THE APPLOG LANGUAGE*
PROLOG vs. LISP - if you can't fight them join them

By

Shimon Cohen.

EECS Computer Science Division

UC Berkeley

ABSTRACT

We discuss the virtues of PROLOG in comparison to LISP, we come to the conclusion that: "If you can't fight them JOIN them". We propose, as a solution, the APPLOG language which is a mixture of LISP and PROLOG. APPLOG is embedded within the PROLOG language and thus the facilities of PROLOG can be used through a simple *goal* function.
APPLOG is an applicative language where functions are applied to arguments. APPLOG variables are compatible with PROLOG variables and serve as a mean for data transfer between APPLOG and PROLOG. APPLOG supports *lambda* and *nlambda* functions definitions and *one-to-one*, *one-to-many* and *mixed* binding mechanism like INTERLISP.

The main advantage of APPLOG is the simple integration of LISP and PROLOG into ONE powerful language which hopefully incorporates the best features of both languages. We also extended APPLOG to a simple relational data-base query language which is similar to Query-by-example and includes: *Aggregates* and *Grouping*.

We provide the full listing of the APPLOG interpreter including: pretty-print, load, trace, toploop, history, autoload, Interface to relational database.

APPLOG has the following advantages over traditional LISP languages:

(1) Pattern directed invocation.
(2) Call by reference.
(3) Interface to PROLOG as a data-base query language.
(4) Operators (infix, prefix and postfix).
(5) Back-tracking.
(6) Generators.

* We acknowledge the help of the EECS, Computer Science Division, UC Berkeley in the production of this paper.

1. Introduction

The idea of 'logic programming' was proposed <Kowalski 74> as the basis for a new programming language. Kowalski concieved a clear and simple deductive language which is based on the LUSH resolution of Horn clauses <Hill 74>. These ideas were realized in the programming language PROLOG by <Colmerauer 73>, later it was shown <Warren 77> that PROLOG programs can run as fast as the equivalence LISP programs. Finally a compiler written in PROLOG <Warren 80> is able to compile Horn clauses into efficient machine Code.

Although PROLOG appears to be useful in a large variety of applications, there are still some doubts concerning PROLOG's efficiency (Back-tracking), "pure" mathematical background (*cut,not*)) etc ,especially when PROLOG is compared to LISP. Other researchers <Sato 83> and <Robinson 82> came to similar conclusions and proposed two different languages for programming in logic: QUTE and LOGLISP. (Both are combination of LISP and PROLOG). In our view, these two languages are too complicated, they introduce: New syntax, special cases, many many new functions WHICH are really not needed.

The goal of this paper is to show that a combination of LISP and PROLOG is **desirable** *and can be* **simply** *achieved.*

In the first chapter we express our view concerning the issue: "PROLOG vs. LISP" and we come to the conclusion: "If you can't fight them join them ...".

In the next chapter we describe the APPLOG (APPlicative LOGic) which is a mixture of LISP and PROLOG. In APPLOG both LISP and PROLOG environments (Variables, Data structures ...) are "unified" and provide an easy way to express one's ideas in "applicative" style or "logic" style. Our main goal is to enable the LISP and the PROLOG users to write programs in their "native" language, while at the same time allow them an easy access to the power and virtues of the other language. The result is a more powerful and efficient language which provides various features like: function application, logic inference, relational data-base query language, generators etc.

We provide (in the appendix) the full listing of our system implemented in PROLOG including facilities like: trace, autoload, history, toploop etc. This definition (written in PROLOG) constitutes the operational semantics of APPLOG.

But before we continue we would like to present an example which explains the idea of *generators* in APPLOG: Suppose we want to put eight queens on a chess board without having two of them on the same line, column or diagonal.

```
def( q8,
 lambda( [Line, Board],
  if( Line = 0, Board,
     try( q8( Line - 1,  Board ), or(1,2,3,4,5,6,7,8))))).

def( try, lambda( [Board, Col],
    if( noaim( Board, Col, 1 ), cons(Col, Board), fail))).

def( noaim,
 lambda( [Board, Col, Add],
  if( eq( Board, [] ), true,
     and(
      Col == car(Board),
      Col+Add == car(Board),
      Col-Add == car(Board),
      noaim(cdr(Board), Col, Add+1)
     )))).
```

To call *q8* try: q8(8,[])

meaning: "try to put 8 queens on an empty board". Function *try* is called with a partially filled board and with a **generator** *or* which generates the numbers 1 to 8. *try* tries to put the next queen in the suggested column *Col*. If it can't do that, it fails (see *fail* latter), then *or* generates a new possible *Col* number and *try* is re-tried.

1.1. LISP vs. PROLOG

Both languages are quite similar, they were designed to deal with: Understanding Natural languages, Mathematical Logic, Algebraic manipulation and other AI applications. Arguments made by PROLOG and LISP fans usually involve some of the following issues:

* Mathematical back-ground.
* Efficiency.
* Programming Style.
* Readability and Clearity.

* Side effects.

* Special Features: Unification, Back-tracking, Functional.

