# Formal Program Testing

Robert Cartwright
Rice University

**Abstract**

This paper proposes a practical alternative to program verification -- called formal program testing -- with similar, but less ambitious goals. Like a program verifier, a formal testing system takes a program annotated with formal specifications as input, generates the corresponding verification conditions, and passes them through a simplifier. After the simplification step, however, a formal testing system simply evaluates the verification conditions on a representative set of test data instead of trying to prove them. Formal testing provides strong evidence that a program is correct, but does not guarantee it. The strength of the evidence depends on the adequacy of the test data.

## 1. Introduction

After more than a decade of vigorous research on program verification, it is still not feasible to prove the correctness of "production" programs. Moreover, program verification has not won widespread acceptance among programming professionals as a potential programming tool. Given the technical limitations of existing verification systems and the indifference of programmers to the concept, practical program verification seems little closer to reality than it did a decade ago.

There are two fundamental reasons for the discouraging results of program verification research. First, proving that a computer program is correct ultimately reduces to proving a set of formal theorems (called verification conditions) in a first order theory of the program data domain -- an exceedingly tedious, yet demanding intellectual activity that we do not yet understand how to automate. Although most verification systems wisely rely on the programmer to guide the generation of proofs, we still do not how to identify and reliably mechanize the "routine" arguments that form the bulk of nearly all formal proofs. Consequently, the programmer must continually delve into the tedious details of symbolically complex, but conceptually simple sub-proofs and manually complete them. As a result, verifying non-trivial programs is an arduous process. For production programs, the burden is overwhelming.

The second reason for the disappointing performance of program verification systems is lack of expressiveness in the specification (assertion) languages typically used to describe program behavior. Current systems rely on specification languages consisting of Boolean program expressions augmented by free variables, quantifiers, and occasional ad hoc extensions.[1] In essence, the programming language is being used as its own specification language. While this approach works well for some experimental, very high level languages (e.g. SETL [Schwartz 79], pure LISP dialects [Boyer and Moore 75,79; Cartwright 76ab]), it does not make sense in the context of

This research has been supported in part by National Science Foundation grant MCS 78-05850.



[1] Some systems such as the Stanford PASCAL Verifier allow the programmer to introduce new predicates by axiomatizing them. See [von Henke and Luckham 74, Luckham and Suzuki 79].

conventional programming languages (e.g. PASCAL, C, PL/I, FORTRAN). The data objects and operations supported by conventional languages are too machine-oriented to describe computations simply and abstractly. As a result, conventional language specifications are often more difficult to read and to understand than the programs they purport to document. In these cases, the practical value of program verification is dubious; it merely establishes that the programs satisfy incomprehensible specifications, not that they are intuitively correct.

In this paper, we develop a practical alternative to program verification -- called formal program testing -- with similar, but less ambitious goals. Like a program verification system, a formal testing system takes a program annotated with formal specifications (invariant assertions) as input, generates the corresponding verification conditions, and passes them through a simplifier. A fast simplifier incorporating sophisticated, computationally efficient decision procedures such as the Stanford PASCAL Simplifier [Nelson and Oppen 79] will reduce many verification conditions to **true**. After the simplification step, formal testing systems and formal verification systems follow divergent paths. Instead of trying to prove the remaining verification conditions, a formal testing system simply tests each one by evaluating it on a representative set of data values. If no errors are detected, the tested verification conditions are accepted as true statements. Formal testing provides strong evidence that a program is correct, but does not guarantee it. The strength of the evidence depends on the adequacy of the test data; we will discuss this issue in depth in Section 6.

## 2. Specification Languages for Program Testing

Since formal program testing is applicable only to formally annotated programs, its viability as a practical tool depends on the development of concise, readable specification languages. In addition, formal program testing imposes a special constraint on program specifications: all of the operations appearing in program assertions must be computable. Otherwise, it is not possible to evaluate the program verification conditions on test data values. As a result of this constraint, specification methods that rely on the axiomatic definition of abstract operations are not applicable to formal program testing.

