A SUMMARY OF LUCID FOR PROGRAMMERS
(1981 Version)

by

E.A. Ashcroft*

&

W.W. Wadge**

Technical Report CS-82-57

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1

* Department of Computer Science, University of Waterloo,
Waterloo, Ontario, Canada, N2L 3G1

** Computer Science Department, University of Warwick,
Warwickshire CV4 7AL, England

ABSTRACT

This paper presents the programming language Lucid, both its syntax and semantics, and discusses some possible ways of thinking about the operational behaviour of Lucid programs. (The actual operational behaviour is really completely different, but we find that having some sort of operational ideas is often very useful when designing Lucid programs.)

This paper does not consider the manipulation (transformation) rules, or the inference rules, that are an integral part of Lucid and which allow reasoning about programs. It is concerned mainly with Lucid as a programming language.

The commonplace expressions of arithmetic and algebra have a certain simplicity that most communications to computers lack.

[*P.J. Landin, 1966*]

0. INTRODUCTION

It is apparent that the goals of language designers and logicians are quite similar: to develop systems for precisely specifying objects and properties. In both cases this means the study and development of the syntax and semantics of purely formal, as opposed to natural, languages.

It is also apparent that logicians have been eminently more successful. The logicians' languages, such as predicate calculus, are simple, elegant and, above all, semantically well defined. Programming languages, by contrast, are complex, clumsy and, above all, semantically very poorly defined. It is often said that they are "illogical".

Furthermore, the languages of logicians were developed in conjunction with rules of inference, so that reasoning about properties of objects could proceed by simple finite manipulations completely within the language itself. By contrast, formal reasoning about programs, to the extent that it is possible at all, has to be carried out in a separate formal system in which the manipulations are performed on comments on, or a translation of, the original program.

The obvious conclusion is that logic (and mathematics in general) could be usefully applied to the study and design of programming languages. Few would dispute this; but there have always been two points of view about the relationships between logic and computer science.

One point of view sees mathematics as playing primarily a passive role, being used to describe, to model and to classify. The other point of view sees mathematics as playing primarily an active role, being used not so much to describe existing objects as to plan new objects.

These two approaches, which we might call the *descriptive* and *prescriptive* approaches [1], are well illustrated by two important papers by Landin, "A Correspondence between Algol 60 and Church's λ Notation" and "The Next 700 Programming Languages", [2, 3].

In the first Landin defines a translation from Algol 60 into the λ-calculus and so uses logic to describe Algol. In the second he begins with λ-calculus and develops a simple non-procedural language (ISWIM†), with a naturally defined construct (the "where" clause) which introduces local variables in a way similar to the Algol block, but which is actually based on the λ-calculus. Landin's first paper represents the descriptive approach and the second represents the prescriptive approach.

It is clear that if we want to develop computer languages having the elegance of mathematical languages it is the prescriptive approach that we must adopt.

It has become almost accepted without question that computer languages can not hope to have the simplicity we desire. For example, Scott and Strachey [6] say that "computer oriented languages differ

---

† If you See What I Mean.

from their mathematical counterparts by virtue of their *dynamic* character. An expression does not generally possess one uniquely determined value ... but rather the value depends on the state of the system at the time of initialization of evaluation ... . Therefore the "algebra" of equivalences of such expressions need not be as "beautiful" as the well-known mathematical examples. This does not mean that the semantics of such languages will be *less* mathematical, only an order more complex". We feel that this attitude is a result of trying to mathematically describe existing languages. By taking the prescriptive approach, and basing new languages on existing mathematical languages, such as mathematical logic, more positive results can be obtained, and equivalences of expressions can be "beautiful".

Our goal here is to follow Landin's lead in the second paper mentioned above and develop an uncompromisingly logical language which has "facilities" for functions and scope and has non-trivial and useful operational interpretations.

The language, Lucid, is in fact similar to ISWIM. Landin, however, gave no direct semantics (ISWIM is a syntactic variant of a subset of the λ-calculus), and neither did he give an inference system for the verification of ISWIM programs. (He did give a system, of sorts, for transforming programs, but it is not very useful.) Of course, these omissions are not Landin's fault, because at that time semantics and program verification were in their infancy. With Lucid we fill in the gaps in Landin's treatment, and the fact that the semantics and transformation and inference rules turn out to be simple, natural and elegant is a vindication of Landin's mathematical approach to language design. In this paper we will not consider Lucid's inference system.