Mathematical back-ground: PROLOG is based on the simple and well-investigated ground of LOGIC theory. However only a subset of first-order logic known as Horn Clauses are expressible in pure PROLOG. pure PROLOG does not include: Control, negation and functionals. Soon enough the PROLOG implementors added few "dirty" control structures <Warren 82> which enable to express functions which are otherwise not computable in pure PROLOG (*cut*, *not*, *setof*). Pure LISP on the other hand is based on Lambda-calculus which is also a well founded logic theory. Pure LISP supports Functionals and negation and virtually everything can be written in pure LISP.

Efficiency: In <Warren 80> they claim that PROLOG programs can be compiled and run as fast as LISP programs. That indeed is not a surprise if we take their advice and use *cut* in the programs (i.e.: make it deterministic). In addition they recommend to define for each functor the input and output variables. Although these additions are not pure LOGIC they help indeed to eliminate back-tracking and thus enable PROLOG programs to run as fast as LISP.

Lets take another example: Suppose we want to write a "program" that answers the following question: "Who is the father-of the wife-of the brother-of X ?" In PROLOG we will naturally write:

```
query1(X,Y) :- father-of(Y,Z), wife-of(Z,W), brother-of(W,X).
```

This can result with a long search of the data-base starting with *father-of* relation. In the worst case PROLOG will search all the facts in the data-base about *father-of* relation and for each Z it will search relation *wife-of* and then for each W it will try to determine whether W is the *brother-of* X. It would be much more efficient to write it in LISP:

```
(lambda (X) (father-of (wife-of (brother-of X))))
```

Or even in PROLOG:

```
query2(X,Y) :- brother-of(W,X), wife-of(Z,W), father-of(Y,Z).
```

(**conclusion**: sometime one has to "reverse" his "logic" reasoning)

Programming Style and Readability: In the above example we see the versatility of PROLOG, the predicate *query* (in either version) can be used:

1. As a predicate (if both X and Y are instantiated.)
2. As a two way function (if either one is instantiated).
3. As a generator (if both are not instantiated)

LISP fans must admit that they can't find a matching feature in LISP.

Other points: PROLOG supports "flat programming" while LISP supports nested function calls. In PROLOG when one wants to pass a value computed by one clause to the following clauses he needs dummy variables:

f3(X,Y), f2(Y,Z), f1(Z,W).

Y is passed to *f2* which passes Z to *f3*. In LISP nested functions are used (like: (f1 (f2 (f3)))) and there is no need for dummy variables. However, when one needs to pass the same value to more than one function, one needs to use lambda expression:

((lambda (x) (f1 x (f2 x))) (f3 ...))

The value of (f3 ...) is bound to *x* and then *x* is passed both to *f1,f2*. In PROLOG on the other hand, one simply uses the same variable name in both places like in: f1(I1,O1), f2(O1,O2), f3(O1,O3).

(O1 is passed from *f1* to *f2,f3*)

One of the main advantage of PROLOG over LISP is that it supports backtracking, while in LISP one has to hardwire it into his programs. However the implicit use of backtracking in PROLOG leads to an extensive use of *cut*. The reasons are: (1) Usually programs are deterministic, (2) To imitate the if-then-else construct.

Thus some might prefer the explicit use of backtracking and some might prefer its implicit use.

Side effects: We argue that when one wants to write real efficient programs he usually needs some "dirty tricks" like "side effect". Indeed the Logicians can say that: logic is above all, but is it really acceptable to

use an unbalanced tree as a symbol table as proposed by Warren <Warren 80> for his compiler. Or when one implements a window package to store a window as a list of characters ...

Let's take a look at the following problem: Suppose we have a directed graph which is represented in PROLOG:

```
e(a,b). ; edge from a to b.
e(b,c). ; ...
...
```

We want to define the relation path(X,Y,P) which is true if P is the path from X to Y.

```
path1(X,X,[X]). ; trivial case
path1(X,Y,[X|Pz]) :- e(X,Z), path1(Z,Y,Pz).
```

This "perfect" definition (logic-wise) might fail in reality because of cycles, for example if the graph consists of: e(a,b). and e(b,a). To solve the problem we have to think a little bit more logically and introduce a new relation *path1*.

```
path2(X,Y,P) :- path2l(X,Y,P,[]).
path2l(X,X,[X],L) . ; trivial case
path2l(X,Y,[X|Pz],L) :- e(X,Z), not( member(Z,L)), path2l(Z,Y,Pz,[X|L]).
```

The fourth argument is the list of nodes that we already visited and we don't want to visit again. The third parameter is also the list of nodes that we find along the path, HOWEVER the third list is built when we return from the recursion and the fourth is built on the way in. Why can't we use only one list ? why do we have to build the same list twice ? Well it has to do with PROLOG special treatment of variables. There are ways to overcome this inefficiency:

```
path3(X,Y,L) :- path3l(X,Y,L,[]).
path3l(X,X,[X|L],L).
path3l(X,Y,P,L) :- e(X,Z), not(member(Z,L)), path3l(Z,Y,P,[X|L]).
```

Only when we reach the end of the search (second line) we assign the value of the fourth parameter to the third. Note that you get the reverse list (i.e the list of nodes from Y to X).

This is not the end of the story: If you try to run this particular program on a very big graph you will wait a long time before you will get the result. The reason is *member* which consumes most of the time trying to figure out if we already pass through a particular node. Another problem: If we get to a node and

there is no way from it to the target then we have to backtrack from it. BUT there is no way to remember that this node is a "bad" one, so if we reach this node using a different edge we might as well try again the whole search from this node.

