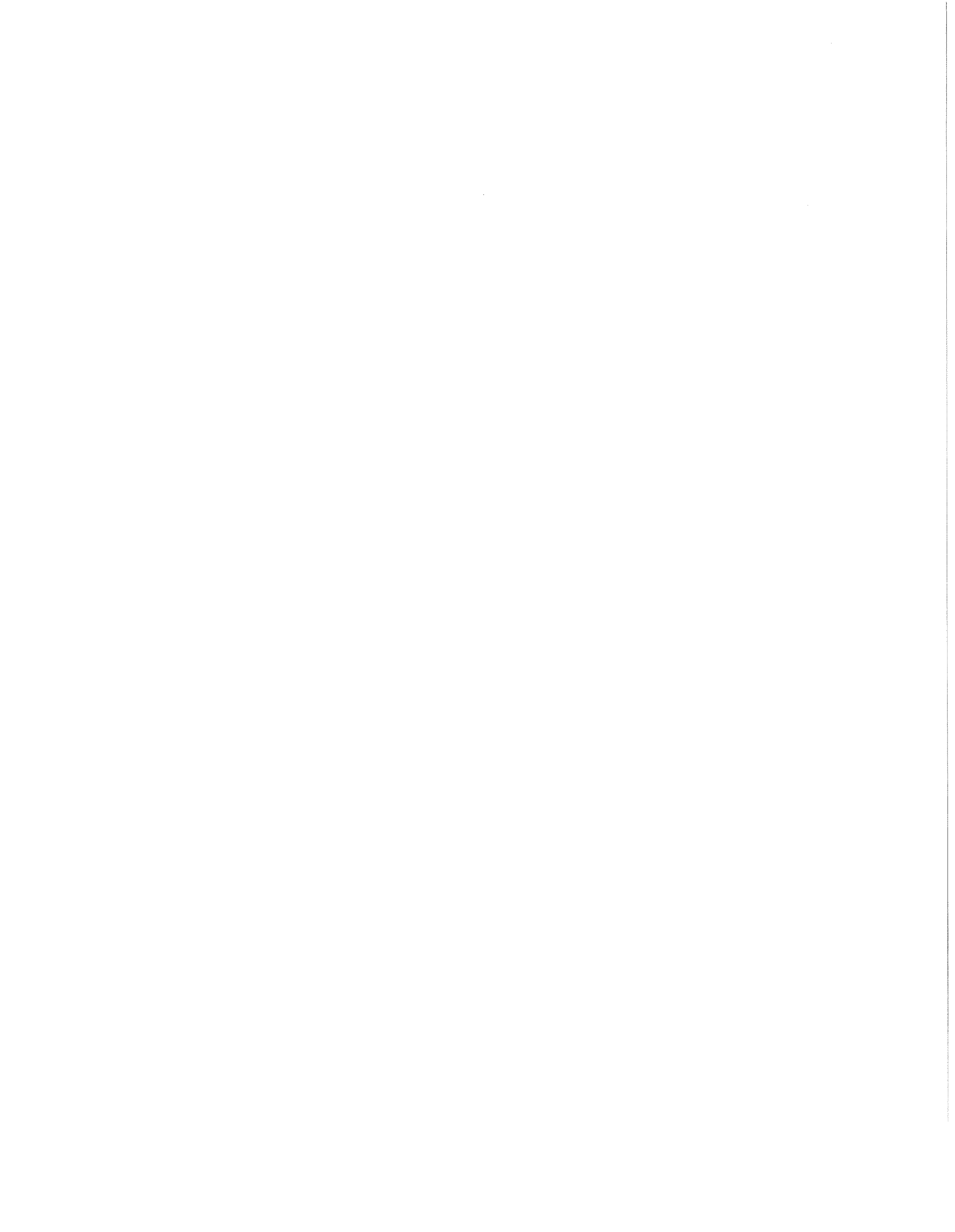
LOGIC FOR LOGIC PROGRAMMERS

by

Kenneth Kunen

Computer Sciences Technical Report #884

October 1989



LOGIC FOR LOGIC PROGRAMMERS

Kenneth Kunen[1]

Computer Sciences Department
University of Wisconsin
Madison, WI 53706, U.S.A.
kunen@cs.wisc.edu

October 27, 1989

ABSTRACT

The goal of logic programming is that the program, or database, can be understood by logic alone, independently of any execution model. Attempts to realize this goal have made it clear that the logic involved must go beyond ordinary first-order logic. This survey will explore several topics of current interest in the logical meaning of logic programs, with particular attention paid to: (1) The meaning of negation; this still remains problematical, although many partial results are known. (2) The meaning of recursions; these imply a least fixed-point computation in deductive databases, and something else in Prolog. (3) The meaning of the Prolog builtin predicates, such as the evaluation of numeric terms. (4) The semantic meaning of the order in which a stream of answers is returned.

§0. INTRODUCTION.

This is a survey of some current issues regarding the declarative semantics of logic programming. It will raise more questions than it answers. This Introduction explains why you should care at all about declarative semantics, and outlines the questions which will be discussed in greater detail in the rest of the survey.

Importance of declarative semantics. The key feature which separates logic programming from other forms of programming is that the program says something in logic, and any answers obtained are logical consequences of what the program says. Thus, any answers obtained are guaranteed to be correct, *by logic.* At least, this is the ideal behind logic programming. The following pages will explain why this ideal is not realized in Prolog, and will explore how close we are to realizing it in any practical logic programming language.

My description of ideal logic programming implies that a logic programming language be set up as database query language. That is, the program (or database) is (or denotes) a set of statements in predicate logic, and the program is run each time the user queries the

[1] This research was supported by NSF Grant DMS-8501521.

database; in such a "run", the computer decides whether the user's query is a logical consequence of the database. I'm not claiming that all desired applications of programming can be so set up, but a lot of them can. Certainly, there are many "obviously" database-style examples; the program may be a set of facts about parenthood and a definition of ancestry, the query may be ?– *ancestor*(*abraham*, *jacob*), and the answer *yes* signifies that the statement *ancestor*(*abraham*, *jacob*) is a logical consequence of the database. But also, a call for a numerical computation can be phrased as a database query. For example, the program may be a set of axioms about numbers, the query might be ?– $factorlist(100, X)$, and the answer $X = [2, 2, 5, 5]$ indicating that $factorlist(100, [2, 2, 5, 5])$ is a logical consequence of the axioms. Of course, the computer doesn't know English, and the programmer is responsible for the formal symbol, *ancestor*, really axiomatizing our informal notion of the meaning of ancestor, or *factorlist* axiomatizing our usual understanding of a list of factors. However, assuming we have correctly axiomatized our informal notion, we know, *by logic*, that any answer obtained is correct; unlike other approaches to programming, we don't need to prove a correctness statement, which, if expressed formally, would be in some formal language other than our programming language.

The program finds its answers by implementing some form of automated deduction. Since automated deduction in general is plagued by the well-known combinatorial explosion, it is reasonable to ask whether this style of programming is even feasible. The best argument for feasibility I know of is given by Kowalski's famous statement, "Algorithm = Logic + Control" [15]. As I understand it, this means that the program (the Algorithm) is (or denotes) a disjoint union of a set, S, of statements in predicate logic (the Logic) plus a set of hints to the theorem prover which help it to find deductions (the Control). Only the logic part is relevant to the correctness of answers; thus, if ϕ is a computed answer (such as $factorlist(100, [2, 2, 5, 5])$), then ϕ is a logical consequence of S, and the computer will have found (something denoting) a formal proof of ϕ from S. With good control, the computer will find proofs quickly; with bad control, the computer will find proofs slowly, or maybe not at all.

This separation into logic and control makes logic programming at least seem feasible. At our current stage of knowledge, we cannot expect computers to find, without guidance, non-trivial conclusions from sets of axioms. This would require truly intelligent computers, which exist only in science fiction. By making intelligence (control) the responsibility of the programmer, it is at least conceivable that the computer can handle the formal logic and guarantee correctness.

By *declarative semantics*, I mean the precise description of how to translate a logic program into formal logic. The reason why this translation isn't completely trivial is discussed next. In this discussion, it is important to distinguish declarative semantics from procedural semantics. The *procedural semantics* is an implementation-independent description of what the interpreter does – that is, how it searches for a deduction. Note the difference in issues. Given a language such as Prolog, it is fairly easy to describe its procedural semantics; obviously, programmers must have a fairly good intuitive grasp of that to be able to program, and it is easy to make this intuition mathematically precise. Thus, the issues involved are how or whether to improve the procedural semantics. On the other hand, it is not always clear what the declarative semantics *is* in a given language,

such as Prolog, let alone asking how to improve it in other logic programming systems. We discuss these issues further in this paper; first let us see why it isn't trivial.