As a solution to the specification problem, we advocate using a separate very high level, applicative programming language -- including a constructive abstract data type definition facility -- as the specification language. The presence of a data type definition facility is critical because it allows program specifications to be written at a high level of abstraction -- omitting irrelevant, machine-oriented details. To accommodate formal testing, abstract data type definitions must provide an effective definition[2] for every primitive operation. Axiomatic data type definitions such as algebraic specifications [Guttag and Horning 78; Goguen 77] do not satisfy this critierion because they fail to provide algorithms (either implicitly or explicitly) for evaluating the primitive operations. In particular, it is impossible to compute whether or not two data objects (ground terms) from an axiomatically defined data domain are equal; the problem is undecidable.

To support this approach to program specification, we have designed a very-high-level programming language called TTL, based on the the TYPED LISP language described in [Cartwright 76]. In comparison with its predecessor, TTL includes a much richer constructive data type definition facility and a more comprehensive collection of built-in types. In addition to "abstract data types", all of the machine-oriented data types in conventional procedural programming languages such as PASCAL, FORTRAN, and ADA are easily defined in TTL. A description of the salient feature of the constructive data type definition facility used in TTL appears in [Cartwright 80].

As a programming language, TTL resembles Pure LISP generalized to a rich, extensible data

[2] If an operation is partial (undefined for some inputs), the effective definition may diverge on arguments that lie outside of the domain of the operation. For example, an effective definition of integer division must either diverge or return an error object when given an argument list of the form (x,0).

domain. As in Pure LISP, new operations (other than primitive operations implicitly defined by data type definitions) are introduced by recursive definitions. Since all of the primitive operations in TTL are continuous, recursive definitions in TTL are always well-defined. The meaning of a recursively defined function is the least fixed point of the corresponding functional.

The semantics of TTL have a natural formalization within conventional first order predicate logic. A TTL "program" composed of type definitions and recursive definitions of operations generates a corresponding first order theory including axioms for all of the defined operations. The domain and functions of the generated theory are simply the data domain and the defined operations of the TTL program. Each recursive type definition in the program generates a collection of axioms analogous to Peano's (first-order) axioms for the natural numbers. Similarly, each recursive definition of a TTL operation generates a corresponding recursion equation. This approach to formalizing the semantics of applicative programming languages is described in more depth in [Cartwright and McCarthy 79], and [Cartwright 80].

In contrast to conventional specification languages, TTL enables the programmer to describe the program clearly and concisely. In fact, we believe that formally documenting programs in TTL is worthwhile even when a formal program testing system is not available. the behavior a program and its subparts in formal, yet concise,

## 3. Specifications as Formal Documentation

A specification language like TTL constitutes a formal system for documenting programs. As [Liskov and Berzins 79] have argued, formal specifications are superior to informal ones in several respects. First, formal documentation enables the programmer to describe program behavior in mathematically precise terms. Every symbol appearing in a program has a well-defined mathematical interpretation. Expressing program documentation in appropriate formal notation forces the programmer to clarify his explanations. When someone tries to read formally documented code, he can determine exactly what each program component is supposed to do by reading the annotation. In contrast, informal specifications written in English or other natural language are inherently imprecise and ambiguous. A good illustration of this phenomenon is the widely acknowledged inadequacy of informal definitions of programming language semantics.

A second advantage of formal documentation is that it forces the programmer to use standardized notation; the documentation must obey formal syntactic rules. A programming language parser can check that program documentation is well-formed, just as it checks that program text is syntactically correct. As a result, many clerical errors in program documentation that would otherwise go undetected will be found and corrected.

The advantages of formal documentation, however, are academic if program specifications are difficult to write and to understand. Unless we develop and promote concise, very high level specification languages, the potential benefits of formal program documentation will go unrealized.

## 4. A Comparison with Conventional Testing Methods

Conventional program testing attempts to establish the reliability of software by the following principle of inductive inference: if the program works for a representative set of test inputs, then it should work in nearly all other cases. The programmer is responsible for selecting the test data and inspecting the output for each test case to verify that it is correct. As a tool for validating software, conventional testing is saddled with several significant liabilities. First, in the absence of formal specifications, it is often impossible to tell whether a particular program output is correct. For example, how does one determine whether the output of a compiler is correct? Without a formal definition of the source and target languages (e.g. an abstract interpreter for each written a formal specification language), the whole question of correctness is moot. In some cases it may be feasible for the programmer to write a test program that verifies that each test output is "correct". In this case, the test program indirectly serves as a formal

program specification. Unfortunately, this form of implicit specification does not constitute a concise, readable description of program behavior.