To overcome these problems we need to use side effect (*assert*) and store in the database those nodes that we have visited.

```
path4(X,X,[X]).
path4(X,Y,[X|Pz]) :- e(X,Z), try_edge(Z), path4(Z,Y,Pz).
try_edge(Z) :- was_visited(Z),!,fail.
try_edge(Z) :- assert(was_visited(Z)).
```

The first time we visit node Z we use *assert* to store this information in the data-base (4th clause), next time we visit the same node Z we *fail* (3rd line)

To solve the same problem in LISP we use property lists as the data-base and we hard-wire backtracking into our functions.

```
(def path (lambda (x y)
  (if (eq x y)
    (list x)
    (try_edges x y (get x 'edges))))

(def try_edges (lambda (x y edges)
  (cond
    ((null edges) nil)
    ((get (first edges) 'was_visited)
     (try_edges x y (rest edges)))
    (t (putprop (first edges) 'was_visited)
    (try1 x y (path (first edges) y))))))

(def try1 (lambda (x y nodes)
  (if (null nodes)
    (try_edges x y (rest edges))
    (cons x nodes))))
```

This definition is obviously less attractive than the previous ones (in PROLOG). It is much more complicated and hard to understand, but this is not a surprise because LISP has no builtin mechanism for backtracking. We ran these programs on a 100 nodes graph with 1000 edges. The programs were interpreted (not compiled) and the results:

LISP vs. path4 1:4

path4 vs. path3 1:4

It means that the pure logic program *path3* is slower almost 16 times than the LISP program. (no surprise: the LISP program is not pure)

conclusion: Logic program is not necessarily an efficient program.

Special Features:

Unification

No doubt "unification" is a more powerful tool than the simple binding mechanism of LISP. PROLOG fans claim that one clause can replace different types of functions: constructors (like *cons*), modifiers (like *rplaca*) and predicate (like *listp*). and I believe they are right, for example:

```
conscell( [X|Y], X, Y).
```

This definition will do:

```
conscell( C, 1, 2). -> C = [1|2] ; constructor
conscell( C, 1, _). X = 1 ; rplaca (well not if the CAR has a value)
conscell( C, Car, Cdr). ; Selectors
```

PROLOG fans also claim that "unification" helps to avoid repeated use of *car,cdr* combinations to break the structure of list (or other constructs). In the following example we show that this is not always true.

The problem is to add two polynomials, so we define the predicate **add(P1,P2,P3)** which is true if P3 is the sum of P1 and P2.

```
add( [], P2, P2).
add( P1, [], P1).
add( [ c(C1, E1) | P1 ], [ c(C2, E2) | P2 ] , [ c(C3, E2) | P3 ] ) :-
  E1 == E2,!,
  C3 is C1 + C2,
  add(P1,P2,P3).
add( [ c(C1, E1) | P1 ], [ c(C2, E2) | P2 ] , [ c(C2, E2) | P3 ] ) :-
  E2 < E1,!,
  add( [ c(C1,E1) | P1] , P2, P3).
add( [ c(C1, E1) | P1 ], P2 , [ c(C1, E1) | P3 ] ) :- add(P1,P2,P3).
```

Note that again and again (clauses 3,4,5) we unify the heads of the polinoms with the patterns. We need to do it because **if-then-else** is not included in the PROLOG language.

In LISP we would write:

```
(defun polyadd( p q) ;; p = ((exp.coef)(exp.coef) ....)  same for q
 (cond
   ((null p) q) ;; simple case
   ((null q) q)
   (t (prog (e1 e2 c1 c2) ; temp vars
       (setq c1 (caar p) e1 (cdar p) c2 (caar q) e2 (cdar q))
       (cond
         ((eq e1 e2) (cons (cons (plus c1 c2) e1) (polyadd (cdr p) (cdr q))))
         ((greaterp e1 e2) (cons (car q) (polyadd p (cdr q))))
         (t (polyadd q p)  ; switch p q
        ))|
```

the LISP function looks more efficient since it breaks the structures only once with *setq*. PROLOG fans will say at this point that this problem is solved by the compiler which generates a good code for unification and avoid the multiple unification, BUT so can a LISP compiler.

Backtracking:

In <Warren 82> he says: "A difficulty which programmers new to PROLOG soon come up against is that they need to combine information generated on alternative branches of the program. However, in pure PROLOG, all information about a certain branch of the computation is lost on backtracking to an earlier point."

Warren suggest to use *setof*:

setof(X,P,L) is true if L is the set of instances of X that satisfy P.

This is a general mechanism, however it does not allow to record information from a subset of branches. In LISP one can record information using the dynamic environment and the function *setq* to set free variables (global), while in PROLOG it is quite impossible because variables are defined and used only inside one clause.

1.2. CONCLUSIONS

We summarize by reviewing the main issues:

Mathematical background - I can only quote Drew McDermott <McDermott 80> :"*Most published description (of PROLOG) are wretchedly misleading. The notion that programming in PROLOG is pro-*

gramming in logic is ridiculous. Some **simple** *clauses can be thought of as first-order implications, but not most."*

Efficiency - Pure recursive programs in LISP can be rewritten in PROLOG and run as efficient as the corresponding LISP programs, In addition special type of programs (as I mention before: natural language processing and relational data-base etc) can be run effectively more efficiently in PROLOG. However, in many other cases, programs can become almost impossible to be run in PROLOG.