1. ISWIM AND LUCID

ISWIM is based on the where clause which is an expression qualified by auxiliary definitions. For example

$$x^2 + y^2 \quad \text{where} \quad x = a + b$$
$$y = a - b$$
$$\text{end}$$

is a typical where clause. Such a clause is a term, i.e., has a value; in the above example, it is the same as that of the term $(a+b)^2 + (a-b)^2$. These constructs can of course be nested, i.e., expressions occurring anywhere in a where clause may contain where clauses. The variables defined by the equations in the right arm of the where clause are the locals: their definitions apply only to the left-hand side expression of the clause. In addition, functions are defined with the formal parameter list on the left hand side, e.g.,

$$f(x, y) = x^2 - p \cdot q \quad \text{where} \quad p = a \cdot y + b$$
$$q = a \cdot x - b$$
$$\text{end}$$

The formal parameters may represent functions, so "higher-order" functions can be defined. ISWIM also has a "whererec" clause in which circular (recursive) definitions are allowed, (the definitions in the whererec clause apply also to the right arm of the clause).

One difference between Lucid and ISWIM is that the former is based on a slightly different construct, namely the valof phrase. A valof phrase consists of a set of definitions (like those in the right arm of a where clause) called the body, enclosed by the 'brackets' valof (i.e., "value of") and end . One of the variables defined in the body must be result , whose purpose is to indicate the value of the phrase. Functions are defined as in ISWIM, except that the formal parameters may *not* represent functions; there are no definitions of "higher-order" functions.

The following are legal valof phrases:

```
valof
        x = a + b ;
        y = a - b ;
   result = x**2 + y**2 ;
end
```

```
valof
        nglobal n,m ;
        p = 1 fby p + m**2 ;
        q = n**2 - 1 ;
    f(c, d) = 1 + valof
                    pglobal d ;
                    eglobal c,q ;
                    p = c**2 - d**2 ;
                result = p**2 + 2*p*q asa p > c ;
                end ;
    result = f(3, q) ;
end
```

An important difference between valof phrases and where clauses is that in the former recursiveness is assumed, i.e., the valof phrase corresponds to the whererec clause of ISWIM. This eliminates one of the most confusing features of ISWIM, namely the fact that the same variable can refer to different things on the opposite sides of the same equation. It would be possible to base Lucid on the whererec clauses of ISWIM, and in fact in the latest versions of Lucid this is exactly what we do. Here we will consider Lucid with valof phrases because that is the Lucid that is handled by the 1981 Version of the Lucid interpreter [5].

All Lucid programs in this paper correspond to the 1981 Version of the interpreter. It was written by Calvin Ostrum at the University of Waterloo (while he was an undergraduate student!). It runs on a VAX, under Berkeley UNIX. Improvements have been made to the interpreter by Tony Faustini at the University of Warwick, and the late 1982 Version handles a slightly different language, with where clauses as well as valof phrases. This paper does not consider the 1982 Version.

2. SYNTAX OF LUCID

We wish to achieve a clear separation between two aspects of a programming language, the set of given or primitive things and the ways of expressing things in terms of other things. We thus get a family of languages, each of which is, in Landin's words, "a point chosen from well-mapped space". The coordinate of a point in this well-mapped space is the set of chosen primitives. Since Lucid is based on expressions, to specify the primitives we have to supply a domain of data objects, a collection of operations on these objects, and a collection of symbols used to denote the operations. In other words, a member of the Lucid family is determined by an algebra A ; we will call the corresponding language $\mathrm{Lucid}(A)$. The syntax of $\mathrm{Lucid}(A)$ is determined only by the signature of A .

Suppose that we are given an algebra A with signature Σ . That is, Σ is a collection of constant symbols of various arities ("individual constants" being of arity 0). We follow the usual terminology of symbolic logic and refer to the elements of Σ as "constant symbols", even though it is only those of arity 0 which are what computer scientists usually refer to as "constants". This is understandable when we realise that, for example, + , like 3 , has the same meaning in all contexts.

