

Logic Programming II: Applications

This is the second of two columns on logic programming. It presents examples of typical logic programming applications using the query evaluator developed in the first column. Among the examples considered are implementations of Prolog-style lists, nondeterministic automata, backtracking and exhaustive search, theorem proving, and deductive databases.

Roman E. Maeder

Lists in Prolog

Prolog uses the same representation of lists as Lisp. A list consists of a first element, the *head*, and the rest of the elements, the *tail*. A typical pattern in Prolog is $[H|T]$, standing for the list with first element H and rest T . In *Mathematica*, we would write this pattern as $\{h_, t_ \}$ and then use $\{t\}$ for the rest (t itself is only the *sequence* of elements, so we have to wrap it in a list). Because our simple unifier cannot handle sequences, we use Lisp lists, imported from the package `Lisp.m` (developed in Volume 2, issue 3 of this *Journal*).

In this notation, we use `cons[h_, t_]` for the list with first element h and rest t ; `car[l]` is the first element of l , and `cdr[l]` is the rest of l . `nil` is the empty list. `list[e1, e2, ..., en]` generates the list $(e_1 e_2 \dots e_n)$.

Here is a predicate that expresses that a list is ordered.

```
Assert[ ordered[List[] ]
Assert[ ordered[List[_] ]
Assert[ ordered[cons[e_, r_]], e_ < car[r_], ordered[r_] ]
```

The empty list is always ordered, as is any list with one element. Otherwise, if the list has at least two elements, it is ordered if its first element is smaller than the first element of its rest and the rest is also ordered.

In *Mathematica* and in Lisp, there is a function `Join[list1, list2]` that joins two lists. In Prolog, such functions are expressed as predicates, where the desired result is an additional argument. Therefore, the predicate `join` will have three arguments. The predicate `join[list1, list2, list]` expresses: “the join of $list_1$ and $list_2$ is $list$.” Let us find rules that characterize this predicate.

```
In[1]:= << LogicProgramming.m
```

The join of the empty list and l is l :

```
In[2]:= Assert[ join[ nil, l_, l_ ] ]
```

The join of the list with first element e and rest r and a list l is the list with the same first element e and rest s , provided that the join of r and l is equal to s :

Roman Maeder is one of the designers of Mathematica and the author of three books on Mathematica-related topics. He is now a professor of computer science at the Swiss Federal Institute of Technology (ETH) in Zürich, Switzerland.

```
In[3]:= Assert[ join[ cons[e_, r_], l_, cons[e_, s_] ],
                join[ r_, l_, s_ ] ]
```

That’s all! We haven’t really written a program to compute the join of two lists. But our predicate works and it can do more than compute the join of two lists.

This query computes the join of $(a b)$ and $(c d)$ and “returns” the result as a binding for the one variable present. It resembles an ordinary function call `res = join[(a b), (c d)]`.

```
In[4]:= Query[ join[List[a, b], list[c, d], res_] ]
Out[4]= {res -> (a b c d)}
```

This query asks the question: “which list, when joined to $(a b)$, gives $(a b c d)$?”

```
In[5]:= Query[ join[List[a, b], l2_, list[a, b, c, d]] ]
Out[5]= {l2 -> (c d)}
```

This query finds all possible ways to join two lists to give $(a b c d)$:

```
In[6]:= QueryAll[ join[l1_, l2_, list[a, b, c, d]] ]
{l1 -> (), l2 -> (a b c d)}
{l1 -> (a), l2 -> (b c d)}
{l1 -> (a b), l2 -> (c d)}
{l1 -> (a b c), l2 -> (d)}
{l1 -> (a b c d), l2 -> ()}
```

As you can see, there is no distinction between input and output parameters. Not all Prolog programs are that flexible, but it is a good idea to try to write code that can be used this way, if possible. Often, arithmetic inequalities prevent the generation of all possibilities. In our first example (the predicate `ordered`), we unfortunately cannot use `Query[ordered[l_]]` to enumerate all sorted lists. There is an infinite number of them, anyway.

Here is one last list example. We define a predicate `member[e, l]` that says “ e is a member of list l .”