Style - Again Mcdermott: *"It is often claimed that PROLOG programs can be used in more than one way, and simple one can. — Everyone* **quickly** *learns how seldom a program works this way. Use (of this style) will* **often** *introduce gross inefficiency or infinite loops. In* **practice** *this means programmers have to devote as much time to think about the different task a relation might do, as they would in writing a set of functions for these tasks in any other language"*

Readability - There is no clear evidence that PROLOG programs are more clear or easy to understand than LISP programs or vice versa.

Side Effects - There are ways to write side effect programs in PROLOG (*assert, retract*) and our experience (and other) shows that they are often used for reasons of efficiency, simplicity and even clarity.

Special features - No doubt PROLOG has two important features (Unification, Backtracking) which are missing in LISP. It seems that they are powerful and well incorporated into the language.

It seems that both LISP and PROLOG are valuable tools for a variety of applications. In some applications (like natural languages processing or relational data-base management) PROLOG is better than LISP, with its unification (like pattern matching) and backtracking (for non deterministic) features. LISP, on the other hand, appears to be more general and applicable to a large variety of applications. In my view LISP can be thought of as the C of AI where one has more control over his computation.

The solution is maybe to create one system which will support both LISP and PROLOG. In LISPLOG <Robinson 82> the solution is to augment LISP with a package of functions (called LOGIC), the result is one system with two different languages running under the same roof.

The QUTE language <Sato 83> is another attempt to "amalgamate" LISP and PROLOG by introducing

a new syntax and a new notion of variable. We prefer our approach because it is simpler and straightforward: PROLOG remains exactly the same language and it is extended to enable "applicative" type expressions. The LISP fans will find this "applicative" style very much the same as ordinary LISP expressions.

2. Language Definition

2.1. APPLOG Syntax

ATOM: starts with a small letter and followed by either: letters, digits or underscores.

Examples: qwqw, aa_1, mnm_asksa

If you want other chars in the atom use single quotations like: '$$$12A'. Unlike traditional LISP, atoms in APPLOG are not variable names, therefor the value of an atom is the atom itself and it needs not be quoted. Several atoms have special meaning when they are evaluated like: *fail, terpri, exit* (they will be explained latter).

Example: eval(aaaa) –> aaaa

NUMBERS: Integers or reals.

LIST: Square brackets (to be compatible with PROLOG) and elements are separated with commas.

Example: [1,2,3,4,5] or [1,2,[4,5,6,6],7,8]

VARIABLES: start with big letter followed by letters or digits or underscores.

APPLOG variables behave like PROLOG variables, they can be uninstantiated (i.e without a value) or instantiated (with a value). To check this status we have the predicate *var(X)* which succeeds if X is uninstantiated (and returns X as its value).

TERMS: We introduce into APPLOG the notion of *term* which is similar to *record* in PASCAL.

Term syntax:

<term> ::= <atom> | <number> | <list> | <variable> | <structured term>

<structured term> ::= <atom><left parenthesis><elements><right-parenthesis>

<elements> ::= <term> <more-elements>

<more-elements> ::= NULL | <comma> <elements>

Examples:

a(1,2), bbb(aaa,ccc,ddd), fff(aaa(1,2), bbb(3,4))

Lists <-> terms

To transform lists to terms and vice versa use:

list_to_term(List).

term_to_list(Term).

Examples:

list_to_term(list(a,2,3)) -> a(2,3).

term_to_list(quote(a(2,3))) -> [a,2,3].

NOTE: *list_to_term* will succeed only if the first element of *List* is an *atom*.

FORMS:

APPLOG *forms* are written a-la PROLOG style as *terms*.

Examples:

cons(1,2). -> [1|2]

list(1,2,3,cons(3,4)). -> [1, 2, 3, [3 | 4]]

List Notation is also possible although it is not too convenient:

[list, 1, 2, 3] -> [1, 2, 3]

[cons, 2, 3] -> [2 | 3]

OPERATORS *Infix, prefix* and even *postfix* operators are recognized in APPLOG forms:

Examples:
3*4. -> 12
3*2+(2+3). -> 11
cons(2+3,3*5). -> [5|15]

Operators are APPLOG primitive functions which are stored as terms inside APPLOG memory, thus the expression: 2*3 is actually the term: *(2,3). User's defined functions can be declared as operators too.

2.1.1. PREDICATES (SUCCESS and FAILURE)

Success and failure

Unlike traditional LISP we adopt PROLOG idea of success and failure. Any function is also a predicate which can succeed or fail, success means *true*, and failure means *false*. When a function succeeds, it always returns a value, predicate functions, like *eq*, return the atom *true* when they succeed, BUT when they fail they DON'T return *nil* but cause the failure of the *form* that contains them. This failure chain is stopped by the conditional functions: *if*, *cond*, or used in the predicates: *and*, *or* etc.
This mechanism of failure can be used to achieve the effects of back-tracking and generators (See the ICON language).

Example:

if(b(X), then_part(Y), else_part(Z)).

