Logic Programming I: The Interpreter

This is the first of two columns on logic programming. It gives an introduction to the subject and develops a query evaluator in *Mathematica*. This query evaluator is an interpreter for a subset of the programming language Prolog. A prerequisite for it is *unification*, which is a generalization of the pattern matching that underlies *Mathematica*'s own evaluator.

Roman E. Maeder

Introduction

In functional programming, a program consists of a number of function definitions. The body of a function applies other functions to its arguments. In a procedural program, a sequence of instructions, including conditional statements, loops, and assignments, performs a computation step-by-step. In a *logic* or *declarative* program, a number of *inference rules* is given. A query is then formulated and the evaluator tries to prove it, given the knowledge of the rules in its database. No procedural interpretation is intended in a pure logic language. The rules simply state logic properties about certain predicates. Since any evaluator for such a language defines implicitly a procedural or operational semantics, we will encounter differences between mathematical logic and a logic language. Nevertheless, we will clearly see the difference between logic programming and traditional procedural programming. The examples in the second part of this column will show the types of tasks for which a logic language is much better suited than any other kind of language.

Here is a simple logic program that will serve as our main example to show how the query evaluator works and what can be done with logic programming:

```
Assert[ arc[a, b] ]
Assert[ arc[a, c] ]
Assert[ arc[c, d] ]

Assert[ path[x_, x_] ]
Assert[ path[x_, z_], arc[x_, y_], path[y_, z_] ]
```

LISTING 1: Logic rules for a DAG.

The program describes a small *directed acyclic graph* (DAG) depicted in Figure 1. In such a graph, edges (called *arcs*) are directed, that is, they can be traversed only in one direction (like one-way streets). There is an arc from vertex a to b, but not from b to a. A *path* is any sequence of consecutive arcs. Since there is an arc from a to c and one from c to d, there is a path from a to d. There is also a (trivial) path from a to a.

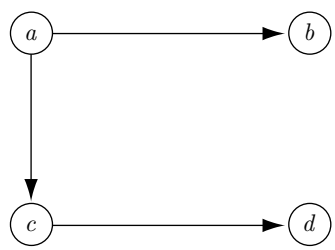

FIGURE 1. A small DAG.

Since we cannot (yet) define our own input syntax in *Mathematica*, we have to give the logic rules as arguments to the function `Assert`. In Prolog, the example would look like this:

```
arc(a, b).
arc(a, c).
arc(c, d).
path(X, X).
path(X, Z) :- arc(X, Y), path(Y, Z).
```

LISTING 2: The program from Listing 1 in Prolog.

All logic statements are terminated by a period. The left side of the rule (the conclusion) is separated from the right side (the preconditions) by the operator `:-`. Preconditions are separated by commas.

The first three rules have no right side. Such rules are called *facts*. They simply state properties that are unconditionally true, and they correspond to axioms in logic. The fourth rule is also a fact, but it contains a *variable*, denoted by the familiar underscore in *Mathematica* and by *capitalization* in Prolog. This rule simply means that for any value of `x` whatsoever there is a path from `x` to `x`. The last rule says that there is a path from `x` to `z` *provided* there is an arc from `x` to some `y` *and* there is a path from `y` to `z`. This rule has two

Roman Maeder is one of the designers of Mathematica and the author of Programming in *Mathematica* (*Addison-Wesley, 1991*). *He is now a professor of computer science at Swiss Federal Institute of Technology in Zürich, Switzerland.*

clauses on the right side, implying a conjunction: *both* must be satisfied for the conclusion to be valid. Note that we must use the underscore also on the right side of the rules, which is different from ordinary rules in *Mathematica*. I can predict from experience that you will forget this underscore many times before you get used to it. Since both the query and the rule may contain variables, there is no way around this.

I have just explained the meaning of the short example program. What can we do with it? If we assert these logic rules in a logic programming language, we can ask questions about arcs and paths. This concept is very powerful, as we will show with a few examples.

```
In[1]:= << LogicProgramming`
In[2]:= << DAG.m
```

The two rules for the predicate **path** are in the package **DAG.m**. Reading the package defines the rules as logic values:

```
In[3]:= LogicValues[ path ] // TableForm
Out[3]//TableForm= path[x_, x_]
                   path[x_, z_] :- (arc[x_, y_] && path[y_, z_])
```

Note that the two clauses in the second rule are expressed as an explicit conjunction (using *Mathematica*'s logic **And** function, written as **&&**).

If we now ask whether there is an arc from a to c, the evaluator starts comparing our query **arc[a, c]** with the left sides of the logic values for **arc**. Since no variables are involved, the procedure is quite simple. The first fact doesn't compare, but the second one does. The answer we get is simply "yes," meaning that the query can be fulfilled:

```
In[4]:= Query[ arc[a, c] ]
Out[4]= Yes
```

Now let us ask, "For which values of **x** is there an arc from **a** to **x**?" Our query is **arc[a, x_]**. It can be unified with the first two rules for **arc**, with the bindings **x -> b** and **x -> d**, respectively. The answer of the evaluator is the first successful binding:

```
In[5]:= Query[ arc[a, x_] ]
Out[5]= {x -> b}
```

We can ask the evaluator to *try again!* Now it finds the second successful binding:

```
In[6]:= Again[]
Out[6]= {x -> c}
```

There are no more possibilities, so on the next try the evaluator says "no."

```
In[7]:= Again[]
Out[7]= No
```

The command **QueryAll** prints all possible answers:

```
In[8]:= QueryAll[ arc[a, x_] ]

        {x -> b}
        {x -> c}
```

For a query with two variables, there are three possible unifications with all three rules:

```
In[9]:= QueryAll[ arc[x_, y_] ]

        {x -> a, y -> b}
        {x -> a, y -> c}
        {x -> c, y -> d}
```

