones), and the input may also be restricted by a constraint using a **where** clause. Examples:

```
<<< read(X,Y)?
```

```
? 1,2.
X=1 ∧ Y=2
<<< read(X)?
? Y.
X=Y
<<< read(s(X))?
? s(s(a)).
X=s(a)
<<< read(s(X))?
? Z.
X=X ∧ Z=s(X)
<<< read(k)?
? k.
TRUE
<<< read(k)?
? X.
X=k
<<< read(k)?
? X where X/=k.
FALSE
```

write(arguments**)**

The values of the arguments are printed in order. The term **nl** is interpreted specially to cause a new line of output to begin. If any of the argument terms contain variables, the output will include any constraints that apply to those variables.

Examples:

```
<<< X=6, write('X is ',X,nl)?
X is 6
<<< X=k, Y=f(Z), Z/=k, write(X,nl,Y,nl,Z,nl)?
k
f(Z) WHERE Y=f(Z) ∧ Z/=k
Z WHERE Y=f(Z) ∧ Z/=k
```

readline(X**)**

When executed, the input prompt ("? ") is displayed, and program execution pauses until some input is entered (the input is terminated by a period "." or question-mark "?"). The input is split up into individual symbols, which are composed into a list. The list is unified with the argument X, which should be an uninstantiated variable.

Example:

```
<<< readline(X)?
? the cat sat on the mat.
X=[the,cat,sat,on,the,mat,.]
```

equalsdotdot(A**,**B**)**

Performs the function of Prolog's **=..** operator, converting between complex terms and lists of less complex terms. Example:

```
<<< equalsdotdot(X, [f,a,b])?
X=f(a,b)
<<< equalsdotdot(f(a,b), Y)?
Y=[f,a,b]
```

```
<<< equalsdotdot(f(X,Y), [A,B,k])?
X=X ∧ Y=k ∧ A=f ∧ B=X
```

assert(term**)**

Similar to Prolog's assert; the term is translated into a predicate with an identical syntactic form, and that predicate is accepted as a new rule. Asserted predicates are considered to be dynamic, and predicates entered in the normal way are considered to be static. It is inadvisable to have static and dynamic predicates with the same names. Example:

```
<<< assert(p(k))?
TRUE
<<< write('who is a cat'), read(X),
                equalsdotdot(P, [cat,X]),
                assert(P)?
who is a cat ? tom.
X=tom ∧ P=cat(tom)
<<< cat(C)?
C=tom
```

call(term**)**

Similar to Prolog's call; the term is translated into a predicate with an identical syntactic form, and that predicate is executed in the normal way. Example:

```
<<< mouse(jerry).
<<< call(mouse(X))?
X=jerry
```

defined(name**)**

Succeeds if a definition for the named predicate exists. If the name does not specify an explicit "arity", it will detect any definition with the same base name. Example:

```
<<< p(a).
<<< q(a,b).
<<< defined(p/1)?
TRUE
<<< defined(p)?
TRUE
<<<defined(X)?
X=p
X=q
```