Why isn't it trivial – or, at least, all well known by now? It is all well known in the case of pure Horn logic, so let us give an example here to see what the issues are. Now, as for the logic part, the program is a set of Horn clauses. For example, consider the following program, which gives two facts about parenthood (p), together with the definition of the ancestor relation (a).

$p(i, j)$. %1
$p(j, k)$. %2
$a(X, Y) :- a(Z, Y), p(X, Z)$. %3
$a(X, Y) :- p(X, Y)$. %4

This isn't quite standard predicate logic, but there is an obvious translation into standard logical syntax. Declaratively, the program simply stands for the following set, S, of four sentences of predicate logic:

1. $p(i, j)$
2. $p(j, k)$
3. $\forall X, Y, Z(a(Z, Y) \wedge p(X, Z) \Rightarrow a(X, Y))$
4. $\forall X, Y(p(X, Y) \Rightarrow a(X, Y))$

Call this the *naive reading* of the program. For the query, $?-\ a(X, k)$, the answer $X = i$ is supported by the declarative semantics – meaning that the sentence $a(i, k)$ is a logical consequence of S. The relevant abstract procedural semantics is called SLD, which is really just the search for a proof in a kind of Resolution. SLD is essentially a formalization of what ordinary Prolog does, *except* that it is non-deterministic – that is, it is free to choose the clause or literal to reduce by next, whereas standard Prolog reduces left-right. Here, the sequence of reductions in SLD is:

$$a(X, k) \longrightarrow_3 a(Z, k), p(X, Z) \longrightarrow_{1; X=i, Z=j} a(j, k) \longrightarrow_4 p(j, k) \longrightarrow_2 true \quad ,$$

where the subscript indicates the clause used in the reduction. SLD satisfies both a *Completeness* Theorem and a *Soundness* Theorem. Soundness says that any answer obtained by SLD is indeed a logical consequence of the program. Completeness says, conversely, that any answer which is a logical consequence of the program will be obtained by SLD; more formally, if the query is $?-\ \phi$ and σ is a substitution such that $\forall\phi\sigma$ is a logical consequence of the naive reading of the program, then σ or some more general substitution will be returned by SLD. The completeness theorem [3] was non-trivial at the time (1982), but is by now well-known and in standard texts [21].

Note the role of *control* here. Completeness does not say that answers will be returned by Prolog. In fact, in this example, Prolog will not return at all. In pure Horn Prolog, the only control mechanism is the ordering of the clauses and the ordering of the literals in the clause body, which is orthogonal to the declarative semantics. This ordering directs the non-determinism in SLD. Here, if the two literals in the body of clause 3 are transposed, the answer will be found by standard Prolog. Intelligence (control) is the responsibility of the programmer. Some further remarks about the relation of control to completeness are contained in §4.

In general, as we consider various questions about semantics, you should be aware of the difference in importance between soundness and completeness theorems. Soundness is of vital importance – it corresponds to the key feature of logic programming that the correctness of any answer obtained is guaranteed *by logic*. Any "logic programming" system which fails to be sound is not really logic programming. Completeness, although nice, is not vital, and holds only partially in more complex situations. It is a little harder to evaluate what completeness means in practical terms. For one thing, mathematical completeness results talk about an answer being found *eventually*, not whether the answer will be found in a feasible amount of time. Also, logic programming completeness results usually refer to some non-deterministic search for deductions, and leave open the issue of whether there is an easily implementable control strategy (see §4). Nevertheless, partial completeness results are of interest and give us confidence that the claimed declarative semantics has some relationship to the evaluation procedure.

Anyway, the reason that the subject is non-trivial and doesn't end with SLD is that pure Horn logic does not have sufficient expressive strength for many applications. In fact, the whole subject can be discussed in terms of a tradeoff between *expressive strength* and *efficient evaluation*.

Expressive strength by itself is no problem; one can always just use full first-order logic; or, if necessary, second-order logic. However, these logics not correspond to efficient evaluation procedures. We would expect our declarative semantics to be at least *effective*; that is, the set of supported queries should be recursively enumerable. If this fails, then there is no evaluation procedure *in principle* for the semantics. Of course, even if it is effective, there remains the question of whether there is an evaluation procure which can be made efficient with suitable control. So, we reject second-order logic immediately because it is not effective. First-order logic is effective; that is, there are standard proof procedures (such as Resolution) which will, in theory, enumerate all logical consequences of a set of axioms. Nevertheless, we reject full first-order logic for now because we do not know of an *efficient* evaluation procedure for it. Pure Horn logic seems promising because we have an efficient evaluation procedure for it, but it is too weak for many applications.

So, there has been much work, including, of course, within Prolog itself, which tries to stay within the framework of the Prolog-style execution model, and add features to remedy the lack of expressive power. Unfortunately, while these features have sometimes led to efficient programming methods, their semantic meaning is often obscure or completely non-existent. The challenge of finding a well-defined declarative semantics which yields both expressive power and efficient evaluation has to a large extent not been met. In the following sections we discuss a number of issues along this line – for now, we just briefly outline the problems.

One issue, to which much literature has been devoted, is negation. The restriction in a Horn clause, that the literals in the body must all be positive, seems too restrictive, so it is natural to consider allowing the literals to be negative as well. For example, once we have defined *ancestor*, it would be natural to talk about "$\neg$*ancestor*". If we were only concerned with expressive strength, there would be no problem here – but we would be back in full first-order logic with its corresponding lack of efficient evaluation. Attempts to have negation and preserve the standard Prolog execution model, have lead to the notion

of negation-as-failure. However, there is some problem defining precisely what negation-as-failure means. There are by now several mathematically precise definitions, but then, after making the semantics precise, it is not always clear whether it corresponds to an efficient evaluation procedure, or even to any evaluation procedure at all. These issues are discussed in more detail in §1.

Another issue, of particular interest now in deductive databases, is the issue of fixed-points. The ancestor relation, a, as axiomatized above, is a standard example. In many database contexts, it is desired not that a is *just any* relation satisfying the axioms, but that it is the *least* such relation. There are semantic problems saying exactly what "least" means, especially in the presence of programs with negation. For more details, see §2.

There are also semantic problems regarding the many built-in functions added to most Prolog systems, such as numerical evaluation. If we actually wrote out the code for the predicate *factorlist* discussed above, we would undoubtedly use the machine evaluation for arithmetic operations. Problems involving the semantic meaning of this are discussed in §3.

In the usual discussions of declarative semantics, a logical meaning is given only to each answer *individually* as it is printed on the terminal. However, in practice, many programs rely on the *order* in which the answers obtained. Whether this order can be described in logic is discussed in §5.

The Ph.D. effect. This describes the tendency of many descriptions of declarative semantics in the literature to require a Ph.D. in logic to understand. Is this bad? That depends on whom we think we are talking to. There are two possible answers here. One is that we talking among ourselves. This is OK if the point of the semantics is to prove theorems about logic programming and its implementations. For example, once we fix on a declarative semantics, we might wish to use it to prove that a given implementation of Prolog, perhaps using some non-trivial optimizations or control mechanisms, is correct in the sense that all the answers that it returns are correct according to this semantics. However, if the point of declarative semantics is that the program has a meaning, it might be desired that ordinary Prolog programmers be able to understand what their programs mean. Such programmers will more-or-less understand ordinary first-order logic (which means what you think it means) but may not be experts in fine points in model theory, multiple-valued logics, etc. Perhaps these fine points may be necessary to explain arbitrary pathological programs, such as ones with deeply nested or unusual uses of negation, but we would hope that the ordinary kinds of programs which ordinary people write will have ordinary explanations. We strive for this in the following, although we are not completely successful yet.

Even if you have a Ph.D. in logic, you should bear in mind the following remark. The point of logic programming is really that the program should have a *simple and clear* translation into logic. Otherwise, there really is nothing special about logic programming. *Anything* can be formalized in logic, including the procedural semantics of any procedural language. For example, one could write a set of sentences in predicate logic which axiomatizes the progression through time of an abstract computer working on a Pascal program. But this axiomatization introduces a lot of new constructs which are not part of Pascal. The challenge is to obtain the simplicity and clarity of pure Horn logic in a more expressive

programming language.

