Intelligent Information Systems,
Proceedings of the Workshop held in Deblin, Poland,
Polish Academy of Sciences, 1996, ISBN 83-900820-6-3

Programming in Logos[1]

S. Murrell and Jan A. Plaza

University of Miami Department of Mathematics and Computer Science
P.O. Box 249 085
Coral Gables, Florida 33124
U.S.A.
http://www.cs.miami.edu/~janplaza/logicprogramming

Abstract. We present a new constraint logic programming language, Logos, which has an intuitive declarative semantics. Logos computes over an open universe of terms, and returns as answers substitutions constrained by formulas involving disequalities. Logos allows all operators of classical logic including classical negation, existential and universal quantifiers and equality. Moreover it has fully logical input and output facilities. (The final version of Logos is expected to include meta-programming, and have facilities for selection of control which distinguish between sequential and parallel conjunctions of goals.)
Logos is meant as a high level programming language for rapid prototyping in the area of symbolic computation and for construction of rule-based expert systems with logical consistency constraints.
A preliminary version, Logos 0.7 is available via www, both as a Macintosh binary file and a portable source code in Common Lisp.

Keywords: constraint logic programming, declarative programming, symbolic computation, rule-based expert systems.

1 Introduction

It has long been accepted that logic programming could provide a convenient basis for the implementation of rule-based expert systems. The basic concepts are the same: logic programs are collections of rules which specify conditions that must be met for new facts to be deduced and actions to be taken; Prolog (and other logic programming languages) normally execute programs through a kind of backward chaining strategy; the predicates from which definitions in logic programming are constructed, describe exactly the same kinds of operations as the conditions and actions from which rule-based systems are constructed.

However, logic programming so far failed to establish itself as a serious tool for work in this area; very few expert systems are implemented in logic programming languages. This is mainly because the archetypal logic programming language, Prolog, does not implement logic programming, but instead, it is a procedural programming language with operations which only resemble those of classical logic. There are also newer alternatives to Prolog, but all fail to some extent to provide the necessary full logic programming environment.

The advantages of logic programming for rule-based expert systems become apparent if we consider the following simple example:

```
cheese(X) :- colour(X,yellow) /\ state(X,solid).
cheese(X) :- colour(X,blue) /\ smelly(X) /\ state(X,solid).
cheese(X) :- colour(X,white) /\ state(X,runny) /\ state(X,solid).
wine(X) :- ( colour(X,white) \/ colour(X,red) \/ colour(X,pink) )
           /\ state(X,liquid).
item(X) :- X=brie \/ X=gorgonzola \/ X=cheddar \/ X=chablis \/
           X=port \/ X=burgundy \/ X=camembert \/ X=sauternes \/
           X=sally_smith \/ X=joe_lyons \/ X=mr_banjo \/
           X=mrs_banjo.
colour(cheddar,yellow).
colour(port,red).
colour(chablis,white).
state(port,liquid).
state(cheddar,solid).
state(chablis,liquid).
state(joe_lyons,solid).
```

With this knowledge base and database, any logic programming language would be able to make correct deductions; the queries `wine(port), cheese(cheddar), and cheese(port)` would all produce the correct responses (`yes, yes, no`). Even the generative query `wine(X)` would produce under Prolog the correct list of answers (`X=port; X=chablis`).

As we see from this example, while using a logic programming language, the implementation of expert systems could be a simple, error-free process; expert system rules expressed in a very natural intuitive way could be directly executed as programs in a logic programming language. The whole error-prone coding phase would be eliminated, and debugging would be a matter of correcting errors in the original rule specifications, not a complex programming task.

However there is a problem: under currently existing Prolog-like languages, it would not be possible to make the rule base more sophisticated than this without using non-declarative methods. For instance, we might want to add a rule saying that anything solid other than cheese, is a guest:

```
guest(X) :- not cheese(X) /\ state(X,solid).
```

In Prolog-like languages, this would work to a certain extent: specific queries, such as `guest(joe-lyons)` would produce the correct response, but generative queries, such as `guest(X)` would not.

This is because these languages do not implement 'proper' logical negation, but negation as failure, under which the sub-goal not `cheese(X)` is solved by attempting to solve `cheese(X)` and converting success to false, and failure to true. This scheme is adequate when `X` is already known, but does nothing to find a possible value for `X` otherwise. This weakness, which effectively means that negation is not provided, severely restricts the utility of logic programming for expert system implementations.

Another important logic operation, especially for verification and validation, is universal quantification. The universal quantifier, allows automatic consistency and validation checks to be included in the database of the expert system. For example, we may wish to specify that every item at the party is either cheese, wine, or a guest:

```
(all X item(X) -> (cheese(X) \/ wine(X) \/ guest(X))
```

or we may wish to demand that nothing is both cheese and wine:

```
(all X item(X) -> not(cheese(X) /\ wine(X))).
```

If either of these queries answers false, meaning that the conditions are not satisfied, it is of course important to be able to find the cause of the problem: `exist X item(X) / cheese(X) / wine(X) / write("failure: ",X,nl)`. These operations are outside the scope of traditional Prolog-like languages, which makes the tasks of verification and validation unnecessarily difficult to automate.