A second problem with conventional program testing is selecting a reprentative set of test inputs. Recently, [Budd, DeMillo, Lipton, and Sayward 80] have developed an ingenious method -- called mutation analysis -- for evaluating the adequacy of test data. To evaluate the quality of his test data, the programmer feeds his program and test data sets to a mutation analyzer, which generates a large collection of mutant programs and runs them on all of the test data sets. Each mutant program is identical to the original except that it contains a minor syntactic change mimicking a program error. The mutant analyzer exhaustively checks each mutant program on the test data sets until it produces an output that distinguishes it from the original program. Of course, some mutants may be semantically indistinguishable from the original program. The programmer must manually verify that every indistinguishable mutant is equivalent to the original program. If not, then the test data are inadequate and the programmer must devise test cases to distinguish the mutant from the original. The philosophical motivation for mutation analysis is that the original program is correct except for a few minor clerical errors.

Budd, et. al. report excellent results in detecting program errors when using conventional testing augmented by mutation analysis. The primary drawback to mutation analysis is that it requires large expenditures of programmer and machine resources. The mutant analyzer must run a very large collection of mutant programs on the test data and the programmer must check that mutants duplicating the original program's behavior are in fact semantically equivalent to the original program.

A final disadvantage of conventional program testing is that it attacks program correctness on a global rather than a local basis. A small change in a single program subroutine necessitates retesting the entire program. An individual subroutine cannot be rigorously tested in isolation of the subroutines that it calls (directly or indirectly). Moreover, smaller program building blocks such as loop bodies usually cannot be tested as independent units because their input and output states are too complex for the programmer to check reliably.

With the exception of generating representative test data, formal testing overcomes the pitfalls of conventional testing. Since it relies on constructive formal specifications, there is no question whether a particular test output is correct.[3] Furthermore, since verification conditions correspond to distinct linear paths in a program, formal testing is inherently local. Changing one statement in a program only requires retesting the verification condition for the path containing the statement -- assuming that the specifications for the path remain unchanged. Furthermore, when formal testing detects an error, it is easy to locate because it must occur within the path producing the erroneous test.

## 5. Generating Test Data

Automatically choosing a representative set of test data is a difficult problem that warrants further study. In many respects, automatic test data generation resembles automatic theorem proving. For each program path S with pre-assertion P, the test data generator must find sets of bindings for the free variables of P and S that satisfy P. Nevertheless, we believe that automatically generating test data for a program segment is a more tractable problem than automatically proving the corresponding verification condition. Programmers certainly find it much easier to generate test data than to prove programs correct. Furthermore, unlike program verifiers, test case generators do not have to attain a very high level of competence before they are useful in practice. When a test generator fails, the programmer can simply revert to generating the data on his own -- a task he must perform anyway if an automatic system in not available.

---

[3] Unless the output assertion involves an embedded, unbounded quantifier. In this case, the formal testing system must resort to testing the output assertion on a representative set of values for the quantified variable. In practice, embedded, unbounded quantifiers are rarely necessary.

One possible approach to generation test data for a verification condition is to symbolically evaluate the condition deferring the binding of every variable until the last possible moment. At each predicate forcing a variable binding, the generator makes a "non-deterministic" choice for the binding from a small set of heuristically generated values based on the particular predicate. If the variable belongs to inductively defined type T, then the base cases and first level constructions of T are obvious choices for members of the set. The generation of test cases proceeds by backtracking until all possible non-deterministic choices have been tried (given the generator sets a small bound on the maximum recursion depth allowed in evaluating recursively defined predicates and functions in the specification language).

For verification conditions involving repeated occurrences of complex expressions, it may help to generalize the verification condition (as in the Boyer-Moore theorem prover [Boyer and Moore 75]) before generating test values and then invert the substitutions (if possible) to produce test values for the original formula. In fact, much of the analysis that Boyer and Moore perform to determine their stategy in proving a formula could be profitably applied to the automatic generation of test cases. For example, the induction variable chosen by their analysis probably warrants a proportionally larger number of test values than the other variables appearing in a verification condition. Another obvious heuristic for generating test values is to use the output test values from each path as test inputs to subsequent paths.

A sophisticated formal testing system should do more than generate test data for the program and check that every test case satisfies the corresponding verification condition. It must also show that the generated data is sufficiently diverse to establish that the program is almost certainly correct. In the next section, we show how to adapt the concept of mutation analysis to formal program testing.