§1. NEGATION.

There are many cases where one wishes to use a negation in the body of a clause. For example, if *member* is defined in the standard way, we may wish to call for *not member*, as in the following simple program:

$$member(X, [X|Tail]).$$
$$member(X, [Y|Tail]) :- member(X, Tail).$$
$$addin(X, L, L) :- member(X, L).$$
$$addin(X, L, [X|L]) :- not\ member(X, L).$$

Here, *addin* adds a new element to the head of a list if it's not already a member; $?- addin(a, [a, b], L)$ results in $L = [a, b]$, whereas $?- addin(a, [c, b], L)$ results in $L = [a, c, b]$.

Procedurally, the way Prolog arrives at $M = [a, c, b]$ for $?- addin(a, [c, b], M)$ is known as *negation-as-failure*. In the course of the derivation, the system considers the subgoal, $\neg member(a, [c, b])$, whereupon it attempts to derive $member(a, [c, b])$ and fails; it then concludes that the negation must be true. The question is whether this evaluation procedure corresponds to anything declaratively meaningful.

If we simply consider the program as shorthand for its naive reading, as described in the introduction, then there is no way the statement $addin(a, [c, b], [a, c, b])$ is supported by the semantics, since the logic does not imply that $\neg member(a, [c, b])$. In fact, the naive reading of any pure Horn definition is consistent with all predicates being universally true. Clark [8] suggested saying that the logical meaning of a program is not exactly it's naive meaning, but something slightly more complicated, known now as the Clark completion. The Clark completion contains the completed definition of each predicate. Roughly, instead of the "natural" implications ($\Rightarrow$) read off the program, the completed definition is an iff ($\Leftrightarrow$) statement which says that the *only* way a predicate can be true is via one of the clauses in the program. For formal definitions, see [8,21,16]. We may get the completed definition in two steps. First, replace each clause in the definition of the predicate by a *normalized* one in which all the unifications are in the body. In the case of *member*, we would have:

$$member(A, B) :- A = X, B = [X|Tail].$$
$$member(A, B) :- A = X, B = [Y|Tail], member(X, Tail).$$

Second, we write, as the completed definition of *member*, the statement that these two ways are the only ways $member(A, B)$ can be true:

$$\forall A, B\Big\{ member(A, B) \Leftrightarrow$$

$$\Big(\exists Tail(B = [A|Tail]) \quad \vee \quad \exists Y, Tail(B = [Y|Tail] \wedge member(A, Tail)) \quad \Big)\Big\}$$

We also need to add some axioms about equality to be able to derive $\neg member(a, [c, b])$. These axioms say, for example, that $a \neq c$ and $a \neq b$; these facts, taken for granted in Prolog, do not follow by pure logic alone. More generally, Clark's equality axioms contains the usual axioms in logic about $=$, plus the universal closure of $\tau_1 \neq \tau_2$ whenever τ_1 and τ_2 are terms which are not unifiable. Then $\neg member(a, [c, b])$ does follow logically from our equality axioms plus the completed definition of *member*.

The Clark completion gives semantic support to these negation-as-failure derivations in Prolog. To express this support mathematically, one defines a formal deductive system, called SLDNF (see [21]). SLDNF is, roughly, a mathematical formalization of what Prolog actually does. SLDNF can then be proved to be sound, in the sense that any statement derived by SLDNF is in fact a logical consequence of the completion of the program. We now note some good and bad features of the completion approach. Remember, we are trying to balance expressive strength with efficient execution.

On the positive side, we are adding the expressive strength of negation in examples such as this, where we want to express it. In fact, in some sense, we are getting a little more. With negation-as-failure, we don't have to explicitly program "non-membership". This is a minor point here, but becomes very useful with a large relational database, where we don't have to explicitly list all tuples for which relations are false. As for efficient evaluation, we are preserving the idea behind Prolog-style depth-first-search execution model, whereas a more "naive" addition of negation would put us back into ordinary resolution logic. It is true that the completion of a program is a little more complicated than it's "naive" logical reading, but this is still a relatively straightforward syntactic transformation – we can view the program declaratively as an abbreviation for its completion.

Another good feature is that we are not changing the naive reading in the case where there was no problem with it – namely, for affirmative answers to positive programs and queries. That is, if P is a positive program and ϕ is a positive query, then ϕ follows from the naive reading of P iff ϕ follows from the completion of P. Note, however, that even here, the agreement is only in the case of affirmative answers to the queries. If the answer is "no", there has been a change. Say the query if $?- \; q$. In the "naive" reading, "no" meant only that the system has been unable to come up with a derivation of q, whereas now it means that q is refuted – i.e., $\neg q$ follows from the completion.

A third positive feature is that the completion handles very neatly the non-monotonic character of the logic. It is sometimes said that Prolog is an example of non-monotonic reasoning – that is, the set of logical consequences of the program is not a monotonic function of the program. For example, you might have $?- \; \neg q$ return "yes", but expanding the program (for example, by adding q), $?- \; \neg q$ will return "no". The notion of non-monotonic logic seems very pathological to many logicians. Now, the only non-monotonicity is in the passage to the Clark completion; after that, we use ordinary monotonic logic. For more on the relation of Prolog to non-monotonic logic, see Przymusinski's [26].

On the negative side, there are many semantic problems which make the situation not quite as simple as it might seem. In specific, I mention five problems.

Problem 1. The statement that SLDNF is a formalization of what Prolog does must be taken with a grain of salt. As with SLD, it is non-deterministic – presumably, this non-determinism is handled by the "control". More fundamentally, Prolog's treatment of

negated non-ground literals is wrong (i.e., unsound), and would have to be revised to be in accordance with SLDNF (see [21]). SLDNF requires that the interpreter *flounder*, or produce an error message and quit, if asked to evaluate a negated non-ground literal. This is done correctly in the language NU-Prolog [27].

Problem 2. The negation-as-failure interpretation renders control mechanisms such as the cut or *var* unsound. In the purely positive case, these could be viewed as purely control, with no logical meaning. As long as "no" meant "no proof found", any mechanism which serves only to prune the search tree is sound. But now, if a "no" to $?-\ q$ means that $\neg q$ is supported, these are no longer sound. For a purely propositional example,

$q :-\ a, !, b.$
$q.$
$a.$

In Prolog, $?-\ q$ will return a "no" so that $?-\ \neg q$ will return a "yes", whereas q, rather than $\neg q$ follows from the program (either in its naive reading, or its completion). Thus, it seems that in this interpretation of negation, the only valid control mechanism are ones which affect the ordering of goals. A sound Prolog should really "flounder" here – more generally, it should flounder rather than produce any negative answer to a goal or subgoal if the search tree was pruned while searching for that answer.

Problem 3. SLDNF is not complete. Although there are completeness results for specific classes of programs (see [7, 17, 18, 19]), the current belief now is that there is no complete evaluation procedure which resembles Prolog and handles negation. But, as explained in the Introduction, this is perhaps not a serious deficit. The partial completeness results give us some confidence that the declarative semantics is on the right track, and there is ongoing research into improving SLDNF to better handle negated goals. In any case, the semantics is effective, since it is defined as the set of logical consequences of a decidable set of axioms.

The three problems above might be called *procedural* problems – that is, the declarative semantics seems clear enough, but we need to to revise Prolog somewhat to put it into better conformity with the semantics. The next two problems are *declarative* ones, in that they point out defects in the semantics itself.

Problem 4. The completion of the program can be inconsistent. The standard example is:

$q :-\ not\ q.$
$r.$

The completed definition of q is $q \Leftrightarrow \neg q$, which is inconsistent. Thus, declaratively, any statement follows from the completion; so, for example, a "yes" answer to $?-\ \neg r$ is correct. This is not a mathematical problem, but it does seem to violate the Prolog spirit. The pathology in the definition of q should not affect our reasoning about r, whose definition does not at all involve q. More generally, we would expect a certain *locality* in the semantics of Prolog resembling that of other programming languages. For example, if $P = P_1 \cup P_2$, where P_1 and P_2 use disjoint sets of predicate symbols, then we would expect that the answers to a query using only symbols of P_2 would be the same under the program P_2 as under P.