Questions involving paths are more interesting since they involve a rule that has a right side. Let us ask "for which **v** is there a path from **a** to **v**?" We have to unify the query **path[a, v_]** with the left sides of the two rules for **path**. The first answer is immediate:

```
In[10]:= Query[ path[a, v_] ]
Out[10]= {v -> a}
```

The query returns with the binding for the variables occurring in the query. With **v** set to **a**, the first rule for **path** is applicable. If we try again, we have to turn to the second rule. To fulfill it, the queries in its right side have to be fulfilled first. There is an arc from **a** to **b** and a path from **b** to **b**, so there is a path from **a** to **b**:

```
In[11]:= Again[]
Out[11]= {v -> b}
```

A few additional possibilities can be obtained by trying some of the auxiliary queries again. Here are all solutions, that is, all points that can be reached from **a**:

```
In[12]:= QueryAll[ path[a, v_] ]

         {v -> a}
         {v -> b}
         {v -> c}
         {v -> d}
```

Let's look at some more examples of typical queries. Here are all points that can reach **b**:

```
In[13]:= Query[ path[u_, b] ]
Out[13]= {u -> b}
```

Queries can also contain several clauses. This query asks for all points reachable from **a**, excluding **a** itself:

```
In[14]:= QueryAll[ path[a, v_], v_ != a ]

         {v -> b}
         {v -> c}
         {v -> d}
```

The following query tests whether our graph is indeed acyclic. In an acyclic graph, there cannot be a path from u to v and back. Note that we must exclude the trivial case $u = v$.

```
In[15]:= Query[ path[u_, v_], u_ != v_, path[v_, u_] ]
Out[15]= No
```

We can also use anonymous pattern variables. The following query could be phrased as: "Is there a path starting at `a` (leading somewhere)?" The anonymous variable is not shown in the result:

```
In[16]:= Query[ path[a, _] ]
Out[16]= Yes
```

Unification

The process of *unification* should be easy to understand for *Mathematica* users, since a weaker form of it—*pattern matching*—is the fundamental operating principle of *Mathematica*'s evaluator. In pattern matching, a *pattern* is compared with an expression. Pattern matching means finding replacements for the *pattern variables* occurring in the pattern that make the instantiated pattern identical to the expression. If this is possible, we say that the expression matches the pattern. Here is an example:

```
In[1]:= f[3, h[3]] /. f[x_, h[x_]] :> x
Out[1]= 3
```

When *Mathematica* applies the rule, it tries to match expressions occurring in `f[3, h[3]]` with the pattern `f[x_, h[x_]]`, which forms the left side of the rule. Here it succeeds with `x -> 3` and then applies this substitution to the right side of the rule.

The following expression does not match, so it is left alone:

```
In[2]:= f[3, h[4]] /. f[x_, h[x_]] :> x
Out[2]= f[3, h[4]]
```

The pattern matcher itself, which tries to find the variable bindings, is not available as a user-callable function. We provide a simple pattern-matcher in the package `Unify.m`. It takes as arguments the pattern and the expression to match, and it returns the list of bindings or `$Failed` if a match cannot be found.

Here is the matching that takes place in the example above. If the pattern variable `x` is set to 3, the pattern becomes identical to the expression:

```
In[1]:= << Unify.m
In[2]:= Unify[ f[x_, h[x_]], f[3, h[3]] ]
Out[2]= {x -> 3}
```

This expression does not match the pattern:

```
In[3]:= Unify[ f[x_, h[x_]], f[3, h[4]] ]
Out[3]= $Failed
```

But it matches this pattern with two pattern variables:

```
In[4]:= Unify[ f[x_, h[y_]], f[3, h[4]] ]
Out[4]= {x -> 3, y -> 4}
```

The function `Unify[]` is not as sophisticated as the built-in matcher. It understands only simple pattern variables in the form *name*`_` and does not take attributes into account (that is, it is not an associative/commutative pattern matcher).

In one important respect our function is more powerful than the built-in pattern matcher: it allows *unification*. Unification is two-way matching, where both arguments of the `Unify` function can contain pattern variables. It returns the most general variable substitution that makes the two patterns identical:

```
In[5]:= Unify[ f[x_, a], f[b, y_] ]
Out[5]= {x -> b, y -> a}
```

If we set `x` to `a` and `y` to `b`, the two patterns become identical.

In the following example, no substitution can be found since `x` would have to be equal to `a` and `b` simultaneously:

```
In[6]:= Unify[ f[x_, a], f[b, x_] ]
Out[6]= $Failed
```

The next example is already a bit tricky to figure out by hand:

```
In[7]:= Unify[ f[x_, g[y_]], f[g[3], x_] ]
Out[7]= {x -> g[3], y -> 3}
```

Comparing the first arguments of `f`, we find that `x -> g[3]`. Comparing the second arguments with this binding in mind, we find that `g[y_]` must unify with `g[3]`, which gives the binding for `y`.

Here is another important example:

```
In[8]:= Unify[ f[x_], f[g[y_]] ]
Out[8]= {x -> g[y]}
```

We find that with `x -> g[y]` the two patterns are identical, whatever value we choose for `y`. The result therefore does not contain a binding for `y`.

The next example is similar:

```
In[9]:= Unify[ g[x_, y_], g[y_, y_] ]
Out[9]= {y -> x}
```

If the two variables have identical values, the two patterns become identical, whatever that value is. Therefore, we replace one of the variables by the other one.

Unification is a fairly simple process. Two identical expressions always unify without any bindings. A variable unifies with any expression by being bound to it. If the two expressions are composite, they must have the same head and the same number of elements; elements are then unified in turn. Bindings obtained in this way are merged, and if a conflict is found, unification fails. The code is in the package `Unify.m`.

