Prolog vs. Lisp

Carl Ponder

Computer Science Division
University of California
Berkeley, CA 94720

August 17, 1989

Abstract

Prolog and Lisp benchmark timings are compared on the VAX 8600. The measured Lisp-to-Prolog time ratio varies between 0.143 and 2.83. These differences between Prolog and Lisp performance can be explained by studying the structure of the benchmarks and the language implementations. Previous authors have used such measurements as evidence that one language is "better" than the other; their works are summarized. The issues involved in comparing two languages are treated, as well as the assumptions involved in interpreting language performance data.

Acknowledgment

This work was supported in part by the Army Research Office, grant DAAG29-85-K-0070, through the Center for Pure and Applied Mathematics, University of California at Berkeley, and the Defense Advanced Research Projects Agency (DoD) ARPA order #4871, monitored by Space & Naval Warfare Systems Command under contract N00039-84-C-0089, through the Electronics Research Laboratory, University of California at Berkeley.

1 Introduction

> ...a very well-known professor of Computing Science made the statement that "Algol 60 was a very inefficient language", while what he really meant was that, with the equipment available to him, he and his people had not been able to implement Algol 60 efficiently. (That is what he meant, he did not mean to say it!)
>
> – Edsger W. Dijkstra, 1974

The first Prolog compiler was presented by Warren [15]. He argued that his Prolog compiler produced reasonably efficient code by comparing timings of benchmarks written in Prolog, Lisp, and Pop-2. He went further to explain why some benchmarks ran faster in Prolog and others ran faster in Lisp [14][15]. Warren's Prolog compiler used an intermediate form to represent program information between the syntax-analyzer and code-generator. This intermediate form evolved into what is known as the *Warren Abstract Machine* (WAM) [16].

The WAM is a very high-level instruction set; early Prolog implementations compiled Prolog into WAM and executed the WAM instructions on a software simulator, or macroexpanded the WAM instructions into native code. More recent implementations use microcode to execute WAM instructions [2], or use more sophisticated code-generators to produce better native code. These newer implementations appear to provide a significant jump in performance. It is reasonable to wonder how advances in Prolog implementation technology compare to similar advances for Lisp, which likewise include compiler optimizations and microcode or other forms of architectural support.

The intent of this paper is to provide a framework for comparing Lisp and Prolog implementations both now and in the future, as technologies continue to advance. Specifically it will

- Discuss the issues involved in comparing the performance of two languages.
- Summarize previous efforts.
- Present a number of benchmark results, and explain differences in performance.
- Discuss the interpretation of the results and the assumptions involved.

2 Comparing Languages X & Y

Comparing two languages is difficult. There are a number of ambiguities to resolve, in terms of

1. What is being compared: Programmability? The dollar cost of the compiler? Execution speed? Memory requirements?
2. What comparison to make: Benchmark tests? Formal analysis? Many important aspects of programming languages are difficult to quantify.
3. How to interpret the results: Are the results absolute, valid across all programs? Do they indicate trends? To what extent?

On the most abstract level, a programming language can be characterized as a mapping of programs to the functions that they compute. We can look at a few questions from this perspective:

1. Program size: for a given function F, which language maps the shortest program to F?
2. Program semantics: languages X and Y map program P to functions F_X and F_Y, respectively. How do F_X and F_Y compare? For example, one might be an extension of the other.
3. Language semantics: in an enumeration of programs, how are the computable functions ordered under each language?

The answers to these questions might tell us whether one language is inherently more compact or more efficient than another. Unfortunately proofs on such an abstract level tend only to work only for examples too simple or too technical to be of any interest.

More concrete questions of immediate interest are as follows:

1. *How easy is the language to program in?* This could refer to programming in general, construction of a particular program, or programming for a given class of applications. It is difficult to measure how easy a language is to learn and use. Ease of programming is strongly affected by outside factors, such as the available documentation and the programming environment.

2. *How easy is the language to implement?* The implementation includes aspects of the hardware, compiler, and runtime system. Some languages make certain optimizations more obvious. An implementation may be easy because it performs poorly or provides little debugging information. A more abstract language may be harder to implement.

3. *How efficiently does the language execute?* Efficiency is dependent upon the implementation. A highly optimizing compiler may speed up execution time at the expense of debug time, making the system harder to program. One may wonder how efficiently a language can be implemented given the hardware resources at hand; available theoretical approaches appear to provide no answer, although lowerbound techniques might be applied in specific cases [10].

Execution efficiency is the focus of this report. The approach taken is to compare execution times of benchmark programs, on reasonably fast implementations of Lisp and Prolog. There are a number of interrelated considerations affecting the performance of a given benchmark:

1. *Use of an Interpreter.* Programs tend to run 5-10 times more slowly interpreted than compiled. Interpreters are generally designed to provide a high degree of functionality without taking a long time to build; program execution time usually suffers.

2. *Efficiency of the runtime model.* An interdependency exists between the representation of the program state at runtime, the design of the compiler code-generator, and the functionality of the hardware. All three of these factors affect performance. Design of the runtime system can make up for apparent inadequacies in the hardware functionality (the Smalltalk SOAR design, for example [13]).

3. *Compiler optimizations.* Optimizations have been applied to "standard" languages like Fortran for quite some time. For newer languages like Prolog these have yet to be explored. Two "good" compilers for the same language and machine can produce object code differing by 10-25% in performance.

4. *Architecture.* Both the speed and the functionality of the architecture has a strong impact on performance. The raw speed of the Cray computer, for example, makes it one of the fastest Lisp machines.

5. *"Language Power".* Some languages might not be able to express algorithms as efficiently as others, beyond the capability of any compiler to correct. Nothing concrete is known about this, though some hypotheses have been made. In particular, it is hypothesized that the lack of destructive operations in Prolog and Pure Lisp is a liability [10].

The difference in execution time for two benchmarks reflects a combination of these factors. The degree to which each consideration affects a given benchmark can be determined by analyzing the benchmark and its steps of execution. It is difficult, however, to extrapolate the "general" case from specific benchmarks; such results are always questionable.

Considerations 2 & 3 are obviously driven by language semantics, but are still conditioned by the way we think about hardware organization. The real relationship between language semantics and

hardware is poorly understood. It is often the case that one system will run faster for some programs and more slowly for others, which makes it difficult to decide which individual design decisions are superior.

3 A Summary of Previous Efforts

By now we have seen a number of ambiguities regarding the comparison of two languages. These ambiguities can potentially cause sloppy analysis of results and erroneous conclusions. Some previous language comparison efforts are described here.

3.1 Lisp/Prolog Performance Comparisons

Gutierrez compared the (DEC-10) performance of a theorem-prover written in both Lisp and Prolog, finding the Lisp version to run about 3 times faster than the Prolog version [4]. He stated that the experiment had been undertaken to "provide a basis for choosing a language for a large research project." A general factor of 3 in performance would probably be a good reason to favor one language over another, were there no other factors involved - after all, why not use assembly language? Regardless, O'Keefe rebutted this [8], rewriting the theorem-prover to be faster in both Lisp and Prolog; the new Prolog version generally outran the new Lisp version by 30-50%. In both Gutierrez' and O'Keefe's discussions there were underlying issues which were not treated:

1. Why were the programs structured as they were? To be most efficient? Most natural? The theorem-prover written by Gutierrez looks reasonable at first glance; O'Keefe showed that it handicapped the Prolog implementation several ways. This argument is difficult to resolve; how can you claim to have made the "ultimate rewrite" of a program?

2. What efficiency issues are purely implementation-dependent? For example, O'Keefe stated that Prolog is faster for manipulating records and Lisp is faster for manipulating lists. This is purely an artifact of the implementations; the runtime representations of the data structures can be adjusted if they penalize performance significantly. Another example is the "lazy cons" tail-recursion optimization that Lisp compilers ignore but Prolog compilers do not; this optimization is regarded as an important efficiency enhancement in Prolog, whereas Lisp implementors don't consider it worth the effort. The Prolog syntax encourages a style of programming which the compiler can readily optimize; the Lisp syntax does not.

3. What efficiency issues are purely language-dependent? The "inherent efficiencies" of the languages are ostensibly the issue in these papers, but it is not clear that anything other than programming style and implementation tradeoffs are being compared. The fundamental difference in language semantics suggest that some inherent difference in performance must exist [11], but as yet yields no insight for real cases.

These issues will considered in our later discussion.

Tick [12] made a more analytic comparison using the Gabriel benchmarks [3] *Tak*, *Boyer*, *Deriv*, and *Puzzle*. He took counts of procedure-calls, memory references, and instructions executed, and compared raw performance on a Sun-3. In particular, the timing ratios of Lisp-to-Prolog were 0.1, 0.59, 0.67, and 3.47, respectively. (The measurements for *Puzzle* presented later show Lisp to take about 50% of the time of Prolog. The remaining results are relatively consistent). Major details of the compilers and architectures which would tend to bias the results were made explicit. The Lisp was native-code compiled from a portable intermediate form, and the Prolog was compiled to an intermediate form which was interpreted. Neither of these implementations sounds efficient; any differences in performance could have been due to poor expressiveness of the the Lisp intermediate form, or the interpretation overhead of the Prolog intermediate form.

