Research Report AI-1989-08
Efficient Prolog: A Practical Guide

Michael A. Covington

Artificial Intelligence Programs
The University of Georgia
Athens, Georgia 30602

August 16, 1989

Abstract

Abstract: Properly used, Prolog is as fast as any language with comparable power. This paper presents guidelines for using Prolog efficiently. Some of these guidelines rely on implementation- dependent features such as indexing and tail recursion optimization; others are matters of pure algorithmic complexity.

Many people think Prolog is inefficient. This is partly because of the poor performance of early experimental implementations, but another problem is that some programmers use Prolog inefficiently.

Properly used, Prolog performs automated reasoning as fast as any other language with comparable power. It is certainly as fast as Lisp, if not faster. There are still those who rewrite Prolog programs in C "for speed," but this is tantamount to boasting, "I can implement the core of Prolog better than a professional Prolog implementor."

This paper will present some practical guidelines for using Prolog efficiently. The points made here are general and go well beyond the implementation-specific advice normally given in manuals.

1 Think procedurally as well as declaratively.

Prolog is usually described as a declarative or non-procedural language. This is a half-truth. It would be better to say that most Prolog clauses can be read two ways: as declarative statements of information and as procedures for using that information. For instance,

```
in(X,usa) :- in(X,georgia).
```

means both "X is in the U.S.A. if X is in Georgia" and "To prove that X is in the U.S.A., prove that X is in Georgia."

Prolog is not alone in this regard. The Fortran statement

```
X=Y+Z
```

can be read both declaratively as the equation $x = y + z$ and procedurally as the instructions LOAD Y, ADD Z, STORE X. Of course declarative readings pervade Prolog to a far greater extent than Fortran.

Sometimes the declarative and procedural readings conflict. For example, Fortran lets you utter the mathematical absurdity X=X+1. More subtly, the Fortran statements

```
A = (B+C)+D

A = B+(C+D)
```

look mathematically equivalent, but they give profoundly different results when B=10000000, C=-10000000, and D=0.0000012345.

Analogous things happen in Prolog. To take a familiar example, the clause

```
ancestor(A,C) :-
      ancestor(A,B), ancestor(B,C).
```

is part of a logically correct definition of "ancestor," but it can cause an endless loop when Prolog interprets it procedurally.

The loop arises because, when B and C are both unknown, the goal `ancestor(A,B)` on the right is no different from `ancestor(A,C)` on the left. The clause simply calls itself with essentially the same arguments, making no progress toward a proof. But if the clause is rewritten as

```
ancestor(A,C) :-
      parent(A,B), ancestor(B,C).
```

there is no loop because ancestor cannot call itself with the same arguments.

The moral is that to use Prolog effectively, one must understand not only the declarative reading of the program but also the procedures that the computer will follow when executing it. The limitations of Prolog's built-in proof procedures are not flaws in the implementation; they are deliberate compromises between logical thoroughness and efficient search.

2 Narrow the search.

Searching takes time, and an efficient program must search efficiently. In a knowledge base that lists 1000 gray objects but only 10 horses, the query

```
?- horse(X), gray(X).
```

can be 100 times as fast as the alternative

```
?- gray(X), horse(X).
```

because it narrows the range of possibilities earlier.

Many opportunities to narrow the search space are much more subtle. Consider the problem of determining whether two lists are set-equivalent — that is, whether they have exactly the same elements, though not necessarily in the same order.

Two lists are set-equivalent if and only if one of them is a permutation of the other. One strategy, then, is to generate all the permutations of the first list and compare them to the second list:

```
set_equivalent(L1,L2) :-
      permute(L1,L2).
```

The trouble is that an N-element list has N! permutations; testing the set-equivalence of two 20-element lists can require 2.4×10^{18} comparisons. I have actually seen someone attempt this in a Prolog program.

