Intensional Programming*

A. A. Faustini William W. Wadge

Abstract

Imperative programming languages such as FORTRAN and Pascal have always played an important rôle in problem solving. In this paper we present a purely declarative programming language which we believe to be much more problem oriented than the conventional imperative languages. The language Lucid is based on intensional logic, which is itself a relatively new branch of mathematical logic. Intensional logic is concerned with assertions and expressions whose meanings depend on an implicit context. Natural language abounds with these kinds of assertions and we use natural language all the time for problem solving.

A number of Lucid programs that solve problems in the area of numerical analysis are included. In particular, programs are included to solve two partial differential equations, namely Laplace’s equation and the heat conduction equation.

1 Introduction

Computer scientists and logicians share a common interest in symbolic languages, though for different reasons. Logicians use formal languages to clarify basic foundational questions of mathematics. They are concerned primarily with discovering what can be stated or proved in principle—without regard to questions of efficiency. Computer scientists, on the other hand, use languages as practical tools: specification languages for stating problems in unambiguous terms, and programming languages for expressing mechanical solutions.

The languages of the logicians (such as the predicate calculus) certainly deserve the adjective “formal”. They are powerful but extremely simple, and both syntax and semantics are precisely and unambiguously defined.

The languages of the Computer Scientists are, by the same criteria, rather less successful. The high-level languages (beginning with FORTRAN) have indeed borrowed many important concepts from mathematical logic; but nevertheless they can hardly be considered as formal systems. Even the more modest languages (such as Pascal) are extremely complex, and only the syntax is really clearly defined. The semantic definitions (language manuals) use informal, anthropomorphic terminology, and explain almost everything in terms of the actions of some (vaguely described) machine. The various language features are usually dealt with in isolation, and the results of the interactions are often completely unspecified (or even inconsistently specified). Under these circumstances, it is extremely difficult to produce software that is correct and reliable, and impossible to prove such properties formally.

Computer scientists are well aware of this highly unsatisfactory state of affairs, and a series of efforts have been made to base programming languages more directly on the languages of mathematical logic. It is in fact possible to extend and adapt (say) the λ-calculus to produce a programming language which is at the same time a formal system. The language LISP (invented by McCarthy and others around 1960 [7]) was originally intended to be just such an adaptation of the λ-calculus, but it diverged in its treatment of variable binding and higher-order functions. Shortly after, however, Peter Landin corrected the discrepancies in LISP and produced ISWIM, the first true functional language [6].

*Published as University of Victoria Computer Science Research Report DCS–55–IR. Slightly revised version of the paper that appears in the proceedings of the Second Conference on Languages and Problem Solving, Applied Physics Laboratory, Johns Hopkins University, June 1986.
These ‘logical’ programming languages such as ISWIM are in many respects vastly superior to the more conventional ones. They are much simpler and better defined and yet at the same time more regular and much more powerful. These languages are notationally closer to ordinary mathematics and are therefore much more problem-oriented. Finally, programs are still expressions in a formal system, and are still subject to the rules of a formal system. It is therefore much easier to reason formally about their correctness, or to apply meaning-preserving transformations. With these languages, programming really is a respectable branch of applied mathematical logic.

Unfortunately, these logic-based languages have proved to be difficult to implement efficiently. The problem is that they are not well suited to the sequential, storage-based form of computation offered by conventional ‘von-Neumann’ machines. As a result interest in declarative languages declined soon after the promising initial work by McCarthy and Landin.

Recently, however, new advances in digital electronics (VLSI) have made it possible to consider radically different architectures more suited to declarative (as opposed to imperative) languages. This in turn has reawakened interest in declarative languages, and brought about a series of new ‘second-generation’ declarative languages, such as PROLOG [2] and KRC (recently renamed Miranda) [9].

In this paper we will discuss Lucid, one of these second-generation declarative languages. Lucid is based not so much on the classical logical systems, as on intensional logic—itself a relatively new branch of logic [8] which reached maturity during the period (1965–75) in which declarative programming languages were in eclipse.

2 Intensional Logic

Intensional logic is concerned with assertions and other expressions whose meaning depends on an implicit context.

This type of logic was originally developed to help understand natural languages, in which such expressions abound. For example, the expression

five degrees less than yesterday’s temperature

obviously denotes a numerical value. This value clearly depends on a numerical quantity called temperature. It also depends on the time of utterance, and on the place—even though there is no explicit reference to either of these two parameters.