## 6. Mutation Analysis in the Context of Formal Program Testing

To evaluate the adequacy of test data generated for a particular verification condition, we simply apply mutation analysis to the corresponding program fragment. Specifically, we apply mutant operators to the program fragment S producing a set of program fragments which are identical to S except for minor clerical errors. We accept a set of test data as adequate for path S if for each mutant S´ that is not semantically equivalent[4] to S, it includes an input value that falsifies the verification condition for S´.

We believe that mutant analysis in the context of formal testing has several potential advantages over the original scheme proposed by [Budd et. al. 80]. First, since the analysis is local rather than global, the cost of testing mutants is much lower; only a short program fragment containing the mutation is executed rather than the entire program. Second, conventional testing in conjunction with mutant analysis will not necessarily detect a systematic error (such as a consistent "off-by-one" mistake) since no mutation corrects more than a single occurrence of the error. None of error-correcting mutants will necessarily behave correctly on a larger class of inputs than the original program. On the other hand, in formal testing each program path corresponding to a verification condition is independently tested. If some path contains only a single occurrence of the systematic error, then mutant analysis will almost surely discover it.[5] Finally, the presence of program annotation improves the prospects for automatic detection of equivalent mutants. Determining whether or not a mutant is equivalent to the original program (or fragment) is one of the most time-consuming parts of mutation analysis and a possible source of error if it is done manually. In the context of formal program testing, proving that a mutant is

[4] A program fragment S´ is semantically equivalent to the annotated program fragment {P} S {Q} iff the formulas {P} S´ {Q} and {P} S {Q} are logically equivalent.

[5] If the single occurrence of the systematic error is the only error on the path and the error is corrected by some mutant operator, then mutant

equivalent to the original fragment reduces to proving that the mutant's verification condition is equivalent to the original one. Instead of proving two large programs are equivalent, the system only has to prove two program fragments are equivalent. Of course, even in the latter case, it is unrealistic to expect that an automatic detection system will work most of the time, since the competence of existing automatic theorem provers is limited.

## 7. An Extended Example

To illustrate how a formal testing system based on TTL might work, we present an example taken from [Luckham and Suzuki 79]. The following "extended" PASCAL procedure annotated in TTL maintains an event queue -- a linear list of records each containing a key and count. The procedure takes an event queue and a key as input, increments the count field of the record corresponding to the key in the queue, and moves that record to the front of the list. If no record corresponding to the key exists, the procedure creates one at the front of the list. To eliminate annoying special cases, the list includes header and trailer records containing no data. For the sake of notational clarity, we have augmented PASCAL with Dijkstra's guarded command control structures.

```
Specification:

type event_queue = map[int,int]

function Abstr(head,tail: int): event_queue ≡
  global event#class;  (* event#class is the *)
                       (* heap array for     *)
                       (* record type event  *)

  let first = head↑.next;

  if first = tail then empty
  else {(first↑.key,first↑.count)}
            Abstr(first↑.next,tail);

function Add_event(q: event_queue,
                   e: int): event_queue ≡
  if e  Domain(q) then q(e ; q[e]+1)
    (* ( q - {(e,q[e])} ) ∪ {(e,q[e]+1)} *)
  else q ∪ {(e,1)};
```

```
Program:

type ref = ↑event;
   event = record key: int;
                  count: int;
                   next: ref
           end

procedure Search(ref : Head, Tail; int : X);
   logical var Q:event_queue;
   pre : Abstr(Head,Tail): event_queue ∧
         Abstr(Head,Tail) = Q;
   post: Abstr(Head,Tail)=Add_event(Q,X);
   var P, R : ref;
   begin
   P:=Head;
   Tail↑.key:=X;
  invariant: Abstr(Head,Tail)=Q ∧
              X ∉ Domain(Abstr(Head,P↑.next))
   do P↑.next↑.key≠X →  P:=P↑.next od;
   if P↑.next=Tail →
         new(R);
         R↑.next :=Head↑.next;
         R↑.count:=1;
         R↑.key  :=X;
         Head↑.next:=R;
   [] P↑.next≠Tail →
         R:=P↑.next;
         R↑.count:=R↑.count+1;
         P↑.next :=R↑.next;
         R↑.next :=Head↑.next;
         Head↑.next:=R;
   fi
   end Search
```

The data type definitions in the **Specification** section preceding the **Program** do not include the definitions either of "built-in" TTL types such as polymorphic sets and maps or concrete PASCAL types defined in the program. Every PASCAL **type** definition implicitly generates a corresponding TTL type definition -- formalizing PASCAL data objects and operations within TTL. The program specification uses an abstraction function (a concept introduced by [Hoare 72]) written in TTL to define the abstract data object (a map from **int** into **int**) corresponding to a low-level event-queue representation.

