PROLOG FOR IMPERATIVE PROGRAMMERS

Amruth N Kumar
Ramapo College of New Jersey
505 Ramapo Valley Road
Mahwah, NJ 07430-1680
amruth@ramapo.edu

ABSTRACT

Once students master the imperative programming paradigm, they find it hard to learn to work with a declarative language such as Prolog. In order to help our students make the transition from imperative to pure declarative programming, we have identified many common misconceptions and pitfalls to which imperative programmers are susceptible. We have attempted to recast the semantics of commonly misunderstood data and control constructs in Prolog in terms of the more familiar imperative constructs. Furthermore, in order to promote declarative rather than imperative programming in Prolog, we have devised a set of programming templates, organized by input-output characteristics of the problems to which they apply. We have found that these templates are helpful in teaching logic programming in our *Comparative Programming Languages* course.

In this paper, we will present the common misconceptions of imperative programmers as they begin programming in Prolog, and our attempts to dispel them. We will also present and analyze the templates we have developed to help our students make the transition from imperative to declarative programming. Finally, we will list some of the problems that our students have successfully solved over the years, as evidence of the effectiveness of the tools and techniques we present.

1 INTRODUCTION

In most schools, Computer Science students begin programming in an imperative programming language. Many of them find it very hard to learn a declarative language such as Prolog after getting into an imperative mind-set. We have identified common misconceptions and pitfalls to which imperative programmers are susceptible as they begin to learn Prolog. In order to counter these misconceptions, we have recast the semantics of Prolog constructs in terms of imperative constructs.

Often, when imperative programmers learn Prolog, they tend to write imperative rather than declarative code in Prolog syntax, using `assert()`s and `retract()`s.

In order to introduce declarative rather than imperative programming in Prolog, we have devised a set of templates. We have arranged these templates according to the characteristics of the problems to which they are applicable, an arrangement that helps students recall and reuse the right template in a programming situation. We have used these templates to teach Prolog in our *Comparative Programming Languages* course for several years. We have found that teaching with templates expedites the instruction of Prolog, while at the same time providing students with a better understanding of declarative programming in Prolog. In this paper, we will present and analyze several of these templates.

In Section 2, we will present some common misconceptions among imperative programmers about the data types and variables in Prolog. In Section 3, we will recast the semantics of Prolog constructs in terms of imperative constructs. We will present and analyze several templates for writing declarative programs in Prolog in Section 4. Finally, in Section 5, we will list some of the problems our students in *Comparative Programming Languages* course have successfully solved in pure Prolog using the material we present in this paper.

2 THE DATA TYPES AND DATA VARIABLES

One of the stumbling blocks for imperative programmers learning Prolog is variables. Variables in Prolog are different in many ways from their counterparts in imperative languages:

- Variables in Prolog are not typed. Therefore, we need not "declare" variables before using them. Imperative programmers are often so wedded to the concept of declaring variables before using them (a good/required practice in imperative programming) that unless they are disabused of this notion for Prolog, they are reluctant to "create" new variables "on the fly". This leads them to write incorrect code as described next.
- Variables in Prolog cannot be assigned, but only instantiated. Once a variable is instantiated to a value, the only way one can change its value is by un-instantiating it first through backtracking. Imperative programmers, who assume that instantiation and assignment are one and the same tend to write code such as:

```
factorial( Number, Factorial ):-
        NewNumber is Number - 1,
        factorial( NewNumber, Factorial ),
        Factorial is Number * Factorial.
```

  Note that the same variable `Factorial` is used to save both the value of `Number`! and (`Number – 1`)! Needless to say, this code fails to work.
- The behavior of the matching algorithm confounds the understanding of imperative programmers who are already unclear about the distinction between assignment and instantiation. When an uninstantiated variable is matched with a constant, the variable is instantiated to the constant. However, when an instantiated variable is matched with a constant, the variable is *compared* with the constant, returning true if the two values are the same, and false otherwise. The following clause, written once too many times by imperative programmers, illustrates this point:

```
Number is Number – 1
```

  Whereas imperative programmers expect `Number` to be assigned the value of `Number – 1`, instead, the value of `Number` is compared with `Number – 1`, a

comparison that inevitably fails. So, even though the code does not generate a syntax error, it does not seem to work "as intended", leaving imperative programmers quite puzzled.