Stranger still is the way in which the value of the expression depends on that of temperature. If we look outside and see that the thermometer reads (say) 35, we cannot conclude that the value referred to above is 30; in fact, we cannot conclude anything about the value, because the value of the expression on any given day depends on the value of temperature on the previous day.

In other words, the expression quoted seems to correspond to a mathematical expression of the form \( y(t) - 5 \), where \( t \) is temperature and the function \( y \) corresponds to yesterday’s. It is obvious, however, that there is no function \( y \) from integers from integers which makes the value of the mathematical expression corresponding to that of the cited phrase. The expression in question seems to violate the basic principle of referential transparency: that the meaning of a whole expression depends only on the meaning of its parts.

For many years examples such as these were considered by logicians to be simply further evidence of the nonmathematical and illogical nature of natural languages. It was recognized that expressions like the one given could be translated into a ‘respectable’ language like the predicate calculus, but only by introducing variables which refer explicitly to those factors (such as place and time) which implicitly determine the meaning of the natural language expression.

It is only comparatively recently that logicians discovered a more direct and natural way to capture formally these ‘context-sensitive’ operators and expressions. The solution to this long-standing puzzle is based on the distinction between what is called the extension and the intension of expressions [8].

The extension of an expression is the value in a given context; for example, a truth value or (as with the example given above) an integer. A natural-language expression can obviously
have different extensions in different contexts. The *intension* gathers all these different extensions together, and captures the way in which the extensions depend on the contexts. In other words, the intension is the function which assigns to each context the value of the expression in that context.

The various paradoxes disappear once we realize that the intension is the true meaning of a natural language expression. Consider again the temperature cited earlier. The intension of *temperature* is essentially a table giving the temperature on each day in question; part of the table might look like this:

```
.. ... .. ..
30 Dec 84 23
31 Dec 84 21
 1 Jan 85 25
 2 Jan 85 23
 3 Jan 85 19
.. ... .. ..
```

The intension of *five degrees less than yesterday's temperature* is a similar table, which might look in part like the next table.

```
.. ... .. ..
31 Dec 84 18
 1 Jan 85 16
 2 Jan 85 20
 3 Jan 85 18
 4 Jan 85 14
.. ... .. ..
```

It is now easy to see that the meaning of *five degrees...* really does depend on the meaning of *temperature*, if we take the intensions to be these meanings. In fact we can obtain the intension of the former from that of the latter by (i) subtracting 5 from all the temperatures in the right-hand column; and (ii) advancing all the dates in the left-hand column by one day. We can even find a mathematically respectable function \( y \) which accurately captures the meaning of the phrase *yesterday's*. The function \( y \), which maps intensions to intensions, simply increases all the dates by one day.

It is possible, of course, to formalize these ideas in a conventional, extensional logical system in which variables and expressions denote context-to-value functions. We could declare these objects to be the real extensions, thereby restore referential transparency, and dispose of the notion of intension. In practice, however, this approach is unnatural and impractical. Intensions are complicated mathematical objects. No one in their right mind would think of *temperature* as denoting some vast infinite table; nor would they consider statements about the temperature to be assertions about infinite tables. Furthermore, assertions involving a hidden context have a logic of their own which is in itself quite simple but which cannot be studied in a purely extensional system without ugly explicit context 'indices'.

Programmers very often think intensionally about programs in conventional languages. For example, when examining the body of a procedure declaration, we cannot know exactly what value it has—that depends on an implicit parameter, the ‘call’ of the procedure in question. In the same way, the variables in the body can have many possible values—depending on another hidden parameter, namely the iteration index of the loop in question (and the indices of the enclosing loops).

We would not, however, characterize these languages as being “intensional”. They lack intensional operations—operations which, like *yesterday's*, give the programmer access to extensions in contexts other than the current one. In Pascal, for example, there is no operator which yields the value an expression had on the previous iteration. Nor is there an operator which, in a procedure body, gives the value an expression had in the ‘environment’ from which the procedure was called.
It would be possible to extend Pascal with intensional operators like those just described; but it would probably not be a good idea. The new operators might be useful but they would interact in complicated ways with other features, such as side effects and aliasing. The extended language would have intensional ‘features’, but it would not be based on intensional logic.