Unification plays the same role in query evaluation as matching does in *Mathematica*: A list of rules is searched sequentially, until one is found that unifies with the query. The unifying variable binding is then applied to the right side of the rule, and query evaluation continues.

Query Evaluation

A query establishes a *goal* to be satisfied. To do so, further subgoals may have to be satisfied. If a goal fails, the evaluator will try to resatisfy an earlier goal: it performs *backtracking*. If none of the subgoals can be redone, the original goal fails.

In close analogy to *Mathematica*'s downvalues, which are used to store definitions made for a symbol, logic rules will be kept in a list `LogicValues[`*symbol*`]` attached (as an upvalue) to *symbol*.

If we view the collection of rules for a predicate as a function, this function has *four* connections to its environment, unlike an ordinary function, which has just two: call and return. When we try to satisfy a goal for the first time, we *call* the predicate. There are two possible outcomes: *success* and *failure*. In case of success, we can later *redo* the predicate. This is often depicted as in Figure 2. For purposes of backtracking, a conjunction of goals, $goal_1$ `&&` $goal_2$ `&&` … `&&` $goal_n$ can be thought of as a connection of the corresponding procedure boxes, as shown in Figure 3. A succeeding call will invoke the next procedure until we succeed with the last one and thus with the complete goal. If one of the predicates fails, it invokes the redo entry of the preceding one. Control may pass back and forth many times until we either succeed the last goal or fail the first one.

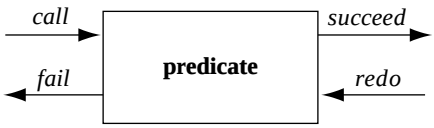

FIGURE 2. The procedure box model.

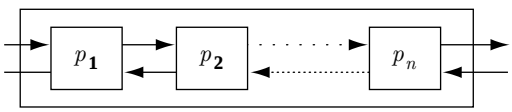

FIGURE 3. Backtracking a conjunction of goals.

Let us return to the query `path[a, v_]` from the introduction and show in detail how the answers are obtained.

We have to unify the query with the left sides of the two rules for `path`. The first answer is immediate. The unification with the left side `path[x_, x_]` of the first rule succeeds with these bindings:

```
In[1]:= Unify[ path[x_, x_], path[a, v_] ]
Out[1]= {x -> a, v -> a}
```

The query returns only the bindings for the variables occurring in the query itself:

```
In[2]:= Query[ path[a, v_] ]
Out[2]= {v -> a}
```

If we try again, we have to turn to the second rule. Let us now show step-by-step what happens inside the evaluator. The unification with the left side of the second rule succeeds with these bindings:

```
In[3]:= Unify[ path[x_, z_], path[a, v_] ]
Out[3]= {x -> a, z -> v}
```

The right side of the second rule, with the values for `x` and `z` inserted, becomes `arc[a, y_] && path[y_, v_]`. (Note that the value of `z` is again a variable.) Now, we try to fulfill the first clause of the conjunction by a recursive call of the evaluator. We succeed with this binding for `y`:

```
In[4]:= Query[ arc[a, y_] ]
Out[4]= {y -> b}
```

Next, we apply this additional binding to the remaining clauses of the right side (only one more is left) and get `path[b, v_]` as the remaining goal to satisfy. Again, a recursive call is used to fulfill this query. It succeeds by using the first rule for `path`. The result is the binding for our original query variable `v`:

```
In[5]:= Query[ path[b, v_] ]
Out[5]= {v -> b}
```

To find additional solutions to our original query `path[a, v_]`, we use *backtracking*. We turn to the last subgoal satisfied and try to redo it. There is no other way to satisfy `path[b, v_]`; that is, there is no arc starting at `b`:

```
In[6]:= Again[]
Out[6]= No
```

Since the second clause could not be redone, we turn to the first one: `arc[a, y_]`. It can be redone, producing the next solution:

```
In[7]:= Again[]
Out[7]= {y -> c}
```

We apply the new binding for `y` to the second clause and try to satisfy it (for the first time), producing our second solution to the original query:

```
In[8]:= Query[ path[c, v_] ]
Out[8]= {v -> c}
```

If we try our original query again, we backtrack and try to redo the last query. This time there is an additional solution. Backtracking therefore stops here, and we get the third solution:

```
In[9]:= Again[]
Out[9]= {v -> d}
```

The Implementation

Our query evaluator has become quite sophisticated. Nevertheless, it shows that a more-or-less complete Prolog interpreter turns out to be a rather short program.

Internally, pattern variables like $x\_$ are represented as `Var[`x`]`. This representation protects them from *Mathematica*'s evaluator, for which pattern variables are also meaningful. The code for converting back and forth is in the auxiliary package `Unify.m`. The user-level unifier `Unify[`*pattern*$_1$, *pattern*$_2$`]` converts the pattern variables to internal form, calls the internal code in `Unify0`, and finally restores the pattern variables for output.

```
Unify[x_, y_] :=
    Unify0[ x /. Pattern2Var, y /. Pattern2Var ] /. Var2Symbol
```

The value of `Pattern2Var` is a list of two rules for turning $x\_$ into `Var[`x`]` and `_` into `Var[$, n]`, where n is an integer that is different for each anonymous variable encountered. The exact form of these rules is a case study in writing patterns for manipulating patterns. A pattern object can be protected from *Mathematica*'s pattern matcher by enclosing its *head* in `Literal[`*object*`]`.

```
anon = 0 (* counter for unique values *)

Pattern2Var = {Literal[Pattern][x_, Literal[Blank][]] :> Var[x],
               Literal[Blank][] :> Var[$, anon++] };
```

To understand these rules, recall that the internal form of $x\_$ is `Pattern[`x`, Blank[]]`. After wrapping `Literal` around the two occurring pattern objects (`Pattern` and `Blank`), we get the template for the left side of the first rule. The `x` inside it is now a pattern variable for *this* rule!