Another important consideration is the simplicity of the semantic model. In an ideal situation, the semantic (or meaning) function should be declarative i.e. obey rules of the following sort.

```
M(x /\ y) = M(x) /\ M(y)
M(x \/ y) = M(x) \/ M(y)
M(not x) = ~M(x)
```

meaning that the implementations of logical operators should behave in the way that the logical operators themselves are expected to behave in normal use. If these rules are obeyed, logic programming becomes a natural task, requiring no particular programming expertise, and demanding no special care to avoid the unexpected situations that would otherwise occur. To illustrate, consider the following rules:

```
ancestor(X,Z) :- ancestor(X,Y) /\ ancestor(Y,Z).
ancestor(X,Z) :- parent(X,Z).
```

Logically, they do correctly define the ancestor-descendant relationship, but under the semantics of most logic programming languages (strictly top-to-bottom, left-to-right evaluation) would not deliver any results, but instead enter an infinite loop in response to any ancestor related query. The presence of such inconsistencies makes the direct and transparent implementation of expert systems in logic programming languages impossible, requiring an additional, complex translation phase in which the natural design of the expert system is forced to conform to the unnatural semantics of the implementation language.

In this paper we introduce a new programming language, Logos, which solves these problems. Preliminary version, Logos 0.7 is available on www. It is an experimental implementation designed for ease of experimentation, not for speed. It relies on a text based user interface under which the user types commands, programs, or queries, and the system responds interactively.

2 A Sample Session

When Logos is able to accept input, the standard prompt is `<<<` (if the previous line was not complete, this is replaced by the continuation prompt `<`). Commands, programs (i.e. definitions of predicates), and queries may be entered at any time that the prompt is shown. Commands and definitions must be terminated by a full-stop or period (.), and queries must be terminated by a question-mark (?). Programs may by entered interactively, or loaded from a text file as explained in the next section. Logos program files have names ending in .l (Logos source); programs are preprocessed and saved in files with extensions .ql (Quick Load) which can be loaded and executed next time without preprocessing; for the purposes of documentation, a readable version of preprocessed program is saved in a file with extension .pp (Pretty Printed).

The following example shows a complete Logos session, annotated with explanations.

```
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 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 or in .pp files:

```
<<< 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, e.g. `~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 behavior are defined: cats eat mice, and tigers eat everything but tigers. The symbol $\wedge$ represents the logical and-operation as opposed to Prolog's sequential and-operation represented by a comma, (which is also available in Logos).

```
<<< 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 operation $\wedge$, 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 by `write`.

```
<<< 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. Notice also that universal quantifiers defined in terms of existential quantifiers and negation will not behave correctly in Prolog because the negaation does not behave correctly. Logos performs the universal quantification 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=[]
```

These examples with `append` 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
```

3 Basic Syntax

A *variable* consists of a capital letter followed by any combination of letters and digits; underlines may be used between other characters. *A name* consists of a lower-case letter followed by any combination of letters and digits; underlines may be used between other characters. *Predicate names* and *function names* are arbitrary names. 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. Function names of arity 0 are called *constants. A term* is a variable, or an integer, or a function name possibly followed by a list of terms separated by commas and in parentheses; examples of terms: `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 possibly 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 $\wedge$ – *Sequential and parallel conjunctions*
`;` and $\vee$ – *Sequential and parallel disjunctions*
`->` 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 is evaluated and unified with the first operand.
`/=` – Test whether the operands are not unifiable.
`< =< >= >` – 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 body* is a formula, *the guard* is a formula containing no predicates except `=`. If the guard is *true*, the keyword `where` and the guard may be omitted; if the body is `true`, the symbol `:-` and the body 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.

3.1 Commands

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

`exit.` Exit from Logos.

`show.` List all user-defined predicate names.

`show(predicate-name).` Display the preprocessed 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`

`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).

```
selection(sequential).
p(a).
p(b).
selection(parallel).
q(a).
q(b).
```

The query: `p(X)?` always produces the solutions `X=a, X=b,` 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.

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 define 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:

```
positiveonly.
parent_child(alf,ben).        parent_child(alf,cal).
parent_child(ben,dan).        parent_child(ben,elf).
parent_child(cal,fran).       parent_child(cal,gab).
parent_child(cal,hen).        parent_child(dan,ida).
parent_child(elf,jane).       parent_child(elf,ken).
parent_child(fran,log).       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.

3.2 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 (free variables of the query), or new ones, and the input may also be restricted by a constrained 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(=8CX is =8C,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
```

`stop` interrupts the execution of a Logos program.

References

1. K. R. Apt, Introduction to Logic Programming, in J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, North Holland, 1989.
2. K. R. Apt and R. N. Bol, Logic Programming And Negation: A Survey, *Journal of Logic Programming* Vol.19/20, pp. 9-71, 1994.
3. J. W. Lloyd, *Foundations of Logic Programming*, second extended edition, Springer, 1987.
4. J. A. Plaza and S. Murrell, Foundations of Logic Programming over an Open Universe of Term, I: Fix Points, submitted for publication.
5. J. A. Plaza and S. Murrell, Foundations of Logic Programming over an Open Universe of Terms II: Theory of Equality, submitted for publication.
6. J. A. Plaza and S. Murrell, Foundations of Logic Programming over an Open Universe of Terms III: Resolution System, submitted for publication.
7. L. Sterling and E. Shapiro, *The Art of PROLOG*, MIT Press, 1986.