It is much faster to sort both lists and compare the results:

```
set_equivalent(L1,L2) :-
      sort(L1,L3), sort(L2,L3).
```

An N-element list can be sorted in about $N\, log_2\, N$ steps — i. e., about 86 steps per 20-element list — and each "step" involves considerably less work than generating a new permutation. So this technique is faster than the first one by a factor of more than 10^{16}.

3 Let unification do the work.

As a classroom exercise I ask my students to write a predicate that accepts a list and succeeds if that list has exactly three elements. Some of the weaker answers that I get look like this:

```
has_three_elements(X) :-
      length(X,N),
      N = 3.
```

Slightly better are those that say

```
has_three_elements(X) :-
      length(X,3).
```

thereby letting the built-in pattern-matcher test whether length returns 3. But the best students cut the Gordian knot by writing:

```
has_three_elements([_,_,_]).
```

The point is that `[_,_,_]` matches any three-element list and nothing else. Unification does all the work.

Unification can even rearrange the elements of a data structure. Here is a predicate that accepts a list and generates, from it, a similar list with the first two elements swapped:

```
swap_first_two([A,B|Rest],[B,A|Rest]).
```

Again, unification does all the work. More precisely, the data structures `[A,B|Rest]` and `[B,A|Rest]`, or templates for them, are created when the program is compiled, and unification gives values to the variables at run time.

4 Avoid `assert` and `retract`.

Beginners tend to overuse the `assert` and `retract` predicates to modify the knowledge base. There are two good reasons not to do so: `assert` and `retract` are relatively slow, and, perhaps more importantly, they lead to messy logic.

Even the slowness is twofold. It takes appreciable time to perform an `assert` or `retract`. Further, in most implementations, a predicate that has been (or can be) modified by `assert` or `retract` cannot run at full compiled speed., (*ALS Prolog* is a striking exception.)

More importantly, the haphazard use of `assert` and `retract` confuses the program logic. The effects of `assert` and `retract` are not undone by backtracking. By contrast, most predicates return their results by instantiating variables; these instantiations are discarded if the overall goal fails, and you get only the results of computations that have succeeded. If you use `assert` as a general way to store temporary data, you will end up unable to tell whether the data came from successful computations. This can make programs very hard to debug.

The normal way to store temporary information is to pass it along from one step to the next as arguments to procedures. The legitimate uses of `assert` and `retract` are to record new knowledge in the knowledge base (in a program that "learns") and, less commonly, to store the intermediate results of a computation that *must* backtrack past the point at which it gets its result. Even in the latter case, the built-in predicates `bagof` and `setof` often provide a better way to collect alternative solutions into a single structure. They are implemented in hand-optimized machine code and are faster than anything you could construct in Prolog.

5 Understand tokenization.

The internal memory representation of data in Prolog can be quite different from the printed representation. The fundamental unit is the term, of which there are three types: numbers, atoms, and structures. Numbers are stored in fixed-point or floating-point binary, the same as in most other programming languages. Atoms and structures have representations specific to Prolog.

Atoms are stored in a symbol table in which each atom occurs only once; atoms in the program are replaced by their addresses in the symbol table. This is called *interning* or *tokenization* of the atoms, and it is performed whenever Prolog reads atoms and recognizes them as such — when loading the program, accepting queries from the keyboard, or even accepting input at run time with the read predicate. Whenever an atom exists, it is in the symbol table.

Because of tokenization, a Prolog data structure can be much shorter than it looks: repeated occurrences of the same atom take up little additional space. Despite its appearance, the structure

```
f('What a long atom this seems to be',
  'What a long atom this seems to be',
  'What a long atom this seems to be')
```

is very compact — possibly smaller than `g(aaaaa,bbbbb,ccccc)`. The memory representations of these two structures are shown in Figure 1.

Further, atoms can be compared more quickly than anything else except numbers. To compare two atoms, even long ones, the computer need only compare their addresses. By contrast, comparing lists or structures requires every element to be examined individually.