Therefore, we encourage our students to never reuse variables, but rather to generously create and use new variables to hold the intermediate results.

We encourage our students to limit themselves to the purely declarative subset of Prolog [1] when they write programs. In this spirit, we encourage them to use the list as the universal data structure. This is a recursive data structure that permits only sequential access. Although imperative programmers feel constrained by its lack of random access (a la' arrays), they quickly realize that when combined with untyped variables, the list data type subsumes both the array and the structure types of imperative programming languages.

3 THE CONTROL CONSTRUCTS

Teaching in terms of what students already know is a standard practice in educational circles. In this spirit, we present Prolog in terms of the control constructs found in all imperative programming languages: sequence, selection, repetition and abstraction

Sequence: The equivalent of statements in imperative languages is clauses in Prolog. The delimiter for clauses is a period. Just as statements in imperative languages are executed top to bottom, clauses in Prolog databases are tried top to bottom, and literals in a goal are attempted left to right.

Even though the literals in a Prolog goal are executed left to right, they are not executed sequentially, but rather conditionally. E.g. consider the following goal:

```
?- subgoal-One(), subgoal-Two(), subgoal-Three().
```

On the face of it, imperative programmers assume that the above goal is equivalent to the following sequence of imperative statements:

```
subgoal-One();
subgoal-Two();
subgoal-Three();
```

In fact, the Prolog goal is executed as:

```
if( subgoal-One() )
   if( subgoal-Two() )
      if( subgoal-Three() )
         return true;
```

This refers to the simple case when all the three sub-goals can be satisfied without any backtracking. If backtracking is involved, the equivalent imperative code would be:

```
while( subgoal-One() )
   while( subgoal-Two() )
      if( subgoal-Three() )
         return true;
```

If more than one answer is desired for the goal, the equivalent imperative code would now be:

```
while( subgoal-One() )
   while( subgoal-Two() )
      while( subgoal-Three() )
         if( enough answers found )
            return true;
```

The above statement is simply a procedural description of the depth-first behavior of the inference engine of Prolog. However, we find that casting this behavior in terms of the execution of imperative statements is helpful to imperative programmers.

Selection: Two/Multi-way selection in Prolog is implemented simply by writing a separate clause for each case. The clauses are listed in the database in the order in which they should be tried in the program. E.g., the C++ code to determine the letter grade corresponding to a numerical grade is:

```
if( grade >= 90 )
   letter_grade = 'A';
else if( grade >= 80 )
   letter_grade = 'B';
else if( grade >= 70 )
   letter_grade = 'C';
else if( grade >= 55 )
   letter_grade = 'D';
else
   letter_grade = 'F';
```

The corresponding Prolog code would consist of five distinct clauses in the database:

```
grade( Number, Letter ):- Number >= 90, Letter = a.
grade( Number, Letter ):- Number >= 80, Letter = b.
grade( Number, Letter ):- Number >= 70, Letter = c.
grade( Number, Letter ):- Number >= 55, Letter = d.
grade( _, Letter ):- Letter = f.
```

Imperative programmers find it unusual that these five clauses are not "syntactically encapsulated" – something they are used to expect in imperative languages. Therefore, we find it necessary to emphasize that these five distinct clauses are together semantically equivalent to the single nested `if-else` statement in C++. Anonymous variables are another concept with which imperative programmers are unfamiliar. This example illustrates the use of an anonymous variable – if the execution reaches the last clause, we want the goal to match the clause regardless of the value of the numerical grade.

Repetition: Recursion is used to implement repetition in Prolog. A recursive definition includes one or more base cases and one or more recursive cases. Base cases are usually facts, and recursive cases are rules in the database. Each case is a separate clause, and the order in which they are listed in the database is important to the correctness of the program: base clauses must be listed before recursive clauses. We will consider several recursive examples in the next section in the context of input/output templates.