One way out is to use 3-valued logic rather than 2-valued logic [12, 16]. Here, we have a third truth value **u**, which could be read as "undefined" or "undetermined". The idea that 3-valued logic might be relevant to Prolog occurred first in [23,20]. Then, Fitting [12] showed how, given any program, there was a natural construction of a 3-valued model of the completion built on the Herbrand universe. This construction proceeded through transfinite steps, and the semantics obtained from it was not effective. However, a related 3-valued approach, described in [16] is effective. There, we keep the Clark completion unchanged, but say that a statement is supported by the semantics iff that statement is true in all 3-valued (not necessarily Herbrand) models of the completion. The effectiveness of the semantics follows from the fact that the set of logical consequences of a decidable set of axioms is always recursively enumerable.

With either this semantics or Fitting's, it is easy to prove the *locality* mentioned above. Both approaches agree for propositional programs and queries; the differences appear only when the programs use variables and function symbols. In the specific program above, all 3-valued models for the completion give q the value **u** and r the value **t**; thus, any interpreter which answers "yes" to $?-\ \neg r$ is now unsound, as expected. On the propositional level, this is a complete success, in the sense that SLDNF is sound and complete for this 3-valued semantics [16,17]. With terms and variables present, we still have soundness; completeness is lost, but we would expect that anyway, as remarked above, although there are completeness results [17] for specific classes of programs and queries. For example, a query is called *allowed* iff every variable which occurs in it occurs in at least one positive literal, and a program clause is called *allowed* iff its body is allowed and every variable which occurs in the head occurs at least once in the body. If the program and the query are allowed, then every answer substitution supported by the 3-valued semantics is ground, and is computed by SLDNF. Unfortunately, many common Prolog programs are not allowed, such as the standard definition of *member*.

A negative feature of this 3-valued semantics is the Ph.D. effect, mentioned in the Introduction. Most ordinary programmers might be a little mystified by 3-valued logic, and the correct choice of 3-valued truth tables is a little technical. This is partially ameliorated by some sufficient syntactic criteria involving "strictness" [17], satisfied by many (but not all) programs written in practice, which imply that the 3-valued and 2-valued semantics are equivalent. More precisely, a program is called *strict* iff no predicate depends (hereditarily) on itself negatively and no predicate depends on any other predicate both positively and negatively. A query is called *strict* iff it does not depend on any predicate both positively and negatively. The result in [17] says that if the program and query are both strict, then 2-valued and 3-valued logic support the same answers.

As another pathological example, the program consisting of the one clause, $p :-\ p$, is strict, but the query, $?-\ p, \neg p$ is not. In 2-valued logic, a "no" answer is supported, since $\neg(p \land \neg p)$ is a tautology. The 3-valued semantics allows a **u** value to p and hence to all propositional formulas constructed from p, and does not support either a "yes" or a "no" to this query. This corresponds nicely to the fact that Prolog will not halt with this query.

Unfortunately, non-strict constructions are not limited to such pathologies, and are in fact common programming practice. For example, the definition of *addin* at the beginning of this section is non-strict, since it depends on *member* both positively and negatively. In

this case one can verify that 2-valued and 3-valued logic agree on the "intended" queries, of the form $addin(\alpha, \beta, X)$, where α and β are ground terms. However, if the query is $?-\ addin(a, L, L), addin(a, L, [a|L])$, the 2-valued semantics supports a "no" but the 3-valued semantics does not support any answer. The reason is that there are 3-valued models for the completion which contain a (non-Herbrand) element λ for which $member(a, \lambda)$ has value **u**. As a curiosity, if you ask this query in Prolog, you "should" get an infinite loop, but in fact, unless your Prolog does an occur check, you will get "infinite" term when L is unified with $[a|L]$. The occur check is, of course, another problem with Prolog, but we do not discuss it in this paper because it is really an implementation problem, not a semantic one.

Non-strict programming is encouraged by Prolog's if-then-else construct, which hides an explicit "not". For example, it is more efficient to program *addin* by the semantically equivalent

$$addin(X, L, M) :- \quad member(X, L) \rightarrow M = L \ ; \ M = [X|L].$$

With this form, *member* is only called once in the evaluation of *addin*. Since the semantics of this if-then-else immediately reduces to that of negation, it is not necessary to take up if-then-else as a separate topic, but we point out that any program which uses it is not strict.

There is ongoing research into weakening the strictness criterion to something which really did include all programs and queries used in practice. If this succeeds, then ordinary programmers could think in terms of ordinary 2-valued logic, even though 3-valued logic would be required to understand arbitrary pathological uses of negation. One could even have Prolog systems print a warning when such a pathological construct was used.

The purpose of this paper is not to dwell on the fine points of 3-valued logic, and further examples will in fact satisfy the strictness criterion, so that we may emphasize instead the further semantic problems which need to be discussed.

There are other approaches to avoiding the inconsistency problems with negation, using modal logic [13] or intuitionistic logic [22]. They also suffer from the Ph.D. effect.

Problem 5. The completion semantics is not sufficiently expressive for many database applications, and a wholly different approach has been advocated. This defect is not sensitive to the distinction between 2-valued and 3-valued logic, and becomes apparent even in programs which do not use negation at all. We take it up in the next section.

§2. MINIMAL MODELS.

There are many examples in database-style applications where the completion approach yields a semantics which is so weak in expressive strength that it becomes awkward to express some very commonly used notions. Often, we want to say that a relation is not just *some* relation satisfying the axioms, but *the least* such relation. By "least", we mean, roughly, that the relation being defined is to be *false* unless it "has to" be true, in some sense which we must define precisely. In fact, this is the same motivation behind negation-as-failure, where we considered $member(a, [b, c])$ to be false becauses it doesn't "have to" be true, but the Clark completion semantics doesn't make things "least enough" for some useful applications.

An example to illustrate "least" can be given even in the purely propositional setting, although a propositional example will not correspond to an interesting application. If the program is:

$$p :- p. \quad ,$$

then the naive reading, $p \Rightarrow p$, and the completed definition, $p \Leftrightarrow p$ are both tautologous – i.e., say nothing about the truth of p. This corresponds to the fact that the query $?-\ p$ does not return in Prolog. A person who wants the *least* model would say that p is false. Corresponding to this, an answer of "no" should be returned for $?-\ p$, or "yes" to $?-\ \neg p$. Clearly, such a system will not resemble the Prolog-style evaluation, but perhaps it deserves further examination.

That it deserves further examination is apparent when we turn to predicate logic, where we see some practical applications of this "least model" semantics. Let us return to the transitive closure (ancestor) example above, with minor modifications. Actually, if you want a program for the transitive closure of a given relation, p, several possibilities come to mind; we write out two, $t1$ and $t2$, together with a small sample p:

$$p(a,b).\ \ p(b,a).\ \ p(c,d).$$
$$t1(X,Y) :- p(X,Y).$$
$$t1(X,Y) :- p(X,Z), t1(Z,Y).$$
$$t2(X,Y) :- p(X,Y).$$
$$t2(X,Y) :- t2(X,Z), t2(Z,Y).$$

Then the least relation satisfying the definition of $t1$ given by the program (either in its naive reading, or its completion) is the transitive closure of p. Likewise for $t2$ – i.e., the two definitions define the same relation. However, from the point of view of Prolog, which corresponds to logical consequences of the completion, these definitions are not equivalent, and neither one of them correctly defines transitive closure.

The completed definition of $t1$ is:

$$\forall X,Y(t1(X,Y) \Leftrightarrow (p(X,Y) \vee \exists Z(p(X,Z) \wedge t1(Z,Y)))) \quad ,$$

while the completed definition of $t2$ is:

$$\forall X,Y(t2(X,Y) \Leftrightarrow (p(X,Y) \vee \exists Z(t2(X,Z) \wedge t2(Z,Y)))) \quad .$$

Although the completion of this program is *consistent* with $t1$ and $t2$ being the correct transitive closures, this is not required. In particular, the completion is also consistent with $t2$ being true of everything. Thus, for example, the truth value of $t2(c,a)$ is not determined from the program; there are models of the completion in which it is true, and other models in which it is false. This corresponds nicely with the fact that Prolog will not return from $?-\ t2(c,a)$. On the other hand, the completion logically implies that $t1(c,a)$ is false, and Prolog will return a "no" to $?-\ t1(c,a)$. However, $t1$ also does not describe the transitive closure, since the truth of $t1(a,c)$ is not determined from the completion; although $t1(a,c)$ is false of the true transitive closure, there are models of the completion in which $t1(a,c)$ and $t1(b,c)$ are both true. Again, this corresponds nicely with the fact that Prolog will not return from $?-\ t1(a,c)$.