We also assume that we have available an unlimited number of variables of various arities (these are what are usually called identifiers). The set of variables is the same for all algebras A .

The nullary variables will also be called "individual variables". Strictly speaking, the constant symbols and variables should be typed to indicate the number of arguments expected, but in practice we will omit these types. Non-nullary constant symbols will often be called operation symbols, and non-nullary variables will often be called function variables.

A Lucid program is simply a term*, but to define the class of terms we must also define the classes of definitions and phrases simultaneously and inductively.

A *term* is either

(i) an n-ary constant symbol together with a sequence of n terms as operands (n will be zero if the symbol is an individual constant);

(ii) an n-ary variable together with a sequence of n terms as actual parameters (n will be zero if the variable is an individual variable);

or

(iii) a phrase.

A *phrase* consists of an unordered set of declarations and an unordered set of definitions, no two of which have the same definiedum, and exactly one of which has the individual variable result as its definiendum.

A *declaration* consists of the word nglobal or eglobal followed by a list of nullary variables.

---

* Actually, the 1981 Version of the interpreter considers programs to be phrases without declarations. All globals of these phrases are implicitly nglobals .

A *definition* consists of a definiens which is a term, and of a term consisting of an n-ary variable ϕ (the definiedum) together with an ordered set of n distinct individual (nullary) variables (the formal parameters). ϕ is called a *local* variable of the phrase in which the definition occurs.

All variables which are not local variables of a phrase are called *global* variables of the phrase. All the nullary globals must be declared*; eglobal stands for "elementary global" and nglobal stands for "nonelementary global".

This, of course, is an abstract syntax of Lucid (in the sense of McCarthy [4]), analogous to Landin's abstract syntax of ISWIM. In our examples, (like those already given) we will use a fairly obvious *concrete* linear (or, more realistically, two dimensional) representation in which terms are written using infix notation, definitions are written as equations (with the definiens on the right), and each phrase is written as a sequence of declarations followed by a sequence of equations, enclosed by the keywords valof and end.† We will not give a precise definition of the concrete syntax. Such a definition would clearly not be particularly complex, but it can not just take the form of a context-free grammar because of the restriction that the formal parameters in a function definition be distinct and the restriction that no variable have two definitions in the same phrase.

---

† The order of the declarations or definitions in a sequence will not be important.

* Actually, in the 1981 Version, non-nullary globals must be declared also, as nglobals. Moreover, if a variable is "inherited" through two or more levels of phrases, it must be declared in the same way in each phrase. (See the example on page 24.)

3. SEMANTICS OF LUCID

Before proceeding to the semantics of Lucid it is necessary first to define precisely the algebra $Lu(A)$, the class of elementary history functions, the operation of freezing, and so on.

Suppose then that we are given a continuous algebra A with signature Σ. The set Σ is usually considered to be a set of ranked operation (constant) symbols but we will assume (for simplicity) that Σ contains two extra symbols, namely the sort symbol U for the universe of A and the relation symbol $\sqsubseteq$ for approximation in A. This assumption allows us to consider A to be simply a function with domain Σ, one which assigns to U a nonempty set, which assigns to $\sqsubseteq$ a partial order on $A(U)$ which makes $\langle A(U), A(\sqsubseteq)\rangle$ a cpo, and which assigns to each n-ary operation symbol an $A\sqsubseteq$-continuous operation over $A(U)$. We also assume that Σ contains the nullary symbols true, false and Ω, with $A(\Omega)$ the $A(\sqsubseteq)$-least element of $A(U)$, usually denoted $\perp$.

We require that Σ be 'normal' in that it does not contain the special Lucid operation symbols, which for our next purposes we take to be the unary symbols first and next and the binary symbols asa and fby. This allows us to define $Lu(\Sigma)$ to be the result of adding these symbols to Σ, and to define $Lu(A)$ to be the function H with domain $Lu(\Sigma)$ such that:

(i) $H(U)$ is the set of all infinite sequences of elements of $A(U)$, i.e., the set of all functions from the set $\{0, 1, 2, \ldots\}$ of natural numbers to $A(U)$ (elements of $H(U)$ will be called A-histories or simply histories);

