LOGICON: an Integration of Prolog into ICON

Guy Lapalme
Suzanne Chapleau

Département d'informatique et de
recherche opérationnelle (I.R.O.)
Université de Montréal
C.P. 6128, Succ. A
Montréal, Québec
Canada, H3C 3J7

ABSTRACT

This paper describes the coupling of logic programming with Icon, which is a programming language aimed at string processing. Icon and Prolog have many similarities and their integration is feasible and desirable because the weaknesses of one can be compensated for by the strengths of the other. In our case, a Prolog interpreter was written as an Icon procedure that can be linked and called by an Icon program. This interpreter deals with all Icon data types and can be called in the context of the goal directed evaluation of Icon. We give an example showing the power of this symbiosis between these two languages where a Prolog call in Icon is a generator and an Icon call in a Prolog clause is a built-in predicate.

Keywords:

Icon, Prolog, logic programming, language integration.

INTRODUCTION

Prolog and Icon are two high level programming languages with different concepts. Prolog is based on predicate logic giving facts and relations between these facts. Execution tries to prove a new fact from the ones already known.

Icon is more traditional; it is an expression-based language with a syntax similar to C and Pascal, but each expression may return a value or not; the latter case is interpreted as failure. Prolog and Icon share two main features: an expression can generate many different solutions and does so by backtracking. This feature allows a real "symbiosis" between these languages where weaknesses of Prolog in sequential computing can be overcome by the power of Icon in this respect.

MAIN FEATURES OF ICON

Icon [1] was developed as a language for processing non-numeric data. Superficially, Icon programs resemble those in Algol-like languages, but they have some unusual features: an Icon expression can yield more than one result. The evaluation mechanism does a depth-first search with backtracking over all generative components of an expression to give either the first result or all possible results. An expression that can give more than one result is called a generator; on its initial evaluation, it gives a first result, but it can be reactivated by a suitable operator or construct to give all other solutions one by one.

For example,

```
upto(SetOfCharacters, String)
```

gives the positions of a letter of SetOfCharacters in String.

```
upto('a', "abracadabra")
```

returns 1 on its first call and 4,6,8 and 11 on each of the succeeding resumptions. {1,4,6,8,11} is called the sequence of results of that expression.

Expressions produce a value, if evaluation succeeds. When the expression has no result to give, it fails. The control flow of an Icon program is driven by success and failure.

In addition to the classical assignment, arithmetic, string and comparison operators, the following Icon operators deal with sequences of results.

Operators	*Resulting sequence of results*
e1 **to** *e2*	*e1*, *e* + 1, *e1* + 2, ... *e2*
!*e1*	all elements of *e1* where *e1* is a string, a list or a record
e1||*e2*	results of *e1* followed by results of *e2*
||*e1*	repeatedly all results of e1
e1 \e *e2*	only the first *e2* results of *e1*
**e1*	number of results in *e1*

There are a number of built-in functions on character strings that can be used as generators. Users can also define generators by using **suspend** instead of **return** to end a function.

In this case, the function returns the value of the expression following **suspend** and may be subsequently resumed to get the next result of the sequence.

For example:

```
procedure LCM(min,max)          procedure CM(min,max)
local mult                      local mult
if min > max then min:=:max     if min > max then min:=:max
mult:=min                       mult:=min
repeat{                         repeat{
  if mult % max=0 then            if mult % max=0 then
    return mult                     suspend mult
  mult:=mult+min}                 mult:=mult+min}
end                             end
```

LCM is a procedure giving the lowest common multiple of its two parameters while CM is a generator giving a new common multiple at each call.

As illustrated in the previous example, Icon has a set of "traditional" control structures for selective and iterative constructs, but they are expressions that yield results or fail. The main control mechanism for generators is

```
    every e1 do e2
```

which evaluates *e2* for each result in the sequence generated by *e1*.

```
    every i:=upto('a', "abracadabra") do write(i)
```

writes 1 4 6 8 11 on the standard output.

```
    while i:=upto('a', "abracadabra") do write(i)
```

loops undefinitely writing 1 each time it goes through the loop because upto is not called within a generating context; each call starts the sequence of results anew.

If **every** is followed by an expression with many generators, then all valid combinations are tried in a depth-first strategy. For example,

```
    every write (0 to 3, 0 to 9, 0 to 9)
```

writes all numbers less than 400 in increasing order: the last generator gives all its results before the preceding one changes, when it does the last one starts its sequence anew.

Icon has 10 predefined types. Variables are dynamically typed; declarations specify only scope (local or global) or allocation (static or dynamic). The type of a value can be determined at run-time by calling the built-in function type and is initially of the &null type. Icon types are: integer, real, character, file, procedure, variable length character strings, of any type, associative tables and records. The latter permit the definition of new types by combining values of other types.