By “Intensional Programming” we mean programming in a language which is at the same time a formal system based on intensional semantics. Intensional programmers should be encouraged to think intensionally (when appropriate), and should be provided with ‘context-switching’ operators which allow values from different contexts to be combined without explicit context manipulation. We now turn our attention to Lucid, which is just such a language.

3 The Intensional Language Lucid

Lucid is a dataflow programming language invented by E. A. Ashcroft and W. W. Wadge [10]. Lucid was originally intended to be a purely declarative (nonprocedural) language in which iterative algorithms could be expressed easily and naturally. It was soon realized, however, that the language was well suited for expressing algorithms based on a dataflow view of computation—one in which data flows through a network of asynchronously operating processing stations.

The development of Lucid began in 1974, and at that time it was widely believed that declarative languages were inherently incapable of describing dynamic activity in any very natural way. The problem, so it seemed, was that logical languages dealt only with unchanging extensions, whereas iteration requires values which are constantly changing.

The paradox was resolved by basing the language on an intensional logic in which the values (extensions) of expressions and variables depend on an implicit natural-number time parameter. Lucid programmers do not manipulate time indices explicitly; instead they use intensional operators such as next and fby (pronounced followed by). Lucid otherwise looks very much like a functional programming language, and its where construct is copied almost directly from Landin’s ISWIM. The inventors of Lucid were aware of the relationship between intensional logic and Lucid, through works such as [4].

Here is a Lucid program which calculates the partial sums of the series $1 + \frac{1}{2} + \frac{1}{3} + \cdots$ for the number $e$.

\[
\begin{align*}
  \text{sum(} & \text{term) where} \\
  & \text{term} = 1 \text{fby} \text{ term}/(i + 1); \\
  & i = 0 \text{fby} \text{ i + 1}; \\
  & \text{sum(a) = s where} \\
  & s = a \text{fby} s + \text{next a}; \\
  \end{align*}
\]

The Lucid program is obviously nothing like the program one might produce if one formulated the same algorithm in Pascal. Nevertheless, its meaning should be clear enough once the significance of the special Lucid operators is understood. The operator next takes us one moment ahead in time; the value of next $X$ at a given point in time is the value $X$ has at the next point in time. In the same way, fby takes us back one moment in time. The value of $A$ fby $B$ is the value $B$ had on the previous instant—unless the time is 0, in which case the value of $A$ fby $B$ is the current (time 0) value of $A$.

Now consider the equation $i = 0 \text{fby} i + 1$. Like all equations in a program, it is taken to be true at all points in time. This means that the value of $i$ at any point in time is that of $0 \text{fby} i + 1$. In other words, the value of $i$ at any time point is the value of $i + 1$ at the previous instant—or 0, if the time is currently 0. We can rephrase this as follows: $i$ is initially 0, and the value at the next point in time is 1 plus the current value. We can therefore imagine that the values of $i$ are produced by a computation in which $i$ is initially assigned the value 0 and then repeatedly incremented by 1. This is the sense in which Lucid programs can describe iterative algorithms.
The computational meaning of the definition of term should now be clear as well. It is initially assigned the value 1, and repeatedly updated by dividing its current value by the current value of \(i + 1\). Note that the definition could have been written

\[
\text{term} = 1 \text{fby term / next } i;
\]

because next \(i\) is always \(i + 1\).

To finally understand the whole program, we need only understand the function sum. This function is defined by the user (in the program itself), but it is nevertheless an intensional one. It should not be hard to see that sum ‘accumulates’ the values of its arguments; the value of sum(a) at any point in time is the sum of the values (extensions) of a for all points up to and including the present one.

The meaning of the program as a whole is the value of the expression sum(term), with sum and term as defined. The extension of the program at a given point in time is therefore the sum of the extensions of term for time points up to and including the present one. In other words, we can imagine that the program is computing the partial sums one by one. In Lucid the output values are defined to be its extensions. The program therefore outputs the sequence of partial sums of the series for \(e\).

Of course we could still forget about intensional logic and consider the above to be an ordinary ISWIM program in which the data objects are infinite sequences. In this view the operator next, given a sequence \((x_0, x_1, x_2, \ldots)\), returns the sequence \((x_1, x_2, x_3, \ldots)\). The operator + adds sequences componentwise, even numerals like 1 denote infinite sequences (such as \((1, 1, 1, \ldots)\)) whose components are all the same. This approach is mathematically correct but in practice is extremely confusing and misleading. It is our experience that people who understand Lucid from this purely extensional point of view find it difficult if not impossible to write even simple programs.