Tick's conclusion was that Prolog has a greater "semantic content" than Lisp; also that Lisp is better mapped onto current machines than Prolog. This "semantic content" is based on the the ratio of instruction to data transfer rates during execution; as such the "semantic content" is purely an artifact of the Prolog and Lisp runtime models. The superior mapping of Lisp onto the hardware may be true *for those implementations;* very likely we will find better ways to implement Prolog in the future.

3.2 Benchmark Suites

Gabriel presented a "standard" suite of Lisp benchmarks and compared the results for a large number of implementations [3]. This caused a stir in the Lisp community not only because it gave customers a comparison upon which to choose, but because it forced implementors to realize the effects of their design decisions. A suite of Prolog benchmarks was developed by Wilk [17]. Okuno [9] presented the results for a combination of Lisp and Prolog benchmarks; only a few of the benchmarks were written in both Lisp *and* Prolog. Additional benchmarks for Prolog continue to appear; a particularly "meaty" collection was presented recently [6].

Most of the early Prolog and Lisp/Prolog benchmarks fall into the category of *diagnostic* or *microscopic benchmarks*, as they measure specific parameters such as the speed of procedure-call or list operations. Diagnostic benchmarks are perhaps more useful for tuning an implementation, as they identify potential performance bottlenecks. The performance results for diagnostic benchmarks tend to be easier to interpret, since the benchmarks are short or spend most of their execution time within small sections of code. The alternative is *application* or *macroscopic benchmarks*, such as the theorem-provers of Gutierrez and O'Keefe. Most of the Gabriel suite falls into this category; a few of these have been converted to Prolog, the results of which will be presented later.

3.3 Other Work

McDermott [7] discussed the pros and cons of the Prolog language from the perspective of the contemporary AI community, focusing on functionality aspects.

An interesting experiment was reported in [5], where programmers were timed while they coded a set of AI-related problems into Ada, Lisp, and Prolog. Their performance for Lisp and Prolog were roughly comparable up to the debugging stage, where the Prolog programmers were bogged down. The Ada programmers performed worst or near-worst all around. The results of such experiments are all too fallible, but are interesting to consider.

4 The Benchmarks & Measurements

In this section we measure the performance of Quintus Prolog 2.0 and Franz Lisp Opus 43.1 running on a VAX 8600 ("Vangogh" at UC Berkeley). Seven benchmarks are used, named *Nreverse*, *Tak*, *Boyer*, *Browse*, *Frpoly*, *Prover*, and *Puzzle*. The numbers are summarized below in table 1. Each benchmark was run 5 times, and the variation in timings was less than 10%. The entries for garbage-collection time were omitted when the value was zero (the Prolog system supports garbage-collection, but didn't perform any for these cases). "CPU" is the total execution time minus "GC", which is the time spent in garbage-collection. Franz Lisp "localf" declarations were used for fast function-call. Vectors were used in the *Puzzle* benchmark. "Fixnum" operators were used in most cases, which assume the results of integer operations stay within a fixed range.

Table 1 - Benchmark Timings for Lisp and Prolog (in seconds)				
		Prolog	Lisp	
Benchmark	Case	CPU	CPU	GC
Nreverse		0.0134	0.015	
Tak		2.45	0.35	
Boyer†		14.75	25.28	
Browse‡		18.84	53.33	
Frpoly	1	0.10	0.067	
	2	1.24	1.48	
	3	5.73	9.22	
Prover	1	0.034	0.033	
	2	0.13	0.083	
	3	0.10	0.05	
	4	0.083	0.033	
	5	0.10	0.083	
	6	0.18	0.13	
	7	0.20	0.13	
	8	0.35	0.35	0.25
	9	0.40	0.32	
	10	0.70	0.63	
Puzzle		10.34	5.04	
†12.18 seconds CPU/14.29 seconds GC for 8600 Common Lisp [3]. ‡38.69 seconds CPU for 8600 Common Lisp [3].				

The benchmarks are listed in Prolog in the appendix; the Lisp versions are in Gabriel [3], with the exception of *Prover* and *Nreverse* whose Lisp translations are also included in the appendix. We will examine the benchmarks one by one and try to explain the timing difference.

Nreverse

The *Nreverse* benchmark is a test of the so-called *lazy cons* tail-recursion optimization of Prolog. The clause

```
concatenate([X|L1],L2,[X|L3]) :- concatenate(L1,L2,L3).
```

when used with instantiated first and second arguments, will allocate a list cell for the third argument; the recursive call (destructively) "fills in" `L3`. This tail-recursive clause is compiled into an iterative loop by all WAM-based Prolog compilers.

The Lisp form of that same line is

```
(cons (car L1) (concatenate (cdr L1) L2))
```

which defers setting up the list cell until *after* the recursive call to concatenate. Rearranging this form to be tail-recursive gives

```
(setq L (cons (car L1) nil))
(dconcatenate L (cdr L1) L2)
```

where *dconcatenate* destructively attaches the result onto L. Tail-recursion optimization is now applicable. To do this automatically on the first form would require analysis techniques which are beyond any existing Lisp compiler. Programming in the style of the second form is awkward, so Lisp does appear to be at a disadvantage here.

The Prolog version of *Nreverse* runs about 10% faster than the Lisp version, which I (following the lead of Warren [14]) would attribute to the removal of tail-recursive procedure-call overhead.

Tak

The *Tak* benchmark, on the other hand, favors the Lisp system over the Prolog system. This is a quadruply-recursive program which performs integer arithmetic. WAM-based Prolog implementations are at a disadvantage here; the flow of control is strictly deterministic by the semantics of the arithmetic operations, but work is nonetheless performed to set up choice-points and maintain backtracking information. Data flow analysis would be sufficient to correct this, but it is not clear that Prolog programs would generally benefit from this optimization.

Boyer

Profiling the Prolog version of the *Boyer* benchmark shows that roughly 90% of the executed WAM instructions were in the routines

```
rewrite(Atom,Atom) :-
        atomic(Atom),!.
rewrite(Old,New) :-
        functor(Old,F,N),
        functor(Mid,F,N),
        rewrite_args(N,Old,Mid),
        ( equal(Mid,Next),
          rewrite(Next,New)
        ; New=Mid
        ),!.

rewrite_args(0,_,_) :- !.
rewrite_args(N,Old,Mid) :-
        arg(N,Old,OldArg),
        arg(N,Mid,MidArg),
        rewrite(OldArg,MidArg),
        N1 is N-1,
        rewrite_args(N1,Old,Mid).
```

The results will be very sensitive to how this portion executes. The Lisp version was not profiled, but the corresponding routines are considerably more complicated:

```
(defun rewrite (term)
  (cond ((atom term) term)
        (t (rewrite-with-lemmas (cons (car term)
                                      (rewrite-args (cdr term)))
           (get (car term) (quote lemmas))))))
```

```
(defun rewrite-args (lst)
  (cond ((null lst) nil)
        (t (cons (rewrite (car lst))
                 (rewrite-args (cdr lst))))))

(defun rewrite-with-lemmas (term lst)
  (cond ((null lst) term)
        ((one-way-unify term (cadr (car lst)))
         (rewrite (apply-subst unify-subst (caddr (car lst)))))
        (t (rewrite-with-lemmas term (cdr lst)))))


(defun apply-subst (alist term)
  (cond ((atom term)
         (cond ((setq temp-temp (assq term alist))
                (cdr temp-temp))
               (t term)))
         (t (cons (car term) (apply-subst-lst alist (cdr term))))))

(defun apply-subst-lst (alist lst)
  (cond ((null lst) nil)
        (t (cons (apply-subst alist (car lst))
                 (apply-subst-lst alist (cdr lst))))))


(defun one-way-unify (term1 term2)
  (progn (setq unify-subst nil)
         (one-way-unify1 term1 term2)))

(defun one-way-unify1 (term1 term2)
  (cond ((atom term2)
         (cond ((setq temp-temp (assq term2 unify-subst))
                (equal term1 (cdr temp-temp)))
               (t (setq unify-subst (cons (cons term2 term1)
                                          unify-subst))
                  t)))
        ((atom term1) nil)
        ((eq (car term1) (car term2))
         (one-way-unify1-lst (cdr term1) (cdr term2)))
        (t nil)))

(defun one-way-unify1-lst (lst1 lst2)
  (cond ((null lst1) t)
        ((one-way-unify1 (car lst1) (car lst2))
         (one-way-unify1-lst (cdr lst1) (cdr lst2)))
        (t nil)))
```

As McDermott pointed out [7], Prolog's real strength lies in the ability to describe pattern-matching processes in a concise way. The Lisp version of the benchmark has to define its own pattern-matcher which explicitly decomposes argument data-structures and matches basic patterns. This is done implicitly by the unification routines in the Prolog runtime system. The Lisp version pays

the overhead of the procedure-calls in the unification process, making it considerably slower. The Prolog version permits optimization of the tail-recursive call in *rewrite-args*. The Prolog version also represents data differently, using functors in some places rather than lists. Nonetheless, the 8600 Common Lisp results reported in [3] are more competitive with the Prolog measurements presented here.