The last type is the co-expression, which relates to an expression as a coroutine relates to a procedure. Co-expressions are needed because the scope of a generator is limited to the context of the expression in which it appears. For example,

```
every write (upto ('a', "abracadabra"))\e3
every write (upto ('a', "abracadabra"))\e2
```

writes 1 4 6 followed by 1 4 because the second expression "reevaluates" the call and does not reactivate it.

There is no mechanism for explicitly resuming an expression to get the next result. To overcome this restriction, co-expressions "capture" an expression and its context so that it can be resumed at any time or at any place in a program. The operations defined on a co-expression are:

create *expr* : gives a value that can be used to activate the expression keeping its current context.

@expr : activates the co-expression.

^expr : gives a copy of the co-expression as it was initially defined.

For example,

```
up :=  create upto ('a', "abracadabra")
every write (||@up \e 3)
every write (||@up \e 2)
```

writes 1 4 6 followed by 8 11. Co-expressions can be used to build coroutines and form a powerful control flow mechanism.

MAIN FEATURES OF PROLOG

Prolog [2, 3] is a non procedural programming language where a problem is described by facts and relations between them. The execution is started by trying to prove a fact from the existing facts and relations. Relations follow this format:

predicate : - list of predicates

where a predicate is the name of a relation followed by a parameter list, each being an identifier or a variable (given here as an identifier starting with a capital letter). For example

```
brother(X,Y) :- male(X), parents(X,Ma,Pa), parents(Y,Ma,Pa)
```

states that X is the brother of Y provided X is a male, and that X and Y have the same parents. The call

```
? - sister (X, alice)
```

yields all sisters of "alice" one by one, much like an Icon generator. This behavior is accomplished by finding all possible values for X such that sister can be proven from the existing relations. The proof is done using unification and the resolution principle [4]. Unification is a generalization of pattern matching and is the way to pass information between predicates. Another

important characteristic of Prolog is the reversibility of the programs: the same specification can be used to construct a result from its components or for finding all components that can be combined to obtain a given result.

The following example defines the relation between three lists such that the third is the concatenation of the first two. The DEC-10 Prolog Syntax [5] is used: [X|Y] indicates a list whose X is the head and Y is the tail. A three-element list is represented as [do,re,mi] where the head is do and the tail is [re,mi]. [] denotes the empty list.

```
append ([], L, L).
append ([A|B], C, [A|L ]) :- append (B, C, L).
```

These two relations state that if the first list is empty, then concatenation is the second list, otherwise the resulting list is composed of the first element of the first list A followed by L defined recursively as the concatenation of the rest of the first list B and the second list C

If we now try to prove the relation

```
?-append ([do, re, mi], [fa, sol, la], X)
```

the first relation does not match because the first list is not empty, but the second one does and thus

```
A = do
B = [re, mi]
C = [fa, sol, la]
```

and L will be the third parameter of

```
append ([re,mi], [fa, sol, la], L)
```

Append is called recursively three times, until the first parameter is empty and it then returns the second list as the result. At each level, the first element of the list is saved in the partially constructed third parameter; when the last element is known, all of the "calls" return and the result

```
X = [do, re, mi, fa, sol, la]
```

is constructed.

This example showed the usual list concatenation where the first two lists are known and constant. But these relations can also be used for other purposes: if we try to prove the relations with the first argument free (i.e., with a variable) and the other two constant, we get the list that should be concatenated to the second to get the third.

For example,

```
?-append (X,[fa, sol, la], [do, re, mi, fa, sol, la])
```

returns

```
X = [do, re, mi]
```

By the same reasoning, if the first two parameters are free and the third constant, we get all possible combinations such that their concatenation gives the list given as the third parameter.

```
?-append (X, Y, [do, re, mi, fa, sol, la])
```

yields the following sequence of results:

```
X = []    Y = [do, re, mi, fa, sol, la]
X = [do]  Y = [re, mi, fa, sol, la]
  :
X = [do, re, mi, fa, sol, la]   Y = []
```

An order to write an Icon procedure with the same behavior, a case analysis of the parameters and a different algorithm for each case would be required. Here are a few cases:

```
procedure append (x,y,z)
  local i
   ....
  # x,y are given and z is the result
     z := x ||| y
   return
   ....
  # y,z are given and x is the result
     i := *z;
     while y[i] = z[i] do i := i-1
     x := z[1:i]
     return
   ....
  # z is given and x and y are free
   every i := 1 to *z do {
         x := z[1:i]
         y := z[i:0]
         suspend
     }
   ....
```