If b(X) succeed then *then_part(Y)* is called. If it fails *else_part(Z)* is called.

f(or(1,2,3,4,5)).

Function *f* is called with 1, if it fails then we try with 2 etc. until all the possibilities are exhausted, in which case *f* returns with failure.

* To cause failure try: *fail*

and, *cond* and *or* are recognized with arbitrary number of arguments:

Example:

```
cond(
   (eq(X,Y), Z)
   (X>Y,     X)
   (t,       Y)
  ).
```

2.2. USER's DEFINED FUNCTIONS

Function's parameters are usually specified as a list of *variables*. However, in addition one can write any legal *term*. APPLOG functions seem to be more readable with the inclusion of operators and additional schemas for parameter passing: By reference, Pattern directed invocation. *Nlambda* functions (as in INTER-LISP) are also available.

Examples:

Definition:
```
def(fact, lambda( [X], if (X=1, 1, X*fact(X-1)))).
```

Note: the use of infix operators
Example: fact(4) -> 12

Definition:
```
def( app, lambda( [L1,L2],
   if( eq(L1,[]), L2, cons( car(L1), app( cdr(L1), L2))))).
```

Example: app(list(1,2,3)), list(4,5,6)) -> [1,2,3,4,5,6].

Definition:
```
def( faaa, lambda( [ aaa(A,B), C], list( A, B, C))).
```

Note: *aaa* is a record passed to *faaa*.
Example: faaa(quote(aaa(4,5)), 8). -> [4,5,8].

Definition:
```
def( f4, lambda( [A,B], setq(B,A+A*A))).
```

Note: B is passed "by reference"
Example: list(f4(2,X), X+1). -> [6,7].

Definition:
```
def( fibo, lambda( [N], if( N<2, 1, fibo(N-1) + fibo(N-2)))).
```

```
(def fibo (lambda (n) (if (lt n 2) 1 (+ (fibo (- n 1)) (fibo (- n 2))))))
```
Note: compare APPLOG fibo def. to LISP

Definition:

```
def( last, lambda( [ [X|L] ], if( L=[], X, last(L)))).
```
Note: pattern-directed invocation:
first parameter of *last* must be *car-cdr*, where X is bound
to the *car* and L to the *cdr*.
same function in PROLOG will be:
```
last([X],X).
last([X|L],Y) :- last(L,Y).
```

Definition:
```
def( mylist, lambda(L,L)).
```

Note: one to many binding.
Example: mylist(1,2,3,list(7,8),4,5,6). -> [1,2,3,[7,8],4,5,6].

Definition:
```
def( f5, nlambda( [A,B] , list(A,B,eval(A),eval(B)))).
```

Example: f5(2*3, 5+6) -> [2*3, 5+6, 6, 11]

Definition:
```
def( f6, lambda( [A,B|C], list(A,B,C))).
```

Note: C is a "catch all" variable as in INTERLISP (lambda (A B . C) ...).
Example: f6(1,2,3,4,5) -> [1,2,[3,4,5]]

2.3. APPLOG <-> PROLOG

APPLOG --> PROLOG

PROLOG is called from APPLOG using the *goal* function. There are two ways to use *goal*:

1. goal(Goal,Form).

A PROLOG *Goal* and an APPLOG *Form*: The *Goal* is satisfied by PROLOG and then the *Form* is executed by APPLOG.

Recall that if the *Form* fails it backtracks into the *Goal* and then if the *Goal* can be satisfied again then *Form* is evaluated again.

Simple Example:

If the database contains a(3) then:

goal(a(X),X+X). -> 6.

Backtracking Example:

If the data-base contains: b(3,4), b(6,7) then:

goal(b(X,Y), if(X>5, X+Y)).

Then b(X,Y) will find b(3,4) but when *if* fails it will back-tracks and will come again with b(6,7) and then the result will be 13.

2. goal(Goal).

Satisfy the *Goal* and return it as the value.

Example: goal(a(X)). –> a(3)

To enter new facts to the database use *assert*:

Example: to insert the fact a(1,2,3)

Use: assert(quote(a(1,2,3))).

or: assertq(a(1,2,3)).

PROLOG --> APPLOG

To call the APPLOG interpreter from PROLOG use:

eval(Form,Result).

2.4. System Utilities

Top Loop

To call the friendly APPLOG top-loop function type in PROLOG: top.

History expressions can be re-executed by: h(N).

History listing: h(From,To).

loading APPLOG programs

Use: load(File).

The file <File> may contain any APPLOG forms. *load* reads and executes these forms.

Pretty print

Use: pp(Term).

or: pp(Term, Linelength).

2.5. Relational Database Interface

We use PROLOG database and APPLOG evaluation methods to propose a simple extension of APPLOG as a relational database query language. We use "lazy evaluation" to access "relations" and thus save in space. First we describe the simple functions: *rel* and *all*.

rel <relation>

rel is a prefix operator (unary function) which behaves as a generator. It generates the "tuples" of relation *relation*. One can avoid the use of *rel* by declaring it as a relation:

def(<relation name>, relation).

and then only the relation name has to be mentioned.

all(Func,Generator)

or as an infix operator: Func all Generator

all behaves almost like the LISP function *maplist*, it applies the function *Func* to ALL possible values which can be generated by *Generator*.