The code for `Assert[`*lhs*`,` *rhs*...`]` appends the given logic rule to the list already stored as `LogicValues` of the head of *lhs* (Listing 3).

```
Assert[lhs_, rhs___] := insertRule[ makeRule[lhs, rhs], Append ]
Asserta[lhs_, rhs___] := insertRule[makeRule[lhs, rhs], Prepend]

insertRule[r_, op_] :=
    With[{h = dispatch[r]},
        If[ h === Null, Return[$Failed] ];
        LogicValues[h] ^= op[ LogicValues[h], r ];
    ]

makeRule[lhs_] := rule[lhs] /. Pattern2Var
makeRule[lhs_, True] := makeRule[lhs]
makeRule[lhs_, rhs_] := rule[lhs, rhs] /. Pattern2Var
makeRule[lhs_, rhs__] := makeRule[lhs, And[rhs]]

dispatch[rule[lhs_, ___]] := predicate[lhs]
predicate[(h_Symbol) | (h_Symbol[___])] := h
predicate[e_] :=
    (Message[Logic::nohead, e]; Null) (* no predicate found *)

LogicValues[_Symbol] := {} (* for symbols with no rules yet *)
```

LISTING 3: Data type for rules.

The auxiliary constructor `makeRule[`*lhs*`,` *rhs*...`]` takes care of rules without a right side (turning them into facts) and rules with more than one clause on the right side (wrapping `And` around the clauses). `rule` is our data type for facts and rules. A right side consisting of `True` can also be treated as a fact since it is always satisfied. The output form of rules approximates Prolog's syntax. The function `predicate[`*expr*`]` returns the predicate to which the rule must be attached, either *expr* itself if it is a symbol, or the head of a composite *expr*. This definition is best expressed with an alternative. Downvalues in *Mathematica* are treated in the same way.

The last definition for `LogicValues` takes care of defining the first rule for a symbol. Without it, the call to `LogicValues[`h`]` occurring in the code for `insertRule` would not return a proper value. This simple trick avoids a cumbersome test in the code. Since nonempty rule lists are stored as *upvalues*, the definition will apply only when there are indeed no rules yet.

`Assert` appends new rules to the end of the list of rules. `Asserta` prepends rules at the beginning. There is also a command `Retract[`*rule*...`]` that removes a rule from the database. The first rule matching the argument is removed.

Evaluating Predicates: State Information and Backtracking

A user-level query `Query[`*goal*`]` is transformed into internal form by converting the pattern variables as we have just seen. It is then handed to our internal query evaluator `qeval`.

A query is either a built-in predicate (treated in the next section) or a user-defined predicate for which we may have defined logic values.

The internal query evaluator is called as `qeval[`*goal*`,` *state*`]`. It takes the goal predicate and a *state* as arguments and returns two items: a list of bindings or `$Failed` if it cannot be satisfied, and the new state. The second argument of `qeval` describes the internal state of the evaluator, which is needed for backtracking. When the goal has to be retried, it is the caller's responsibility to call `qeval[` *goal*`,` *state*`]` with the state that was returned in the last call. The special value `initialState` is used in the first call, and a value of `finalState` means that the goal cannot be retried.

Mathematica does no backtracking when applying rules. Therefore, its evaluator does not need any state information (the state is implicit in the run-time stack). Since backtracking is an essential feature of logic programming, we need to manipulate the state in this complicated way.

If the goal is a user-defined predicate, we try all logic values in turn until we find one whose left side unifies with the goal (Listing 4). The state information needed to continue the search consists of the list of additional rules to try, the current rule, and its state. For efficiency, we keep a version of the current rule with its variables renamed. Renaming variables is necessary to avoid conflicts between variable names in rules and in queries.

The loop is initialized by setting the list of remaining rules to the rest of the rules of the predicate, the current rule to the renamed version of the first rule, and its state to `initialState`. If there are no rules, we immediately return `no`, which is defined as `{$Failed, finalState}`. The general case takes the

```
(* apply rules: state is triplet
   {rest of rules, instance of first rule, its state} *)

qeval[ expr_, initialState ] :=
    With[{rules = LogicValues[predicate[expr]]},
        If[ Length[rules] == 0, Return[no] ];
        qeval[expr,
              {Rest[rules], rename[First[rules]], initialState}]
    ]

qeval[ expr_, {rules0_, inst0_, rstate0_} ] :=
   Module[{rules = rules0, inst = inst0, rstate = rstate0, res},
        While[True,
            res = tryRule[ expr, inst, rstate ];
            If[ !failedQ[res],
                Return[{bindings[res],
                        {rules, inst, state[res]}}] ];
            (* try next rule *)
            If[ Length[rules] == 0, Break[] ]; (* no more *)
            inst = rename[First[rules]];
            rstate = initialState;
            rules = Rest[rules];
        ];
        no
    ]
```

LISTING 4: Searching for a rule.

```
(* for facts *)

tryRule[ expr_, r:rule[lhs_], initialState ] :=
    Module[{bind, inst, res},
        bind = Unify0[ expr, lhs ];         (* unify lhs *)
        If[ bind === $Failed, Return[no] ]; (* does not unify *)
        bind = closure[bind];
        {bind, finalState}
    ]

(* for rules proper *)

tryRule[ expr_, r:rule[lhs_, rhs_], initialState ] :=
    Module[{bind, res},
        bind = Unify0[ expr, lhs ];         (* unify lhs *)
        If[ bind === $Failed, Return[no] ]; (* does not unify *)
        tryRule[ expr, r, {bind, rhs //. bind, initialState} ]
    ]

tryRule[ expr_, r_rule, {bind_, inst_, stater_} ] :=
    Module[{res, outbind},
        res = qeval[ inst, stater ]; (* recursion *)
        If[ failedQ[res], Return[res] ];
        outbind = closure[bind, bindings[res]];
        {outbind, {bind, inst, state[res]}}
    ]
```

LISTING 5: Applying a rule.

current rule in the state information and tries it, passing the saved state information along. If this attempt succeeds, we return the bindings obtained and the updated state information. If it doesn't succeed, we have to try the next rule. If there are no more rules, we return with `no`.