Both Prolog and Icon do a depth-first with backtracking, of their solution spaces. For example, in Prolog, given the following relations:

```
note (do).      octave (first).
note (re).      octave (second).
note (mi).      octave (third).
note (fa).
note (sol).
note (la).

combination (O,N) :- octave (O), note (N).
```

Solving

```
combination (O,N)
```

gives

```
O=first     N=do
O=first     N=re
  ...       ...
O=second    N=do
  ...       ...
O=second    N=la
O=third     N=do
  ...       ...
O=third     N=la
```

In Icon

```
    octave := ["first", "second", "third"]
    note   := ["do", "re", "mi", "fa", "sol", "la"]

  every write (!octave, !note)
```

writes

```
  first do
  first re
   .....
```

in exactly the same order as the Prolog predicate. The logical aspect of Prolog presented here makes it harder to write programs because it lacks such "mundane" features as input/output, arithmetic operators, tests on the status of a variable, etc. Predicates have been defined to fill those needs. They are not used for their logical results, which are usually meaningless, but for their side effects (e.g. writing or reading).

INTEGRATION OF PROLOG INTO ICON

Solving a Prolog clause and executing an Icon expression have the following similarities:

- Prolog matches one clause at a time and Icon evaluates one expression at a time.

- A Prolog clause can refer to other clauses, each of them being able to give more than one result. An Icon expression can be decomposed into many sub-expressions each of them possibly generating more than one result.

- A Prolog system can give all solutions of a request by doing a depth-first search with backtracking on the different alternatives. Icon uses a similar strategy to generate the sequence of results.

- A Prolog clause and an Icon expression terminate by failing when their sequence of results is exhausted.

- Prolog and Icon derive much of their power from being able to express pattern matching concisely: Prolog uses the unification algorithm to match and even construct data structures; Icon uses functions generators and assignments.

The integration of Prolog and Icon is done on these grounds of similarity, but the "personality" of each language is retained. A Prolog request in Icon acts as a generator and Icon code in a Prolog request is used as a built-in predicate having the same attributes as another Prolog goal.

One major difference between these languages is the way results are generated. In Prolog, results are returned as the final values of the variables (which is why Prolog interactive systems often show the final values of the variables after a request has succeeded). On the other hand, in Icon the result is the final value of the expression; assignments done to variables have no direct consequences on the final result of an expression.

We have implemented a Prolog interpreter as an Icon generator that deals with the partially defined variables that can be completed during a proof. These effects are undone automatically when backtracking; the variables get values by unification with a data base of facts that must be maintained. The next section shows how this can be implemented, but first we present the way to insert Prolog code in an Icon program as a generator (we call it the external interface) and how to put Icon code in Prolog to behave as a goal (internal interface). These insertions are done without any preprocessing; this is "standard" Icon code calling precompiled procedures and using predefined Icon records.

Prolog term representation

For constant terms, the Icon types real, integer, string, cset, table, files, procedure and co-expression can be used; lists are dealt with below. The last five types are not usually found in Prolog. In pure Prolog, the only operation is unification, which we implement as the Icon equality test, defined for all Icon types. The actual data types matter only in the built-in predicates, which are defined outside of Prolog, so these additional types have no impact on the Prolog interpreter.

One novel feature of Prolog is the logical variable that can only be assigned once but can retain undefined parts, which can be filled in later. An example of this behavior appears in the append relation where the third parameter was built from a value and of another logical variable that would be defined later. A logical variable is represented by an instance of the Icon record.

```
record logical (value)
```

A free Prolog variable has &null as the value of its corresponding Icon component.

Thus, the Prolog interpreter procedure can differentiate between variables and constants by testing the type of the value of the variable at run time.

The complex terms of Prolog are composed of a functor with a fixed number of components. These components correspond exactly to the record structure of Icon. The only implication for Prolog programmers is that all functors used in a program must be declared.

A list could be defined using a binary functor linking its head and its tail. This notation is quite cumbersome and the implementation of this useful construct can be more efficient with a linear structure instead of a tree. Moreover, Icon has an extensive set of built-in operators and functions for lists. So, Prolog lists are kept as Icon lists. This choice is useful and convenient for the programmer, since the Icon list notation in almost identical to the DEC10-style list notation. This choice has the unfortunate consequence that the unification algorithm is more complicated, but this complication is hidden from the programmer.

External Interface: Prolog into Icon

To enter facts or relations in the Prolog data-base, the Icon procedure assert (head, goals) in called, and to execute a request prolog(request) is used.