It is somewhat paradoxical that people should reject a form of thinking (namely, intensional thinking) which forms the basis for their natural language and of the way they think about problems.

The Lucid approach to iteration can to some extent be mimicked in the newer functional languages, such as Turner’s Miranda. These languages, however, are purely extensional and consequently do not encourage problem solving in a context-dependent manner.

4 Field Lucid

Intensional reasoning arises naturally in many problems and usually involves contexts other than (or in addition to) time. Consider, for example, a classic problem in engineering: heat transfer in a solid. Suppose that we have a long thin metal rod which is initially cool (temperature 0) but whose left-hand end touches a heat source (temperature 100). The heat will gradually diffuse through the rod, with parts nearer the heat source at first warming more quickly than those further away. The problem is to determine the temperature in various parts of the rod after a given interval.

We can compute an approximate solution to the problem by thinking of the rod as a sequence of small slices, each of which at any point in time has a uniform temperature. We then apply the basic law of heat transfer, which says that the flow of heat between two bodies is proportional to the temperature difference. Let \(T_1\) be the temperature of one of the slices (at some given time point) and let \(T_0\) and \(T_2\) be the temperatures of the slices on the left and right, respectively (in general \(T_0 > T_1 > T_2\)). In any small interval of time the middle slice will gain an amount of heat proportional to \(T_0 - T_1\), and lose an amount of heat proportional to \(T_1 - T_2\). The new temperature of the middle slice will therefore be

\[
kT_0 + (1 - 2k)T_1 + kT_2
\]

for some small constant \(k\) depending on the length of the time interval and the properties of the metal.
Using this formula we can repeatedly ‘sweep’ the rod and determine the heat distribution at successive time instants. In so doing, we are really using a simple discrete (central) difference approximation to solve the partial differential equation

\[ \frac{\partial T}{\partial t} - a^2 \frac{\partial^2 T}{\partial s^2} = 0 \]

(where \(a\) is a constant related to \(k\)).

This problem and the solution described involves an intension (temperature again) which varies in space (1-dimensional in this case) and time. Can our solution reflect this intensional point of view?

The solution to the heat flow problem expression in a conventional language such as FORTRAN might use a pair of arrays, say \(T\) and \(NEWT\). The \(T\) array would initially hold information about the heat source and the cold rod. The program would repeatedly execute a main loop which counts time instants. Nested inside the main loop we would find another loop which sweeps through the array in space, executing the assignment statement

\[ NEWT(I+1) = K*T(I) + (1-2*K)*T(I+1) + K*T(I+2) \]

To mimic this in Lucid we could always extend the simple numerical Lucid described earlier by allowing extensions to be entire vectors or matrices. This would not be particularly difficult—the existing pLucid interpreter already supports streams (intensions) whose value (at any given point in time) can be arbitrarily complex LISP-like data structures. This approach would not, however, allow us to think and write intensionally about space. Our language would be significantly less problem-oriented.

The authors and E. A. Ashcroft have recently taken an alternate approach, and have extended the language (and its interpreter) to allow intensions to vary in space as well as time. If we need only one dimension of space (as in the present example), it is enough to define our intensions as functions of two natural number parameters \(t\) and \(s\).

We can solve the current problems with only two primitive spatial intensional operators, \(\text{right}\) and \(\text{sby}\) (pronounced \(\text{succeeded by}\)), which correspond more or less to \(\text{next}\) and \(\text{fby}\). The value of \(\text{right} X\) at a given spacepoint is the value of \(X\) at the spacepoint immediately to the right. The value of \(A\ \text{sby} B\) at a given spacepoint is the value of \(B\) at the spacepoint immediately to the left if there is such a point; otherwise it is the value of \(A\) at the origin. More formally,

\[
\text{right}(F)^t_s = F^{t+1}_s \\
\text{sby}(F,G)^t_s = \begin{cases} F^t_s, & s = 0 \\
G^{t-1}_s, & s > 0. \end{cases}
\]

The following equation formalizes the FORTRAN solution given above:

\[ T = 100 \ \text{sby} \left(0 \ \text{fby} \ (k*T + (1-2*k)*\text{right} T) + k*(\text{right right} T)\right); \]

Thus at any time \(t\) and space point \(s\), \(T^s_t\) is the temperature of slice \(s\) at time step \(t\).