We may also use backtracking with `fail` to implement repetition, especially when we want to generate all the possible answers to a query. E.g., if a library lists all

its books in a database using the predicate `holding(author, title, year)`, the following goal would repeatedly print the titles of all the books acquired in 2001:

```
?- holding( _, Title, 2001), write( Title ), nl, fail.
```

Abstraction: The equivalent of functions in imperative languages are clauses: facts and rules:

- Facts are trivial functions. They accomplish their purpose through matching of arguments.
- Rules are equivalent to the traditional functions in imperative languages, with a header (head) and a body. There are some significant differences between functions in imperative languages and rules in Prolog:
    - Since variables are not typed, programmers should feel free to generously use new variables while writing the body of a rule, without having to declare them in advance as they would in an imperative language. We have noticed that without this explicit encouragement, students tend to "reuse" existing variables and abstain from creating new ones, which results in incorrect code as described earlier.
    - Due to the nature of the variables and the matching algorithm, parameters in Prolog clauses are passed either by value (copied from the goal's argument to a database clause's argument at the time the goal is matched with the database clause) or by result (copied from a database clause's argument to the goal's argument at the time the sub-goals generated by the database clause are satisfied), and *never* by reference or value-result (See Table 1). Copy semantics is always used for parameter passing.

- Each goal literal is a function call. Therefore, the body of a rule is nothing but a series of function calls, and their execution is conditional rather than sequential as described under the Sequence section.

Since variables are not typed, Prolog clauses naturally implement the polymorphic behavior of templates in imperative languages. Not only does template polymorphism come naturally to Prolog, the fact that it predates the inclusion of template polymorphism in C++ can be used to give students an appreciation for the importance of knowing the history of computing and the evolution of languages.

Prolog clauses permit the equivalent of "function overloading" in imperative languages. We can write multiple clauses for a given predicate: clauses with the same signature are considered to be alternatives, whereas clauses with differing signatures are considered to be unrelated.

Pure Prolog does not permit global variables. The boolean value returned by individual clauses controls only the conditional execution of the Prolog program, as described earlier in the Sequence section. So, *all* the results of computation in a Prolog program *must be* communicated among the clauses through their arguments, i.e., "parameters". Imperative programmers must realize this before they can write any large Prolog programs consisting of multiple clauses.

In Table 1, we summarize the behavior of scalar variables used as parameters. In Table 2, we present the behavior of list variables used as parameters. Note that when using list variables as parameters, the parameters themselves may be used for elementary list operations such as accessing elements and constructing a list from its

elements – we do not have to introduce any additional literals in a clause to do these jobs.

Argument in Goal Literal (Actual Parameter)	**Argument in Database Clause (Formal Parameter)**	**Significance**
Uninstantiated Variable	Uninstantiated Variable	*Aliasing*
Uninstantiated Variable	Instantiated Variable	*Pass By Result*
Instantiated Variable	Uninstantiated Variable	*Pass By Value*
Instantiated Variable	Instantiated Variable	*Comparison for Equality*

Table 1: Matching Scalar Arguments of a Goal Literal with those of a Database Clause

Argument in Goal Literal (Actual Parameter)	**[H | T] Argument in Database Clause**	**Significance**
Uninstantiated Variable	Uninstantiated Variables H, T	*Aliasing*
Uninstantiated Variable	Instantiated Variables H, T	*List Construction*
Instantiated Variable	Uninstantiated Variables H, T	*List Access*
Instantiated Variable	Instantiated Variables H, T	*List Comparison*

Table 2: Matching Vector Arguments of a Goal Literal with those of a Database Clause

4 THE TEMPLATES

We have devised a set of templates to help our students make the transition from imperative programming to declarative programming in Prolog. These templates present idiomatic Prolog clauses for typical applications. We have organized the templates in terms of the input-output characteristics of the problems they can be used to solve. This organization helps students recall and reuse the right template for a

given problem. We will discuss our templates for the following input-output combinations:

- Scalar Input
  - Output by Side-Effect
  - Scalar Output
  - Vector Output
- Vector Input
  - Output by Side-Effect
  - Scalar Output
  - Vector Output

Note that scalars are terms (constants and variables) and vectors are lists. Side-effect is through `read()` and `write()` clauses. In our *Comparative Programming Languages* course, we use similar templates to also teach LISP [3], thereby enabling our students to compare and contrast logic-based programming with functional programming.