Browse

The main routine *match* of the *Browse* benchmark consumes > 90% of the execution time. In the Lisp version it is 52 lines long; in the Prolog version it is 10 lines long. There are a number of places where operations are repeated, such as `(car pat)`, or dereferences are repeated in taking the `car` and `cdr` of the same variable. Such repeated work can be eliminated by the compiler if it confirms that there are no side-effects tampering with the intermediate pointers, so these pointers can be saved rather than recomputed. As with the "lazy-cons", such an optimization may not be generally useful.

The program can be rewritten to use temporary variables to hold intermediate values. The compiler must then be smart enough store these variables in registers rather than in memory, where the dereferencing would have to occur anyway. Registers are generally not used in the Franz Lisp implementation.

Frpoly

The *Frpoly* benchmark was designed to mimic computations in the Macsyma system. As a Lisp benchmark it would give an indication of Macsyma performance on a particular Lisp system. *Frpoly* symbolically expands the expression $(x+y+z+1)^n$ for $n = 5, 10, 15$, where intermediate expressions maintain a canonical form at certain steps of the computation.

The Lisp version uses some messy data-structure manipulations, drawn from the Macsyma polynomial-manipulation package. Destructive operations are used, presumably to save time and reduce the number of scratch list-cells used. These destructive operations did not translate to Prolog, so a simpler but "less efficient" scheme is used. The Lisp version uses a routine to explicitly put expressions into canonical form, as is done in Macsyma, while the Prolog version maintains the canonical form all along. Since the algorithmic structure is not the same for both cases we cannot really judge the languages by this benchmark.

Prover

The *Prover* benchmark from O'Keefe does not appear to favor or penalize either language. The program structures are virtually identical, and neither relies on any "unpopular" language features which may not be well-implemented. In O'Keefe's original paper the Prolog implementation tended to be a bit faster; in these tests the Lisp implementation tends to be faster.

Puzzle

The *Puzzle* benchmark finds ways of packing pieces in a 5x5x5 cube by trying all possible combinations. The Lisp version uses an array to represent the arrangements. Failed combinations are undone and retried with different choices.

The Prolog coding uses backtracking to undo failed combinations: in order to keep track of the number of combinations attempted, a global variable is maintained using the *set/access* or *assert/retract* operations. In the Quintus implementation a full assert/retract is used, which involves a substantial runtime overhead. The Prolog coding has a further disadvantage of using lists instead of arrays, which imply more sequential accesses to data.

4.1 A Note on the Code

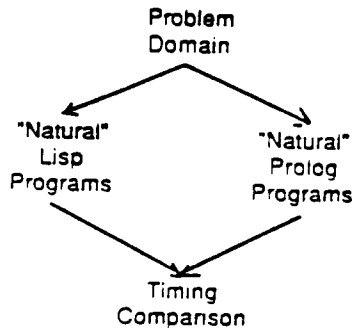
Looking at the output of the Lisp compiler shows that there is room for further improvement. The following sequence is from *apply-subst* in the inner-loop part of the *Boyer* benchmark:

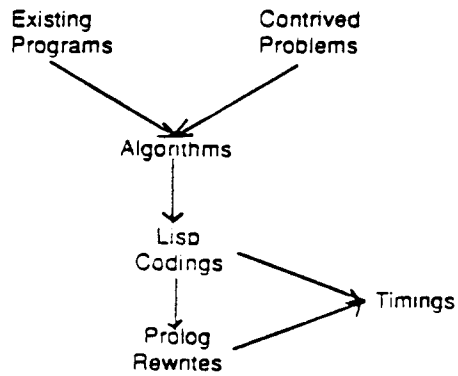
```
L18:
        subl2   $8,r6
        movl    r0,r0
        movl    28(r8),r1
        movl    4(r1),r1
        movl    r0,0(r1)
        jeql    L15
        movl    28(r8),r1
        movl    *4(r1),r0
        movl    r0,r0
        movl    0(r0),r0
        jbr     L14
```

The *movl r0,r0* operations are redundant, as well as the second *movl 28(r8),r1* operation: overall there is a lot of stacking and unstacking of variables, as registers are *not* used to transmit parameters in procedure-calls. Unfortunately the Prolog compiler does not produce symbolic output for us to examine. Very likely *both* compilers could use additional optimizations....

5 Interpretations

Ideally this paper would compare the usefulness, or at least the relative performance of Prolog and Lisp, as might be expressed by the diagram on the left. As outlined in section 2, there are a number of complicating issues which cannot be readily resolved. We are instead forced to make a more pragmatic treatment – evaluating a measure that is easy to quantify (VAX performance) on a domain of programs we can deal with (benchmark programs). This pragmatic comparison is better expressed by the diagram on the right:





The concepts on the right only crudely approximate the concepts on the left. Rather than an AI problem domain, we have algorithms scavenged from existing programs with a few contrived ones thrown in. Rather than natural Lisp and Prolog codings we have direct Lisp and Prolog rewrites of these algorithms. For those we have specific timings.

The natural program style have may have a significant bearing on the inherent usefulness of a language. Language X might efficiently express function F, but be unrepresentative because real programmers are unable to utilize X so effectively. The naturalness of a piece of code depends on the experience of the programmer writing (or critiquing) it, the aesthetic notions of programming style, etc. and is difficult to measure. The benchmarks in this study are small enough that there is little freedom of choice in rewriting them. Thus any distinctions in "natural" programming style between Lisp and Prolog would not be very pronounced in these tests.

Do these results have any bearing on the ideal comparison? We have already exposed issues involved in interpreting these results. If we read the results of our pragmatic comparison *ver batim* it appears that the Prolog and Lisp implementations are roughly competitive, with Lisp slightly favored. In the extreme cases, Prolog was 7x slower than Lisp (*Tak*), and Lisp was 2.8x slower than Prolog (*Browse*). Among the larger benchmarks, Prolog was only 2.5x slower than Lisp (*Prover*, case #4). Even among the results for *Prover* there was a substantial variation, with Lisp being equal to Prolog in case #8 and 1.7x slower if we include GC time.

Studies like this typically assume that the performance ratio will stay the same as implementations improve. Otherwise measurements made on interpreters would be uninteresting - if the interpreter for language X loses against the interpreter for language Y, we might conclude that Y was inherently faster than X. But we can rest assured that if we compile language X, it will run faster than the interpreter for language Y. So until we compile language Y as well, X will appear faster. What happens when we compile language Y? The comparison usually changes as the implementations improve. Extrapolating the performance ratio from one generation of implementations to another will not work, any more than does extrapolating the performance of one benchmark to a suite of benchmarks. A prime example is in comparing Gutierrez's [4] results to O'Keffe's [8], and O'Keefe's results to the ones here - clearly the performance ratio was not preserved!

Further improvement is likely to continue. Examination of the Franz Lisp object files shows room for further optimizations. There are a number of obvious, unexploited high-level optimization techniques available for both Lisp and Prolog. Already there are faster Prolog systems in the works.

As implementations get faster, more and more deficiencies in the system design will have to have been "ironed out" (i.e. poor garbage-collection mechanisms, inefficient register usage by the compiler, etc.). If there is such a thing as "inherent efficiency" of a programming language, we should approach this limit as the efficiency of existing implementations improves.

However, we need to distinguish between improvements which reflect a speedup for all or "most" programs, and those which only apply to a given benchmark or small set of benchmarks. Comparing results for the *Tak* benchmark, for example, it appears that Prolog could greatly benefit from compiletime determinacy analysis. But this is not so clear from the other benchmarks; *Tak* is unusually sensitive to determinacy optimization.

In fact we could make optimizations even more benchmark-specific by building a system with lookup-tables, which identify a given benchmark and print its results without actually executing it. The "fastest" system for executing a given benchmark would have to use precisely this approach! As we consider "increasingly efficient" systems we must keep this point in mind, whether our "increasingly efficient" implementations are evolving toward the "generally most efficient" system or merely one with a large lookup table. The successive improvements in the implementations should be made with generality in mind.

In my opinion, for sufficiently well-implemented Lisps and Prologs there will always be programs running faster in one than the other if translated in a "natural" way. This discrepancy can be patched one way or the other by rewriting the benchmark, modifying the system, or both, but the benchmark programs end up having a contrived look or the system will become optimized for highly

unlikely cases.

Acknowledgments

Prof. Alvin Despain prodded me to do this study, after I had done some initial measurements on *Nreverse* and *Tak*. Bill Bush, Peter Van Roy, Bruce Holmer, and Hervé Touati made suggestions along the way. Hervé Touati modified the *Browse* benchmark (originally converted by Tep Dobry), and profiled the executions of a number of the benchmarks. Rick McGeer converted the *Frpoly* benchmark. The *Boyer*, *Puzzle*, and *Tak* benchmarks are from Evan Tick, the *Prover* benchmark is from R.A. O'Keefe [8], and the *Nreverse* benchmark is from Warren's original thesis [15]. Franz Lisp versions of *Browse* and *Puzzle* were supplied by Charlie Cox.

References

[1] Dijkstra, E.W. *Selected Writings on Computing: A Personal Perspective.* Springer-Verlag, New York, N.Y. (1982).

[2] Dobry, T.P., Despain, A.M., Patt, Y.N. Performance Studies of a Prolog Machine Architecture. *12th International Symposium of Computer Architecture* (June 1985).