To try out a rule, we have to unify its left side with the goal, apply the bindings obtained to its right side, and call `qeval` recursively with the right side (Listing 5).

The initial call does the unification. For efficiency, the result (the list of bindings) is retained as part of the state information, as is the instantiated right side. The first definition takes care of facts; that is, rules without a right side. The function `closure[`*bindings*`]` from `Unify.m` combines bindings.

The actual code in `LogicProgramming.m` contains an array of additional details, which we have no room here to discuss.

Built-in Predicates

The built-in predicates are similar to built-in functions and commands in *Mathematica*. They are used to do things that cannot be expressed by simple logic rules. Each built-in predicate p is implemented by special code for `qeval[`p`[`*args*…`]`, *state*`]`. This code is attached to p as an upvalue, which avoids a large number of (sequentially searched) rules for `qeval`.

Conjunction (And)

We have encountered one built-in predicate so far: `And[`$goal_1$, $goal_2$, …`]`. Evaluation happens by first (recursively) trying the first goal. If this goal succeeds, we apply the bindings obtained to the rest of the query `And[`$goal_2$, …`]`. If this query also succeeds, we return the combined bindings. If this second query fails, we retry the first one until it fails as well or until we find a success of the second query. The state information needed is the result of the first query and the state of the second query. For efficiency, we also keep the instance of the rest of the query (Listing 6).

Because of the state information, the recursive treatment of the rest of the query is simpler to implement than a loop over all arguments of `And`. The same kind of recursion is used in Lisp.

Disjunction (Or)

The disjunction $goal_1$ || $goal_2$ || … || $goal_n$ is evaluated as follows. The first goal is called. If it succeeds, we are done. If it fails, we discard it and try the rest of the disjunction $goal_2$ || … || $goal_n$ until we find a goal that succeeds. To redo a disjunction, we try to redo the goal that succeeded last time. The state information consists simply of the state of the first goal and the state of the rest.

The following rule says that an *edge* is an undirected arc, which means that there is an arc either from `x` to `y` or in the opposite direction:

```
In[10]:= Assert[ edge[x_, y_], arc[x_, y_] || arc[y_, x_] ]
```

Here are all edges having `c` as one of their endpoints:

```
In[11]:= QueryAll[ edge[c, v_] ]
         {v -> d}
         {v -> a}
```

```
(* And: state is triplet
   {result of first clause, instance of rest, rest state} *)

And/: qeval[ and:And[g1_, goals__], initialState ] :=
    Module[{res},
        (* call first goal to initialize things *)
        res = qeval[g1, initialState];
        If[ failedQ[res], Return[res] ]; (* failure *)
        (* apply bindings to rest and call it *)
        qeval[ and,
          {res, And[goals] //. bindings[res], initialState} ]
    ]

And/: qeval[ And[g1_, goals__], {res10_, rest0_, stater0_} ] :=
    Module[{res1 = res10, rest = rest0, stater = stater0,
            res, binds},
        While[ True,
            res = qeval[ rest, stater ]; (* eval rest *)
            If[ !failedQ[res],
                binds = closure[bindings[res1], bindings[res]];
                Return[{binds, {res1, rest, state[res]}}];
            ];
            (* else try first one again *)
            res1 = qeval[g1, state[res1]];
            If[ failedQ[res1], Return[res1] ];
            stater = initialState; (* reset for next attempt *)
            rest = And[goals] //. bindings[res1];
        ];
    ]
```

LISTING 6: Evaluating a conjunction of goals.

Negation (Not)

The goal `Not[`*goal*`]`, or `!` *goal*, succeeds if *goal* itself fails. No bindings are generated, and it cannot be redone. Its implementation is therefore quite simple:

```
Not/: qeval[Not[g_], initialState ] :=
    If[ failedQ[ qeval[g, initialState] ], yes, no ]
```

Negation provides the best illustration that logic programming is not mathematical logic. Here is the standard example:

```
In[12]:= Query[ !human[Joe] ]
Out[12]= Yes
```

Logically, this result would mean that "Joe is not human." In logic programming, we interpret negation as failure, so it really means "there is not enough information in the database to prove 'Joe is human'."

This use of `Not` is called the *closed world assumption*. We assume that our database is complete and anything not deducible from it is therefore false.