We will now present each template with an example. Since these templates present recursive solutions, we present them in terms of base case(s) and recursive case(s). In the recursive case, we differentiate between local action, i.e., the work done locally within a clause, and the work done by the recursive call. The generalized structure of the templates is:

```
base-case().
recursive-case():- local-action(), recursive-call().
```

4.1 Scalar Input, Output by Side-Effect

Problem: Write a program to print the squares of all the numbers from a given number down to 1.

```
% Base Case – Stop printing when Number is 0
printSquares( 0 ).
% Recursive Case – Print Number, call recursively on Number - 1
printSquares( Number ) :- Square is Number * Number,
                          write( Square ), nl,
                          NewNumber is Number - 1,
                          printSquares( NewNumber ).
```

Analysis: We begin with this example to illustrate how the base case and recursive case are two distinct clauses, listed in that order in the database. The base case is a fact, and the recursive case is a rule. The scalar input is passed as an argument to the clauses. The clause for the recursive case changes the value of the input `Number` in the direction of its base case value, viz., 0.

If the database is queried with a negative number, the result would be infinite recursion! This could be used as a lead-in to discuss making end conditions robust. In general, the *weakest* end condition(s) must be used. Whereas the above end condition tests for `Number` being 0, a more robust design would test for negative numbers as well. This could be done simply by changing the base case into a rule as follows:

```
% Base Case – Stop printing when Number is 0 or less
printSquares( Number ):- Number =< 0.
```

This example also illustrates the concept of *tail recursion*, where the recursive call is the last literal in the body of the recursive case. During tail recursion, all the local action in a clause is done before the recursive call, and none after returning from the recursive call, as may be clarified using a recursion tree. Imperative programmers find it interesting to note that any clause with tail recursion can be rewritten as a loop. This also drives home the more general point that recursion is just an alternative to iteration: it demystifies recursion for imperative programmers.

In order to illustrate *head recursion,* where the recursive call precedes all local action in a recursive case, we will consider the next problem.

Problem: Modify the above program to print the squares of all the numbers from 1 up to a specified number.

```
% Base Case – Stop printing when Number is 0 or less
printSquares( Number ):- Number =< 0.
% Recursive Case – Call recursively on Number–1, then print Number
printSquares( Number ) :- NewNumber is Number - 1,
                          printSquares( NewNumber ),
                          Square is Number * Number,
                          write( Square ), nl.
```

Analysis: The only difference between this and the previous code is that the recursive call occurs before the local action in the recursive case. This is an example of *head recursion*, where all the local action is done after returning from the recursive call. It is enough to reorder the literals in the body of the recursive rule to reverse the order in which the squares are printed!

An extension of this discussion might be to consider the rule wherein the recursive call is performed both before and after the local action in the body of the rule. Therefore, the squares would be printed first in descending order, followed by ascending order.

4.2 Scalar Input, Scalar Output

Problem: Write a program to calculate the factorial of a number.

```
% Base Case – Factorial of 0 is 1.
factorial( 0, 1 ).
% Recursive Case – N! = N * (N – 1)!
factorial( Number, Factorial ):-
           NewNumber is Number - 1,
           factorial( NewNumber, NewFactorial ),
           Factorial is Number * NewFactorial.
```

Analysis: Since the output is a scalar, and parameters are the *only* means through which Prolog clauses can communicate results with each other, we need a second argument for the scalar output. Typically, the input argument is passed by value and the output argument is passed by result. In the recursive case, the result obtained from the recursive call is *composed* with the input to calculate the output.

