I. INTRODUCTION

1. Procedural vs. Descriptive Languages

Languages for computation are often organized into two types, procedural and descriptive. Procedural languages, such as FORTRAN, ALGOL, COBOL, SNOBOL, PL/1, are directly concerned with sequences of actions to be performed by a computer, they have an imperative tone, their parts are often thought of as commands. The process of writing compilers and interpreters for such languages is fairly well understood, and very successful practical results have been obtained. A primary weakness of such languages is that it is often very difficult to understand the relationship between a program and the problem which it solves (i.e., the function which it computes).

Descriptive languages, such as the mathematical languages of functional and relational expressions and the predicate calculi, are oriented more toward defining mathematical objects, functions and relations without direct reference to computational techniques. These languages are declarative in tone, and their parts have the forms of formulae and assertions. Mathematical logic provides us with an elegant semantics for these languages, which illuminates very well the correspondence between formulae and the objects which they describe. Descriptive languages are not as popular for practical computation as procedural languages, partly because the efficient implementation of such languages may be very difficult. The results of this paper show how practical interpreters may be derived for a large class of mathematical languages in such a way that the interpreter precisely satisfies the mathematical semantics of the language.

Mathematical logic contains one study which seems relevant to computing - proof theory. The definitions of one or more functions may be treated as a set of axioms or postulates; and computing the value of an expression E to be the constant c is basically the same as proving that E=c. Unfortunately, traditional mathematical proof theory tells us how to generate correct proofs, but not how to efficiently prove assertions of the particular form above. Even an efficient decision procedure which recognized true statements in a language would not allow us to efficiently compute the value of E above, since we apparently would have to search for the appropriate constant symbol c. Proof theory may yet be helpful, since if the computation of a value for E closely mimics established proof techniques, we will be confident at least that any computed answer is correct. Also, we may be able to apply traditional completeness results to show that our computation halts whenever

it should. So, we approach the design of interpreters for mathematical languages by trying to efficiently generate certain proofs. Theoretical results of this nature have been obtained for the predicate calculus [vEK76], but in order to produce efficient practical methods, we limit our attention to the weaker language of equational logic.

2. Equational Logic and Computation

The language of equational logic contains only assertions of the form $Exp_1 = Exp_2$. The language of equations is much less expressive than the predicate calculus, but it is sufficient for describing many interesting functions in a natural way. All of LISP 1.0 and the Red languages, and large parts of Lucid and APL may be described by sets of equations. In mathematics, the lambda calculus, the combinator calculus and theories of recursively defined functions may be viewed as special systems of equations.

The general problem of computing in equational systems may be stated as follows:

> Given a set of equations A (axioms), and an expression E, find an expression $F \in N$ such that E = F is a consequence of A, where N is some fixed set of simple expressions.

For instance, A might be an infinite set of equations defining + and * for the integers, and N might be the set of integer constant symbols $N = \{...,-3,-2,-1,0,1,2,3,...\}$. Then, given 8+3*5-23, we may find the equivalent expression 0. Solutions to the computing problem depend critically on the sets A and N, and the way that they are presented (A and N are usually infinite, so we work with finite descriptions rather than the actual sets).

Suppose that, for each equation $Exp_1^i = Exp_2^i$ in A, Exp_2^i is in some sense simpler or clearer that Exp_1^i. For example, in the equation 8+3*5-23 = 0 the expression 0 is intuitively simpler because it is a single symbol. In the recursive definition of the factorial function, F(x) = If x=0 then 1 else F(x-1)*x, the right-hand side is longer than the left, but it is clearer in the sense that the value of the function at 0 is immediately apparent, and some information about the value at other integers n (e.g., that the value at $n \neq 0$ is divisible by n) can be easily seen. The left-hand expression F(x) shows none of this information. Axioms of this type may be thought of as reduction rules allowing the reduction of the left-hand expression to the right-hand expression.

In such cases, it is reasonable to process an expression E_0 by successively reducing subexpressions of the form Exp_1^i to the equivalent

but simpler or clearer forms Exp_2^i, generating a sequence (E_j). If some E_n has no subexpression of a form Exp_1^i (that is, E_n cannot be reduced further) it is said to be in normal form. This work is concerned with the computing problem for sets of ordered equational axioms A, where $N = \{F \mid F \text{ is in normal form}\}$. The essential formalism for this study is the Subtree Replacement System (SRS) of Brainerd [Br69] and Rosen [Ro73]. Subtree Replacement Systems may be used to formalize important proofs in many equational theories, and the formal versions of these proofs may be implemented straightforwardly by standard programming techniques using pointer structures.

The main contribution of this work is to develop sufficient conditions on Subtree Replacement Systems under which:

1) For each expression E there is at most one normal form F equivalent to E, and F may be found by reducing E;
2) a large class of computations succeeds in finding the normal form F for E whenever such exists;
3) a simple computational strategy for finding F is optimal.

These results are applied to the study of the lambda calculus, the combinator calculus, recursive equations, LISP and Lucid, producing some new theorems and reproducing some old ones in a uniform way. The application to LISP suggests an interpreter with several provable advantages over traditional LISP interpreters. Hopefully, the study of reduction in Subtree Replacement Systems, begun here and in [Ro73], may serve as the basis for a rigorous and practical computing theory for equational logic which may yield a uniform methodology for developing interpreters for many descriptive languages.