Equality and Inequality

In logic programming, we interpret equality as unification and inequality as non-unification. We have already seen an example of inequality in the query `path[a, v_] && v_ != a` in the introduction. Since *Mathematica* turns `Not[`a `==` b`]` into a `!=` b, we need the definition for inequality, even though it is equivalent to `Not[`a `==` b`]`. Unification cannot be redone.

```
Equal/: qeval[ e1_ == e2_, initialState ] :=
    Module[{u},
        u = Unify0[e1, e2];
        If[ u === $Failed, no, {u, finalState} ]
    ]

Unequal/: qeval[ e1_ != e2_, initialState ] :=
        If[ Unify0[e1, e2] === $Failed, yes, no ]
```

Input and Output

It is hard to fit input and output into the logic framework. Prolog suffers from the *single-paradigm syndrome:* Everything has to be forced into the single programming paradigm advocated by the language, even if it does not fit naturally. We provide only rudimentary I/O facilities (different from the ones found in Prolog). `input[`v, *prompt*`]` reads an input expression using the (optional) prompt and succeeds if the input read can be unified with v. Most often, v is a variable and unification always succeeds. This case corresponds roughly to *Mathematica*'s v = `input[`*prompt*`]`. If the end-of-file character (CTRL-D) is encountered, it fails. `input` can be redone; it simply asks for new input.

The "predicate" `print[`*args*`…]` prints its arguments (using `Print[`*args*`…]`) and always succeeds. It cannot be redone.

The following loop repeatedly asks for a vertex and prints `yes` if there is a path from `a` to it, and `no` otherwise. The final `fail` is a predicate that always fails. It forces backtracking and thus implements the loop.

```
In[13]:= Query[ input[v_, "vertex: "],
                path[a, v_] && print["yes"] ||
                    !path[a, v_] && print["no"],
                fail ]

      vertex: a
      yes
      vertex: e
      no
      vertex: b
      yes
      vertex: ^D

Out[13]= No
```

Procedural Interpretation of Logic Rules

In pure logic, the predicates `True` and `False` would not alter the meaning of a program. The expression p `&& True` can be simplified to p (*Mathematica* does this). Since a logic programming has—besides its declarative interpretation—a procedural meaning, defined by the actions of the query evaluator, such predicates can alter the behavior of the program. The predicate `true` always succeeds. We provide it (in addition to `True`) because it is not affected by the built-in simplification above. The predicates `fail` and `false` always fail. Their implementation is trivial: Any undefined predicate will fail. For efficiency reasons, we nevertheless give a definition for them:

```
qeval[ True, initialState ] := yes
qeval[ False,initialState ] := no
fail/: qeval[ fail, initialState ] := no
true/: qeval[ true, initialState ] := yes
false = fail
```

This is the skeleton of a Prolog-level implementation of **QueryAll**. The final **false** forces backtracking until all possibilities are exhausted:

```
In[14]:= Query[path[a, v_], print[v_], false]
         a
         b
         c
         d

Out[14]= No
```

It does not work with the built-in **False**, since *Mathematica* simplifies **And**[$goal_1$, $goal_2$, … , **False**] to **False**:

```
In[15]:= Query[path[a, v_], print[v_], False]
Out[15]= No
```

The predicate **repeat** succeeds (like **true**), but it can be redone infinitely often. It can be used to form loops. It can be implemented directly in the language:

```
Assert[ repeat ]
Assert[ repeat, repeat ]
```

The Cut

There is one remaining predicate that is the topic of endless discussions: the *cut*. The cut prevents backtracking and is, therefore, not part of a pure logic programming language, but it seems unavoidable and often leads to big gains in program efficiency. (This might be viewed as a corollary to the single-paradigm syndrome: Every such language has a feature that breaks its single paradigm.) If a cut is encountered on the right side of a rule, no backtracking will occur to the left of the cut. If the cut should be redone, the current goal fails immediately. There is no attempt to redo the predicates before the cut, nor does it try a different rule for the same predicate! A cut, therefore, commits the choices made so far.

One use is for programming mutually exclusive alternatives. The conditional definition

```
f[x_] := If[ pred[x], then[x], else[x]]
```

needs to be programmed like this (using Prolog syntax):

```
f[X] :- pred[X], then[X].
f[X] :- ! pred[X], else[X].
```

The negation of the predicate in the else clause is necessary to avoid backtracking into the second rule if the predicate is true. With a cut, we can commit ourselves to the first clause, once the predicate is known to be true:

```
f[X] :- pred[X], cut, then[X].
f[X] :- else[X].
```

Negation can also be implemented in terms of the cut as

```
not[P] :- (P && cut && fail) || true.
```

The cut prevents backtracking from **fail**, once **P** is known to be true.

The cut is implemented through the use of a special state that acts like a failure state but is detected by the code for **And**, **Or**, and the loop that tries rules in turn. When the cut state is encountered, no further attempts to redo the goal are made and failure is returned.

Equation Solving

In Prolog, the special predicate **is**[*var* = *expr*] is used to evaluate arithmetic expressions. Because all queries in our logic evaluator are evaluated by *Mathematica* in the standard way, we do not need it for this purpose. We use **is**[*eqlist*] to tap some of *Mathematica*'s power. This predicate succeeds if the occurring variables can be bound to a solution of the list of equations *eqlist* (using **Solve**). If this query is redone, the next solution is returned, and so on. Implementation is quite simple, since **Solve** already returns lists of replacements for the variables. The state information is simply the list of remaining solutions.

This query returns the first solution of the quadratic equation:

```
In[16]:= Query[ is[ x_^2 == x_ + 1 ] ]

                    1 - Sqrt[5]
Out[16]= {x -> ----------- }
                        2
```

Now we get the second solution:

```
In[17]:= Again[]

                    1 + Sqrt[5]
Out[17]= {x -> ----------- }
                        2
```

Meta-logic Predicates

As in most other languages, there are some predicates that leave the primary domain of the language (logic, in our case). Examples are **explode** in Lisp, which turns an atom into a list of characters (like **Characters**[**ToString**[*symbol*]] in *Mathematica*), and **DownValues**[*symbol*] in *Mathematica*, which allows messing around with definitions. In Prolog, there is **assert**[*rule*], which works like the outside **Assert**[*rule*]; **retract**[*rule*], corresponding to **Retract**[*rule*]; **var**[v], which is true if v is an uninstantiated variable; **ground**[*expr*], which tells whether *expr* is free of variables (a so-called *ground term*); and **name**[v, *list*], which succeeds if *list* is the list of character codes that make up the symbol v.