Note that the base case returns the *identity element* for the operation used in the recursive case to compose the result of the recursive call with the input, e.g., 1 for multiplication, 0 for addition.

4.3 Scalar Input, Vector Output

Problem: Write a program to return the list of factorials of all the numbers from a given number down to 0.

```
% Base Case – If the Number is negative, return a null list
listFactorials( Number, [] ):- Number < 0.
% Recursive case – Insert the factorial of the current number
%   as the head of the list returned by the recursive call
listFactorials( Number, Vector ):-
            NewNumber is Number - 1,
            listFactorials( NewNumber, NewVector ),
            factorial( Number, Factorial ),
            Vector = [ Factorial | NewVector ].
```

Analysis: Since the output is a vector, we need a second argument for the vector output. In the recursive case, the result of the local action is inserted as the head and the result of the recursive call is inserted as the tail of a new list – illustrating a classic technique of list construction used in Prolog. In the base case, the null list is returned, which is the identity element for list construction.

Whenever the body of a Prolog rule contains an equality predicate with a formal parameter on one side, the predicate can be eliminated in the body and the formal parameter in the head of the rule can be replaced with the other side of the equality predicate. E.g., in the recursive case above, the formal parameter `Vector` can be replaced by `[ Factorial | NewVector ],` and the equality predicate eliminated from the body as follows:

```
listFactorials( Number, [ Factorial | NewVector ] ):-
            NewNumber is Number - 1,
            listFactorials( NewNumber, NewVector ),
            factorial( Number, Factorial ).
```

Although novice programmers find this substitution confusing at first, once they understand how work is done in Prolog programs during the matching of arguments (e.g., see Tables 1 and 2), they adopt this style. (For a procedural interpretation of this substitution, please see [2], page 2078.)

What if we wanted to generate the list of factorials in ascending rather than descending order? Simply exchanging the head and tail during list construction as follows will not produce the desired result:

```
listFactorials( Number, [ NewVector | Factorial ] ):- …
```

Since `Factorial` is not itself a list, we will end up with "dotted pairs". Enclosing `Factorial` in list notation will not solve the problem either:

```
listFactorials( Number, [ NewVector | [ Factorial ] ] ):- …
```

We will end up with a list of nested lists rather than a simple list. The solution is to use `append()` predicate to append the result of the recursive call to the list of the result of the local action:

```
listFactorials( Number, Vector ):-
            NewNumber is Number - 1,
            listFactorials( NewNumber, NewVector ),
```

```
            factorial( Number, Factorial ),
            append( NewVector, [ Factorial ], Vector ).
```

This example illustrates another classic technique of list construction in Prolog.

4.4 Vector Input, Output by Side-Effect

Problem: Write a program to print the elements of a list, in order.

```
% Base Case – Stop when the list is empty
printElements( [] ).
% Recursive Case – Print the first element,
%    Call recursively on the rest
printElements( [ First | Rest ] ):-
           write( First ), nl,
           printElements( Rest ).
```

Analysis: Since the input is a vector, the base case checks to see if the input is an empty list. In the recursive case, the local action is performed on the head of the list, and the tail of the list is handed to the recursive call. These characteristics are common to all the templates for vector input that follow.

The above template is once again an example of *tail recursion*. By using *head recursion*, we can reverse the order in which the elements of the list are printed. This involves reversing the order of the local action (`write( First )`) and the recursive call in the body of the recursive rule.

If we want to print the *factorials* of the elements of the input list, we simply insert the `factorial()` predicate in the body of the recursive rule, as follows:

```
printElements( [ First | Rest ] ):-
           factorial( First, FirstFactorial ),
           write( FirstFactorial ), nl,
           printElements( Rest ).
```

This illustrates the ease with which PROLOG clauses can be composed to build larger programs. It illustrates why Prolog is well suited for rapid prototyping applications.