(ii) $H(\underline{\sqsubset})$ is the pointwise extension of $A(\underline{\sqsubset})$, i.e., given any A-histories α and β, $\langle\alpha, \beta\rangle \in H(\underline{\sqsubset})$ iff $\langle\alpha_i, \beta_i\rangle \in A(\underline{\sqsubset})$ for all i;

(iii) for any operation symbol k in Σ, $H(k)$ is the pointwise extension of $A(k)$, i.e.,

$$(H(k)(\alpha, \beta, \gamma, \ldots))_t = A(k)(\alpha_t, \beta_t, \gamma_t, \ldots)$$

(a function over $H(U)$ will be called an A-history function, or simply a history function);

(iv) $H(\underline{\text{first}})$, $H(\underline{\text{next}})$, $H(\underline{\text{fby}})$ and $H(\underline{\text{asa}})$ are the unary history functions first and next and the binary functions fby and asa where

(a) $\text{first}(\alpha) = \langle\alpha_0, \alpha_0, \alpha_0, \ldots\rangle$

(b) $\text{next}(\alpha) = \langle\alpha_1, \alpha_2, \alpha_3, \ldots\rangle$

(c) $\text{fby}(\alpha, \beta) = \langle\alpha_0, \beta_0, \beta_1, \ldots\rangle$

(d) $\text{asa}(\alpha, \beta) = \langle\alpha_i, \alpha_i, \alpha_i, \ldots\rangle$ where i is the least number for which $\beta_i = A(\text{true})$ and $\beta_j = A(\text{false})$ for all $j < i$

$= \langle A(\Omega), A(\Omega), A(\Omega), \ldots\rangle$ $(= H(\Omega))$. if no such i exists

for all histories α and β and all natural numbers t.

We now proceed to give the semantics of Lucid. A *Lucid environment* is a function from the set of variables to the set of history functions which assigns n-ary history functions to n-ary variables. Given a Lucid environment E and a time s and a set w of individual variables, the frozen environment E^s_w is the unique Lucid environment such that $E^s_w(m)() = \langle E(m)(), E(m)(), \ldots \rangle$ for any individual variable m in w, and $E^s_w(\zeta) = E(\zeta)$ for any other variable ζ.

We now define the meaning of a Lucid term t in a Lucid environment E, by induction on the structure of t;

(i) if t consists of the Lu(E) operation symbol k together with operands $u_0, u_1, \ldots, u_{n-1}$, then the meaning of t in E is the result of applying $H(k)$ (i.e., Lu(A)(k)) to the meaning of the u_i in E;

(ii) if t consists of a variable g together with actual parameters $u_0, u_1, \ldots, u_{n-1}$, then the meaning of t in E is the result of applying $E(g)$ to the meanings of the u_i in E;

(iii) if t is a phrase with eglobals w then the meaning of t in E is α. where at any time s, α_s is the meaning of result() in E'_s at time s where E'_s is the least environment which satisfies the definitions in the phrase and agrees with E^s_w except possibly for the values E'_s assigns to the locals of t.

Finally, given a definition d of the form

$$g(x_0, x_1, \ldots, x_{n-1}) = a,$$

we say that d is satisfied by E iff the meaning of $g(x_0, x_1, \ldots, x_{n-1})$ in $\hat{E}$ is equal to the meaning of a in $\hat{E}$ for any environment $\hat{E}$ agreeing with E except possible for some of the values assigned to the formal parameters $x_0, x_1, \ldots, x_{n-1}$.

A Lucid program is simply a term, the free variables of which are called the input variables. An input to a program is simply an association of input values with input variables, and the meaning (or "output") of the program is its value in the appropriate environment. For example, suppose that A, B and C are the input variables of a program; then the meaning of the program given input α, β and γ is its meaning as a term in an environment in which $E(A) = \alpha$, $E(B) = \beta$ and $E(C) = \gamma$.

The above is not strictly speaking a valid definition, because an assumption is made which is not obviously true; namely, that a least Lucid environment E'_s always exists. It is possible to give a more indirect but obviously valid definition, and then to prove that the above is a true statement about the meaning so defined. We will not do this here.

We shall approach the intuitive meaning of Lucid by first considering two languages which are special cases of Lucid, ULU and LUSWIM.