For each relation to be entered in the Prolog database, the user must declare a record type for the relation, declare logical variables used in the relation, and call assert. For example, to enter the append , the user declares

```
record append (f1, f2, f3)
```

and executes

```
a := logical()
b := logical()
c := logical()
d := logical()
```

These global variables are used for getting the results of the Prolog expression into Icon. Finally

```
assert (append ([], d, d))
assert (append ([a, b], c, [a,d]), append (b, c, d))
```

saves the append definition in the Prolog database of facts. The Icon procedure assert checks the syntax of the clause and keeps an internal structure binding the same variable occurrences within a clause, but keeping the ones between two clauses separate (i.e., in this example, variable d is used in both clauses, but is not the same one internally). The final results of a Prolog goal are returned in the global variables named in the clause, but with all substitutions made so that a programmer does not have access to the internal structures of the interpreter. As the prolog procedure behaves like an ordinary Icon generator, all control mechanisms can be applied; for example

```
every prolog (append (a,b,[do,re,mi,[]]) do {...}
```

executes the statements in braces for each possible bindings of a and b such that their concatenation gives the list [do,re,mi,[]] (the list must be terminated by []).

The prolog generator can also be used in a co-expression.

```
Conc := create prolog (append (a,b, [do,re,mi,[]]))
```

and called with @Conc one Prolog execution can be active at once, each one being a distinct generator controlled by the Icon program.

Internal Interface: Icon into Prolog

Ideally, the integration of Icon code in a Prolog clause should be done by substituting Icon code for a Prolog goal. For example

```
assert (append (x,y,z), z.value := x.value |||| y.value)
```

supposing that x and y were already instantiated to lists. Unfortunately, since Icon parameters are passed by value, the expression is evaluated before giving its result to assert. If we use a co-expression, such as

```
assert(append(x,y,z), create z.value := x.value |||| y.value)
```

the expression is captured, but the bindings between the logical variables and the ones of the co-expression are not available to assert. We have to make sure that, when Icon translates the co-expression, it does not create binding between logical and Icon variables that do not necessarily have the same representation. Moreover, the same logical variables (which are global) can be used in different Logicon clauses and thus these values have to be saved before the co-expression is evaluated and subsequently restored. The bindings have to be given using an intermediary call to the icon procedure, which binds the variables correctly; its implementation is given later.

```
assert (append (x,y,z),
  icon (create z.value := x.value ||| y.value, [x,y,z])).
```

Any Icon expression can appear in this context, including generators.

IMPLEMENTATION OF THE INTERPRETER

The implementation is divided into two parts: data structures and algorithms of the interpreter and the processing of Icon built-in predicates within clauses. Only the main data structures and the outlines of the algorithms and given here; further details and the Icon program appear in Ref. 6.

External Interface: Prolog into Icon

The state of the resolution steps is saved in two stacks: the execution stack is a list of environments corresponding to the currently active nodes of the search tree; the backtrack stack saves the information about alternatives yet to be examined at in the non-deterministic environments of the execution stack. All this information is kept as lists of Icon records.

The Prolog interpreter is an Icon generator that uses these two stacks to go through all alternatives and find the values of the variables needed to find appropriate bindings for the Prolog request.

The logical variables are represented using the "structure-sharing" mode of representation: it includes a pointer to a model of the structure and another pointer to the actual values of the variables of the model.

A logical free variable is an Icon value of type logical having &null as the type of its value field; when the logical variable takes a constant value, that field is set to the constant; finally when that value is a structure S whose variables are in some environment E, the field has a value of type composite having two fields: a reference to S and the name of the environment.

The unification algorithm is also standard, but it can deal with all of the Icon types. The equality tests in the unification procedure are done using the Icon equality operator (===) except for lists, which are linear structures used to represent Prolog trees. This special case introduces some complexity into the unification algorithm dealing with embedded logical variables, but doing so permits the efficient list operators and functions of Icon to be used.

Internal Interface: Icon into Prolog

This part deals with Icon code (given as a co-expression) used as a built-in predicate within a Prolog clause. We want the co-expression to behave as a Prolog goal, i.e., it works on (it instantiates) the local variables of the clause in which it appears.

A built-in predicate is a goal having the defined type icon and is implemented in four steps:

- copying of the variables that are in the parameters vector of the co-expression;
- creation of Icon environments on the execution and backtrack stacks;
- activation of the co-expression; and
- return of the results in the Prolog environment.

The following example illustrates these steps: We have the "logical" variables v,w,x,y,z. The call

```
assert (termA(x,y,z), [termB(w), termC(w,y,z),
                icon(coex(x,y,z),[x,y,z])])
```

requires that bindings for x , y and z be found such that termB(w) , termC(w,y,z) and coex(x,y,z) have consistent values, respectively. This Logicon clause is translated in the following Icon data.

```
termA(^v1, ^v2, ^v3) ->
   termB(^v4)
   termC(^v4, ^v5, ^v3)
   icon(coex(x,y,z),[var-param(^x,2),
                     var-param(^y,5),
                     var-param(^z,3)
                        ]
      )
```

where

```
v1===logical(1)
v2===logical(2)
 ...
v5===logical(5)
```

The logical variables have been renumbered consistently; the links between the variables and the Icon ones and done through var-param.