In this example, the problem is not with Prolog; what Prolog returns is in total agreement with the completion semantics. The problem is with the completion semantics itself. This sort of example was discussed by Van Gelder [28] and Przymusinski [25] as illustrations of the weakness of the completion semantics. In fact, not only do these two "natural" definitions of transitive closure not work – there seems to be no really nice way of defining transitive closure at all in Prolog. There are some theoretical limitations; for example [18] shows that there is no definition of transitive closure which is strict and doesn't use function symbols. Using function symbols, one can define it, although rather awkwardly. The fact that you have to be rather tricky to program such a natural concept in an efficient way may be considered to be negative feature of the completion semantics, and leads one to think that the "right" semantics may be some sort of "least model" semantics.

We are faced with two challenges. First, can we make mathematical sense out of this notion of "least"? Second, even if we can, does it correspond to an efficient evaluation? As usual in our attempt to balance expressive strength and efficient evaluation, there is no problem finding logics strong enough to express transitive closure or any other concept we want; the problem is whether these logics yield a feasible programming language.

As for mathematical sense, the answer is "yes", at least for the vast majority of programs one will see in practice. For positive programs, there is the van Emden - Kowalski [11] minimal model. This model is built on the Herbrand universe (the set of ground terms) and is the least model of the program, and also the least model of the completion. If the query is also positive, this is equivalent to the previous approaches; that is, q is true in the minimal model iff q is a logical consequence of the completion of the program iff q is a logical consequence of the naive reading to the program. But, these three approaches diverge as soon as we consider negative queries, or even try to interpret a "no" to a positive query as per negation-as-failure. In particular, this model gives formal support to our informal remarks above about $\{p :- p\}$ and the definitions of $t1$ and $t2$.

There is a natural extension to the minimal model in the case that the program is stratified, due to Apt, Blair, and Walker [1] and Van Gelder [28]. Stratified means that we can partition the set of all predicate symbols into strata, $P_0, P_1, \ldots$, indexed by natural numbers, in such a way that if $p \in P_n$ and q appears in the body of a clause defining p, then $q \in P_i$ for some $i \leq n$; furthermore, if q occurs negatively, then $i < n$. In particular, the predicates in P_0 have purely positive definitions. One may define a minimal model by first minimizing the predicates in P_0 as per van Emden - Kowalski; then fixing these and minimizing the predicates in P_1, and so forth.

There are, in fact, wider classes of programs, such as the locally stratified [25] ones, for which one can define a (2-valued) minimal model in a natural way. If you want to a semantics which includes *all* programs, it still seems natural to go to 3-valued logic. For example, consider

$p :- \textit{not}\ q.$
$q :- \textit{not}\ p.$

The only 2-valued models of this program or of its completion have exactly one of p, q true, and the other false, and there is no natural "least" one among these – so here it would be natural to have p, q both $\mathbf{u}$. Nevertheless, despite this pathology, one can make

sense, mathematically, out of the notion of a (2-valued) least model for the vast majority of programs occurring in practice.

Next, is there an efficient way to compute this model? Consider first purely positive programs, where van Emden and Kowalski [11] actually describes a bottom-up way of computing the least fixed-point by successive approximation. This naive bottom-up execution is inefficient, but there are many optimizations [5] which seem to be rather successful, at least in a database setting, where there are no function symbols. For example, in the case of transitive closure, a naive bottom-up evaluation would compute the full transitive closure, bottom-up, to evaluate a specific query, whereas the optimized versions will only compute as much of the transitive closure as is needed to answer the query.

These optimizations extend somewhat to stratified programs with no functions symbols [4], as well as to some situations with function symbols. However, in the worst case, the minimal model is highly undecidable [2], so that not only is there no efficient evaluation for it, there is no evaluation procedure for it at all. More specifically, for each n, there is a stratified program with n strata such that the set of supported answers is a complete Σ_n^0 set.

So, which is the correct semantics to use (say, in the case of stratified programs, where both approaches make sense)? It is easy to see that the completion semantics, which considers all models for the completion, is always weaker than the minimal model semantics, which picks out one such model. Declaratively, therefore, you would always prefer the least fixed-point semantics, because of its greater expressive strength. However, this expressive strength is meaningless unless coupled with an evaluation procedure. If you are programming in a deductive database setting, where there are good optimizations for computing least fixed-points, then of course you would opt for that semantics. However, if you are programming with complex data structures represented by lists or other terms, you should think in terms of the completion semantics, which, although weaker, gives a more honest appraisal of what your program is really saying.

§3. BUILTINS.

Logic programming often requires some use of arithmetic. This leads to further semantic problems. For example, Prolog usually contains the predicate *is*; the query $?-\ X\ is\ 3{*}4$ unifies X with 12. Of course, in theory, arithmetic could be axiomatized, and something like *is* could be defined in pure logic. We could think of the natural numbers as generated by the constant 0 and the successor function s, and axiomatize *is* by:

$plus(X, 0, X).$
$plus(X, s(Y), s(Z))$:− $plus(X, Y, Z).$
$times(X, 0, 0).$
$times(X, s(Y), Z)$:− $times(X, Y, T),\ plus(X, T, Z).$
$X\ is\ X$:− $numeral(X).$
$numeral(0).$
$numeral(s(X))$:− $numeral(X).$
$X\ is\ TERM1 + TERM2$:− $V1\ is\ TERM1,\ V2\ is\ TERM2,\ plus(V1, V2, X).$
$X\ is\ TERM1 * TERM2$:− $V1\ is\ TERM1,\ V2\ is\ TERM2,\ times(V1, V2, X).$

We might think of the numeral 3 as syntactic sugar for $s(s(s(0)))$. Then X *is* $3 * 4$ would eventually unify X with $s(s(s(s(s(s(s(s(s(s(s(s(0))))))))))))$, which we have decided to sugar to 12. This is very nice for pure theory; for example, a similar argument will establish that all computable functions on natural numbers can be programmed in pure Horn logic; but this is obviously grossly inefficient in practice – and we're ignoring the additional complexity of negative and floating point numbers. The real situation is that numeric terms are evaluated by the hardware to produce an answer. Prolog is not responsible for the correctness of that evaluation, but uses it as a black box. Can we develop a sound semantics to reflect the practical use of arithmetic in logic programming?

The way this is usually explained formally is that implicitly, the program contains, as an infinite database, all true ground statements about *is*, such as

$12\ is\ 3 * 4$
$36\ is\ 3 * 3 * 4$

and so forth. Numbers are now just unstructured constant symbols. Then the query $?-\ X\ is\ 3 * 4$ is simply answered as if by unification with the (implicit) database. This semantics correctly reflects the situation that the Prolog *is* only evaluates expressions – it does not include an axiomatization of arithmetic or decide the truth or falsity of quantified numeric formulas.

This point of view is a good first approximation to the semantics, but there are still problems amalgamating it with our formalization of negation-as-failure. We discuss the completion semantics, but attempts to amalgamate it with the stratified negation semantics would wind up with the same problem. What is the completed definition of *is* ? Naively, it would be

$$\forall X \forall Y (X\ is\ Y \Leftrightarrow (X = 12 \wedge Y = 3 * 4) \vee (X = 36 \wedge Y = 3 * 3 * 4) \vee \cdots)$$

That is, it would be an infinitely long statement. Although these are often treated in mathematical logic, their use as a basis for Prolog semantics leads to two problems, which I would call *false advertising* and *non-effectiveness.*

By "false advertising", I mean that this semantics promises more than Prolog delivers. For example, this semantics says that the answer to the query $?-\ X\ is\ X + 1$ is "no", since it fails to unify with any clause in the implicit database (or, since $\neg \exists X (X\ is\ X + 1)$ is a logical consequence of the completion). In fact, this query produces an error in Prolog, since Prolog only does numerical evaluation, and has no mechanism for reasoning about number theory.

I will describe below a modification of the completion semantics which does not logically decide the truth of $\exists X (X\ is\ X + 1)$. Of course, this new approach has a very weak expressive power compared to the naive approach using infinitary formulas. At first, this seems bad, but of course it is really good, since the point of a declarative semantics is not to be as strong as possible, but to have the property that the formal declarative meaning we assign to a program is a good approximation of what the system can deliver.