The predicate **name** can be used to find the character codes of the name of a symbol:

```
In[18]:= Query[ name[symbol, l_] ]
Out[18]= {l -> (115 121 109 98 111 108)}
```

In typical Prolog manner, it can also be used "backwards" to assemble a list of character codes into a symbol:

```
In[19]:= Query[ name[var_, list[97, 108, 112, 104, 97]] ]
Out[19]= {var -> alpha}
```

Lists are represented in the same way as they are in Lisp. We use the package **Lisp.m** developed in our column about abstract data types in Volume 2, issue 3 of this Journal. Examples with lists in Prolog will follow in the second part of this column.

The ability to modify the database of logic rules during a query with **assert**[*rule*] can be used for dynamic programming in a way similar to *Mathematica*. Here is the example for Fibonacci numbers, taken from [Maeder 1991] and [Maeder 1994]. This is the usual recursive program for Fibonacci numbers in Prolog:

```
Assert[ fib[1, 1] ]
Assert[ fib[2, 1] ]
Assert[ fib[n_, f_],
        n_ > 2, n1_ == n_ - 1, fib[n1_, f1_],
        n2_ == n_ - 2, fib[n2_, f2_], f_ == f1_ + f2_ ]
```

It shows the expected poor performance. If we store any new computed values as new logic rules, they will not have to be recomputed. The new rules need to go to the beginning of the list, so we use **asserta**:

```
Assert[ fibf[1, 1] ]
Assert[ fibf[2, 1] ]
Assert[ fibf[n_, f_],
        n_ > 2, n1_ == n_ - 1, fibf[n1_, f1_],
        n2_ == n_ - 2, fibf[n2_, f2_], f_ == f1_ + f2_,
        asserta[fibf[n_, f_]] ]
```

This query computes the 10th Fibonacci number:

```
In[1]:= Query[ fibf[10, f_] ]
Out[1]= {f -> 55}
```

We can see that all intermediate values have been stored:

```
In[2]:= LogicValues[ fibf ]
Out[2]= {fibf[10, 55], fibf[9, 34], fibf[8, 21], fibf[7, 13],
         fibf[6, 8], fibf[5, 5], fibf[4, 3], fibf[3, 2],
         fibf[1, 1], fibf[2, 1], fibf[n_, f_] :-
          ((n_) > 2 && (n1_) == -1 + (n_) && fibf[n1_, f1_] &&
            (n2_) == -2 + (n_) && fibf[n2_, f2_] &&
            (f_) == (f1_) + (f2_) && asserta[fibf[n_, f_]])}
```

In *Mathematica*, we would use a definition of the form

```
fibf[n_] := fibf[n] = fibf[n-1] + fibf[n-2]
```

When retracting rules, you can use unification to search for the rule to be removed. If there are several matching rules, backtracking will remove one after another. These are the rules known for **arc**:

true	always succeeds
false, **fail**	always fails
cut	succeeds but prevents backtracking
repeat	succeeds and can be redone always
and[*goals*], g_1**&&**g_2**&&**...	succeeds if the goals succeed in sequence
or[*goals*], $g_1 \| \| g_2 \| \| \ldots$	succeeds if one of the goals succeeds
not[*goal*], **!** *goal*	succeeds if *goal* fails
e_1 == e_2	e_1 and e_2 can be unified
e_1 != e_2	e_1 and e_2 cannot be unified
print[*args*]	prints the *args* and succeeds
input[v, (*prompt*)]	unifies v with an expression read
name[v, l]	l is the list of character codes of the symbol v
var[v]	v is an uninstantiated variable
nonvar[v]	v is not a variable
ground[e]	e contains no variables
integer[i]	i is an integer
atomic[a]	a is an atom
atom[a]	a is an atom other than a number
is[*eqlist*]	succeeds if the equations *eqlist* can be solved
assert[*rule*]	succeeds if the rule was entered into the database
asserta[*rule*]	succeeds if the rule was entered at the beginning
assertz[*rule*]	succeeds if the rule was entered at the end
retract[*rule*]	succeeds if the rule was removed from the database
run[*cmd*]	evaluates the *Mathematica* expression *cmd* and succeeds if no messages were produced
run[*cmd*, *res*]	succeeds if the result of *cmd* is *res*
trace	always succeeds and turns on full tracing
notrace	always succeeds and turns off full tracing

TABLE 1. The built-in predicates of our logic language.

```
In[1]:= LogicValues[arc]
Out[1]= {arc[a, b], arc[a, c], arc[c, d]}
```

This query succeeds because a matching rules was found and removed:

```
In[2]:= Query[ retract[arc[c, d]] ]
Out[2]= Yes
```

It is indeed gone:

```
In[3]:= LogicValues[arc]
Out[3]= {arc[a, b], arc[a, c]}
```

This query removes all rules of the form **arc[a,** x**]**. There were two of them:

```
In[4]:= QueryAll[ retract[arc[a, x_]] ]

        {x -> b}
        {x -> c}
```

The following query removes all rules for the predicate **path** that are not facts (that is, those that have a right side). Note

that you do not need to use the same variable names as in the rule; unification takes care of that.

```
In[5]:= QueryAll[ retract[path[u_, v_], rhs_] ]
        {rhs -> arc[u, y] && path[y, v]}
```

Table 1 shows all predicates defined in our interpreter. These are the predicates present in most Prolog interpreters. This table is our reference manual.