If we ignore the issue of termination, the verification conditions for the program are:

---

analysis is guaranteed to find it.

1. ∀Head, Tail, P:ref, Q:event_queue
   Abstr(Head, Tail): event_queue ∧
   Abstr(Head,Tail) = Q ⇒
     wp("P:= Head; Tail↑.key:= X",
        X ∉ Domain(Abstr(Head, P↑.next)) ∧
        Abstr(Head, Tail) = Q
       )

2. ∀Head, Tail, P:ref, Q:event_queue
      X ∉ Domain(Abstr(Head,P↑.next)) ∧
      Abstr(Head, Tail) = Q ∧
      P↑.next↑.key ≠ X ⇒
        wp("P:= P↑.next",
          X ∉ Domain(Abstr(Head, P↑.next)) ∧
          Abstr(Head, Tail) = Q
         )

3. ∀Head, Tail, P:ref, Q:event_queue
      X ∉ Domain(Abstr(Head,P↑.next)) ∧
      Abstr(Head, Tail) = Q ∧
      P↑.next↑.key = X ⇒
        (P↑.next = Tail ⇒
          wp("**new**(R);
              R↑.next := Head↑.next;
              R↑.count:= 1;
              R↑.key  := X;
              Head↑.next:= R",
             Abstr(Head,Tail) = Add_event(Q,X))
        ) ∧
        (P↑.next ≠ Tail ⇒
          wp("R:= P↑.next;
              R↑.count:= R↑.count+1;
              P↑.next := R↑.next;
              R↑.next := Head↑.next;
              Head↑.next:= R",
             Abstr(Head, Tail) = Add_event(Q, X))
        )

where "wp(S,R)" denotes the weakest pre-condition of program seqment S and post-assertion R. Each occurrence of the notation wp(S,R) in the verification conditions above is equivalent to a pure formula R′ where R′ is R modified by a substitution corresponding to the sequence of assignments S. We prefer "wp" notation to explicit substitutions for assignment statements because it is easier to read. It also happens to be a convenient form for the purposes of evaluation and mutation analysis.

Given test values for the universally quantified variables, a TTL interpreter (supporting PASCAL data objects and operations) clearly could evaluate the above verification conditions. On the other hand, automatically generating satisfactory test values is a subtle problem. For example, consider the problem of generating test data for verification condition 1. If we follow the strategy proposed in section 5, we can easily fall into an infinite recursion in evaluating Abstr(Head,Tail) if we happen to bind Head↑.next and Head to the same pointer value and bind Tail to a different one. Placing a bound on maximum recursion depth is an obvious "brute force" solution to this problem, but it rigidly limits the complexity of test cases that can be generated.

## 8. Directions for Further Work

Since formal program verification is far too expensive and time-consuming to be practical in most applications, formal program testing appears to be a promising, cost-effective alternative. Nevertheless, the success of formal testing depends on the willingness of programmers to document their programs with formal specifications. Unfortunately, the bulk of recent research in formal specifications has concentrated on axiomatic descriptions of program data and operations -- an approach that is incompatible with formal testing. The specification language TTL described in this paper is a first attempt at developing a specification language to support formal testing. We hope others will study the problem. Since the subject is still in its infancy, the prospects for significant advances in specification language design are bright.

We are currently building a formal testing system for PASCAL to evaluate the viability of formal testing and the utility of TTL as a specification language. We will report on the results in a subsequent paper.

## 9. References

[Boyer and Moore 75]
Boyer, R.S., and Moore, J S.; "Proving Theorems About LISP Functions," J. ACM 22, 1 (Jan. 1975), pp. 129-144.

[Boyer and Moore 79]
Boyer, R.S., and Moore, JS. A Computational

Logic, Academic Press, New York, 1979.