[3] Gabriel, R.P. *Performance Evaluation of Lisp Systems.* MIT Press, Cambridge, Mass. (1985).

[4] Gutierrez, C. Prolog Compared with Lisp. *ACM Symposium on Lisp and Functional Programming* (1982), 143-149.

[5] Hattori, F., Kushima, K., Wasano, T. A Comparison of Lisp, Prolog, and Ada Programming Productivity in AI Area. *Proc. 1985 Int. Conf. on Software Engineering*, 285-291.

[6] Heygood, R. *A Prolog Benchmark Suite for Aquarius.* UC Berkeley Computer Science Division report #89/503 (April 1989).

[7] McDermott, D. The Prolog Phenomenon. *SIGART Newsletter* 72 (July 1980), 16-20.

[8] O'Keefe, R.A. Prolog Compared with Lisp? *SIGPLAN Notices 18,*5 (May 1983), 46-56.

[9] Okuno, H.G. *The Report of The Third Lisp Contest and The First Prolog Contest.* NT&T Mushino Electrical Communication Laboratories (Sept. 13, 1985).

[10] Ponder, C., McGeer, P., Ng, A.P.C. Are Applicative Languages Inefficient? *SIGPLAN Notices 23,*6 (June 1988), 135-139.

[11] Ponder, C. Benchmark Semantics. *SIGPLAN Notices 23,*2 (Feb. 1988), 44-48.

[12] Tick, E. Memory Performance of Lisp and Prolog Programs. *Proc. Third Int. Conf. on Logic Programming* (E. Shapiro, ed.), Springer-Verlag Lecture Notes in Computer Science (1986), 642-649.

[13] Ungar, D.M. *The Design and Evaluation of a High-Performance Smalltalk System.* UC Berkeley Computer Science Division report #86/287 (Fall 1986).

[14] Warren, D.H.D., Pereira, L.M. Prolog - the Language and its Implementation Compared with Lisp. *SIGPLAN Notices 12,*8 (Aug. 1977), 109-115.

[15] Warren, D.H.D. *Implementing Prolog - Compiling Predicate Logic Programs (Vols. 1 & 2).* Dept. of Artificial Intelligence Research Reports 39 & 40, Edinburgh University (1977).

[16] Warren, D.H.D. *An Abstract Prolog Instruction Set.* Technical Note 309, AI Center, SRI International (1983).

[17] Wilk, P.F. *Prolog Benchmarking.* Dept. of Artificial Intelligence, Edinburgh University (Dec. 1983).

Appendix: the Benchmarks

1.a. Prolog *Nreverse*

```
%%                                                                    %%
%          Prolog version of "nreverse list30" benchmark,             %
%                   from Warren's thesis.                             %
%%                                                                    %%

list30([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30

nreverse([X|L0],L) :- nreverse(L0,L1), concatenate(L1,[X],L).
nreverse([],[]).

concatenate([X|L1],L2,[X|L3]) :- concatenate(L1,L2,L3).
concatenate([],L,L).

main :- statistics,
        list30(L30),
        nreverse(L30,X),
        statistics,
        write(X).
```

1.b. Lisp *Nreverse*

```
;;
;;        Warren benchmark "nreverse list30", for comparison
;;                of various LISPs and PROLOGs.
;;

(defun nreverse (l)
  (cond ((null l) nil)
        (t (concatenate (nreverse (cdr l))
                        (cons (car l) nil)))))

(defun concatenate (l1 l2)
  (cond ((null l1) l2)
        (t (cons (car l1) (concatenate (cdr l1) l2)))))


(defun test ()
  (prog (H I J K list30)
        (setq list30 '(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
                         22 23 24 25 26 27 28 29 30))
        (do ((I 0 (1+ I)))
            ((> I 100))
            (nreverse list30))))


(defun doit ()
  (prog (X Y)
        (setq X (ptime))
        (test)
        (setq Y (ptime))
        (return (cons (- (car Y) (car X))
                      (- (cdr Y) (cdr X))))))
```

2. Prolog *Tak*

```
%/* Copyright Herve' Touati, Aquarius Project, UC Berkeley */

main :- statistics(runtime,_),
        tak(18, 12, 6, X),
        statistics(runtime,[_,T]),
        write(T), nl,
        write(X), nl.

tak(X,Y,Z,A) :-
        X =< Y, !,
        Z = A.
tak(X,Y,Z,A) :-
        X1 is X - 1,
        tak(X1,Y,Z,A1),
        Y1 is Y - 1,
        tak(Y1,Z,X,A2),
        Z1 is Z - 1,
        tak(Z1,X,Y,A3),
        tak(A1,A2,A3,A).
```