4.5 Vector Input, Scalar Output

Problem: Write a program to obtain the sum of all the elements of a list.

```
% Base case – If the list is empty, the sum is 0
addElements( [], 0 ).
% Recursive Case – Add the first element to the partial sum returned
%   by the recursive call on the rest of the list
addElements( [ First | Rest ], Sum ):-
            addElements( Rest, NewSum ),
            Sum is First + NewSum.
```

Analysis: All the features that were introduced in the last template have been carried over to this template. Note that the base case returns the identity element (0) for the operation (+) used in the recursive case to compose the result of the local action with the result of the recursive call. In the recursive case, the local action is performed on the head of the list and the tail of the list is handed to the recursive call.

4.6 Vector Input, Vector Output

Problem: Given a list of numbers, return a list of their respective squares.

```
% Base Case – If the list is empty, return an empty output list
squareElements( [], [] ).
% Recursive Case – Insert the square of the first element
%   as the head of the list returned by the recursive call
squareElements( [ First | Rest ], [ NewFirst | NewRest ] ):-
          NewFirst is First * First,
          squareElements( Rest, NewRest ).
```

Analysis: Since the output is a vector, the base case returns the identity element for list construction, viz., []. In the recursive case, the result of the local action is inserted as the head and the result of the recursive call is inserted as the tail of the list that is returned. Just as in the case of the Scalar Input, Vector Output template, we can reverse the order of the output list by using the `append()` predicate.

In some problems, we may want to process only selected elements of the input list. E.g., given a list of numbers, we may want to return a list of only those numbers that are greater than or equal to a threshold value. We can implement this by introducing a second recursive case in which the result of the local action is *not* composed with the result of the recursive call.

```
% Base Case – If the list is empty, return an empty output list
filterList( [], [], _ ).
% Recursive Case – If the head of the list is
%   greater than or equal to the threshold,
%   insert it as the head of the list returned by the recursive call
filterList( [ First | Rest ], [ First | NewRest ], Threshold ):-
          First >= Threshold,
          filterList( Rest, NewRest, Threshold ).
% Recursive Case– If the head of the list is less than the threshold,
%   simply return the result of the recursive call.
filterList( [ _ | Rest ], NewRest, Threshold ):-
          filterList( Rest, NewRest, Threshold ).
```

4.7 Factoring Out Code

Consider another vector-input, scalar-output problem: the problem of finding the largest element in a list. Beginning Prolog programmers may attempt a solution that includes multiple recursive calls, and is hence inefficient:

```
% Base case – If the list has only one element, it is the largest
findLargest( [ Number ], Number ).
% Recursive Cases – Find the largest element in the rest of the list
%    Compare it with the first element to determine the
%    largest element of the entire list
findLargest( [ First | Rest ], Largest ):-
          findLargest( Rest, RestLargest ),
          First >= RestLargest,
          Largest = First.
findLargest( [ First | Rest ], Largest ):-
          findLargest( Rest, RestLargest ),
          First < RestLargest,
          Largest = RestLargest.
```

After a little reflection we can see that we can eliminate the recursive call in the last clause:

```
% Recursive Cases
findLargest( [ First | Rest ], Largest ):-
          findLargest( Rest, RestLargest ),
          First < RestLargest,
          Largest = RestLargest.
findLargest( [ First | _ ], First ).
```

However, this formulation of the solution is not quite readable. Moreover, if the user attempts to re-satisfy a query, these clauses would generate incorrect results. A better solution would be one that involves **factoring out code**, a principle that is relevant to all the paradigms of programming, but especially so to declarative programming. In this example, we may "factor out" the code to compare two numbers and determine the larger number, and introduce a new predicate to carry out this task:

```
% Base case – If the list has only one element, it is the largest
findLargest( [ Number ], Number ).
% Recursive Cases – Get the largest element in the rest of the list
%    Compare that with the first element to determine the
%    largest element of the entire list
findLargest( [ First | Rest ], Largest ):-
          findLargest( Rest, RestLargest ),
          getLarger( First, RestLargest, Largest ).

% If the first argument is greater than or equal to the second,
%     it is the larger number
getLarger( First, Second, First ):- First >= Second.
% Otherwise, the second argument is the larger number
getLarger( First, Second, Second ):- First < Second.
```

Again, we could skip the comparison `First < Second` in the second `getLarger()` clause, but the clause would produce incorrect results if backtracking occurs.