To illustrate the problem-oriented nature of Lucid, let us derive a program from the mathematics (the heat-transfer equation). As before, we can use user-defined intensional operators to present the same solution in a form very close to the original statement. The differential equation above can be integrated in time to give

\[ T = c + \int \frac{\partial^2 T}{\partial s^2} dt. \]

We already have a time integration function (the sum program in Section 3 can be easily adapted); we need another operator which gives us (an approximation to) the second space derivative. This is quite easy, if we think intensionally. Suppose we have a quantity \(X\) varying in space,
and that \( x_1 \) is the value at a particular spacepoint, and that \( x_0 \) and \( x_2 \) are the values of the points to either side. The ratio

\[
\frac{x_1 - x_0}{ds}
\]

is the derivative one-half step to the left, and

\[
\frac{x_2 - x_1}{ds}
\]

is the derivative one-half step to the right (\( ds \) is the spatial step size). It follows easily that the following is (approximately) the second derivative at the middle spacepoint:

\[
\frac{x_2 - 2x_1 + x_0}{ds^2}
\]

this in turn reduces to

\[
\frac{x_2 - x_1 - x_1 - x_0}{ds^2}
\]

From this we are led to the definition

\[
Ds^2(X) = 0 \text{ sby } (X - 2 \ast \text{right } X + \text{right right } X)/ds^2;
\]

Then for some appropriate constant \( c \) we can rewrite the definition of \( T \) as:

\[
T = 1 \text{ sby } It(0, Ds^2(T));
\]

\[
It(c, v) = w \text{ where } w = c \text{ fby } w + dt \ast v; \text{ end};
\]

\[
Ds^2(F) = 0 \text{ sby } (F - 2 \ast \text{right } F + \text{right right } F)/ds^2;
\]

The heat transfer problem just presented dealt only with a 1-dimensional rod and we were able to solve it by adding two primitive spatial intensional operators. In practice problems of this nature usually involve at least two or three dimensions. To deal with these more general cases the language allows arbitrary numbers of space dimensions. This extended version of the language is called Field Lucid and an interpreter already exists for the language. The name Field Lucid was chosen because the language is well suited to problem solving over an \( n \)-dimensional space that changes with time.

Our next example is a solution to Laplace’s equation using the usual relaxation method [5]. This program was originally written by E. A. Ashcroft [1] and we present it here in a slightly modified form. The relaxation method is well suited to the intensional paradigm, as the value of a point in space is determined by the average value of its four immediate spatial neighbours at the previous point in time. The following is an outline of a program that solves Laplace’s equation over a two-dimensional space.

\[
s \text{ where}
\]

\[
\text{if ELECTRODE then POTENTIAL else 0 fby avg(s) fi;}
\]

\[
\text{avg}(M) = (\text{left } M + \text{right } M + \text{up } M + \text{down } M)/4;
\]

\[
\text{ORIG = if ELECTRODE then POTENTIAL else 0 fi;}
\]

\[
\text{ELECTRODE} \text{ is the characteristic predicate of a region of space (the plane) in which an electrode is present, and at each of these points the electrical potential is fixed as the value of POTENTIAL. The initial potential at all other points in the rectangle is 0. This forms the basis for the relaxation method. At each step of the iteration, the value at a given space point outside the electrode becomes the average of the potentials of the four surrounding space points at the previous time instant. Successive approximations occur until the process settles down in}
\]

7
some finite region of interest. (The above equation defines only the iteration process, without any provision for termination.)

The definition of the $\text{avg}$ method assumes that we have negative space coordinates. The intensional expression $\text{left } M$ denotes the value of $M$ at the current (horizontal) space coordinate minus one, just as $\text{down } M$ denotes the value of $M$ at the current vertical space coordinate minus one. Field Lucid does in fact support negative space and time coordinates. The negative space coordinates permit intensional programming using a full Euclidean space. The negative time coordinates are extremely useful for placing initial conditions used in the main computation (or main inductive definition). Without this these initial conditions would have to be placed in the first few points in time, which is sometimes counter-intuitive. These additional features are defined by intensional operators that we have not defined in this paper.