Examples:

1. print(rel a(X,Y)) . /* to print a single "tuple" of "relation" a */

   or: print(a(X,Y)). if *a* is defined as a relation.

2. print all a(X,Y). /* print all "tuples" of "relation" a */

3. print all a(X,Y) ? (X>Y) /* selection */

4. print all d(X,Y,Z,W) -> e(X,Z). /* project */

5. print all a(X,Y) x b(Y,Z) -> c(X,Y,Z) .

/* Join two relations *a,b* to form a third relation *c*. */

6. sum(personal(Name,Sex,Salary), Salary).

This aggregate function returns the sum of salaries from relation *personal*.

7. sum(personal(Name,Sex,Salary), Salary, Sex).

/* With grouping by *sex*, the functions *sum* is a generator which returns tuples: sum(male,1000000). and then sum(female,1000000). */

3. PATH

We want to show the way APPLOG solve the problem of finding a path between two nodes. Note the mixture of PROLOG and LISP features in the definition of *path*.

```
def( path, lambda( [X,Y],
  if( eq(X,Y), list(X),
     cons(X, try_edges( e(X,Z), Y))))).

def( try_edges, lambda( [ e(X,Z), Y],
  if( was_visited(Z), fail,
    (assertq( was_visited(Z)),
     path(Z,Y))))).

def(e,relation). def(was_visited,relation).
```

Note that relation *e* is a generator which generates all the edges going out from X. In *try_edges* we check if Z was visited and if so we fail. This failure causes backtracking to return to relation *e* which retrieves another edge and then *try_edges* is called again.

4. SUMMARY

It seems that both PROLOG and LISP are here to stay, each with its own group of fans. We showed that with almost no pain at all, one can enjoy the virtues of both languages. As a proof we supply in the appendix the FULL listing of the APPLOG interpreter written in PROLOG.

5. References

<Bergman 75>
Bergman M. & Kanoui H.
SYCOPHANT: Systeme de calcul formel et dintorogation symbolique sur lordinature
Groupe dIntelligence Artificial, U.E.R. De luminy, U. d'Aix-Marsille II

<Bundy 79>
Bundy A., Byrd L., Luger G. Mellish C and Palmer M
"Solving Mechanics problems using meta level inference"
Expert systems in micro electronics age, pp:50-64
(ed Michie D.) Edingurgh: Edinburgh University Press.

<Colmerauer 73>
Colmerauer A. Kanousi H., Pasero R. and Roussel P.
"Un Systeme De Communication Homme-machine en Fracais"
Report Luminy, France: Group Intelligence Artificielle,
Universite d'Aix-Marseille

<Darvas 77>
Darvas F. Futo I and Szeredi P.
"Logic based program system for predicting drug interaction"
Intl. Journal Biomed. Comp.

<Clark 82>
Clark K. L. and McCabe F. G.
"PROLOG: a language for implementing expert systems"
Machine Intelligence 10, 1982.

<Griswold 83>
Griswold R. E.and Griswold M. T.
"The ICON Programming Language"
BOOK, Prentice Hall, 1983.

<Hill 74>
Hill R.
"LUSH resolution and its completeness"
DCL Memo NO. 78, Dept. of Artificial Intelligence, Univ. of Edinburg, 1974.

<Kowalski 74>
Kowalski R. A.
"Predicate Logic as Programming Language"
Proceeding IFIP Congress 1974.

<McDermott 80>
McDermott D.
"The PROLOG phenomenon"
SIGART Newsletter 72, pp:16-20, 1980.

<Robinson 82>
Robinson J. A. and Sibert E. E.
"LOGLISP: An alternative to PROLOG"

Machine Intelligence 10, 1982.

<Sato 83>
Sato M. and Sakurai T.
"QUTE: A PRLOGG/ LISP type language for LOGIC programming"
Proc. of 8th intl. Joint Conf. on AI, Volume 1,
August 83, Karlsruhe, West Germany.

<Warren 77>
Warren D. H. D., Periera L. M. and Periera F.
"PROLOG- The language and its implementation compared with LISP"
Proc. of Symp. on AI and Programming Languages, 1977,
SIGPLAN Notices, 12, No. 8, and SIGART newsletters 64, 109-115.

<Warren 80>
Warren D. H. D.
"Logic Programming and Compiler Writing"
Software Practice and Experience 10, pp: 97-125, 1980.

<Warren 82>
Warren D. H. D.
"High order extension to PROLOG: are they needed ?"
Machine Intelligence 10, 1982.