3. Prolog *Boyer*

```
%/* Copyright Herve' Touati, Aquarius Project, UC Berkeley */

%-------------------------------------------------------------------------
%       Benchmark Program - Boyer
%       Lisp vs. Prolog Study
%
%       Copyright by Evan Tick
%       Date: November 12 1985
%
%-------------------------------------------------------------------------

main :- statistics,
        wff(X),tautology(X),
        statistics.

wff(implies(and(implies(X,Y),
                and(implies(Y,Z),
                    and(implies(Z,U),
                        implies(U,W)))),
            implies(X,W))) :-
        X = f(plus(plus(a,b),plus(c,zero))),
        Y = f(times(times(a,b),plus(c,d))),
        Z = f(reverse(append(append(a,b),[]))),
        U = equal(plus(a,b),difference(x,y)),
        W = lessp(remainder(a,b),member(a,length(b))).

tautology(Wff) :-
        write('rewriting...'),nl,
        rewrite(Wff,NewWff),
        write('proving...'),nl,
        tautology(NewWff,[],[]).

tautology(Wff,Tlist,Flist) :-
        (truep(Wff,Tlist) -> true
        ;falsep(Wff,Flist) -> fail
        ;Wff = if(If,Then,Else) ->
                (truep(If,Tlist) -> tautology(Then,Tlist,Flist)
                ;falsep(If,Flist) -> tautology(Else,Tlist,Flist)
                ;tautology(Then,[If|Tlist],Flist),        % both must hold
                 tautology(Else,Tlist,[If|Flist])
                )
        ),!.

rewrite(Atom,Atom) :-
        atomic(Atom),!.
rewrite(Old,New) :-
```

```
        functor(Old,F,N),
        functor(Mid,F,N),
        rewrite_args(N,Old,Mid),
        ( equal(Mid,Next),        % should be ->, but is compiler smart enough?
          rewrite(Next,New)        % to generate cut for ->?
        ; New=Mid
        ),!.

rewrite_args(0,_,_) :- !.
rewrite_args(N,Old,Mid) :-
        arg(N,Old,OldArg),
        arg(N,Mid,MidArg),
        rewrite(OldArg,MidArg),
        N1 is N-1,
        rewrite_args(N1,Old,Mid).

truep(t,_) :- !.
truep(Wff,Tlist) :- member(Wff,Tlist).

falsep(f,_) :- !.
falsep(Wff,Flist) :- member(Wff,Flist).

member(X,[X|_]) :- !.
member(X,[_|T]) :- member(X,T).


equal(  and(P,Q),                        % 106 rules
        if(P,if(Q,t,f),f)
        ).
equal(  append(append(X,Y),Z),
        append(X,append(Y,Z))
        ).
equal(  assignment(X,append(A,B)),
        if(assignedp(X,A),
           assignment(X,A),
           assignment(X,B))
        ).
equal(  assume_false(Var,Alist),
        cons(cons(Var,f),Alist)
        ).
equal(  assume_true(Var,Alist),
        cons(cons(Var,t),Alist)
        ).
equal(  boolean(X),
        or(equal(X,t),equal(X,f))
        ).
equal(  car(gopher(X)),
        if(listp(X),
        car(flatten(X)),
        zero)
        ).
```

```
equal(  compile(Form),
        reverse(codegen(optimize(Form),[]))
        ).
equal(  count_list(Z,sort_lp(X,Y)),
        plus(count_list(Z,X),
             count_list(Z,Y))
        ).
equal(  countps_(L,Pred),
        countps_loop(L,Pred,zero)
        ).
equal(  difference(A,B),
        C
        ) :- difference(A,B,C).
equal(  divides(X,Y),
        zerop(remainder(Y,X))
        ).
equal(  dsort(X),
        sort2(X)
        ).
equal(  eqp(X,Y),
        equal(fix(X),fix(Y))
        ).
equal(  equal(A,B),
        C
        ) :- eq(A,B,C).
equal(  even1(X),
        if(zerop(X),t,odd(decr(X)))
        ).
equal(  exec(append(X,Y),Pds,Envrn),
        exec(Y,exec(X,Pds,Envrn),Envrn)
        ).
equal(  exp(A,B),
        C
        ) :- exp(A,B,C).
equal(  fact_(I),
        fact_loop(I,1)
        ).
equal(  falsify(X),
        falsify1(normalize(X),[])
        ).
equal(  fix(X),
        if(numberp(X),X,zero)
        ).
equal(  flatten(cdr(gopher(X))),
        if(listp(X),
           cdr(flatten(X)),
           cons(zero,[]))
        ).
equal(  gcd(A,B),
        C
        ) :- gcd(A,B,C).
```

```
equal(  get(J,set(I,Val,Mem)),
        if(eqp(J,I),Val,get(J,Mem))
        ).
equal(  greatereqp(X,Y),
        not(lessp(X,Y))
        ).
equal(  greatereqpr(X,Y),
        not(lessp(X,Y))
        ).
equal(  greaterp(X,Y),
        lessp(Y,X)
        ).
equal(  if(if(A,B,C),D,E),
        if(A,if(B,D,E),if(C,D,E))
        ).
equal(  iff(X,Y),
        and(implies(X,Y),implies(Y,X))
        ).
equal(  implies(P,Q),
        if(P,if(Q,t,f),t)
        ).
equal(  last(append(A,B)),
        if(listp(B),
           last(B),
           if(listp(A),
              cons(car(last(A))),
              B))
        ).
equal(  length(A),
        B
        ) :- mylength(A,B).
equal(        lesseqp(X,Y),
        not(lessp(Y,X))
        ).
equal(  lessp(A,B),
        C
        ) :- lessp(A,B,C).
equal(  listp(gopher(X)),
        listp(X)
        ).
equal(  mc_flatten(X,Y),
        append(flatten(X),Y)
        ).
equal(  meaning(A,B),
        C
        ) :- meaning(A,B,C).
equal(  member(A,B),
        C
        ) :- mymember(A,B,C).
equal(  not(P),
        if(P,f,t)
```

```
        ).
equal(  nth(A,B),
        C
        ) :- nth(A,B,C).
equal(  numberp(greatest_factor(X,Y)),
        not(and(or(zerop(Y),equal(Y,1)),
                not(numberp(X))))
        ).
equal(  or(P,Q),
        if(P,t,if(Q,t,f),f)
        ).
equal(  plus(A,B),
        C
        ) :- plus(A,B,C).
equal(  power_eval(A,B),
        C
        ) :- power_eval(A,B,C).
equal(  prime(X),
        and(not(zerop(X)),
            and(not(equal(X,add1(zero))),
                prime1(X,decr(X))))
        ).
equal(  prime_list(append(X,Y)),
        and(prime_list(X),prime_list(Y))
        ).
equal(  quotient(A,B),
        C
        ) :- quotient(A,B,C).
equal(  remainder(A,B),
        C
        ) :- remainder(A,B,C).
equal(  reverse_(X),
        reverse_loop(X,□)
        ).
equal(  reverse(append(A,B)),
        append(reverse(B),reverse(A))
        ).
equal(  reverse_loop(A,B),
        C
        ) :- reverse_loop(A,B,C).
equal(  samefringe(X,Y),
        equal(flatten(X),flatten(Y))
        ).
equal(  sigma(zero,I),
        quotient(times(I,add1(I)),2)
        ).
equal(  sort2(delete(X,L)),
        delete(X,sort2(L))
        ).
equal(  tautology_checker(X),
        tautologyp(normalize(X),□)
```

```
        ).
equal(  times(A,B),
        C
        ) :- times(A,B,C).
equal(  times_list(append(X,Y)),
        times(times_list(X),times_list(Y))
        ).
equal(  value(normalize(X),A),
        value(X,A)
        ).
equal(  zerop(X),
        or(equal(X,zero),not(numberp(X)))
        ).

difference(X, X, zero) :- !.
difference(plus(X,Y), X, fix(Y)) :- !.
difference(plus(Y,X), X, fix(Y)) :- !.
difference(plus(X,Y), plus(X,Z), difference(Y,Z)) :- !.
difference(plus(B,plus(A,C)), A, plus(B,C)) :- !.
difference(add1(plus(Y,Z)), Z, add1(Y)) :- !.
difference(add1(add1(X)), 2, fix(X)).

eq(plus(A,B), zero, and(zerop(A),zerop(B))) :- !.
eq(plus(A,B), plus(A,C), equal(fix(B),fix(C))) :- !.
eq(zero, difference(X,Y),not(lessp(Y,X))) :- !.
eq(X, difference(X,Y),and(numberp(X),
                          and(or(equal(X,zero),
                                 zerop(Y))))) :- !.
eq(times(X,Y), zero, or(zerop(X),zerop(Y))) :- !.
eq(append(A,B), append(A,C), equal(B,C)) :- !.
eq(flatten(X), cons(Y,□), and(nlistp(X),equal(X,Y))) :- !.
eq(greatest_factor(X,Y),zero, and(or(zerop(Y),equal(Y,1)),
                                  equal(X,zero))) :- !.
eq(greatest_factor(X,_),1, equal(X,1)) :- !.
eq(Z, times(W,Z), and(numberp(Z),
                      or(equal(Z,zero),
                         equal(W,1)))) :- !.
eq(X, times(X,Y), or(equal(X,zero),
                     and(numberp(X),equal(Y,1)))) :- !.
eq(times(A,B), 1, and(not(equal(A,zero)),
                      and(not(equal(B,zero)),
                          and(numberp(A),
                              and(numberp(B),
                                  and(equal(decr(A),zero),
                                      equal(decr(B),zero)))))))
            ) :- !.
eq(difference(X,Y), difference(Z,Y),if(lessp(X,Y),
                                       not(lessp(Y,Z)),
                                       if(lessp(Z,Y),
                                          not(lessp(Y,X)),
                                          equal(fix(X),fix(Z))))) :- !.
```

```
eq(lessp(X,Y), Z, if(lessp(X,Y),
                     equal(t,Z),
                     equal(f,Z))).

exp(I, plus(J,K), times(exp(I,J),exp(I,K))) :- !.
exp(I, times(J,K), exp(exp(I,J),K)).

gcd(X, Y, gcd(Y,X)) :- !.
gcd(times(X,Z), times(Y,Z), times(Z,gcd(X,Y))).

mylength(reverse(X),length(X)).
mylength(cons(_,cons(_,cons(_,cons(_,cons(_,cons(_,X7)))))),
       plus(6,length(X7))).

lessp(remainder(_,Y), Y, not(zerop(Y))) :- !.
lessp(quotient(I,J), I, and(not(zerop(I)),
                            or(zerop(J),
                               not(equal(J,1))))) :- !.
lessp(remainder(X,Y), X, and(not(zerop(Y)),
                             and(not(zerop(X)),
                                 not(lessp(X,Y))))) :- !.
lessp(plus(X,Y), plus(X,Z), lessp(Y,Z)) :- !.
lessp(times(X,Z), times(Y,Z), and(not(zerop(Z)),
                                  lessp(X,Y))) :- !.
lessp(Y, plus(X,Y), not(zerop(X))) :- !.
lessp(length(delete(X,L)), length(L), member(X,L)).

meaning(plus_tree(append(X,Y)),A,
       plus(meaning(plus_tree(X),A),
            meaning(plus_tree(Y),A))
       ) :- !.
meaning(plus_tree(plus_fringe(X)),A,
       fix(meaning(X,A))
       ) :- !.
meaning(plus_tree(delete(X,Y)),A,
       if(member(X,Y),
          difference(meaning(plus_tree(Y),A),
                     meaning(X,A)),
          meaning(plus_tree(Y),A))).

mymember(X,append(A,B),or(member(X,A),member(X,B))) :- !.
mymember(X,reverse(Y),member(X,Y)) :- !.
mymember(A,intersect(B,C),and(member(A,B),member(A,C))).

nth(zero,_,zero).
nth([],I,if(zerop(I),[],zero)).
nth(append(A,B),I,append(nth(A,I),nth(B,difference(I,length(A))))).

plus(plus(X,Y),Z,
     plus(X,plus(Y,Z))
       ) :- !.
```

```
plus(remainder(X,Y),
     times(Y,quotient(X,Y)),
     fix(X)
        ) :- !.
plus(X,add1(Y),
     if(numberp(Y),
        add1(plus(X,Y)),
        add1(X))
        ).

power_eval(big_plus1(L,I,Base),Base,
           plus(power_eval(L,Base),I)
          ) :- !.
power_eval(power_rep(I,Base),Base,
           fix(I)
          ) :- !.
power_eval(big_plus(X,Y,I,Base),Base,
           plus(I,plus(power_eval(X,Base),
                       power_eval(Y,Base)))
          ) :- !.
power_eval(big_plus(power_rep(I,Base),
                    power_rep(J,Base),
                    zero,
                    Base),
           Base,
           plus(I,J)
          ).

quotient(plus(X,plus(X,Y)),2,plus(X,quotient(Y,2))).
quotient(times(Y,X),Y,if(zerop(Y),zero,fix(X))).

remainder(_,                1,zero) :- !.
remainder(X,                X,zero) :- !.
remainder(times(_,Z),       Z,zero) :- !.
remainder(times(Y,_),       Y,zero).

reverse_loop(X,Y,         append(reverse(X),Y)      ) :- !.
reverse_loop(X,□,         reverse(X)                ).

times(X,         plus(Y,Z),         plus(times(X,Y),times(X,Z))        ) :- !.
times(times(X,Y),Z,                 times(X,times(Y,Z))                ) :- !.
times(X,         difference(C,W),   difference(times(C,X),times(W,X))  ) :- !.
times(X,         add1(Y),           if(numberp(Y),
                                       plus(X,times(X,Y)),
                                       fix(X))                          ).
```