The element e is a member of the list whose first element is e :

```
In[7]:= Assert[ member[e_, cons[e_, _]] ]
```

The element e is a member of the list with rest r , provided it is an element of r :

```
In[8]:= Assert[ member[e_, cons[_ , r_]], member[e_, r_] ]
```

We find that a is indeed a member of $(a\ b\ c\ a)$:

```
In[9]:= Query[ member[a, list[a, b, c, a]] ]
Out[9]= Yes
```

It should now come as no surprise that we can invert the question to find an element that is a member of $(a\ b\ c\ a)$:

```
In[10]:= Query[ member[e_, list[a, b, c, a]] ]
Out[10]= {e -> a}
```

Our program has a bug: Elements that occur more than once are listed more than once:

```
In[11]:= QueryAll[ member[e_, list[a, b, c, a]] ]
      {e -> a}
      {e -> b}
      {e -> c}
      {e -> a}
```

Here is a way to correct this problem. The second rule succeeds only if the candidate element is different from the first element of the list. Note that the inequality must come last.

```
In[12]:= Assert[ member2[x_, cons[x_, _]] ];\
      Assert[ member2[x_, cons[y_, _]],
              member2[x_, _, x_ != y_ ]
```

Each element that appears is now listed only once:

```
In[13]:= QueryAll[ member2[e_, list[a, b, c, a]] ]
      {e -> a}
      {e -> b}
      {e -> c}
```

If we give up the requirement that each argument can be used for input or output, we can find a simpler definition to test membership by using a cut as soon as an element has been found:

```
In[14]:= Assert[ member1[x_, cons[x_, _]], cut ];\
      Assert[ member1[x_, cons[_ , _]], member1[x_, _] ]
```

Nondeterministic Automata

A nice application of backtracking is the implementation of nondeterminism. In ordinary programs, the next instruction to execute is always uniquely determined. In logic programming, we can leave open several possibilities and try them out in some unspecified order. This makes it easy to model nondeterministic processes, for example a *nondeterministic finite automaton*.

A finite automaton (FSA) consists of a number of *states*, usually represented as the vertices of a graph. *State transitions* lead from one state to another one, represented as arcs between states. A state transition is triggered by an input symbol. Each arc is therefore labeled by a symbol and only the transition labeled with the current input symbol can be taken. The current input symbol is “used up” by the transition and then the next one is looked at. When the input is exhausted, the automaton halts. A subset of states is designated as *final*. If the automaton halts in a final state, it *accepts* the input; otherwise, it *rejects* it. If no arc labeled with the current input symbol exists in the current state, the computation also terminates and rejects. This automaton is just like a Turing Machine without the tape. (See the article in Volume 3, issue 3 of this *Journal*.)

An automaton is *deterministic* if there is at most one arc with each label starting at each state. In this case, there is never a choice of which transition to take; at most one transition is possible. If this property is not satisfied, the automaton is nondeterministic. Such automata may also allow a different kind of transition: *silent transitions*, which are not labeled and can be taken without using up an input symbol. A nondeterministic automaton accepts if there is at least one computation that halts in a final state. Since there can be several choices for the continuation of a computation, there can be several states in which the automaton halts for the same input. Since we must answer the question whether *one* accepting computation exists, we have to perform an exhaustive search through all possible computations. Prolog can do this with a minimum of programming; we need only three rules!

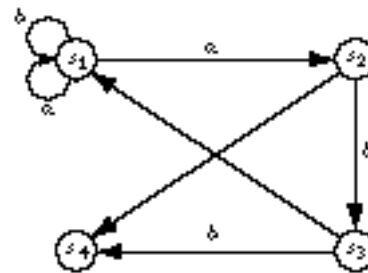


FIGURE 1. A small nondeterministic finite automaton

An automaton is represented as a number of facts that describe the possible transitions and silent transitions. The predicate $trans[s_1, i, s_2]$ describes a transition from state s_1 to state s_2 labeled with symbol i , $silent[s_1, s_2]$ describes a silent transition from state s_1 to state s_2 , and $final[s]$ designates state s as final. The automaton from Figure 1 is coded like this:

```
Assert[ trans[s1, a, s1] ]
Assert[ trans[s1, a, s2] ]
Assert[ trans[s1, b, s1] ]
Assert[ trans[s2, b, s3] ]
Assert[ trans[s3, b, s4] ]

Assert[ silent[s2, s4] ]
Assert[ silent[s3, s1] ]

Assert[ final[s3] ]
```

Now, we want to define the predicate `accepts[s, input]` that says the automaton accepts the input (given as a list of symbols), starting in state s . Here are the rules:

```
Assert[ accepts[s_, nil], final[s_] ]
Assert[ accepts[s_, cons[x_, rest_]],
      trans[s_, x_, s1_], accepts[s1_, rest_] ]
Assert[ accepts[s_, input_],
      silent[s_, s1_], accepts[s1_, input_] ]
```