By "non-effectiveness", I mean that this semantics promises more than than any logic programming can deliver *in principle.* Although Prolog doesn't, a logic programming

system could reason about linear equations and inequalities; in fact the system CLPR [14] does indeed do just that. This is possible because there happens to be an efficient decision procedure for simple statements about + on the real numbers. But this does not extend to + and * together, since this would allow questions about solutions to diophantine equations. So, for example, as τ ranges over terms built from variables, +, and *, the set of queries of the form ?– $0\ is\ \tau$ to which the naive semantics supports a "no" is not recursively enumerable. Here I am thinking of numbers as being natural numbers; but the same problems happen if you think of numbers as being real or complex, since you can define natural numbers anyway by

$natural(0)$
$natural(X)$:– $natural(Y),\ X\ is\ Y + 1$

and then relativize queries to *natural*.

A more honest approach is to say that the completed definition of *is* is the following infinite set of finite statements. For each ground numeric term, τ whose value is n, we have the statement $\forall X(X\ is\ \tau \Leftrightarrow X = n)$. This does support the use of *is* in Prolog, where it is supported only for specific ground numeric terms. It does not decide the truth value of $\exists X(X\ is\ X + 1)$. Since this is a decidable (but infinite) set of *finite* statements, the set of all its logical consequences is recursively enumerable. This may be viewed as a restriction on negation-as-failure; that is, ?– $X\ is\ X + 1$ does not return "no", even though it fails to unify with any clause in the database. In the formal semantics, $\exists X(X\ is\ X + 1)$ has truth value **u**, not **f**.

There are a number of other built-ins provided by the system, such as lexical term comparison and other numerical operations, whose natural declarative interpretation would be as an infinite set of clauses which is implicitly contained in every program. A general approach for handling these is described in [19]. The revised completion is always an infinite set of finite statements. The idea is the following. If P is a possibly infinite program, we accompany it by a set, D, of positive literals, which we call the "declarations". We assume that each element of D unifies with only finitely many heads of clauses in P. The informal idea here is that we are applying negation-as-failure only to literals which are instances of members of D. In the usual "default" case, for an n-place predicate p defined by the user with finitely many clauses, D contains $p(X_1, \ldots, X_n)$ – that is, negation-as-failure applies to all uses of p. In the case of *is*, D contains all $X\ is\ \tau$, where τ is a ground term. Thus, negation as failure does not apply to $X\ is\ X+1$, which is not an instance of any declaration, but it does apply to to $[A, B]\ is\ 2 + 3$; in the formal semantics, $\exists A, B([A, B]\ is\ 2 + 3)$ has truth value **f**.

The completion is now an infinite set of finite statements, so that (assuming some decidability conditions on D and P), the set of its logical consequences is recursively enumerable. The completion always contains Clark's equality axioms and the sentences obtained by the naive reading of each clause in P. Next, for each $\alpha \in D$, it contains an "iff" statement similar to the Clark completion. Say α has variables $X_1, \ldots, X_n$, and β :– ϕ is in P, with β and α unifiable, with mgu σ, where each $X_i\sigma$ contains only variables other than the X_j. Say $\tau_i = X_i\sigma$. The *normalized clause* corresponding to α and β :– ϕ is α :– ψ, where ψ is

$$\exists Y_1 \ldots \exists Y_k(X_1 = \tau_1 \wedge \ldots \wedge X_n = \tau_n \wedge \phi\sigma) \quad ,$$

where $Y_1 \ldots Y_k$ enumerate the variables other than the X_i which occur in the τ_j and in $\phi\sigma$. Let the m normalized clauses corresponding to α be $\alpha :- \psi_i$, for $i = 1, \ldots, m$; then the completion statement corresponding with α is

$$\forall X_1 \ldots X_n(\alpha \Leftrightarrow \psi_1 \vee \ldots \vee \psi_m) \quad ,$$

If $m = 0$, we replace the empty disjunction by *false*. Although we have written this as a $\Leftrightarrow$, note that the $\Leftarrow$ direction already follows from the naive reading of the clauses in P.

The right way to formalize the semantics of CLPR is to use the same sort of approach we use for Prolog, but add in the axioms for densely ordered Abelian groups.

§4. CONTROL.

What is control, anyway? This is problematical even for purely positive programs with positive queries, where all approaches to the declarative semantics seem to agree. Consider the program:

$p(X, Y) :- p(X, T).$ % 1
$p(X, Y) :- p(V, Y).$ % 2
$p(a, b).$ % 3

Then $\forall A \forall B p(A, B)$ is a semantic consequence of the program (even without passing to the completion), and by the completeness for SLD, the identity substitution is returned by SLD to the query $?- \; p(A, B)$. This answer is in fact obtained by using the three clauses in the order 1,2,3. However, it is easy to see that there is no way of ordering the clauses to have this answer returned in Prolog. In the order given, Prolog will keep applying clause 1, and never return.

In this particular example, there are more complex control schemes, not part of standard Prolog, which would work. For example, one could backtrack as soon as a subgoal is subsumed by a parent goal. It would be easy to come up with more complicated examples along these lines which defeats these schemes as well. Also, in practice, there are dangers to incorporating such more intelligent control strategies. Besides the fact that subsumption checks would decrease efficiency, real Prolog programming makes use of the fact that the control is simple-minded depth-first search, and therefor predictable. With more complex control mechanisms, the programmer might loose the necessary direct contact with the deduction process. That is, we know Prolog is stupid, but that also means that we know exactly what it will do.

Such examples cast doubt on the relevance of mathematical completeness proofs to real programming. What exactly does the completeness of SLD mean in practical terms? We would like to say that it means that any supported answer will be obtained with a suitable tweaking of the control, but we don't really know a definition of "control" which makes this statement into a theorem, other than implementing some sort of breadth-first search through the SLD tree, which, of course, negates the practical advantages of Prolog-style search.

§5. ORDERING, LISTS, AND STREAMS.

The point of declarative semantics is that the correctness of any answer obtained to a query is guaranteed to be correct – that is, to follow logically from the program. However, often in Prolog programming, we depart from the database query mode and write a program which calls for a specific side effect, such as a *print* statement. As a simple example, say we have a database of people and facts about them. So, our database contains, among other things, facts such as $person(alice)$, $person(bob)$, and so forth. Now, say we want a simple routine which prints out all the people in the order listed. This is standard in Prolog – for example, we could call ?– *printall*, defined by:

$$printall :- person(X), print(X), nl, fail.$$
$$printall.$$

Or, more primitively, we could just call ?– $person(X)$, and type semicolons at the terminal, safe in the knowledge that we will get the answers in the order in the database. In either case, the problem is that the additional information contained in the *order* in which the facts get printed, which is part of our informal understanding of the correctness of the answer, is not expressed in the logic. Another way of thinking about it is that if *bob* gets printed before *alice*, we would like to say that our interpreter is incorrect, but our treatment of declarative semantics gives us no way of expressing this incorrectness. For the query ?– $person(X)$, *bob* and *alice* are logically correct answers, and for the query *printall*, which contains no variables, the "answer" is, declaratively, just "yes" by the second clauses, no matter what *print* means. In either case, the incorrectness of printing *bob* first would have to be explained in terms of the *side effect* of printing the answer.

Of course, we would only expect the logic programming paradigm to apply in cases where all we want from the computer is the answer to a question. There are programming situations in which we really want the program to *do* something – e.g., control a robot – presumably these situations do not fall under the heading of logic programming *at all.* But what I am looking at is situations where we really only want the answer to a question – i.e., something declarative, but the only convenient way to get it in standard Prolog is to program with side effects.

Note the difference between this and the previous section. We have already seen the relevance of ordering to *control* – how quickly the computer *finds* the answer – but that is a different issue. Now we are looking at situations where the order in which the answers are returned is involved in our informal idea of *correctness*, and we are asking how to make this informal idea formal.

We could begin by using concepts already in Prolog. We could declare, in our semantics, that the order of separate answers has no declarative meaning (that is, the implementation is not responsible for this), and demand that if an ordered sequence is important, the user call for it as a *list.* Then, to get the list of answers to a query, you can use the standard Prolog *bagof*; in the example under consideration now, you can simply query ?– $bagof(X, person(X), L)$. Informally, this should unify L to the list of objects, X, satisfying $person(X)$, where the ordering is in the *standard backtracking order* in which the answers would be computed in SLD.