Note that this solution to Laplace’s equation in two dimensions would work equally well in three dimensions by simply changing the definition of the average function to:

$$\text{avg}(M) = (\text{left } M + \text{right } M + \text{up } M + \text{down } M + \text{front } M + \text{rear } M)/6;$$

where $\text{front}$ and $\text{rear}$ are similar to $\text{left}$ and $\text{right}$, but in the third space dimension.

5 The Role of Extensional Thinking

In Lucid, programmers are (as we have already seen) more or less forced to think intensionally about time. In our experience, it is usually not a good idea to think of Lucid variables as giant immortal extensions. We therefore chose names such as $\text{next}$ and $\text{whenever}$ which strongly suggest an intensional interpretation.

In Field Lucid, variables vary in space as well as in time, and we have already seen examples in which spatial intensional thinking was natural and reflected the original problem statement. On the other hand, it seems that sometimes a ‘global’, non-intensional view of space-varying entities is also appropriate.

Perhaps we should not be suprised that time and space differ in this fundamental way. After all, in the real world we are more or less forced to experience time moment-by-moment. Sometimes we experience space in the same way: we are immediately aware only of the local region of space which we are occupying: the room we are in, the street where we live, the city we are visiting. Sometimes, however, we are able to perceive whole regions of space at once: a floor plan of a building, a whole city seen from an airplane, a highway map of a state or province. We should therefore expect to be able to use both global and local (intensional) concepts in stating and solving problems involving space.

Consider, for example, the problem of enumerating the prime numbers using the ‘sieve’ method of Erastothenes. We can formalize this algorithm as a simple Field Lucid iteration, but one in which the variable being repeatedly ‘updated’ denotes an infinite sequence of natural numbers—a different sequence on each step of the iteration.

The iteration begins with the sequence of all natural numbers greater than 1. We choose the first item of this sequence, 2, as the first prime; then we remove all multiples of 2 from the sequence. On the second step of the iteration, the sequence enumerates the odd numbers. The first item in the sequence is 3, which we take as the second prime number. Then we remove from the sequence all numbers which are divisible by 3.

On the third step of the iteration the sequence therefore begins

$$\langle 5, 7, 11, 13, 17, 19, 23, 25, 29, \ldots \rangle$$

Its first element, 5, is the third prime, and we proceed to the fourth step of the iteration by removing all multiples of 5 from the sequence. In general, then on the $n$-th step of the iteration the sequence enumerates those numbers ($> 1$) which are not divisible by the first $n - 1$ primes.
The next program is the Field Lucid program for the sieve.

\[
\begin{align*}
    p & \\
    \text{where} & \\
    p &= \text{side } D; \\
    D &= \text{N}_2 \ fby \ (D \ \text{wherever} \ (D \ \text{mod} \ p \ \text{ne} \ 0)); \\
    \text{N}_2 &= 2 \ \text{aby} \ \text{N}_2 + 1; \\
    \text{end}
\end{align*}
\]

It is not hard to see that the given program corresponds closely to the informal description of the algorithm. The infix operator \textit{wherever} passes on only those extensions on its left that have a corresponding right argument (always a predicate) that is true. The operator \textit{side} is used to select the space 0 value at each point in time.

6 Conclusion

Intensional Programming languages like Lucid are very useful tools in problem solving. Indeed, programs written in these languages are indeed problem oriented. Another way of viewing this is that programs in a language like Lucid are executable specifications [9].

Intensional programming languages bring with them new problems of implementation and of course new solutions. In the case of Lucid a novel model of parallel computation called \textit{eduction} has been developed [3]. This model of parallel computation is the basis for a parallel architecture being developed jointly by the authors and E. A. Ashcroft.

Lucid is by no means the only computer science example of an application of intensional logic. Another example is in the context of a tree-oriented file system such as is found in UNIX. In such a system, moving from a lower directory to a higher one can be accomplished by the command \texttt{cd ..../...}. This type of command is indeed intensional; in this case the implicit context is not a hidden time or space parameter, rather it is an implicit tree structure. One does not need to know where one is when issuing the above command; all one needs to know is that it will move two levels up the tree.

We recommend that those interested in this approach to problem solving write to Arizona State University for a copy of the Field Lucid Interpreter, which runs under the UNIX operating system (all requests should be accompanied by a small magnetic tape).

References