The first rule says that the empty input is accepted if we are in a final state. The second rule says that the input starting with symbol x is accepted if we perform a transition labeled with x leading to a new state s_1 and then accept the rest of the input in this state. The third rule handles silent transitions that do not modify the input in any way.

```
In[15]:= << FSA.m
```

This query shows that the input `aaab` is accepted from state `s1`:

```
In[16]:= Query[ accepts[s1, list[a, a, a, b]] ]
Out[16]= Yes
```

Here, we find all states from which the input `ab` is accepted:

```
In[17]:= QueryAll[ accepts[s_, list[a, b]] ]
      {s -> s1}
      {s -> s3}
```

Here are all inputs of length three that are accepted from state `s1`:

```
In[18]:= QueryAll[ accepts[s1, list[x1_, x2_, x3_]] ]
      {x1 -> a, x2 -> a, x3 -> b}
      {x1 -> b, x2 -> a, x3 -> b}
```

If we want to see the accepting inputs as lists, we can formulate the query this way:

```
In[19]:= QueryAll[ input_ == list[_, _, _], accepts[s1, input_] ]
      {input -> (a a b)}
      {input -> (b a b)}
```

This example shows how flexible logic programming is and how easy it is to solve tasks that involve searching a large number of cases.

Backtracking and Exhaustive Search

Games are a good example of backtracking and searching. In a finite two-person game with complete information, the outcome can in principle be computed by traversing the whole game tree. For interesting games, this tree is much too large for an exhaustive search. Other methods are used to approximate exhaustive search, such as static evaluation of non-final positions to cut the search depth and *alpha-beta prun-*

ing to cut the search breadth. Prolog programs for these techniques are discussed in [Bratko 1986].

Here, we show how exhaustive search is programmed for the game of NIM. This simple game can be solved by other means, but we shall ignore them here. (For an excellent description of the mathematical theory behind NIM and many other games, see [Berlekamp et al. 1982]).

The game of NIM involves two players and a number of *heaps*, each consisting of a number of *tokens*. Each player moves in turn. A move consists of taking any number (at least one) of tokens from one heap. If all tokens in a heap are taken, the heap vanishes. If there are no more heaps left, you cannot move and you lose. This means that you win if you take the last token.

We will describe a position in the game by a list of the numbers of tokens in the heaps. Thus, `(3 4 5)` describes a position consisting of three heaps with 3, 4, and 5 tokens. A move is described by the predicate `take[heap, amount]`. It describes the number of the heap from which the tokens are taken and the number of tokens removed. The predicate `play[pos1, move, pos2]` expresses the relation that `pos2` is obtained from `pos1` by performing the move `move`. Finally, `win[pos, move]` says that we can win in position `pos` by performing the move `move`. These rules are in the package `NIM.m`.

The whole strategy of the game is expressed in a single rule:

```
Assert[ win[pos_, t_], play[pos_, t_, pos1_], !win[pos1_, _] ]
```

which says that you can win in position `pos` with move `t` if it is impossible to win in the resulting position `pos1`. Since it is then the opponent's turn, this implies that you win!

The rules for `play` are a bit tricky. It would be a simple matter to find rules that could be used to compute the final position if both the initial position and the move were given, as in `play[list[3, 4, 5], take[1, 2], pos1_]`. But we have to make sure the predicate also works with other parts uninstantiated. In particular, we want to use `play[list[3, 4, 5], t_, pos1_]` to *generate* all possible moves in a given position. Here is a set of rules that works:

```
(* take all *)
Assert[ play[ cons[n_, rest_], take[1, n_], rest_ ] ]

(* take one *)
Assert[ play[ cons[n_, rest_], take[1, 1], cons[n1_, rest_]],
      n_ > 1, n1_ == n_ - 1 ]

(* one more than a move with smaller heap *)
Assert[ play[ cons[n_, rest_], take[1, m_], cons[nr_, rest_]],
      n_ > 2, n1_ == n_-1,
      play[ cons[n1_, rest_], take[1, m1_], cons[nr_, rest_]],
      m_ == m1_+1 ]

(* move in rest of heaps *)
Assert[ play[ cons[n_, rest_], take[i_, m_], cons[n_, rest1_]],
      play[ rest_, take[j_, m_], rest1_], i_ == j_+1 ]
```