Running the Evaluator

If you want to experiment with logic queries, you can read in the package **LogicProgramming.m** and then set up some logic rules with **Assert**. You have already seen how to formulate queries. **Again** is implemented by storing the state returned from the internal evaluator in a static private variable.

While developing your programs, you can clear all existing logic values with **Clear[***symbol***]** in the same way you would clear *Mathematica* definitions before entering them again.

Table 2 shows the commands available for working with the logic query evaluator. Please note that all assertions and queries are still evaluated by *Mathematica* in the usual way.

The variable **prolog** is bound to a procedure that simulates a Prolog-style main loop. In this loop, you can enter queries directly at the prompt **?-** and you are prompted to redo them. If you enter **;**, the query is redone; otherwise, it is abandoned. Here is a sample session:

```
In[1]:= <<DAG.m
In[2]:= prolog
        ?- arc[a, x_]
        {x -> b} ?;
        {x -> c} ?;
        No

        ?- path[a, x_]
        {x -> a} ?;
        {x -> b} ?

        ?- ^D
```

Debugging

Since the flow of control is not obvious in a logic program, good debugging facilities are essential. Tracing is the most useful of these. It shows which goals are attempted and with what bindings they succeed. **Spy[***symbol***]** turns on tracing for rules attached to *symbol* (like *Mathematica*'s **On[***symbol***]**). **NoSpy[***symbol***]** turns tracing off. **NoSpy[]** turns it off for all symbols.

Let's look again at the examples with the DAG:

```
In[1]:= Spy[path]
```

Now we will see all calls to the procedure **path**:

```
In[2]:= Query[ path[a, x_] ]
          Goal is path[a, x_]
          Yes, with {x -> a}
Out[2]= {x -> a}
```

Assert[*fact***]**	assert *fact*.
Assert[*lhs***,** $g_1, \ldots, g_n$**]**	assert *lhs* **:-** $g_1, \ldots, g_n$.
Asserta[*rule***]**	enter *rule* at the beginning
Assertz[*rule***]**	enter *rule* at the end
Retract[*rule***]**	remove *rule*
LogicValues[*symbol***]**	the list of logic rules for *symbol*
Clear[*symbol***]**	remove all logic rules defined for *symbol*
Query[$g_1, \ldots, g_n$**]**	evaluate the query **?-** $g_1, \ldots, g_n$.
QueryAll[{$g_1, \ldots, g_n$**]**	print all solutions
Again[]	redo the last query
Spy[*symbol***]**	turn on tracing for rules attached to *symbol*
NoSpy[*symbol***]**	turn off tracing for *symbol*
NoSpy[]	turn off tracing for all symbols
traceLevel	trace level (initially 1); 0 disables all tracing, 2 shows details
Yes	indicates a successful query without variables
No	indicates an unsuccessful query
{var_1->val_1, …, var_n->val_n}	bindings that satisfy a query

TABLE 2. Commands and values of the query evaluator.

```
In[3]:= Again[]
          Goal is path[a, x_]
            Goal is path[b, x_]
            Yes, with {x -> b}
          Yes, with {x -> b}
Out[3]= {x -> b}
```

If you set **traceLevel** to 2, you will also see all rules that are tried, together with the instantiated subqueries:

```
In[4]:= traceLevel=2;
```

In this form of output, variables are given in their internal form. The suffixes distinguish the variables in different invocations of the same rule.

```
In[5]:= Again[]
          Goal is path[a, x_]
            >redo: path[x_, z_] :- (arc[x_, y_] && path[y_, z_])
            +new goal: arc[a, y_1] && path[y_1, x_]
              Goal is path[b, x_]
               >call: path[x_, z_] :- (arc[x_, y_] && path[y_, z_])
                +new goal: arc[b, y_4] && path[y_4, x_]

                <failed
              No.
              Goal is path[c, x_]
                >call: path[x_, x_]
                <ret:  {x_9 -> c, x_ -> c}
              Yes, with {x -> c}
            <ret:  {x_1 -> a, z_1 -> c, y_1 -> c, x_9 -> c, x_ -> c}
          Yes, with {x -> c}
Out[5]= {x -> c}
```

Further Reading

The classical reference to Prolog is the manual of its first implementation by Clocksin and Mellish [1981]. Many examples in this column were taken from [Bratko 1986], which is an excellent introduction to Prolog with many motivating examples from those parts of artificial intelligence that have turned out to be practical and useful.

Acknowledgments

We would like to thank Robert Marti for his help in developing the interpreter, and Georgios Grivas and Stephan Zahner for the first version of the unifier.

A Look Back at Earlier Columns

This article is the 10th in the *Mathematica Programmer* series. The first columns were written for Version 1.2 of *Mathematica*. Many things have happened since then. I am happy to announce that the collection of the first nine columns has now appeared as a book entitled *The Mathematica Programmer*. All columns and programs have been expanded and updated to Version 2.2 of *Mathematica*. An introduction about the different programming styles, many examples, and a color insert have been added. All programs and examples are contained in a floppy disk included with the book.

References

Bratko, Ivan. 1986. *Prolog Programming for Artificial Intelligence*. Addison-Wesley.

Clocksin, F. W., and C. S. Mellish. 1981. *Programming in Prolog*. Springer-Verlag.

Maeder, Roman E. 1991. Fibonacci on the fast track. *The Mathematica Journal* 1(3): 42–46.

Maeder, Roman E. 1992. Abstract data types. *The Mathematica Journal* 2(3): 47–52.

Maeder, Roman E. 1994. *The Mathematica Programmer*. Academic Press Professional.

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`, and `DAG.m`. The programs work with Version 2.2 of *Mathematica* on any machine. `Lisp.m` is identical to the version distributed with Volume 2, issue 3.