Suppose that the current environment of the execution stack has the clause termA with the following vector of variables (in structure sharing mode):

current environment 1	environment 6
(v) 1: ...	1: &null
(x) 2: composite (termD(v1,v1,v3),6)	2: ...
(z) 3: 123	3: "hello"
(w) 4: ...	
(y) 5: &null	

The second variable (x) is instantiated to a record termD whose elements are variables of the environment number 6. termC has just been solved and we are now about to execute the Icon goal.

Step 1: copying the parameters of the co-expression. Since the structure sharing mode of representation of the variables cannot be used in Icon, those parameters are copied for the use of the co-expression:

```
Copies [1] <== termD ( a,  a, "hello")
                a = logical (&null)
Copies [2] <== logical (&null)
Copies [3] <== 123
```

We also find a list of all free variables where the only tangible results of the co-expression will occur. At the end, we will transfer those values to the "real" variables on the stack.

Step 2: Icon environment creation. An Icon environment of the following type is created on the evaluation stack.

```
record env-icon (father, copies, free, param, coex, vector)
```

where father is the number of the enclosing environment, copies is the list of values of the variables used in the co-expression, free is a list of free variables of the copies, param is the list of the external reference of the co-expression, coex is a copy of the original co-expression (we need a copy to deal correctly with recursive procedures) and vector contains the internal variables of the co-expression.

The record

```
record bt-icon (redo, unbind)
```

where redo is an environment number on the execution stack and unbind gives the variables to unbind on backtracking (list), is kept on the backtracking stack.

Step 3: co-expression execution. The first two steps are executed whenever a first Icon goal is encountered, but this step is executed either after the first two or when backtracking to an Icon environment.

Before the co-expression is activated, its parameters are initialized to the values of the copies list with

```
every i := 1 to *param do
    param[i].ref.value := copies [i]
```

where param[i].ref is a pointer to the global variable (in our example ^x , ^y and ^z). If this co-expression fails, the environments are removed from the execution and backtrack stacks and backtracking occurs. Upon success, then the next step is executed.

Step 4: Returning results. The results are the final values of variables of free list in the Icon environment (env-icon). For example, given that

```
free [1].ptparam.value ==> [m,n, "goodnight"]
```

where m and n are internal variables of the co-expression. As global variables can cause problems in case of recursion, they are replaced by new logical variables.

So we have

```
free [1].ptparam.value ==>
            [logical ([^v1, env coex]),
             logical ([^v2, env coex]),
             "goodnight"]
```

The contents of the variables that were previously free are transferred to the execution stack. The variables that will have to be freed upon backtracking are put in the unbind list of the backtrack stack record (bt-icon).

Icon code in a Prolog goal has to execute in a "clean" environment (without pointers into Prolog database) and return its results to Prolog. The copies of the variables are necessary to keep the interpreter free of erroneous user program interferences. The overhead incurred by this

copying can be an advantage: it corresponds exactly to the one used in a nonstructure sharing implementation [7,8].

An astute programmer can introduce a dummy co-expression into a clause that always succeeds and declare as parameters all variables that should be copies. Resolution will then proceed with those variables represented in non-structure-sharing mode.

"Sameness" of the internal and external interfaces

Both the internal and external interfaces are implemented using the same procedures and data structures. The distinction established between them is purely formal and was made for expository purposes.

There are two situations in which the Prolog interpreter suspends and passes control back to Icon code: upon completion of the last goal of a request (external interface) and when encountering a built-in predicate (internal interface).

In both situations, some global references are copied. In the first case, the interpreter may be resumed explicitly. It then tries to go to the last backtrack node. In the second case, a built-in predicate may succeed or fail. If it succeeds, control returns to the interpreter and the next goal is executed. Upon failure, the interpreter resumes to the last choice point. This behavior is quite similar to the first situation, which can then be seen as a call to a built-in predicate that always fails.