4. Prolog *Browse*

```
%/* Copyright Herve' Touati, Aquarius Project, UC Berkeley */

%/* obtained from Tep Dobry */
%/* modified by Herve' Touati 01/15/87 */

main :- statistics,
        init(100,10,4,
                [[a,a,a,b,b,b,b,a,a,a,a,a,b,b,a,a,a],
                 [a,a,b,b,b,b,a,a,[a,a],[b,b]],
                 [a,a,a,b,[b,a],b,a,b,a]
                ],
                Symbols),
        randomize(Symbols,RSymbols,21),!,
        investigate(RSymbols,
                [[star(SA),B,star(SB),B,a,star(SA),a,star(SB),star(SA)],
                 [star(SA),star(SB),star(SB),star(SA),[star(SA)],[star(SB)]],
                 [_,_,star(_),[b,a],star(_),_,_]
                ]),
        statistics.

init(N,M,Npats,Ipats,Result) :- init(N,M,M,Npats,Ipats,Result).

init(0,_,_,_,_,_) :- !.
init(N,I,M,Npats,Ipats,[Symb|Rest]) :-
        fill(I,□,L),
        get_pats(Npats,Ipats,Ppats),
        J is M - I,
        fill(J,[pattern(Ppats)|L],Symb),
        N1 is N - 1,
        (I == 0 -> I1 is M; I1 is I - 1),
        init(N1,I1,M,Npats,Ipats,Rest).

fill(0,L,L) :- !.
fill(N,L,[dummy(□)|Rest]) :- N1 is N - 1, fill(N1,L,Rest).


randomize(□,□,_) :- !.
randomize(In,[X|Out],Rand) :-
        length(In,Lin),
        Rand1 is (Rand * 17) mod 251,
        N is Rand1 mod Lin,
        split(N,In,X,In1),
        randomize(In1,Out,Rand1).

split(0,[X|Xs],X,Xs) :- !.
split(N,[X|Xs],RemovedElt,[X|Ys]) :-
```

```
        N1 is N - 1,
        split(N1,Xs,RemovedElt,Ys).


investigate([],_).
investigate([U|Units],Patterns) :-
        property(U,pattern,Data),
        p_investigate(Data,Patterns),
        investigate(Units,Patterns).


get_pats(Npats,Ipats,Result) :- get_pats(Npats,Ipats,Result,Ipats).

get_pats(0,_,[],_) :- !.
get_pats(N,[X|Xs],[X|Ys],Ipats) :-
        N1 is N - 1,
        get_pats(N1,Xs,Ys,Ipats).
get_pats(N,[],Ys,Ipats) :-
        get_pats(N,Ipats,Ys,Ipats).

property([],_,_) :- fail.  /* don't really need this */
property([Prop|RProps],P,Val) :-
        functor(Prop,P,_),!,
        arg(1,Prop,Val).
property([_|RProps],P,Val) :-
        property(RProps,P,Val).

p_investigate([],_).
p_investigate([D|Data],Patterns) :-
        p_match(Patterns,D),
        p_investigate(Data,Patterns).

p_match([],_).
p_match([P|Patterns],D) :-
        (match(D,P), fail; true),
        p_match(Patterns, D).

match([],[]) :- !.
match([X|PRest],[Y|SRest]) :-
        var(Y),!,X = Y,
        match(PRest,SRest).
match(List,[Y|Rest]) :-
        nonvar(Y),Y = star(X),!,
        concat(X,SRest,List),
        match(SRest,Rest).
match([X|PRest],[Y|SRest]) :-
        (atom(X) -> X = Y; match(X,Y)),
        match(PRest,SRest).

concat([],L,L).
concat([X|L1],L2,[X|L3]) :- concat(L1,L2,L3).
```

5. Prolog *Frpoly*

```
%%
%
%       "FRPOLY" symbolic polynomial powering algorithm.
%       Converted from Lisp (Gabriel Suite) by Rick McGeer.
%
%       To run, say
%                       ?-setup.
%                       ?-bench(5).
%                       ?-bench(10).
%                       ?-bench(15).
%
%%

main :- statistics(runtime,_),
        setup,
        statistics(runtime,[_,A]),
        bench(5),
        statistics(runtime,[_,B]),
        bench(10),
        statistics(runtime,[_,C]),
        bench(15),
        statistics(runtime,[_,D]),
        write(A), nl,
        write(B), nl,
        write(C), nl,
        write(D), nl.



% Polynomial addition

poly_add( poly(Var, Terms1), poly(Var, Terms2), poly(Var, Terms3) ) :-
    !,
    add_terms(Terms1, Terms2, Terms3).


poly_add( poly(Var1, Terms1), poly(Var2, Terms2), poly(Var1, Terms3) ) :-
    Var2 @> Var1,
    !,
    add_To_Zero_Term(Terms1, poly(Var2, Terms2), Terms3 ).

poly_add( poly(Var1, Terms1), poly(Var2, Terms2), poly(Var2, Terms3) ) :-
    Var1 @> Var2,
    !,
    add_To_Zero_Term(Terms2, poly(Var1, Terms1), Terms3 ).
```

```
poly_add( poly(Var1, Terms1), N, poly(Var1, Terms3)) :-
    !,
    add_To_Zero_Term(Terms1, N, Terms3 ).


poly_add( N, poly(Var2, Terms2), poly(Var2, Terms3)) :-
    !,
    add_To_Zero_Term(Terms2, N, Terms3 ).


% Straight addition of numbers

poly_add(N, M, T) :-
    T is N + M.

% adding Terms

add_terms(□,X,X) :- !.
add_terms(X,□,X) :- !.

add_terms([term(Exp,C1)|Terms1],[term(Exp,C2)|Terms2],[term(Exp,C)|Terms]) :-
    !,
    poly_add(C1, C2, C),
    add_terms(Terms1, Terms2, Terms).

add_terms([term(E1,C1)|Terms1],[term(E2,C2)|Terms2],[term(E1,C1)|Terms]) :-
    E1 < E2,
    !,
    add_terms(Terms1,[term(E2,C2)|Terms2],Terms).

add_terms(Terms1,[term(E2,C2)|Terms2],[term(E2,C2)|Terms]) :-
    add_terms(Terms1, Terms2, Terms).

add_To_Zero_Term([term(0,C1)|Terms],C2,[term(0,C)|Terms]) :-
    !,
    poly_add(C1, C2, C).

add_To_Zero_Term(Terms,C,[term(0,C)|Terms]).

% Polynomial Multiplication

poly_mult(poly(Var, Terms1), poly(Var, Terms2), poly(Var, Terms3)) :-
    form_poly_product(Terms1, Terms2, Terms3).

poly_mult( poly(Var1, Terms1), poly(Var2, Terms2), poly(Var1, Terms3) ) :-
    Var2 @> Var1,
    !,
    multiply_through(Terms1, poly(Var2, Terms2), Terms3 ).

poly_mult( Poly1, poly( Var2, Terms2), poly(Var2, Terms3) ) :-
    !,
```

```
    multiply_through(Terms2, Poly1, Terms3 ).

poly_mult( poly( Var2, Terms2), Poly1, poly(Var2, Terms3) ) :-
    !,
    multiply_through(Terms2, Poly1, Terms3 ).

poly_mult(C1, C2, C) :-
    C is C1 * C2.

multiply_through([], _, []) :- !.

multiply_through([term(N,T1)|Terms], Poly, [term(N,NewT1)|NewTerms]) :-
    poly_mult(T1, Poly, NewT1),
    multiply_through(Terms, Poly, NewTerms).

form_poly_product([],_,[]) :- !.

form_poly_product(_,[],[]) :- !.

form_poly_product([T1|Terms], Terms2, Terms3) :-
    form_single_product(Terms2, T1, Ta),
    form_poly_product(Terms, Terms2, Tb),
    add_terms(Ta, Tb, Terms3).

form_single_product([],_,[]) :- !.

form_single_product([term(Exp1,C1)|Terms], term(Exp2,C2), [term(Exp,C)|Products]) :-
    Exp is Exp1 + Exp2,
    poly_mult(C1, C2, C),
    form_single_product(Terms, term(Exp2,C2), Products).

% Polynomial Exponentiation

poly_expt(0, _, 1) :- !.

poly_expt(N, P, Result) :-
    evenP(N),
    !,
    M is N // 2,
    poly_expt(M, P, NextRes),
    poly_mult(NextRes, NextRes, Result).

poly_expt(N, P, Result) :-
    M is N - 1,
    poly_expt(M, P, NextRes),
    poly_mult(P, NextRes, Result).


%poly_expt(N, P, Result) :-
%   poly_expt( N, P, 1, Result).
```

```
%poly_expt(0, _, Result, Result) :- !.
%
%poly_expt(N, P, ResSoFar, Result) :-
%    evenP(N),
%    !,
%    M is N // 2,
%    poly_mult(ResSoFar, ResSoFar, NextRes),
%    poly_expt(M, P, NextRes, Result).
%
%poly_expt(N, P, ResSoFar, Result) :-
%    M is N - 1,
%    poly_mult(P, ResSoFar, NextRes),
%    poly_expt(M, P, NextRes, Result).
%

setup :-
    poly_add(poly(y,[term(0,1),term(1,1)]),poly(z,[term(1,1)]),Tmp),
    poly_add(poly(x,[term(1,1)]),Tmp,R),
    assert(test_case(R)).

evenP(X) :-
    N is X // 2,
    X is N * 2.



print_poly(poly(Var, Terms)) :-
    !,
    print_Terms(Terms, Var).

print_poly(X) :-
    write(X).


print_Terms([],_) :- !.

print_Terms([term(_, 0)|Terms],Var) :-
    !,
    print_Terms(Terms, Var).

print_Terms([Term],Var) :-
    !,
    print_Term(Term, Var).

print_Terms([Term|Terms],Var) :-
    print_Term(Term, Var),
    write(' + '),
    print_Terms(Terms, Var).

print_Term(term(0, P), _) :-
    !,
```

```
    print_poly(P).

print_Term(term(1, C), Var) :-
    !,
    print_Coeff(C),
    write(Var).

print_Term(term(Exp,C), Var) :-
    print_Coeff(C),
    write(Var),
    write('^'),
    write(Exp).

print_Coeff(1) :- !.

print_Coeff(N) :-
    atomic(N),
    !,
    write(N),
    write('*').

print_Coeff(P) :-
    !,
    write('('),
    print_poly(P),
    write(')'),
    write('*').

bench(N) :-
    test_case(X),
    poly_expt(N, X, Y).
```