6. APPENDIX

Here is the listing of the APPLOG interpreter written in PROLOG (Cprolog 1.2). Hopefully, any real PROLOG hacker will be able to adapt this version to his own system.
There are three files:
eval - The main interpreter.
rdb - The relational database interface.
pp - Pretty printer of APPLOG expressions.
Make sure the PATH names are correct in file eval (see instructions there).

```
/*
  A P P L O G   Interpreter        (file: eval)
  ---------------------

  "LISP vs. PROLOG: If you can't fight them, JOIN them"

  Written by: Shimon Cohen (April 1983).

  NOTE: Before loading this file make sure all path names are correct.
  To do so, search for lines with the string: "cohen"
  and insert the right PATH. Make sure you have files: rdb,pp
  in your directory.
*/
/* operators */
initop :-
      op(999,fx,is),
        op(998,fx,q).

:- initop.

/*   simple cases and special atoms   */

eval( X,Y)            :- var(X),!,X=Y.
eval( exit, R)        :- !,abort.
eval( fail, R)        :- !,fail. /* cause failure */
eval( terpri, true)   :- !,nl. /* cause newline */
eval( X, Y)           :- (number(X);var(X);atom(X)),!,Y=X.
eval( value(X),Y )    :- !,Y=X.
eval( (A,B), R)       :- !,eval(A,Ax),eval(B,R).
eval( [A|B], R)       :- !,T =.. [A|B], eval(T,R).

/* arithmetic functions */

eval( A*B ,C)         :- !,eval(A,Ae),eval(B,Be),C is Ae*Be.
eval( A-B ,C)         :- !,eval(A,Ae),eval(B,Be),C is Ae-Be.
eval( A+B ,C)         :- !,eval(A,Ae),eval(B,Be),C is Ae+Be.
eval( A/B ,C)         :- !,eval(A,Ae),eval(B,Be),C is Ae/Be.
eval( A//B ,C)        :- !,eval(A,Ae),eval(B,Be),C is Ae//Be.
eval( A mod B ,C)     :- !,eval(A,Ae),eval(B,Be),C is Ae mod Be.
eval( A ^ B ,C)       :- !,eval(A,Ae),eval(B,Be),C is Ae ^ Be.

eval( A>B ,true)      :- !,eval(A,Ae),eval(B,Be), Ae>Be.
eval( A<B ,true)      :- !,eval(A,Ae),eval(B,Be), Ae<Be.
eval( A=<B ,true)     :- !,eval(A,Ae),eval(B,Be), Ae=<Be.
```

```
eval( A>=B ,true)      :- !,eval(A,Ae),eval(B,Be), Ae>=Be.
eval( A==B ,true)      :- !,eval(A,Ae),eval(B,Be), Ae==Be.
eval( A===B ,true)     :- !,eval(A,Ae),eval(B,Be), Ae===Be.
eval( A=B ,true)       :- !,eval(A,Ae),eval(B,Be), Ae=Be.

/* basic LISPL functions */

eval( cons(X,Y) ,[Xe|Ye]) :- !,eval(X,Xe),eval(Y,Ye).
eval( quote(X) ,Y)     :- !,X=Y.
eval( q X ,Y)          :- !,X=Y.
eval( setq(X,Y) ,Z)    :- !,eval(Y,Z),X=Z.
eval( X is Y ,Z)       :- !,eval(Y,Z),X=Z.
eval( eq(X,Y) ,true)   :- !,eval(X,Xe),eval(Y,Ye),eqq(Xe,Ye).
eqq(A,B)               :- (var(A);var(B)),!,A=B.
eqq(A,B)               :- A===B.
eval( atom(X) ,X)      :- !,(var(X);atom(X)).
eval( car(X) ,Y)       :- !,eval(X,[Y|_]).
eval( cdr(X) ,Y)       :- !,eval(X,[_|Y]).

/* eval, if functions */

eval( if(BoolExpr,ThenForm,ElseForm) ,Te)
        :- eval(BoolExpr,Xe),!,eval(ThenForm,Te).
eval( if(BoolExpr,ThenForm,ElseForm) ,Ze)    :- !,eval(ElseForm,Ze).

eval( eval(Form) ,Result)    :- !,eval(Form,Z),eval(Z,Result).
eval( apply(FnExpr,Args),R) :- !,
              eval(FnExpr,Fn),
              NewExpr=..[Fn|Args],
              eval(NewExpr,R).
/* IO */

eval( read(X), Y)   :- !,read(X),Y=X.
eval( prin1(X), R)  :- !,eval(X,R),write(R).
eval( print(X),R)   :- !,eval(X,R),write(R),nl.

/* interface to prolog */

eval( goal(Goal,Body) ,Result) :- !,call(Goal),eval(Body,Result).
eval( goal(Goal),Goal)         :- !,call(Goal).
eval( goalall(Goal),R)         :- !,setof(Goal,Goal,R).
eval( goalall(Goal,Body),true) :- !,doall(Goal,Body).
doall(G,B)                     :- call(G),eval(B,R),fail.
doall(G,B).
eval( assert(Fact), Xfact)     :- eval(Fact, Xfact),assert(Xfact).
eval( assertq(Fact), Fact)     :- assert(Fact).

/* LOAD RELATIONAL DATABASE INTERFACE */

 :- consult('/na/doe/guest/cohen/prolog/lispl/rdb');true.


printall(G,B) :- eval(goal(G,B),R),write(R),nl,fail.
```

```
printall(G,B).

/* histroy expression */

eval( h(N),Result)      :- !,history(N,Expr),eval(Expr,Result).
eval( h(N1,N2),true)    :- !,h(N1,N2).
h(N1,N2)                :- N1=<N2,history(N1,Expr),
                           write(N1),write(': '),write(Expr),nl,
                           Nx is N1 + 1,h(Nx,N2).