EXAMPLES

The Prolog specification of the append relation is much simpler than the Icon solution. In other cases, Icon gives simpler solutions. The main weakness of Prolog is in the handling of built-in predicates that are "patches" to the logical structure. The following clauses are adapted from Ref. [2]. gensym new string concatenating a given roots and a number; for each root, it remembers what number was last used, so that next time it generates a different string by taking a bigger number. This is easily implemented by the following Icon procedure

```
procedure gensym(root)
  static roots
  initial roots := table(0)
  write(root ||| string(roots[root]+:=1))
end
```

In Prolog, the code is much more convoluted

```
gensym(Root):-
  get_num(Root,Num), integer_name(Num,[],Name),
  append(Root,Name,String), printstring(String),nl.

/*find a given root and saving it */
get_num(Root,Num):-
  retract(current_num(Root,Num1)),!, Num is Num1+1,
  asserta(current_num(Root,Num)).
get_num(Root,1):-asserta(current_num(Root,1)).

/*creating a list of the character representing the integer I*/
integer_name(I,Sofar,[C|Sofar]):- I<10,!,C is I+48.
integer_name(I,Sofar,List):-
  Tophalf is I//10, Bothalf is I mod 10,
  C is Bothalf+48, integer_name(Tophalf,[C|Sofar],List).

append([],L,L).
append([A|B],C,[A|L]):-append(B,C,L).
printstring([]).
printstring([H|T]):-put(H),printstring(T).
```

In cases when string manipulations and imperative statements are necessary, the Icon code is compact and clean.

A program for playing MasterMind illustrates the combination of Prolog and Icon and exemplifies the use of the proper tool at the proper time. Prolog is used giving multiple interpretations to one declaration and Icon is used for the sequential, algorithmic part. MasterMind is a game of logic between two players: the Codemaker and the Codebreaker. The Codemaker chooses an ordered list of four colors out of six in which one color may appear more than once. The Codebreaker must find this code by a series of probes. Each probe consists of a sequence of four colors that are evaluated by the Codemaker against the original code : one black point by correctly-placed color, one white point by correct, but misplaced color. For example, the probe [red,blue,green,yellow] against the code [blue,red,blue,yellow] scores one black point for the yellow and two white points for the red and the blue. The Codebreaker does not know which color earns a black or white point. The Codebreaker proposes probes based upon preceding scores until a perfect match is achieved; that is, four black points. The purpose of the game is to break the code in a minimum number of probes.