Since the backtracking order depends on the order of the clauses, this *bagof* is not usually considered to be declaratively meaningful (see [24]). However, one can in fact make sense of *bagof*. More precisely, for each query clause, ϕ with variables $X_1, \ldots, X_n$, one can augment the program by defining a one-place predicate, $bagof_\phi(L)$, which "says" that L is the list of n-tuples satisfying ϕ. Of course, the definition of $bagof_\phi$ depends on the order of the program. At first, this seems strange, but something like that is inevitable. Our program said, $person(alice) \wedge person(bob)$. If you want the logic to imply that Alice is a person *before* Bob, it seems to me that you have three options. (1) Have $\wedge$ non-commutative – this seems rather pathological. (2) Require the programmer to explicitly program in this order by a statement such as $ordered_persons[alice, bob]$. (3) Have a predicate such as *bagof* whose completed definition depends on the ordering in the program. Option (2) is the logically most straightforward one, but might make the program awkward to edit and update, since one would have to keep *ordered_persons* consistent with *person*. One may then think of (3) as a shorthand for (2).

These options correspond exactly with the ones available with negation. (1) You can take non-monotonic logic as the basic logic – this also seems pathological. (2) The programmer can explicitly assert the completed definition – this would make programming rather awkward. (3) (what we actually do) You can consider the program as shorthand for its completion. The completed definition of a predicate such as *person* depends already on all the clauses in the definition of *person*. The added complication is just that the definition of $bagof_{person}$ depends on the ordering of these clauses as well.

Nevertheless, since the reader may be bothered by such constructions, in further examples in this section, I will actually write a predicate calling for a list, along with the *bagof* definition. The point I wish to make is that either way you do it, other semantic issues are raised, involving the related problems of lazy evaluation and infinite data structures.

If there are just a few answers and the order has declarative importance, it's not too hard to explicitly call for a list of answers. But, what happens if there is a long, or possibly infinite list of answers? Although there are theoretical ways out of these problems, it is not completely trivial to design the theory to correspond with a practical solution.

As a trivial database-style example, say you have your list of 1000 employees, ranked in order of decreasing competence, and you wish to choose 2 distinct ones to form a committee. Of course, you want the pairs printed out with the most competent pairs first – first you maximize the competence of the weaker member. That is, if the people are $a, b, c, d, \ldots$, you want the pairs, $(a, b), (a, c), (b, c), (a, d), (b, d), (c, d), \ldots$. You don't wish them all printed out, but you may wish to backtrack through the first few to think about compatibility. Now that we have seen that the people should really be stored as a list, it's easy to write a predicate, $pair(X, Y)$ which gives you all these pairs in its natural backtracking order. Then $?-\ bagof((X, Y), pair(X, Y), L)$ will unify L with the list of these in that order. If you don't trust the declarative meaning of *bagof*, or if you realize that you can do it more efficiently using the fact that the people listed are all distinct, you might explicitly write a predicate, $all_pairs(L)$, which returns this list:

$person(a).\ person(b).\ person(c).\ person(d).$ % and 996 others
$people([a, b, c, d, \ldots]).$
$pair(X, Y) :- people(P),\ segment(S, Y, P),\ segment(Anything, X, S).$
$segment([], X, [X|Tail]).$
$segment([A|Rest], X, [A|Tail]) :- segment(Rest, X, Tail).$
$all_pairs(L) :- people(P), ap(P, L).$
$ap([], []).$
$ap([X|Rest], L) :- ap1([X|Rest], [X|Rest], Rest, L).$
$ap1(P, [X|Xs], [Y|Ys], L) :- X = Y \rightarrow ap1(P, P, Ys, L);$
$\quad (ap1(P, Xs, [Y|Ys], M), L = [(X, Y)|M]).$
$ap1(P, Xs, [], []).$

Logically, there is no problem here, and in practice it's not very hard to write the program. However, it is clear that if this theory is to find practical use (that is, people will use $all - pairs$), it is necessary to have some sort of lazy evaluation; otherwise, users would have to wait until the whole list of 500500 pairs is developed in order to just see the first few pairs. At first sight, this seems only a matter of designing a friendly user interface. Let us say that, if a query $?- p(L)$ results in L being unified to a large list, the printer could print the first 10 elements of the list as soon as they are known, print *more*, and wait for the user to press a key. Let us say, if the key is 'q', no more is printed, and the computation terminated, whereas if a carriage return is entered, the next 10 elements of the list are printed. Then L is developed lazily, so that you see the first 10 lines as soon as they are known. Similar things could be done with other large data structures.

This idea that a list may be used to represent a stream of answers is of course not new. For example, it is used in concurrent prologs as a way of describing process communication. There too, the list is computed lazily, as its consumer needs it, and you see the elements of the list as they are being computed. Typically, the list remains incomplete during the entire computation, and its elements are communicated to the user via explicit *print* statements.

However, once we have decided on lazy evaluation, we do have a semantic problem – namely, our semantics does not take into account the possibility of process failure. In the particular case at hand, you might query $?- all_pairs(L)$, and retrieve the first ten elements, $L = [(a, b), (a, c), (b, c), (a, d), (b, d), (c, d), (a, e), (b, e), (c, e), (d, e) \cdots\ more\ \cdots$, and then type a 'q' saying you had seen enough. Here, the intended declarative meaning is clear; you have just seen the first 10 elements of the solution to the query. However, in general, it is not clear that any declarative meaning be given to a stream of data, before you have seen the entire stream; since there in fact need not be any answer at all – that is, the deduction could in fact fail if it were carried out. In this example, this would happen if we had omitted the last line in the program, in which case the correct answer to the query would be "no". Thus, it seems that unless it is restricted in some way to queries for which we can prove in some way that there is an answer, it is not clear how to give a declarative meaning to lazy evaluation.

In concurrent prologs, the fact that such use of streams and lazy evaluation is in fact a more fundamental feature of the programming style, and not merely an add-on as we are advocated, means that these semantic problems are relevant to concurrent prologs at

a much more basic level.

Besides the question of failure, an additional complication is due to the fact that in many programs, the answers are either numbers or compound terms and the set of answers in infinite. For a simple numeric example, consider the following (not very efficient) program which backtracks through primes:

$prime(2).$
$prime(Y) :- prime(X), next(X, Y).$
$primelist([2|Tail]) :- pl(2, Tail).$
$pl(P, [Q|Tail]) :- next(P, Q), \quad \% \ print(Q), nl,$
$\quad pl(Q, Tail).$

Here, we leave it to the reader to program a $next(X, Y)$ which, given a positive integer X, unifies Y to the next largest prime. This is intended to work with queries such as $?- \ prime(X)$, which will unify X successively with $2, 3, 5, \ldots$. Again, to make declarative sense out of the order (an interpreter which returns 3 before 2 is incorrect), we are asking the user to call for $?- \ bagof(X, prime(X), L)$; then, since the reader may still be bothered by the $bagof$, we are writing it out explicitly a $primelist(L)$ which unifies L to the list of all primes. Of course, here the list is never completed, and standard Prolog simply fails to return with the query $primelist(L)$, although one could see this list printed as it develops by including the (commented out) *print* statement (again, this has no declarative meaning). In concurrent prologs, one might take this list and feed it to another process (perhaps one which is computing primitive roots of unity modulo the various primes).

In this example, there is no process failure, but it is not clear whether any declarative meaning can be given to an infinite stream. Let ϕ be the statement $\exists L(primelist(L))$. With the "standard" semantics we have described here, neither ϕ nor $\neg\phi$ is a semantic consequence of the completion of the program. In the Herbrand universe, ϕ is false. However, there are other models in which ϕ is true. One such model is described in Lloyd [21], based on earlier work by Andreka, van Emden, Nemeti, and Tiuryn; here, too, the goal is to find some way to describe perpetual processes in some declarative way. Here, one starts with the Herbrand universe and *completes* it to a universe of all infinite terms. If one takes this completed universe as the "standard domain", then ϕ is true here, and the L satisfying $primelist(L)$ is the infinite list of primes. Of course, by fixing a standard domain (which is now even an uncountable set), the semantics is no longer effective. Also, we still have the same problem with process failure – that is, seeing part of the infinite list, evaluated lazily, really has no declarative meaning. Suppose *next* were instead some arithmetical predicate such that $next(X, Y)$ picked out the next prime only if $X < 10^9$, and otherwise would be simply false. Then ϕ would simply be false in all models of the completion, and the first few numbers we saw would simply be the beginning of a failed attempt to satisfy ϕ.