h(_,_).

/* def trace load */

eval( load(File),File)  :- !,see(File),nofileerrors,doload.
doload                  :- read(E),!,doE(E).
doE(E)                  :- E == end_of_file,!.
doE(E)                  :- eval(E,R),doload.
doload                  :- seen.

eval( trace(Fn),Fn)     :- !,(trace(Fn);assert(trace(Fn))).
eval( untrace(Fn),Fn)   :- !,retract(trace(Fn)).
eval( def(F,L), F)      :- !,dodef(F,L).
dodef(F,L) :- putq(L,Lq),
          (retract(def(F,autoload(_)));true),
        assert(def(F,Lq)).

eval( Cond,Result)      :- functor(Cond,cond,N),!,
                        Cond =..[cond|Clauses],!,evcond(Clauses,Result).
evcond([C1|Cr],R)       :- C1 =.. [Stam,P|E],eval(P,Px),!,evand(E,R).
evcond([(P)|Cr],R)      :- eval(P,R).
evcond([C1|Cr],R)       :- evcond(Cr,R).
evcond([],[]).

/* list or and functions */

eval( List,Result)      :- functor(List,list,N),!,
                        List =..[list|El],evlist(El,Result).
evlist([],[]).
evlist([E1|Er],[X1|Xr]) :- eval(E1,X1),evlist(Er,Xr).
eval( Term, Result)     :- functor(Term,term,N),!,
                Term =.. [term| El], evlist(El,Ex),!,
                Result =.. Ex.
/* term <-> list */

eval( list_to_term(L), R)   :- !,eval(L,Lx), R =.. Lx.
eval( term_to_list(T), R)   :- !,eval(T,Tx), Tx =.. R.

eval( Or ,Result)       :- functor(Or,or,N),!,Or =..[or|Er],evor(Er,Result).
evor([E1|Er],Result)    :- eval(E1,Result).
evor([E1|Er],Result)    :- evor(Er,Result).

eval( And ,Result)      :- functor(And,and,N),!,
                        And =..[and|Er],evand(Er,Result).
```

```
evand([E],R)          :- eval(E,R).
evand([E1|Er],Result)  :- eval(E1,Stam),!,evand(Er,Result).

/* apply user functions */

eval( Form,Result)    :-
            Form =..[Fn|Args],
            do_args(Fn,Args,Vars,Body),
            enter_trace(Fn,Vars,L),
            eval(Body,Result),
            exit_trace(Fn,Result,L).

/* binding(B,B).  isn't it simple ??? */

/* arguments evaluation if lambda */

do_args(F,A,V,B) :- def(F, lambda(V,B)),eval_args(A,V).
do_args(F,A,V,B) :- def(F,nlambda(V,B)),A=V.
do_args(F,A,V,B) :- def(F,autoload(File)), consult(File),
                   (retract(def(F,autoload(File)));true),do_args(F,A,V,B).

do_args(F,A,V,B) :- def(F,relation),eval_args(A,V),T =.. [F|V],B=..[goal, T].


def( pp,    autoload( '/na/doe/guest/cohen/prolog/lispl/pp' ) ).
def( pplen, autoload( '/na/doe/guest/cohen/prolog/lispl/pp' ) ).

eval_args([A1|Al],[X1|Xl]) :- eval(A1,X1),eval_args(Al,Xl).
eval_args([],[]).

/* trace printout */

enter_trace(F,X,L)   :- trace(F),!,
                        (retract(trace_lvl(F,N));N=0),L is N + 1,
                       tab(L),
                       assert(trace_lvl(F,L)),
                       write(' -enter- '),write(F),write('    ARGS : '),
                       write(X),nl.
enter_trace(F,X,L).
exit_trace(F,R,L)    :- trace(F),!,
                       retract(trace_lvl(F,N)),
                       tab(L),
                       L1 is L - 1,assert(trace_lvl(F,L1)),
                       write(' -exit - '),write(F),write('    RESULT: '),
                       write(R),nl.
exit_trace(F,R,L).

/* avoid re-evaluation of vars */

putq(A,A)    :- (atom(A);integer(A)),!.
putq(A,value(A)) :- var(A),!.

/* The first argument of sp_fn functions is not "valued" */
```

```
putq(A,B)    :- A=..[F,X1|Xs],
        sp_fn(F),!,
        doputq(Xs,Y),
        B=..[F,X1|Y].
/* simple functions */

putq(A,B)    :- A=..[F|X],
        doputq(X,Y),
        B=..[F|Y].

sp_fn(setq).
sp_fn(is).
sp_fn(goal).
sp_fn(goalall).
sp_fn(q).
sp_fn(quote).
sp_fn(lambda).
sp_fn(nlambda).
sp_fn(assertq).

doputq([],[]) :- !.
doputq([C|D],[Cq|Dq]) :- putq(C,Cq),doputq(D,Dq).

/* friendly top loop */

top :- abolish(history,2),top(1).
top(N) :-
        repeat,abolish(trace_lvl,1),
        write('-'),write(N),write('- '),
        write('APPLOG: '),
        read(E),
      ((functor(E,def,2),Eq=E);putq(E,Eq)),!,
        assert(history(N,Eq)),
        ((eval(Eq,R),write('RESULT IS: '),write(R))
         ;write(' F A I L ')
        ),nl,
        N1 is N + 1,!,top(N1).
```