The Logicon version of the original Prolog version published by Van Emden [9] follows. The same Prolog procedure is used for breaking a code, checking a code and giving the score.

```
record mm(probe,code,score)
record nonmem(el,list)
record black(probe,code,unmatchedScore,unmatchedCode,blackScore)
record diff(color1,color2)
record white(probe,code,score)
record color(color)
record del(el,list2,list3)
record a_probe(probe,score)

global
                        # variables within clauses
  a,b,c,c1,c2,p,p1,pp,s,s1,s2,u,v,v1,y,y1,
  score,probe,probes,code

procedure main ()
                        # Logical variables
  a := logical();  p1 := logical();  v := logical()
  b := logical();  pp := logical();  v1 := logical()
  c := logical();  s := logical();  y := logical()
  c1 := logical();  s1 := logical();  y1 := logical()
  c2 := logical();  s2 := logical();  score := logical()
  p := logical();  u := logical();  probe := logical()
                        # Initialize the system and enter clauses
  init_logicon ()
                        # main clause giving the relation between the number of black and white
                        # points for a given probe and score
                        # It will be used for
                        #   - checking the consistency between a code and a score
                        #   - finding the score given a probe and a score
                        #   - finding codes consistent with a score and a code
  assert(mm(p,c,[s1,s2,"nil"]), [ black(p,c,p1,c1,s1), white(p1,c1,s2)])
                        # count the number of black points
  assert(black("nil","nil","nil","nil","o"))
  assert(black([u,p],[u,c],p1,c1,["b",a]), black(p,c,p1,c1,a))
  assert(black([u,p],[v,c],[u,p1],[v,c1],s), [ diff(u,v), black(p,c,p1,c1,s)])
                        # count the number of white points
  assert(white("nil",c,"o"))
  assert(white([u,p],c,["w",s]), [ del(u,c,c1), white(p,c1,s)])
  assert(white([u,p],c,s), [ nonmem(u,c), white(p,c,s)])
                        # clauses for dealing with lists and colors
                        # delete the first occurence of el in list1 giving list2
  assert(del(u,[u,y],y),cut())
  assert(del(u,[v,y],[v,y1]), [ diff(u,v), del(u,y,y1)])
                        # el is not a member of list
  assert(nonmem(a,"nil"))
  assert(nonmem(u,[v1,v]), [ diff(u,v1), nonmem(u,v)])
                        # color1 and color2 are different
  assert(diff(c1,c2), [ color(c1), color(c2), icon( create c1.value ~== c2.value, [c1,c2])])
                        # give the colors available
  assert(color("blue")); assert(color("pink"));assert(color("green"))
  assert(color("white"));assert(color("red")); assert(color("yellow"))
  master_mind ()
end  #main
```

```
procedure master_mind ()
local
  nuprobe,                  # Integer  : number of current probe
  seqgen,                   # List     : sequence of colors randomly generated
  probe_us,                 # List     : user current probe
  role,                     # String   : "code" or "decode"
  i,c
initial
                            # seed for generating code sequence
  &random := integer(&clock[1:3]) + integer(&clock[4:6]) + integer(&clock[7:9])
repeat {
  writes("Do you wish to code, decode, stop?")
  role := read()
  (role == "stop") & break
  nuprobe := 0
  probes  := []
                            # generate a sequence
  seq_gen := [colors[?6],colors[?6],colors[?6],colors[?6],"nil"]
  if role == "code" then {
                            # PROGRAM IS CODEBREAKER
    code := get_sequence("Enter your code (I have my back turned)!")
    probe.value := seq_gen      # first probe of program
    repeat {
                            # evaluation of the score
      prolog(mm(probe.value,code,score))
      dump_probe_score(nuprobe+:=1,probe.value,score.value)
      if *score.value[1] = 5 then
        break           # code broken!
      put(probes,a_probe(probe.value,score.value))# keep track
                            # generation of the next probe
      prolog([mm(probe.value,probe,score.value),
              icon(create val_probe(probe),probe)])
      }
    dump("The code was ",probe.value[1:5])
    }
  else {
                            # USER IS CODEBREAKER
    code := seq_gen             # try to find that!
    repeat {                # read probe
      probe_us := get_sequence( "Probe nu : " || string(nuprobe+:=1) || " ")
                            # evaluate probe
      prolog(mm(probe_us,code,score))
      if *score.value[1] = 5 then
        break           # code broken!
      dump_probe_score(nuprobe,probe_us,score.value)
      put(probes,a_probe(probe_us,score.value))  # keep track
      write("Your previous probes : ")
      every i:= 1 to *probes do
        dump_probe_score(i,probes[i].probe,probes[i].score)
      }
      write("Congratulations! Number of probes : ",nuprobe)
    }
  }
end # master_mind
```

```
procedure val_probe (code)
                    # Procedure used as an evaluable predicate.  Checks the absence
                    # of contradiction between a so-called code and all previous probes
local e
  every e := !probes do
                    # Checks that all previous probes would have get the same
                    # score evaluated against "code"
  prolog(mm(e.probe,code.value,e.score)) | return &fail
  return &null
end # val_probe


procedure get_sequence(prompt)
                    # reads in a sequence of colors. returns a list.
local
  seq,i,j,k,line,letters
initial
  letters := &ucase ++ &lcase
repeat {
  j := 1
  seq := list(5)
  writes(prompt)
  line := read()
  every i:= 1 to 4 do {
    seq[i] := line[j:(k:=many(letters,line,j))]
    j := (many(' ',line,k) | k)
    }
  seq[5] := "nil"
  }
return seq
end # get_sequence


procedure dump_probe_score (no,probe,score)
                    # dumps a probe followed by its score
  writes("Probe nu ",right(string(no),3," "),": ")
  dump(&null,probe[1:5])    # dump is a general purpose procedure
                    # used to display anything (in logicon)
  writes(" score : [")
  every i:=1 to *score[1]-1 do
    writes("b ")                # black score
  every i:=1 to *score[2]-1 do  # white score
    writes("w ")
  write("]")
end # dump_probe_score
```

This example shows how the cooperation of each language contributes to the simplicity of the solution. The user interface is written in Icon; it is simpler and more sophisticated than the original Prolog code. Prolog is used for the heart of the algorithm which, consists of the clause mm.

There are other Prolog clauses to count the number of black points the number of white points and various other clauses for dealing with lists and colors. Once the clauses are defined, procedure mastermind is called, which uses the mm clause as a judge for computing a score given a probe and a code. More interesting is the use of that same clause when the program is codebreaker, mm is then used in two ways: if only probe and score are specified, mm then generates all codes that give that score given this probe ; if probe , code , and score specified, mm checks that these parameters are coherent.

The overall scheme of use as follows:

```
probe.value := random try
probes := []
repeat{
   print "probe" and read the value of the "score" given by the
   opponent,
   if the "probe" was a total match  then break
   put "probe" at the end of "probes"
   find the next "probe" consistent with the previous scores.
   }
```

The call to prolog instantiates "probe" to the value that will be used on the next move; mm is a generator of probes to be used by the Icon co-expression val_probe , which checks that it is consistent with all previous probes and score. The generation of the next probe is done by a co-expression that calls a Prolog goal while being itself called by a Prolog goal. Ref. 6 gives the full listing of this example and also a Prolog version of that program where both of them can be compared to appreciate the "simplicity" of the right tool at the right time.