Finally, note that the above clauses will fail if the input list is empty. This is not a bug, but rather correct behavior, since an empty list does not have a largest element. This example illustrates another principle of Prolog programming, which imperative programmers find to be rather unintuitive: if we want a goal to fail for certain inputs, we simply do *not* handle those cases of input in the clauses in our database!

4.8 Putting the Templates Together

Finally, we can use the following algorithm to solve complex problems using the various templates:

1. Perform a top-down decomposition of the problem;
2. Analyze the input-output characteristics of the resulting sub-problems;
3. Use the appropriate template to write the clauses for each sub-problem;
4. Finally, compose these clauses to write the overall program.

Problem: Write a program to calculate the average of a list of non-repeating numbers after eliminating the outliers.

Solution:

- Finding an outlier (maximum or minimum) is a vector input/scalar output problem.

- Eliminating an outlier from a list is a vector input/vector output problem.
- Calculating the sum of the elements in a list and counting the number of elements in a list are vector input/scalar output problems.

Our overall solution might look like this:

```
average( Vector, Average ):-
       % Remove the highest element in the list
       findLargest( Vector, Maximum ),
       delete( Vector, Maximum, VectorSansMax ),
       % Remove the lowest element in the list
       findSmallest( VectorSansMax, Minimum ),
       delete( VectorSansMax, Minimum, VectorSansMin ),
       % Calculate the average of the remaining elements
       length( VectorSansMin, Count ),
       addElements( VectorSansMin, Sum ),
       Average is Sum / Count.
```

It is instructive to trace the flow of data among the literals in the above rule with arrows. Such an analysis would demonstrate the "one-way" nature of the flow of data in Prolog. Tracing the arrows head to tail, and starting with the input parameters in the head of the rule, we can see how data is handed back and forth between the clauses in the body of the rule, until eventually, the output parameters are generated.

5 CONCLUSIONS

We have identified common misconceptions and pitfalls to which imperative programmers are susceptible as they begin to learn Prolog. In order to counter these misconceptions, we have recast the semantics of Prolog constructs in terms of imperative constructs. We have also presented a set of templates with which to introduce logic programming in Prolog to imperative programmers. We have arranged these templates according to the characteristics of the problems to which they are applicable, an arrangement that helps students recall and reuse the right template for a given problem.

Our experience suggests that the above set of templates is sufficient for writing Prolog programs to solve most problems. Teaching Prolog from the beginning, up to and including these templates takes us about one week of classroom instruction. We have used the templates and the other material presented in this paper to teach Prolog in our *Comparative Programming Languages* course for several years. Some of the problems our students have solved in this course in pure Prolog include: Battleship / Salvo, Othello / Reversi, Domino Deluxe, Dice games (Pokerdice, Craps), Card games (Spades, Crazy 8, Blackjack), Snakes and Ladders, Stock Market simulation, LOGO and Hare and Tortoise. Our students had three weeks to write these programs, and were programming in Prolog for the first time in their academic career. They were restricted to programming in pure Prolog. We believe that the material we have presented in this paper, when taught in addition to basic Prolog syntax and semantics, helps students quickly make the transition from imperative to declarative programming.

REFERENCES

[1] Clocksin, W.F. and Mellish, C.S.: Programming in Prolog, Springer Verlag, 1981.

[2] Cohen, J.: Logic Programming and Constraint Logic Programming. In *The Computer Science and Engineering Handbook*, (Ed. A.B. Tucker), CRC Press, Boca Raton, FL, 1997.

[3] Kumar, A.: Using Patterns to Teach Recursion in LISP. Proceedings of the Eastern Small College Computing Conference (ESCCC '96), Marywood College, Scranton, PA, 10/25-26/96, 80-84.