ULU is a perfectly well-defined family of languages which has an interesting operational interpretation: the iterations within phrases are synchronized with the iterations in enclosing phrases, which results in the language having the flavour of a data-flow language, with the defined functions behaving like coroutines. Unfortunately, this synchronization means that in ULU it is not possible to define subcomputations.

LUSWIM is, in many ways, a conventional Algol-like language but the same variable means different things inside and outside a phrase, the inner occurrence being "frozen". As a result, phrases can be interpreted as defining subcomputations which return a result.

Lucid is the result of combining ULU and LUSWIM , giving a language which is a superset of both sublanguages and is generally better than either of them individually. The features of the two sublanguages do interact, but constructively, not destructively. This is a direct result of the fact that both sublanguages are mathematically defined.

We should point out again that Lucid is not a "higher-order" language, that is, the defined functions can neither take functions as arguments nor return functions as results. To remove this restriction in Lucid would first require the investigation of the consequences of relaxing the restrictions that the variable result and the formal parameters of a function definition be individual variables. These consequences may not be too dire, but some complications are almost bound to result, so this extension of Lucid is not considered in this paper.

4. THE LANGUAGE ULU(A)

Here is a sample program† in ULU(Q) , where Q consists of the rationals together with the usual arithmetic operations:

```
valof
        I = 1 fby I+1 ;
        J = 3•I       ;
    result = valof
                 nglobal J  ;
                 S = J fby S + next J ;
                 N = 1 fby N+1  ;
             result = S/N  ;
         end ;
end .
```

Since there are no global variables, the meaning of the outer phrase is the value of result in the least environment E_1 satisfying the three definitions in the phrase. It is easy to see that $E_1(I)$ is the history <1, 2, 3, ...> and $E_1(J)$ is the history <3, 6, 9, ...> . The value of E_1(result) is the value of the inner phrase in the environment E_1 , which is the value of result in the least environment E_2 , differing from E_1 only in the values of the locals (S, N and result) of the inner phrase, which satisfies the definitions in the inner phrase.

---

† In this and subsequent papers, we use fby for followed by and asa for as soon as .

Thus $E_2(J)$ is <3, 6, 9, ...> , $E_2(S)$ is <3, 9, 18, ...> , $E_2(N)$ is <1, 2, 3, ...> and E_2(result) is <3, 4.5, 6, ...> . Thus the meaning of the program is <3, 4.5, 6, 7.5, ...> .

Notice that what the inner phrase is doing is maintaining a running average of the values of J . We can use this inner phrase as the body of function called Avg , as we do in the following example:

```
valof
        nglobal J ;
        Avg(X) = valof
                            nglobal X ;
                            S = X fby S + next X ;
                            N = 1 fby N+1 ;
                        result = S/N  ;
                    end ;
                M = Avg(S) asa I eq 10 ;
                I = 1 fby  I+1 ;
            result = Avg((S-M)**2) asa I eq 10 ;
end .
```

Since this program has a free variable S , we can only talk of the value of this program in an environment E which gives a value to S . Let us suppose that $E(S)$ is some history σ . In the inner environment, Avg will be the "running average" function, so that M is the constant history which is everywhere the average of the first

ten values of $E(S)$, that is, the average of σ_0 through σ_9 . Since the value of result will be the average of the first ten values of $(S\text{-}M)^2$, we see that the value of the program in E will be the constant sequence which is everywhere equal to the variance (the second moment about the mean) of the first ten values of S in E .

Probably the best way of viewing the previous program in an operational way is to consider *Avg* as a coroutine, with two invokations, $Avg(S)$ and $Avg((S\text{-}M)^2)$. These invokations are considered as running from time 0 , but only the value of $Avg(S)$ at time 9 and the value of $Avg((S\text{-}M)^2)$ at time 9 are "used". The coroutine activations have to be considered as running from time 0 so that they can keep running sums of S and $(S\text{-}M)^2$ until they are needed at time 9 .