Consider for example the following two tests:

```
a \= b
```

```
aaaaaaaa \= aaaaaaab
```

Without tokenization, the second test would take longer because it would be necessary to compare eight corresponding characters instead of just one. In Prolog, however, the second test is just as fast as the first, because all that it does is verify that two addresses in the symbol table are different. The strings `aaaaaaaa` and `aaaaaaab` were assigned to distinct addresses once and for all when they were first tokenized.

6 Avoid string processing.

Character string handling is rarely needed in Prolog except to convert printable strings into more meaningful structures. The input to a natural language parser, for instance, should be

```
[the,dog,chased,the,cat]
```

rather than

```
"the dog chased the cat"
```

so that the benefits of tokenization can be obtained.

Character strings in Prolog are bulky. Whereas abc is a single atom, the string "abc" is a list of numbers representing ASCII codes, i.e., `[97,98,99]`. Recall that a list, in turn, is a head and a tail held together by the functor '.', so that `[97,98,99]` is really .(97,.(98,.(99,[ ]))), represented internally as shown in Fig. 2. Strings are designed to be easily taken apart; their only proper use is in situations where access to the individual characters is essential.

Arity/Prolog has an alternative type of string, written abc or the like, that is stored compactly but not interned in the symbol table. This is important

because there is usually a limit on the number of symbols in the table; a program with lots of textual messages can avoid hitting this limit by using $-strings instead of long atoms.

The built-in predicate `read` tokenizes its input. Many implementations provide predicates that are like read except that they accept input from a list of characters. With such a predicate, it is easy to preprocess a character string to make it follow Prolog syntax, then convert it to a Prolog term.

7 Recognize tail recursion.

Because Prolog has no loop structures, the only way to express repetition is through recursion. Variables that change value from one iteration to the next must be passed along as arguments, thus:

```
count(N) :-
      write(N), nl,
      NewN is N+1,
      count(NewN).
```

Recursion can be inefficient because, in general, each procedure call requires information to be saved so that control can return to the calling procedure. Thus, if a clause calls itself 1000 times, there will be 1000 copies of its stack frame in memory.

There is one exception. *Tail recursion* is the special case in which control need not return to the calling procedure because there is nothing more for it to do. In this case the called procedure can be entered by a simple jump without creating a stack frame.

Most Prologs recognize tail recursion and transform it into iteration so that repeated execution does not consume memory. In Prolog, tail recursion exists when:

1. the recursive call is the last subgoal in the clause;

2. there are no untried alternative clauses;

3. there are no untried alternatives for any subgoal preceding the recursive call in the same clause.

Figure 3 shows a tail recursive predicate and three predicates that are not tail recursive for different reasons.

A tail recursive predicate normally contains one or more tests to stop the recursion. These must normally precede the recursive clause, thus:

```
count_to_100(X) :-
      X > 100.
      /* succeed and do nothing */

count_to_100(X) :-
      X =< 100,
      write(X), nl,
      NewX is X+1,
      count_to_100(NewX).
```

However, the recursive clause can be followed by other clauses if, at the time of the call, they will have been ruled out by cuts or by indexing (see below).

The lack of conventional loop constructs is not a flaw in Prolog. On the contrary, it makes it easier to prove theorems about how Prolog programs behave. For years, mathematicians have dealt with repetitive patterns by using inductive proofs — that is, by substituting recursion for iteration. Prolog does the same thing. After years of using both Prolog and Pascal almost daily, I find the Prolog approach to repetition less error-prone.

8 Let indexing help.

To find a clause that matches the query

```
?- f(a,b).
```

the Prolog system does not look at all the clauses in the knowledge base — only the clauses for f. Associated with the functor f is a pointer or hashing function that sends the search routine directly to the right place. This technique is known as *indexing*.

Many implementations carry this further by indexing on not only the predicate, but also the principal functor of the first argument. In such an implementation, the search considers only clauses that match f(a,...) and neglects clauses such as f(b,c).