The first three rules all generate or perform moves affecting the first heap. In the first rule, we take all tokens in the first

heap, which makes it disappear. In the second rule, we take one token of the first heap, which consists of at least two tokens (so it doesn't go away). In the third rule, we use recursion. We generate a move in a position in which the first heap has one token *less* than in the given position. If we then take one token *more* than in this move, we arrive at the same final position. This rule applies if there are more than two tokens in the first heap. The last rule performs the moves in the rest of the heaps. All we need to do is add one to the number of the heap from which the tokens are taken.

```
In[20]:= << NIM.m
```

This query performs a move by figuring out the position after the move:

```
In[21]:= Query[ play[list[3, 4, 5], take[1, 2], p1_ ] ]
Out[21]:= {p1 -> (1 4 5)}
```

Here is a list of all possible moves from position (1 2) and the resulting positions after the move:

```
In[22]:= QueryAll[ play[list[1, 2], t_, p_] ]
{t -> take[1, 1], p -> (2)}
{t -> take[2, 2], p -> (1)}
{t -> take[2, 1], p -> (1 1)}
```

Or, we can ask for the move that transforms (1 2) into (2):

```
In[23]:= Query[ play[list[1, 2], t_, list[2]] ]
Out[23]:= {t -> take[1, 1]}
```

Here is a list of all moves from (1 2 3) that do not remove any of the heaps completely (we ask that the new position still has three heaps):

```
In[24]:= QueryAll[ play[list[1, 2, 3], t_, list[_ , _ , _]] ]
{t -> take[2, 1]}
{t -> take[3, 1]}
{t -> take[3, 2]}
```

No move can be made from the empty position. There is no rule expressing this fact, but you can check that all rules for `play` require the initial position to be nonempty, since it is a pattern involving `cons`.

```
In[25]:= Query[ play[nil, t_, p_] ]
Out[25]:= No
```

This query shows that we can win from position (2) by taking all chips:

```
In[26]:= Query[ win[list[2], t_] ]
Out[26]:= {t -> take[1, 2]}
```

The position (1 1) is a *winning position*: If we achieve it after our move, we can win since there is no winning move for the opponent from this position:

```
In[27]:= Query[ win[list[1, 1], t_] ]
```

```
Out[27]:= No
```

This query computes the winning move and the resulting position from position (1 2 1):

```
In[28]:= Query[ pos_ == list[1, 2, 1],
                win[pos_, t_],
                play[pos_, t_, newpos_] ]
Out[28]:= {pos -> (1 2 1), t -> take[2, 2], newpos -> (1 1)}
```

There is no winning move from the new position:

```
In[29]:= Query[ win[newpos /. %, t_] ]
Out[29]:= No
```

Exhaustive search is a slow process. Even computing the winning move from position (3 4 5) takes many hours of computing time.

Theorem Proving

The theoretical foundation of the method used by Prolog for query evaluation is *resolution*, a technique developed for automatic theorem-proving. The language of theorem proving is *predicate calculus*. We shall denote predicates by variables p , q , and r . The logic operators can all be expressed in terms of and (\wedge), or (\vee), and not (\neg). For example, the implication

$$p_1 \wedge p_2 \wedge \dots \wedge p_n \rightarrow q$$

is equivalent to

$$\neg(p_1 \wedge p_2 \wedge \dots \wedge p_n) \vee q$$

and, finally, to

$$\neg p_1 \vee \neg p_2 \vee \dots \vee \neg p_n \vee q.$$

A *literal* is either a predicate symbol or a negated predicate symbol. A disjunction (or connection) of literals is called a *clause*. The *satisfiability problem* consists of a set of clauses, understood to be the conjunction (and connection) of the clauses. The question to be answered is whether the predicate symbols have an assignment of truth values that makes the set of clauses true. For this to be the case, there must be at least one literal in each clause that has the value True assigned. All clauses are then true (since they are disjunctions).

The method of resolution consists of transforming a set of clauses into another simpler, but equivalent, set of clauses. If we ever generate an empty clause, we know the set cannot be satisfied, since there is no literal in this clause that could be assigned true. If we restrict the clauses to *Horn clauses*, such transformations can be derived easily. A Horn clause is a clause that contains at most one non-negated (positive) literal. Note that the clause derived from the implication above is a Horn clause. Axioms, which consist of just one positive literal, are also Horn clauses.