Suppose we wished to generalise this program to compute arbitrary higher moments about the mean. We would need a function *Pow* where $Pow(X, N)$ gives us the running N-th powers of the values of X . We might try to define this as follows:

```
Pow(X, N) = valof
                nglobal X,N  ;
                I = 0 fby I+1  ;
                P = 1 fby X·P  ;
            result = P asa I eq N ;
        end .
```

Now, *Pow*(5, 2) is <25, 25, 25, ...> and *Pow*(6, 3) is <216, 216, 216, ...> but if, say, the value of *A* is <1, 2, 3, ...> then the value of *Pow*(*A*, 2) is <2, 2, 2, ...> ! Moreover, the value of *Pow*(5, *A*) is <5, 5, 5, ...> and the value of *Pow*(5, *A*+1) is <⊥, ⊥, ⊥, ...> ; if the second argument *N* is changing with time, the variable *I* in the definition of *Pow* is chasing a moving target!

Similarly, if we tried to generalise the variance program to give the variance of the first *N* values of the history *X* , we would get rubbish if *N* varies with time.

The reason for all this is that in ULU globals have the same meaning inside a phrase as outside and so phrases are synchronised with their environments. We can have no subcomputations in ULU (but we can have one computation following another, as in the variance example). We have nesting of scope but we can not have nesting of computations, i.e., there are no subloops.

5. THE LANGUAGE LUSWIM(A)

The reason that the definition of *Pow* is considered to be wrong is that we expect it to work pointwise. If A had an explicit exponentiation operator, say $\uparrow$, then, according to Lu(A), if α and β are histories $\alpha\uparrow\beta$ would be $\langle\alpha_0\uparrow\beta_0, \alpha_1\uparrow\beta_1, \ldots\rangle$. This is exactly how we would expect *Pow* to work, but in ULU it clearly doesn't.

The root of the problem is that phrases are not defined pointwise. The value at time t of a phrase, in which G appears as a global, depends on the value of G not just at time t but at other times as well. For example, at any time t, the phrase computing the average of the first $t+1$ values of x depends on $x_0, x_1, \ldots, x_{t-1}$ as well as x_t. This is exactly what we wanted in the "average" example, but it is disastrous in examples like the one to compute the N-th power of x.

What is required is some way of ensuring that the value of a phrase, in environment E, at time t depends only on the values of the globals at time t. We can ensure this by freezing E at time t: given a sequence α and a time t, we define α^t to be the sequence $\langle\alpha_t, \alpha_t, \alpha_t, \ldots\rangle$; then the value the frozen environment E^t assigns to nullary variable G is $E(G)^t$. The value of the phrase at time t is then the value at time t of result in the least environment, differing from E^t only in the values of the locals, which satisfies all the definitions in the phrase.

LUSWIM(A) is the language obtained by using this pointwise interpretation of phrases.

Here is an example of a simple LUSWIM program:

```
valof
        I = 1 fby I ;
        S = 1 fby S + next M ;
        M = valof
                eglobal I ;
                K = 1 fby K+1 ;
                P = I fby I•P ;
            result = P asa K eq I ;
        end ;
    result = S asa I eq 6 ;
end .
```

This program has no global variables, so the difference between ULU and LUSWIM is not apparent until we consider the inner phrase, which has global variable I, which is indicated as an "eglobal", meaning it is frozen, as described above. In the (single) environment E inside the outer phrase, the value of I clearly is $< 1, 2, 3, \ldots >$. The value of M in E is determined separately at each time step. $E(M)_0$ is the value of result at time 0 in E' where E' is the environment obtained by freezing E at time 0 and then choosing values for K, P and result which satisfy the definitions in the inner phrase.

Clearly $E'(P)$ is <1, 1, 1, ...> and E'(result) is <1, 1, 1, ...> . $E(M)_1$ is the value of result at time 1 in E'' where E'' is the environment obtained by freezing E at time 1 and then choosing values for the locals. So $E''(P)$ is <2, 4, 8, ...> and E'(result) is <4, 4, 4, ...> . Continuing this process, we see that $E(M)$ is $<1^1, 2^2, 3^3, ...>$. The value of the program at any time will be $1^1 + 2^2 + 3^3 + 4^4 + 5^5 + 6^6$.

In LUSWIM then, a single outer environment, like E in the example, does not determine a single inner environment, but rather a sequence of environments, like E' , E'' etc. , one for each outer time step.

Although the mathematical semantics of LUSWIM is more complicated than that of ULU , simpler and more conventional operational interpretations are possible. Function evaluation as well as evaluation of phrases can be thought of as subcomputations which take place while the enclosing computation is suspended. Functions without global variables can be thought of like data functions, while functions with global variables are like Algol function procedures whose globals have different values at different times.