[Brand 76]
Brand, D.; "Proving Programs Incorrect," Proceedings Third International Colloquium on Automata, Languages and Programming, Edinburgh U. Press, Edinburgh, 1976, pp. 201-227.

[Brand 78]
Brand, D.; "Path Calculus in Program Verification," J. ACM 25(4), pp. 630-651.

[Budd et. al. 80]
Budd, T., R. DeMillo, R. Lipton, and F. Sayward; "Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Programs," Proceedings Seventh Annual ACM Symposium on Principles of Programming Languages," pp. 213-233.

[Cartwright 76a]
Cartwright, R.S.; "User-Defined Data Types as an Aid to Verifying LISP Programs," Proceedings Third International Colloquium on Automata, Languages and Programming, Edinburgh U. Press, Edinburgh, 1976, pp. 228-256.

[Cartwright 76b]
Cartwright, R.S.; A Practical Formal Semantic Definition and Verification System for TYPED LISP, Stanford Artificial Intelligence Laboratory Memo AIM-296, Stanford University, 1976.

[Cartwright 80]
Cartwright, R.S.; "A Constructive Alternative to Axiomatic Data Type Definitions," Proceedings of 1980 LISP Conference, Stanford University, August 1980, pp. 46-55.

[Earley 73]
Earley, J.; "High Level Operations in Automatic Programming," Computer Science Department, Technical Report 22, University of California, Berkeley, 1973.

[Earley 74]
Earley, J.; "High Level Iterators and a Method for Automatically Designing Data Structure Representation," Electronics Research Laboratory, Memorandum No. ERL-M425, University of California, Berkeley, 1974.

[Gerhart 75]
Gerhart, S.L.; "Correctness Preserving Program Transformations", Proceedings of the Second ACM Symposium on Principles of Programming Languages, pp. 54-66.

[Goguen 77]
Goguen, J., Thatcher, J., Wagner, E., and Wright, J.; "Initial Algebra Semantics and Continuous Algebras," JACM 24, pp. 68-95.

[Guttag and Horning 78]
Guttag, J.V. and JJ. Horning; "The Algebraic Specification of Abstract Data Types," Acta Informatica 10, pp. 27-52.

[von Henke and Luckham 74]
von Henke, F. and D.C. Luckham; "Automatic Program Verification III: A Methodology for Verifying Programs," Stanford Artificial Intelligence Project Memo AIM-223.

[Hoare 72]
Hoare, C.A.R.; "Proofs of Correctness of Data Representation," Acta Informatica 1, 271-281.

[Hoare 75]
Hoare, C.A.R. "Recursive Data Structures," International Journal of Computer and Information Sciences 4,2, pp. 105-132.

[Igarashi, London and Luckham 75]
Igarashi, S., London, R.L., and Luckham, D.C.; "Automatic Program Verification I: Logical Basis and Its Implementation," Acta Informatica 4, pp. 145-182.

[Kennedy and Schwartz 75]
Kennedy, K. and Schwartz, J.T. "An Introduction to the Set Theoretical Language SETL," J. Comptr. and Math. with Applications 1 (1975), pp. 97-119.

[Liskov and Zilles 75]
Liskov, B. and S. Zilles; "Specification Techniques for Data Abstractions," IEEE Transactions on Software Engineering, Vol. SE-1, pp. 7-19.

[Liskov and Berzins 80]
Liskov, B. and V. Berzins; "An Appraisal of Program Specifications," in Research Directions in Software Technology, P. Wegner (ed.), MIT Press, pp. 276-302.

[Luckham and Suzuki 79]
Luckham, D.C. and Suzuki, N.; "Verification of Array, Record, and Pointer Operations in Pascal," TOPLAS 1 (1979), pp. 226-244.

[McCarthy 63]
McCarthy, J.; "A Basis for a Mathematical Theory of Computation," in Computer Programming and Formal Systems, Braffort and Hirschberg, eds., North-Holland, 1963.

[Nelson and Oppen 79]
Nelson, G. and D. Oppen; "Simplification by Cooperating Decision Procedures," ACM TOPLAS 1(2), pp. 245-257.

[Schwartz 79]
Schwartz, J.; Personal communication.

[Warren and Pereira 77]
Warren, D. and Pereira, L.; "PROLOG: The Language and Its Implementation Compared with LISP," Proceedings of the ACM Symposium on Art. Intel. and Prog. Lang., SIGART/SIGPLAN