The resolution algorithm looks for two clauses of the forms $\{p, \neg q\}$ and $\{\neg p, r\}$. These two can be replaced by the single clause $\{\neg q, r\}$, as can be seen by looking at the two possible cases. The literal p must either be assigned true or false. If it assigned true, the second clause can only be satisfied if r is assigned true. If p is assigned false, the first clause can only be satisfied if $\neg q$ is assigned true. Therefore, either r must be assigned true or $\neg q$ must be assigned true, which is exactly what the new clause $\{\neg q, r\}$ expresses. This idea can be generalized to clauses of the form $\{p, q_1, q_2, \dots, q_n\}$ and $\{\neg p, r_1, r_2, \dots, r_m\}$ with $n, m \geq 0$. These two clauses can be replaced by the single clause $\{q_1, q_2, \dots, q_n, r_1, r_2, \dots, r_m\}$.

The classic logical inference rule is *modus ponens*: p and $p \rightarrow q$ imply q . The two formulas are expressed by the clauses $\{p\}$ and $\{\neg p, q\}$. We can apply resolution to replace these two clauses by $\{q\}$. Resolution, therefore, is simply a generalized modus ponens.

If the clauses are of general form (not Horn clauses), the problem of satisfiability becomes much more difficult to solve. This is the reason Prolog rules are restricted to Horn form. The implication $p_1 \wedge p_2 \wedge \dots \wedge p_n \rightarrow q$ is just the Prolog rule

$$q :- p_1, p_2, \dots, p_n.$$

The syntax ($:-$) expresses the fact that the literals on the right side are negated. A fact has one single positive literal and is also a Horn clause.

If the predicates can contain variables, the simple equality test to look for pairs of $p, \neg p$ is replaced by unification. We have seen that unification makes two unified terms equal. Resolution can then be used. The unifying variable bindings have to be applied to the remaining predicates q and r as well. Recall that this is exactly what happens inside the query evaluator after unifying the left side of a rule with the goal.

The classical example with unification is the following. This implication says that men are fallible:

```
In[30]:= Assert[ fallible[x_], man[x_] ]
```

Socrates is a man.

```
In[31]:= Assert[ man[Socrates] ]
```

Therefore, he is fallible.

```
In[32]:= Query[ fallible[Socrates] ]
```

```
Out[32]= Yes
```

Deductive Databases and Expert Systems

Deductive databases are a combination of databases and logic inference. Databases enter naturally into logic programming since the collection of logic facts can be viewed as a database. The tables and relationships of databases can be realized easily as logic predicates (see my article in Volume 3, issue 2 of this *Journal* for an explanation of basic concepts of databases). Here is a sample database about dinosaurs and humans, inspired by a well-known movie. It lists a few of the characters appearing.

```
Assert[ dinosaur[Tyrannosaur] ]
Assert[ dinosaur[Velociraptor] ]
Assert[ dinosaur[Brontosaur] ]
```

```
Assert[ human[Grant] ]
Assert[ human[Tim] ]
Assert[ human[Lex] ]
```

Properties (attributes) of these entities can be defined by additional facts. Since the movie is mainly about who eats whom, we are interested in the composition and dietary habits of the main characters and their food.

```
Assert[ carnivore[Tyrannosaur] ]
Assert[ carnivore[Velociraptor] ]
Assert[ carnivore[Tim] ]
Assert[ vegetarian[Brontosaur] ]
Assert[ vegetarian[Lex] ]
```

```
Assert[ veryBig[Tyrannosaur] ]
Assert[ veryBig[Brontosaur] ]
```

```
Assert[ plant[Tree] ]
Assert[ plant[Grass] ]
Assert[ meat[cow] ]
Assert[ meat[goat] ]
```

The idea behind deductive databases is that not all facts need to be entered case by case. Some facts can be deduced logically from others, as we can do in Prolog. For example, the cow and the goat are not the only characters consisting of meat. There is no need to enter each of the others explicitly, since we know that all dinosaurs and humans are in this class. The following deductive rules replace a larger number of individual assertions:

```
Assert[ meat[x_], dinosaur[x_] ]
Assert[ meat[x_], human[x_] ]
```

Relations between entities can be entered as predicates with several arguments. The relationship “ x can eat y ” is implemented by a logic predicate $eat[x, y]$. Carnivores prefer meat, unless it’s too big to be eaten (or to be caught in the first place), and vegetarians prefer plants:

```
Assert[ eat[x_, y_], carnivore[x_], meat[y_], !veryBig[y_] ]
Assert[ eat[x_, y_], vegetarian[x_], plant[y_] ]
```

All the facts and rules given above are part of the package `JurassicPark.m`, included in the electronic supplement.

```
In[33]:= << JurassicPark.m
```