6.a. Prolog *Prover*

```
%%                                                                    %%
%            Prolog theorem prover to be compared against Lisp        %
%              (R.A. O'Keefe - "Prolog Compared with Lisp?")          %
%                                                                     %
%              (from Sigplan Notices, vol 18 #5, May 1983)            %
%%                                                                    %%

%%%%%%%%                                                              %%%%%%%%
%%%%%%%%       Also to be run with suppressed output, for            %%%%%%%%
%%%%%%%%                computation time only                        %%%%%%%%
%%%%%%%%                                                            %%%%%%%%

main :- statistics(runtime,_),
        timed(10,10),
        statistics(runtime,[_,A]),
        timed(10,9),
        statistics(runtime,[_,B]),
        timed(10,8),
        statistics(runtime,[_,C]),
        timed(10,7),
        statistics(runtime,[_,D]),
        timed(10,6),
        statistics(runtime,[_,E]),
        timed(10,5),
        statistics(runtime,[_,F]),
        timed(10,4),
        statistics(runtime,[_,G]),
        timed(10,3),
        statistics(runtime,[_,H]),
        timed(10,2),
        statistics(runtime,[_,I]),
        timed(10,1),
        statistics(runtime,[_,J]),
        write(A), nl,
        write(B), nl,
        write(C), nl,
        write(D), nl,
        write(E), nl,
        write(F), nl,
        write(G), nl,
        write(H), nl,
        write(I), nl,
        write(J), nl.

:- public
        go/1,           % quick test using stored problems
```

```
        implies/2,      % the prover proper
        timed/2.        % for getting CPU times

:- mode
        add_conjunction(+,+,+),
        expand(+,+,-),
        extend(+,+,+,-,+,-),
        go(+),
        implies(+,+),
        includes(+,+),
        opposite(+,-),
        problem(+,-,-),
        refute(+),
        try(+),
        timed(+,+).

:- op(950, xfy, #).        % disjunction
:- op(850, xfy, &).        % conjunction
:- op(500, fx, +).         % assertion
:- op(500, fx, -).         % denial

implies(Premise, Conclusion) :-
        write('Trying to prove that '), write(Premise),
        write(' implies '), write(Conclusion), nl,
        opposite(Conclusion, Denial), !,
        add_conjunction(Premise, Denial, fs(□,□,□,□)).

opposite(F0 & G0, F1 # G1) :-
        opposite(F0, F1), !,
        opposite(G0, G1).
opposite(F1 # G1, F0 & G0) :-
        opposite(F1, F0), !,
        opposite(G1, G0).
opposite(+Atom, -Atom).
opposite(-Atom, +Atom).

add_conjunction(F, G, Set):-
        write('Expanding conjunction '), write(F & G),
        write(' by Rule 1'), nl,
        expand(F, Set, Mid),
        expand(G, Mid, New), !,
        refute(New).

expand(Formula, refuted, refuted).
expand(F & G, fs(D,C,P,N), refuted) :- includes(D, F & G), !.
expand(F & G, fs(D,C,P,N), fs(D,C,P,N)) :- includes(C, F & G), !.
expand(F & G, fs(D,C,P,N), New) :-
        expand(F, fs(D,[F&G|C],P,N), Mid), !,
        expand(G, Mid, New).
expand(F # G, fs(D,C,P,N), Set) :-
        opposite(F # G, Conj), !,
```

```
        extend(Conj, D, C, D1, fs(D1,C,P,N), Set).
expand(+Atom, fs(D,C,P,N), Set) :- !,
        extend(Atom, P, N, P1, fs(D,C,P1,N), Set).
expand(-Atom, fs(D,C,P,N), Set) :- !,
        extend(Atom, N, P, N1, fs(D,C,P,N1), Set).

includes([Head|Tail], Head) :- !.
includes([Head|Tail], This) :- includes(Tail, This).

extend(Exp, Pos, Neg, New, Set, refuted) :- includes(Neg, Exp), !.
extend(Exp, Pos, Neg, Pos, Set, Set) :- includes(Pos, Exp), !.
extend(Exp, Pos, Neg, [Exp|Pos], Set, Set).

refute(refuted) :- write('Contradiction spotted (Rule 3).'), nl.
refute(fs([F1 & G1|D], C, P, N)) :-
        opposite(F1, F0),
        opposite(G1, G0),
        Set = fs(D, C, P, N),
        write('Case analysis on '), write(F0 # G0),
        write(' using Rule 2'), nl,
        add_conjunction(F0, G1, Set),
        add_conjunction(F0, G0, Set),
        add_conjunction(F1, G0, Set).
refute(Set) :-
        write('Can''t refute '), write(Set), nl,
        fail.


problem( 1, -a, +a).

problem( 2, +a, -a & -a).

problem( 3, -a, +to_be # -to_be).

problem( 4, -a & -a, -a).

problem( 5, -a, +b # -a).

problem( 6, -a & -b, -b & -a).

problem( 7, -a, -b # (+b & -a)).

problem( 8, -a # (-b # +c), -b # (-a # +c)).

problem( 9, -a # +b, (+b & -c) # (-a # +c)).

problem( 10, (-a # +c) & (-b # +c), (-a & -b) # +c).


try(N) :- problem(N, P, C), !,
          implies(P, C).
```

```
timed(0, _) :- !.
timed(K, N) :- (try(N); true), J is K-1, !, timed(J,N).
```