PERFORMANCE

The goal of this project was to show how Prolog could be embedded in Icon and the strength of this combination. Unfortunately, interpreting Prolog with an interpreter in Icon on a Vax 750 gave poor timings. For example, it takes around 200 CPU seconds per move to play MasterMind. This can be explained by the following factors: in Icon, memory management is automatic so we cannot implement our interpreter as efficiently as we would like: arrays and stacks have to be represented as lists; thus access to their elements is not direct. We are currently studying the possibility of modifying the original Icon interpreter so that we can avoid many variable copies that are made only to shield the Prolog environment from the user program. We could also achieve a much better memory management using stacks and implementing tail recursion optimisation.

We are also designing a new version of Logicon synthesizing the syntax of Icon and Prolog that allows the automatic declarations of records and logical variables; a more natural syntax for clauses and evaluable predicates is defined and it is possible to enforce the restriction that a logical variable can not change its type at run-time. Presently, this causes the interpreter to abort. These are mainly cosmetic changes without any great consequences on the basic algorithms we introduced, but which should speed up the interpreter.

CONCLUSION

Mixing Prolog with other languages is not new. Many Prolog implementations permit user-defined built-in predicates in other languages (C or Pascal for example [10] [11]). But in those cases, we cannot call another Prolog goal from this built-in predicate. More complete integrations have been done in LISP [12] which is structurally closer to Prolog than Icon is. To our knowledge, the work described in this paper is the first attempt at an amalgamation of Prolog and a procedural language. The integration is complete: i.e., it can go both ways: Prolog to Icon and vice-versa without any preprocessing. The Prolog can deal with all Icon types and Icon procedures can access the values of the Prolog logical variables. What made this combination possible is that Icon is by itself a very powerful language having constructs to control generation of sequence of results and even ways of defining new control structures. This might lead to an elegant way of achieving a more selective control of the power of Prolog.

Icon has also gained a very powerful and elegant kind of pattern matching: a declarative approach often helps specify certain problems more concisely and more reliably. Logicon gives that tool while retaining the semantics of Icon and complementing the tools already present in the language. The resolution process, which is absent from Icon, can now be used to control the algorithms; from the outside, its behavior is the same as a built-in generator except for the side-effects in its parameters.

Our implementation is still a "toy", but it is very flexible and has shown the usefulness of the approach. We intend improve both on its efficiency and on its ease of use.

ACKNOWLEDGEMENT

This work was partially supported by a grant from the National Sciences and Engineering Research Council of Canada. We also thank anonymous referees who thoroughly commented a first version of that text and led us to great improvements.

REFERENCES

[1] R.E. Griswold, M.T. Griswold, The Icon Programming Language, Prentice-Hall, 1983.

[2] W.F. Clocksin, C.S. Mellish, Programming in Prolog. Berlin: Springer-Verlag, 1981.

[3] A. Colmerauer, H. Kanoui, M. Van Caneghem, Prolog, bases théoriques et développements actuels. Technique et Sciences Informatiques, 2, (no 4), 271-311, (1983).

[4] J.A. Robinson, A Machine-Oriented Logic Based on the Resolution Principle, JACM, Vol. 12, 23-44, (1965).

[5] L.M. Pereira, F. Pereira, D. Warren, User's Guide to DEC System-10 Prolog, DAI Occasional Paper no 15, University of Edinburgh, Scotland, 1979.

[6] S. Chapleau, "Intégration du langage Prolog au langage Icon", Document de travail 154, Département d'informatique et de recherche opérationnelle, Université de Montréal, nov 1984.

[7] C.S. Mellish, An Alternative to Structure Sharing. Logic Programming: recueil d'articles édité par Clark, K.L., Tarnlund, S.A., Londres: Academic Press, 99-106, (1982).

[8] M. Bruynooghe, The Memory Management of Prolog Implementation in Logic Programming, Clark, K.L., Tarnlund, S.-A. ed., Academic Press, 83-98, 1982.

[9] M.H. Van Emden, Relational Programming Illustrated by a Program for the Game of Master-Mind. Computational Linguistics and Computer Languages, 13, 131-150, (1979).

[10] F. Pereira, C-Prolog User's Manual, Version 1.4, SRI International, 1984.

[11] C.R.I.L. Prolog/P, a Pascal Implementation, Conception et réalisations industrielles de logiciel, Puteaux, 1984.

[12] J.A. Robinson, E.E. Sibert, LogLisp: Motivation, Design and Implementation in Logic Programming, J. Clark & Tarnlund ed., Academic Press, 1982.