First-argument indexing is a trade-off. Its intent is to save execution time and, even more importantly, to save memory by reducing the need to record backtrack points. But the indexing process itself complicates the search by requiring more analysis of the thing being searched for. Indexing on the principal functor of the first argument represents a reasonable compromise.

Indexing has two practical consequences. First, arguments should be ordered so that the first argument is the one most likely to be known at search time, and preferably the most diverse. With first-argument indexing, the clauses

```
f(a,x).
f(b,x).
f(c,x).
```

can often be searched in one step, whereas the clauses

```
f(x,a).
f(x,b).
f(x,c).
```

always require three steps because indexing cannot distinguish them.

Second, indexing can make a predicate tail recursive when it otherwise would not be. For example,

```
f(x(A,B)) :- f(A).
f(q).
```

is tail recursive even though the recursive call is not in the last clause, because indexing eliminates the last clause from consideration: any argument that matches x(A,B) cannot match q. The same is true of list processing predicates of the form

```
f([Head|Tail],...) :- ...
f([],...).
```

because indexing distinguishes non-empty lists from `[]`.

9 Use mode declarations.

Normally, Prolog assumes that each of the arguments to a predicate may be instantiated or uninstantiated. This results in compiled code that branches to several alternative versions of each procedure in order to handle all the combinations.

Most compilers provide a mode statement that allows you to rule out some of the alternatives and thereby speed up execution. For instance, the predicate

```
capital_of(georgia,atlanta).
```

can be used with either argument instantiated, or both, or none, but if written as

```
:- mode capital_of(+,-).

capital_of(georgia,atlanta).
```

it requires the first argument to be instantiated and the second to be uninstantiated.

Add mode declarations cautiously after the code has been debugged. At least one manual warns ominously that if the mode declarations are violated, "In some cases, the program will work. In others, your program will produce erroneous results or not work at all."

The ideal Prolog compiler would perform dataflow analysis and generate at least some of its mode declarations automatically.

10 Work at the beginning of the list.

The only directly accessible element of a list is the first one. It pays to perform all manipulations there and avoid, as far as possible, traversing the whole length of the list.

Sometimes this entails building the list backward. After all, there is nothing sacred about the left-to-right order in which lists are normally printed. For example, a program that solves a maze might record its steps by adding them at the beginning of a list. The result is a list giving the path, backward.

Working at the beginning of the list can really pay off in efficiency. A classic example is a way of reversing a list. The familiar, inefficient "naive reverse" predicate is the following:

```
reverse([],[]).

reverse([H|T],Result) :-
      reverse(T,ReversedT),
      append(ReversedT,[H],Result).
```

But reversing an N-element list this way takes time proportional to N^2. One factor of N comes from the fact that reverse is called once for each list

element. The other factor of N comes from append(ReversedT,`[H]`,Result) because append has to step through all the elements of ReversedT in order to get to the end and attach `[H]`. This takes time proportional to the length of ReversedT, which in turn is proportional to N.

A faster way to reverse a list is to extract elements one by one from the beginning of one list and add them at the beginning of another. This requires a three-argument procedure, where the third argument is used to return the final result:

```
fast_reverse(List1,List2) :-
      fr(List1,[],List2).

fr([Head|Tail],SoFar,Result) :-
      fr(Tail,[Head|SoFar],Result).

fr([],SoFar,SoFar).
```

The first clause of fr transfers the first element of `[Head|Tail]` to SoFar, then calls itself to do the same thing again. When `[Head|Tail]` becomes empty, the second clause of fr unifies SoFar with Result. The process takes linear time.

11 Avoid CONSing.

In Prolog, as in Lisp, it is much easier to examine existing structures than to create new ones. Creating new structures (known in Lisp as CONSing) requires dynamic allocation of memory.

If, therefore, the same computation can be done with or without CONSing, the version that avoids CONSing will be faster. Often, CONSing is avoided simply by working at the beginning of the list. Sterling and Shapiro illustrate this with two algorithms to test whether one list is a sublist of another.