6.b. Lisp *Prover*

```
;;                                                                    ;;
;                Lisp theorem prover to be compared against Prolog     ;
;                  (R.A. O'Keefe - "Prolog Compared with Lisp?")       ;
;                                                                      ;
;                  (from Sigplan Notices, vol 18 #5, May 1983)         ;
;                             <FranzLisp Version>                      ;
;                                <corrected>                           ;
;;                                                                    ;;

;;;;;;;;                                                    ;;;;;;;;
;;;;;;;;          Also to be run with suppressed output, for ;;;;;;;;
;;;;;;;;                   computation time only            ;;;;;;;;
;;;;;;;;                                                    ;;;;;;;;

(defmacro time (z)
   `(prog (X Y)
          (setq X (ptime))
          ,z
          (setq Y (ptime))
          (princ (cons (- (car Y) (car X))
                       (- (cadr Y) (cadr X))))
          (terpri)))


(declare
        (special Cases D C P N Refuted))

(defun implies (Premise Conclusion)
        (princ (list "Trying to prove that" Premise "implies" Conclusion))
        (terpri)
        (add-conjunction Premise (opposite Conclusion) nil nil nil nil)
)

(defun opposite (F)
        (prog (O)
                (setq O (car F))
                (return (cond ((eq O '\&)
                               (cons '\# (cons (opposite (cadr F))
                                               (opposite (cddr F)) )))
                              ((eq O '\#)
                               (cons '\& (cons (opposite (cadr F))
                                               (opposite (cddr F)) )))
                              ((eq O '+)
                               (cons '- (cdr F)))
                              ((eq O '-)
                               (cons '+ (cdr F)))
```

```
        ))
))

(defun add-conjunction (F G D C P N)
        (prog (Refuted)
                (princ (list "Expanding conjunction" (cons '\& (cons F G))
                                                        "by Rule 1")) (terpri)
                (setq Refuted nil)
                (expand F)
                (expand G)
                (cond (Refuted
                        (princ "Contradiction spotted")(terpri)
                                (return t))
                      ((not (atom D))
                       (return (split (cadar D) (cddar D) (cdr D)) ))
                      (t
                       (princ (list "Can't refute" D C P N))(terpri)
                       (return nil))
                )
))

(defun split (F1 G1 D)
        (prog (F G)
                (princ (list "Case analysis on" (cons '\# (cons F1 G1))
                                                "(Rule 2)"))(terpri)
                (setq F (opposite F1))
                (setq G (opposite G1))
                (return (and (add-conjunction F G1 D C P N)
                             (add-conjunction F  G D C P N)
                             (add-conjunction F1 G D C P N)
                ))
))

(defun expand (F)
        (prog (O)
                (setq O (car F))
                (cond ((eq O '\&)
                        (cond ((member F D) (setq Refuted t))
                              ((member F C) nil)
                              (t (setq C (cons F C))
                                 (expand (cadr F))
                                 (expand (cddr F)) )) )
                      ((eq O '\#) (setq D (extend (opposite F) D C)) )
                      ((eq O '+) (setq P (extend (cdr F) P N)) )
                      ((eq O '-) (setq N (extend (cdr F) N P)) )
                )
))

(defun extend (F A B)
        (cond ((member F B) (setq Refuted t) A)
              ((member F A) A)
```

```
            (t (cons F A))
        )
)

(defun try (M)
  (implies (vref Cases (* 2  M))
           (vref Cases (1+ (* 2 M)))))

(defun setup ()
   (setq Cases (vector
                        '(- . a)                '(+ . a)
                        '(+ . b)                '(\& (- . b) . (- . b))
                        '(- . nothing)          '(\# (+ . to-be) . (- . to-be))
                        '(\& (- . a) (- . a))   '(- . a)
                        '(- . b)                '(\# (+ . a) . (- . b))
                        '(\& (- . a) . (- . b)) '(\& (- . b) . (- . a))
                        '(- . b)
                                        '(\# (- . a) . (\& (+ . a) . (- . b)))
                        '(\# (- . a) . (\# (- . b) . (+ . c)))
                                        '(\# (- . b) . (\# (- . a) . (+ . c)))
                        '(\# (- . a) . (+ . b))
                                        '(\# (\& (+ . b) . (- . c)) .
                                             (\# (- . a) . (+ . c)))
                        '(\& (\# (- . a) . (+ . c)) . (\# (- . b) . (+ . c)))
                                        '(\# (\& (- . a) . (- . b)) . (+ . c))
                ) ))


(defun timed (K M)
  (prog ()
L       (try M)
        (cond ((greaterp (setq K (sub1 K)) 0) (go L)))
))



(defun doit ()
   (setup)
   (time (timed 10 9))
   (time (timed 10 8))
   (time (timed 10 7))
   (time (timed 10 6))
   (time (timed 10 5))
   (time (timed 10 4))
   (time (timed 10 3))
   (time (timed 10 2))
   (time (timed 10 1))
   (time (timed 10 0)))
```

7. Prolog *Puzzle*

```
/* Copyright Herve' Touati, Aquarius Project, UC Berkeley */

main :- statistics(runtime,_),
        make_board(Board),
        initialize(Board, Pieces),
        statistics(runtime,[_,A]),
        play(Board, Pieces, Board),
        statistics(runtime,[_,B]),
        write(A), nl,
        write(B), nl.

initialize([Spot|_],[[b,c,d,e,f,g,h,i,j,k,l,m],[n,o,p],[q],[r]]) :-
        set(0,0),
        p1(a,Spot).

play([],_,Board) :-
        access(0,N),
        write('Success in '),
        write(N),
        write(' trials.'), nl.
play([s(V,_,_,_)|Rest],Pieces,Board) :-
        nonvar(V), !,
        play(Rest,Pieces,Board).
play([Spot|Rest],Pieces,Board) :-
        fill(Spot,Pieces,NewPieces),
        incr,
        play(Rest,NewPieces,Board).

incr :-
        access(0, Count),
        NCount is Count + 1,
%       write(Count), nl,
        set(0, NCount).

fill(Spot,[[Mark|P1]|T],[P1|T]) :- p1(Mark,Spot).
fill(Spot,[P1,[Mark|P2]|T],[P1,P2|T]) :- p2(Mark,Spot).
fill(Spot,[P1,P2,[Mark|P3]|T],[P1,P2,P3|T]) :- p3(Mark,Spot).
fill(Spot,[P1,P2,P3,[Mark|P4]|T],[P1,P2,P3,P4|T]) :- p4(Mark,Spot).

% 4-2-1
p1(M,s(M,s(M,s(M,s(M,_,C13,_),C12,_),C11,_),s(M,C11,_,_),_)) :-
        C13 = s(M,_,_,_),
        C12 = s(M,C13,_,_),
        C11 = s(M,C12,_,_).
% 2-1-4
p1(M,s(M,s(M,_,_,C11),_,s(M,C11,_,s(M,C12,_,s(M,C13,_,_))))) :-
```

```
        C13 = s(M,_,_,_),
        C12 = s(M,_,_,C13),
        C11 = s(M,_,_,C12).
%1-4-2
p1(M,s(M,_,s(M,_,s(M,_,s(M,_,_,C13),C12),C11),s(M,_,C11,_))) :-
        C13 = s(M,_,_,_),
        C12 = s(M,_,C13,_),
        C11 = s(M,_,C12,_).
% 2-4-1
p1(M,s(M,s(M,_,C11,_),s(M,C11,s(M,C12,s(M,C13,_,_),_),_),_)) :-
        C13 = s(M,_,_,_),
        C12 = s(M,_,C13,_),
        C11 = s(M,_,C12,_).
% 4-1-2
p1(M,s(M,s(M,s(M,s(M,_,_,C13),_,C12),_,C11),_,s(M,C11,_,_))) :-
        C13 = s(M,_,_,_),
        C12 = s(M,C13,_,_),
        C11 = s(M,C12,_,_).
%1-2-4
p1(M,s(M,_,s(M,_,_,C11),s(M,_,C11,s(M,_,C12,s(M,_,C13,_))))) :-
        C13 = s(M,_,_,_),
        C12 = s(M,_,_,C13),
        C11 = s(M,_,_,C12).

p2(M,s(M,s(M,s(M,_,_,_),_,_),_,_)).
p2(M,s(M,_,s(M,_,s(M,_,_,_),_),_)).
p2(M,s(M,_,_,s(M,_,_,s(M,_,_,_)))).

p3(M,s(M,s(M,_,C,_),s(M,C,_,_),_)) :-
        C = s(M,_,_,_).
p3(M,s(M,s(M,_,_,C),_,s(M,C,_,_))) :-
        C = s(M,_,_,_).
p3(M,s(M,_,s(M,_,_,C),s(M,_,C,_))) :-
        C = s(M,_,_,_).

p4(M,s(M,s(M,_,C110,C101),s(M,C110,_,s(M,C111,_,_)),s(M,C101,C011,_))) :-
        C110 = s(M,_,_,C111),
        C101 = s(M,_,C111,_),
        C011 = s(M,C111,_,_),
        C111 = s(M,_,_,_).

make_board(Level0) :-
        make_level(Level0-Level1,Level1-_),
        make_level(Level1-Level2,Level2-_),
        make_level(Level2-Level3,Level3-_),
        make_level(Level3-Level4,Level4-_),
        make_level(Level4-[],X-[]),
        X = [z,z,z,z,z, z,z,z,z,z, z,z,z,z,z, z,z,z,z,z, z,z,z,z,z].

make_level(C-Link,Z-L) :-
        C = [C00,C10,C20,C30,C40,
```

```
            C01,C11,C21,C31,C41,
            C02,C12,C22,C32,C42,
            C03,C13,C23,C33,C43,
            C04,C14,C24,C34,C44|Link],
    Z = [Z00,Z10,Z20,Z30,Z40,
            Z01,Z11,Z21,Z31,Z41,
            Z02,Z12,Z22,Z32,Z42,
            Z03,Z13,Z23,Z33,Z43,
            Z04,Z14,Z24,Z34,Z44|L],

    C00 = s(_,C10,C01,Z00),
    C10 = s(_,C20,C11,Z10),
    C20 = s(_,C30,C21,Z20),
    C30 = s(_,C40,C31,Z30),
    C40 = s(_,  z,C41,Z40),

    C01 = s(_,C11,C02,Z01),
    C11 = s(_,C21,C12,Z11),
    C21 = s(_,C31,C22,Z21),
    C31 = s(_,C41,C32,Z31),
    C41 = s(_,  z,C42,Z41),

    C02 = s(_,C12,C03,Z02),
    C12 = s(_,C22,C13,Z12),
    C22 = s(_,C32,C23,Z22),
    C32 = s(_,C42,C33,Z32),
    C42 = s(_,  z,C43,Z42),

    C03 = s(_,C13,C04,Z03),
    C13 = s(_,C23,C14,Z13),
    C23 = s(_,C33,C24,Z23),
    C33 = s(_,C43,C34,Z33),
    C43 = s(_,  z,C44,Z43),

    C04 = s(_,C14,  z,Z04),
    C14 = s(_,C24,  z,Z14),
    C24 = s(_,C34,  z,Z24),
    C34 = s(_,C44,  z,Z34),
    C44 = s(_,  z,  z,Z44).


% set and access for systems that don't support them

:- dynamic '$set'/2.
set(N, A) :- (retract('$set'(N, _)); true), assert('$set'(N, A)), !.
access(N, A) :- '$set'(N,A), !.
```