It is apparent that these are two very different ways of adding iteration to Lucid, both potentially useful, and both having natural if radically differing operational semantics. It seems very unlikely that one is the right interpretation and the other the wrong one; in fact it is easy to imagine programs which (operationally speaking) use both coroutines and nested computation. An example would be a coroutine which computes a running N-th moment (average of N-th powers) but uses an inner, nested loop which computes the N-th power in a subcomputation.

Clearly the two 'facilities' should be combined in the same language, but it is not immediately obvious how to do this. The two languages are superficially very different in the way they give a meaning to a phrase. In ULU the entire value of the phrase is determined all at once in terms of the entire values of the globals, whereas in LUSWIM it is put together instant by instant in terms of the instantaneous values of the globals. The difference, however, is not as great as it seems, once it is realized that the ULU semantics can also be given in a pointwise manner: the value of a ULU phrase in an environment E at time t is the value of result at time t in the least environment satisfying the definitions in the phrase and differing from E at most in the values assigned to the locals. This definition, of course, makes it clear that the difference between ULU and LUSWIM is that, on each time step, LUSWIM freezes the values of the globals before determining the inner environment. In ULU, the inner environments corresponding to different

times are all the same. To combine the two semantics of phrases we can simply allow both sorts of globals, nglobals and eglobals, one of which (the eglobals) consists of variables which are subject to freezing inside phrases. In this way it should be possible to mix up elementary and nonelementary variables in definitions almost at will.

The language which results from combining LUSWIM(A) and ULU(A) in this way is the language Lucid(A) .

The set of LUSWIM programs is syntatically a subset of the set of Lucid programs. The semantics of LUSWIM is determined by specifying that this inclusion hold semantically as well, in other words, the meaning of a LUSWIM program is its meaning considered as a Lucid program. It is apparent that this definition of LUSWIM conforms with the informal one given earlier because in a LUSWIM program all the globals of a phrase, being elementary, are frozen inside the phrase. Similarly, the meaning of a ULU program is its meaning considered as a Lucid program; since all the globals of a ULU phrase are nonelementary, none will be frozen.

6. OPERATIONAL INTERPRETATIONS OF LUCID

We will illustrate possible operational views of Lucid by looking at several examples of Lucid programs.

The first example combines the two forms of iteration. The phrase has one nonelementary 'input' (free) variable *X* and the program's value at time t is the 10th moment of the first t+1 values:

```
valof
        nglobal  X ;
        S = P fby S + next P ;
       XX = X ;
        P = valof
                    eglobal XX ;
                    Y = 1 fby  Y*XX ;
                    I = 0 fby I + 1 ;
                result = Y asa I eq 10 ;
            end ;
        J = 1 fby J + 1 ;
    result = S/J ;
end .
```

The declaration eglobal *XX* allows the inner phrase to freeze each particular value of *X* in order to compute the 10-th power of that particular value. *XX* is necessary because we cannot say eglobal *X* in the inner phrase. Using *XX* as nglobal instead of eglobal inside the inner phrase would give a completely different result. In general the value of the altered phrase

```
valof
        nglobal XX ;
        Y = 1 fby Y•XX ;
        I = 0 fby I + 1 ;
    result = Y asa I eq 10 ;
end
```

at any time is the value of the product of the first 10 values of XX : If this phrase were used in the previous program, the program's value would also be, at any time, the product of the first 10 values of X .

In writing the above program we could, if we wanted, define a function Pow and just 'call' it to give the 10-th power of X . This program

```
valof
        nglobal X ;
        S = P fby S + next P ;
        P = Pow(X, 10) ;
        N = 1 fby N + 1 ;
    result = S/N ;
Pow(B,K) = valof
                eglobal B,K ;
                Y = 1 fby B•Y ;
                I = 1 fby I + 1 ;
            result = Y asa I eq K ;
            end ;
end
```

has the same value.