Another way to avoid CONSing is to build structures by progressive instantiation rather than by copying. Most Prolog predicates that modify structures do so by building, from the original structure, a new one that is different in some way; `append`, `reverse`, and similar list manipulations are familiar examples. The alternative is to add information to a structure by instantiating parts of it that were originally uninstantiated.

For example, the list `[a,b,c|X]` can be turned into `[a,b,c,d|Y]`, without CONSing, simply by instantiating X as `[d|Y]`. Such a list, with an uninstantiated tail, is called an *open list*. The same principle can be used to build open trees and open data structures of other shapes.

The problem with `[a,b,c|X]` is that the only way to get to the X is to work down the list starting at a. Although this does not require CONSing, it does take time. Processing can go faster if another instance of X is kept outside the list where it can be accessed directly. The resulting structure is called a difference list and has the form

```
f([a,b,c|X],X)
```

where f is any two-argument functor; the infix operators / and - are often used for the purpose, and the above list is written `[a,b,c|X]`-X or the like.

Difference lists can be concatenated very quickly — once — by instantiating the tail of the first list to the whole of the second list. The first list then becomes the result of the concatenation; there is no third, concatenated, list to be produced. This is the Prolog equivalent of the LISP function RPLACD. It is somewhat less destructive because, like all Prolog instantiations, difference list concatenations are undone upon backtracking.

12 Conclusion

All these techniques for improving efficiency share a common thread – awareness of procedural aspects of a declarative language. This does not mean they are all low-level, inelegant "tricks" that purists should ignore.

Some of the techniques *are* low-level, such as indexing and tail recursion optimization. Prolog would still be Prolog if these features were eliminated or changed radically. The decision to index on the principal functor of the first argument is certainly arbitrary, and if indexing went away tomorrow, some programs would lose efficiency but none would lose correctness.

Other techniques, however, are purely algorithmic. Even when stated declaratively, algorithms consist of steps, and one algorithm can have more steps than another. It will always be faster to test the set-equivalence of lists by sorting than by permuting, simply because there are too many permutations, and no conceivable implementation can change this fact.

Between the two extremes are data-structure-dependent techniques such as working at the beginning of a list. The first element of every list is the most accessible, not because of some quirk of implementation, but because the underlying semantics of Prolog says that `[a,b,c]` is really .(a,.(b,.(c,`[]`))). An optimizing implementation might provide faster access to list elements that are theoretically hard to get to, just as an optimizing Fortran compiler can move certain statements outside of loops, but one should not rely on the implementor to make the language more efficient than its semantics calls for.

Acknowledgement
I want to thank Don Potter for helpful suggestions. All opinions and errors in this paper are of course my own.

References

[1] D.H.D. Warren and L.M. Pereira. "Prolog — The Language and its Implementation Compared with Lisp," *ACM SIGPLAN Notices*, Vol. 12, No. 8, August 1977, pp. 109–115.

[2] *Quintus Prolog User's Guide*, Quintus Computer Systems, Mountain View, Calif., version 11 for release 2.0, 1987, p. 98.

[3] *Using the Arity/Prolog Interpreter and Compiler*, Arity Corporation, Concord, Mass., 1987, p. 104.

[4] *ALS Prolog 1.2*, Applied Logic Systems, Syracuse, N.Y., 1988.

[5] R.A.O'Keefe, "On String Concatenation," *Prolog Digest*, Vol. 5, No. 100.

[6] *Arity/Prolog 5.0*, Arity Corporation, Concord, Mass., 1987.

[7] *Building Arity/Prolog Applications*, Arity Corporation, Concord, Mass., 1986, p. 25.

[8] C.S.Mellish, "Some Global Optimizations for a Prolog Compiler," *Journal of Logic Programming*, Vol. 2, 1985, pp. 43–66.