In standard sequential Prolog, unlike in concurrent prologs, the notion of an incompleted (and potential list) is not of fundamental importance, we have the following "way out" of our semantic problems. Disregard the above comments on lazy evaluation and lists of answers. Instead of formalizing $bagof$, we can, for each query clause ϕ, variables $X_1, \ldots, X_n$, and natural number k, define a predicate $answer_{\phi,k}(X_1, \ldots, X_n)$ which says that $(X_1, \ldots, X_n)$ is the kth answer to ϕ. We could then say that this is the declarative

meaning the the kth answer obtained via backtracking. For example, if $p(X)$ is defined by the two clauses $p(X) :- r(X,Y)$ and $p(X) :- q(X)$, then we would have

$$answer_{p,0}(X) :- answer_{r,0}(X,Y).$$
$$answer_{p,0}(X) :- not\ some_answer_{r,0}, answer_{q,0}(X).$$
$$some_answer_{r,0} :- r(X,Y).$$

Unfortunately, this definition, although formally meaningful, suffers from the Ph.D. effect, and thus won't sell. That is, the logic is rather hard to understand, and it introduces complex nested negations even to simple positive programs. One could argue that all it is doing is formalizing the procedural semantics; of course, the procedural semantics of any programming language could be expressed within Prolog. The point of logic programming is that the program has a *simple and clear* declarative meaning. Programs which call for a list explicitly, as in the previous examples, do seem simple and clear enough, but we ran into trouble making declarative sense of them.

§6. CONCLUSION.

As I said in the Introduction, this survey raises more questions than it answers. As of now, one must say that the jury is still out on whether true logic programming is feasible. If it is feasible, we have a truly new form of programming. If not, of course, Prolog-style programming is still important, but it is what I would call logic**ish** programming. One may take the point of view that Prolog is simply a procedural language, which has borrowed some ideas from logic, such as terms and unification, and which has built-in backtracking and symbol processing features, which are useful for some sorts of problems. Similarly, we view Lisp as a procedural language which has borrowed some convenient ideas from functional languages. In that case, there is no need for a formal discussion of declarative semantics. With this point of view, there are still many important issues to discuss about Prolog (compiler optimizations, WAMs, architectures, etc.), but they do not involve declarative semantics. Under this point of view, the issues raised in this survey are mainly irrelevant.

Even granting the importance of declarative semantics, there are some important issues related to it which have not been discussed at all. Perhaps many such will come to mind to the reader, but the following two seem especially glaring:

System features. I mean here things like *assert*, *retract*, *consult*, and system calls. Something like this seems to be necessary if we are to interact with the system, load files produced by a text editor, etc. It seems hopeless to try to handle these declaratively. Perhaps the cleanest way to handle them would be to separate them from the logic and modify the Prolog front end to distinguish more clearly between queries and commands. So, for example, any "query" which called a system call or a *consult* would be processed as it is now, but the reply would contain a warning that there is no logical importance to the answer. This would handle the "honest" use of such features. This would not handle the situations where *assert* and *retract* are used in computations as a kludge for assignable variables (i.e., the Pascal "x := 5" gets kludged to the Prolog $assert(x(5))$). In many simple examples, this sort of kludge can be omitted by a little thought, but, for example,

graph algorithms which operate by marking and unmarking nodes are notoriously hard to write efficiently in "pure" Prolog. It remains a subject for further research as to whether the need for these database kludges could be eliminated by making the language more expressive.

Other approaches. There are a number of well-known interactive theorem provers which seem to conform to the "logic + control" paradigm, but which don't resemble Prolog at all. That is, they are implementations of logic, guaranteeing correctness, but they can interact with the user, who provides intelligence (control). Examples are the Boyer-Moore Theorem Prover [6], Constable's Nuprl [10], and a number of Resolution-based systems developed at Argonne (see [29]). One might say, however, that these are in a different position as is Prolog. Unlike Prolog, their logical foundations are unassailable; that is, their declarative meaning is clearly and and unambiguously expressed within standard predicate logic. However, also unlike Prolog, they are being advocated not as general-purpose programming languages, but rather as tools for verification or aids in discovering theorems. The claims for success for these systems are the kinds of statements that can be verified, not how quickly they can do arbitrary computations. Prolog, by trying to be a viable general-purpose programming language, has a much harder task to justify its existence.

REFERENCES.

[1] Apt, K. R., Blair, H. A., and Walker, A., Towards a Theory of Declarative Knowledge, in: J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufman, Los Altos, 1988, pp. 89-148.

[2] Apt, K. R. and Blair, H. A., Arithmetic Classification of Perfect Models of Stratified Programs, in: R. A. Kowalski and K. A. Bowen (eds.), *Logic Programming* (Proceedings of the Fifth International Conference and Symposium), MIT Press, 1988, pp. 765-779.

[3] Apt, K. R. and van Emden, M. H., Contributions to the theory of Logic Programming, *JACM* 29:841-862 (1982).

[4] Balbin, I., Port, G. S., and Ramamohanarao, K., Magic Set Computation of Stratified Databases, Technical Report 87/3, University of Melbourne, 1987.

[5] Bancilhon, F. and Ramakrishnan, R., Performance Evaluation of Data Intensive Logic Programs, in J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufman, Los Altos, 1988, pp. 439-517.

[6] Boyer, R. S., and Moore, J. S., *A Computational Logic Handbook*, Academic Press, 1988.

[7] Cavedon, L., and Lloyd, J. W., A Completeness Theorem for SLDNF Resolution, Technical Report CS-87-06, University of Bristol, 1987, to appear in *J. Logic Programming.*

[8] Clark, K. L., Negation as Failure, in: H. Gallaire and J. Minker (eds.), *Logic and Databases*, Plenum Press, New York, 1978, pp. 293-322.

[9] Clocksin and Mellish, *Programming in Prolog*, Springer-Verlag, 1981.

[10] Constable, R. L. *et al*, *Implementing Mathematics with the Nuprl Proof Development System*, Prentice-Hall, 1986.

[11] van Emden, M. H. and Kowalski, R., The Semantics of Predicate Logic as a Programming Language, *JACM* 23:733-742 (1976).

[12] Fitting, M., A Kripke-Kleene Semantics for Logic Programs, *J. Logic Programming* 2:295-312 (1985).

[13] Gabbay, D. M., Modal Provability Foundations for Negation by Failure, *to appear.*

[14] Jaffar, J. and Lassez, J-L., Constraint Logic Programming, *Proceedings Fourteenth ACM Symposium on Principles of Programming Languages*, Jan. 1987, pp. 111-119.

[15] Kowalski, R. A., Algorithm = Logic + Control, *Comm. ACM* 22:424-436 (1979).

[16] Kunen, K., Negation in Logic Programming, *J. Logic Programming* 4:289-308 (1987).

[17] Kunen, K., Signed Data Dependencies in Logic Programs, Technical Report #719, University of Wisconsin, 1987, to appear in *J. Logic Programming.*

[18] Kunen, K., Some remarks on the Completed Database, in: R. A. Kowalski and K. A. Bowen (eds.), Logic Programming (Proc. Fifth International Conference Symposium), MIT Press, 1988, pp. 978-992.

[19] Kunen, K., Partial Negation-as-Failure, *in preparation.*

[20] Lassez, J-L., and Maher, M. J., Optimal Fixedpoints of Logic Programs, *Theoretical Computer Science* 39:15-25 (1985).

[21] Lloyd, J. W., *Foundations of Logic Programming*, Second Edition, Springer-Verlag, 1987.

[22] Miller, D. A., A Logical Analysis of Modules in Logic Programming, *J. Logic Programming* 6:79-108 (1989).

[23] Mycroft, A., Logic Programs and Many-valued Logic, *Proc. of Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science* 166:274-286 (1984).

[24] Naish, L., Negation and Control in Prolog, Lecture Notes in Computer Science #238, Springer-Verlag, 1986.

[25] Przymusinski, T. C., On the Declarative Semantics of Deductive Databases and Logic Programs, in: J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufman, Los Altos, 1988, pp. 193-216.

[26] Przymusinski, T. C., Non-monotonic Reasoning vs. Logic Programming: A New Perspective, in: D. Partridge and Y. Wilks (eds.), *Formal Foundations of Artificial Intelligence*, Cambridge University Press, London, 1988, to appear.

[27] Thom, J. A., and Zobel, J. (eds.), NU-Prolog reference manual (ver. 1.1), Technical Report 86/10, Machine Intelligence Project, University of Melbourne, 1986.

[28] Van Gelder, A., Negation as Failure Using Tight Derivations for General Logic Programs, in: J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufman, Los Altos, 1988, pp. 149-176.

[29] Wos, L, *Automated Reasoning – 33 Basic Research Problems* Prentice Hall, 1988.