Database queries now take the form of ordinary logic queries. One question raised in the scene in which Dr. Grant, Tim, and Lex are high up in the tree staring at the Brontosaur, is whether the Brontosaur would eat Lex, since she is—as Tim jokingly points out—also a vegetarian, just like the animal confronting them. Reassuringly, she is safe:

```
In[34]= Query[ eat[Brontosaur, Lex] ]
```

```
Out[34]= No
```

The Velociraptor enjoys a varied diet:

```
In[35]= QueryAll[ eat[Velociraptor, y_] ]
```

```
{y -> cow}
{y -> goat}
{y -> Velociraptor}
{y -> Grant}
{y -> Tim}
{y -> Lex}
```

Deductive databases add such inference capabilities to ordinary database operations. One important class of rules is consistency constraints, which can be used to check that the database is consistent. For example, this consistency check ensures that we did not erroneously enter an item as both a dinosaur and a human:

```
In[36]= Query[ dinosaur[x_] && human[x_] ]
```

```
Out[36]= No
```

This query checks that all dinosaurs have been classified as either carnivore or vegetarian, that is, that our information is complete:

```
In[37]= Query[ dinosaur[x_], !(carnivore[x_] || vegetarian[x_]) ]
```

```
Out[37]= No
```

Expert systems emphasize the deductive part of the database even more. Additional features of expert systems include an *explanation component* that tells the user how an answer was derived, and *fuzzy reasoning* that can deal with incomplete or contradictory information and with inferences that have probabilities attached to them. Such features have been used in medical expert systems, for example, where it is quite common that symptoms observed in a patient can be explained by different causes. The *expert system shell* takes the place of our simple query evaluator and offers these additional capabilities. The preparation of the knowledge base is the crucial step in developing an expert system. The expert who provides the knowledge is often aided by a knowledge engineer, a specialist who knows how to put the expert's knowledge into the right form. Most expert systems have not developed out of Prolog, but have their roots in artificial intelligence. They are, to date, the most successful application of this discipline, which has otherwise not fulfilled the promises with which it was advertised in the 1970s and 1980s.

Prolog has been used successfully in many commercial software projects, especially in *configuration programs*. The configuration of a system, for example a computer system, is characterized by a database of available components and by rules about the interdependence of such components. Certain components require others (for example, a printer requires a printer port and a cable), while some components are mutually exclusive. Configuration programs let us check proposed

configurations (such as a customer's order), or suggest configurations satisfying certain constraints (such as cost constraints).

Conclusions and Acknowledgments

Prolog has always been regarded as something of a curiosity, an interesting idea but not for the real world. However, it has been used in a number of large industrial applications. In some cases, the development costs of a new Prolog program were found to be less than the yearly maintenance of the old program written in the traditional way.

A database of real-world Prolog applications is maintained by Prolog 1000, P.O. Box 137, Blackpool, Lancashire, FY2 0XY, U.K. Its description is as follows:


The Prolog 1000 is a database of real Prolog applications being assembled in conjunction with the Association for Logic Programming (ALP) and PVG. The aim is to demonstrate how Prolog is being used in the real world and it already contains over 500 programs with well over 2 million lines of code. The database is available for research use in SGML format from the Imperial College archive `src.doc.ic.ac.uk:packages/prolog-progs-db/prolog1000.v1.gz`

Many examples in this article are taken from [Bratko 1986]. The information about commercial applications of Prolog was provided by R. Marti.

References

- Berlekamp, Elwyn R., John H. Conway, and Richard K. Guy. 1982. *Winning Ways for your mathematical plays*. Academic Press.
- Bratko, Ivan. 1986. *Prolog Programming for Artificial Intelligence*. Addison-Wesley.
- Crichton, Michael. 1991. *Jurassic Park*. Random House.
- Maeder, Roman E. 1993. Turing machines and code-optimization. *The Mathematica Journal* 3(3).

Roman E. Maeder
ETH Zurich, Institute of Theoretical Computer Science,
ETH Zentrum IFW, 8092 Zurich, Switzerland
maeder@inf.ethz.ch

 The electronic supplement contains the packages `LogicProgramming.m`, `Unify.m`, `Lisp.m`, `FSA.m`, `NIM.m`, and `JurassicPark.m`, as well as the notebook `LogicExamples.ma`, containing the examples from this article. The programs work with Version 2.2 of *Mathematica* on any machine. The first three of these packages were included in the supplement of the last issue. They are repeated here for convenience.