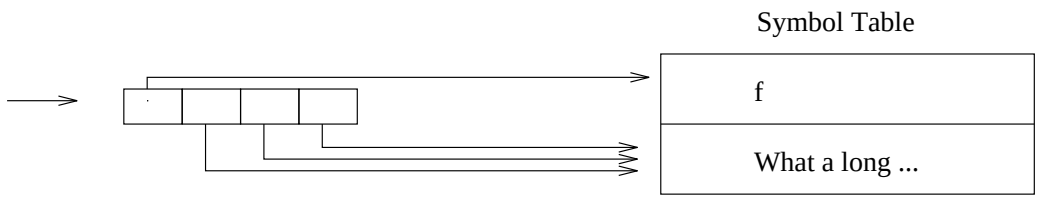
[9] S.K.Debray and D.S. Warren, "Automatic Mode Inference for Prolog Programs," *Proceedings, 1986 Symposium on Logic Programming*, IEEE Computer Society, pp. 78–88.

[10] L.Sterling and E.Shapiro, *The Art of Prolog: Advanced Programming Techniques*, M.I.T. Press, Cambridge, Mass., 1986, p. 194.

Figure 1.

Memory representations of two structures.

```
f('What a long atom this is',
  'What a long atom this is',
  'What a long atom this is')
```




```
g(aaaaa,bbbbb,ccccc).
```

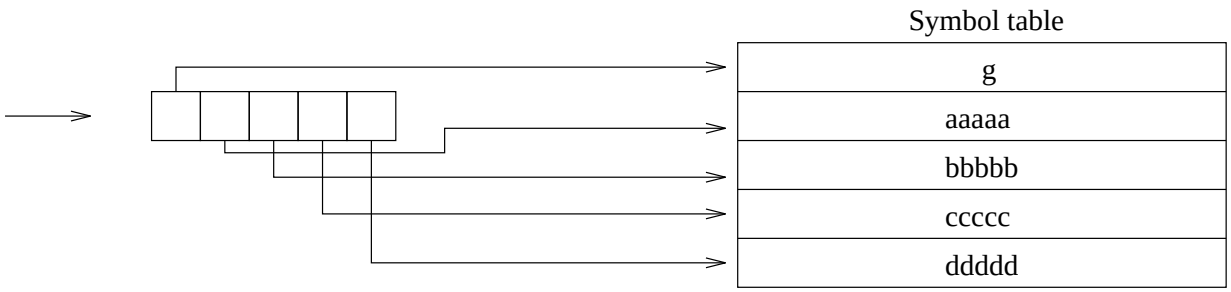


Figure 2.

Internal representation of the list [97,98,99] (equivalent to the string "abc").

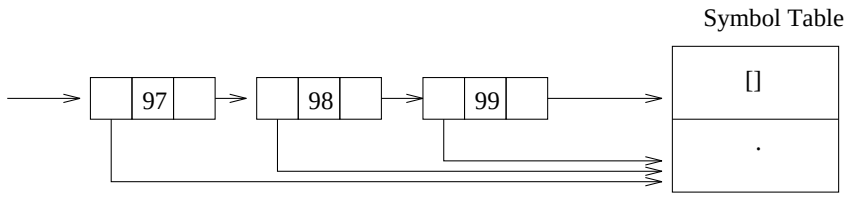


Figure 3.

Recursion does not consume memory if the recursive call is the very last step of the calling procedure.

```
% This predicate is tail recursive
% and can run forever.

test1 :- write(hello), nl, test1.


% This predicate is not tail recursive
% because the recursive call is not last.

test2 :- test2, write(hello), nl.


% This predicate is not tail recursive
% because it has an untried alternative.

test3 :- write(hello), nl, test3.
test3 :- write(goodbye).


% This predicate is not tail recursive
% because a subgoal has an untried alternative.

test4 :- g, write(hello), nl, test4.

g :- write(starting).
g :- write(beginning).
```