The examples just given used both kinds of iteration, but separately: the outer phrase had only a nonelementary variable as its global and could be though of as a ULU phrase, while the inner one had only an elementary variable as its global and could therefore be thought of as an ordinary LUSWIM phrase. Here is an example where the features are combined in one (the inner) phrase. The program computes the running variance of its input variable X, i.e., its value at time t is the variance of the first t+1 values of X:

```
valof
        nglobal X ;
        Avg(V) = valof
                    nglobal V;
                    S = V fby S + next V ;
                    I = 1 fby I + 1 ;
                result = S/I ;
                end ;
            M = Avg(X) ;
        result = valof
                    nglobal X ;
                    eglobal M ;
                result = Avg((X - M)**2) ;
                end ;
end .
```

The outermost phrase is a pure ULU phrase because its only global is nonelementary, and the same is true of the phrase defining the function *Avg* (whose value is the running average of its argument). The second inner phrase, however, is quite different: one of its globals is the elementary variable M, but the other is the nonelementary variable X. The result is that the value of M inside this phrase is always its frozen outer value, whereas that of X depends on the inner time.

The value of this 'mixed' phrase at time 2, for example, depends on the value of M at time 2 only, but on the values of X at times 0, 1, and 2. If the value of X begins <6, 8, 10, ...> then the value of M begins <6, 7, 8, ...> and the value of the phrase at time 2 is $((6 - 8)^2 + (8 - 8)^2 + (10 - 8)^2)/3$ which is 8/3, as required. A mixed phrase is used because this (admittedly naive) algorithm implements directly the definition of variance which requires that the *present* average be subtracted from all previous values.

There are two conceptually different ways of giving an operational interpretation to 'mixed' phrases with both elementary and nonelementary globals. One way is to consider such a phrase as 'basically' an ordinary LUSWIM phrase which has additional special (nonelementary) variables which can be thought of as being 'restarted' at the beginning of every subcomputation. The phrase just discussed can be understood in this way, as can the phrase defining *isprime* in the program

```
valof
                    P =   fby nxprime(P) ;
            nxprime(Q) = valof
                                 eglobal Q ;
                                 N = Q + 1 fby N + 1 ;
                               result = N asa isprime(N) ;
                         end ;
            isprime(M) = valof
                                 nglobal P ;
                                 eglobal M ;
                               result = P**2 ≥ M asa P**2 ≥ M or M mod P eq 0
                         end ;
                  result = P ;
end
```

whose value is the sequence <2, 3, 5, ...> of all primes. The function *isprime* can be thought of as testing its argument for primeness using a simple loop which runs through all the primes starting with the first and checks whether one whose square is less than M actually divides M.

The other way of viewing a mixed phrase is to consider it as a parameterised set of ULU phrases, with each "freezing" of the global elementary variables yielding a ULU phrase. For example the following program

```
valof
        eglobal N ;
        nglobal X ;
        S = P fby S + next P ;
        P = X ↑ N ;
        I = 1 fby I + 1 ;
    result = S/I ;
end
```

has as its value at time t the N-at-time-t-th moment of the values of X up to time t (assuming the data function ↑ is exponentiation). The variable N is the parameter, and each numeric value of N yields an ordinary ULU phrase. In an environment in which N is constantly 2, the phrase is equivalent to the ordinary ULU phrase

```
valof
        nglobal X ;
        S = P fby S + next P ;
        P = X ↑ 2 ;
        I = 1 fby I + 1 ;
    result = S/I ;
end .
```

In an environment in which N is constantly 3, it is equivalent to an ordinary ULU phrase which computes the third moment. In an environment in which the value of N changes irregularly with time between 2 and 3,

the phrase can be considered as sampling the appropriate outputs of two different simultaneously running 'coroutines' computing running 2nd and 3rd moments respectively.

The same interpretations can be applied to *functions*, the definitions of which use both elementary and nonelementary globals in phrases. For example, given the definition

```
Mom2(X, M) = valof
                nglobal X ;
                eglobal M ;
                S = T fby S + next T ;
                T = (X - M)**2 ;
                I = 1 fby I + 1 ;
             result = S/I ;
           end ;
```

the value of *Mom2(A, N)* (in an appropriate environment) at time t is the 2nd moment of the first t+1 values of *A* about the value of *N* at time t. This function can be understood as an ordinary Algol-like function except that its special first argument is restarted every time the function is called.

It is, of course, possible to use two different viewpoints of the same object. We could, for example, also regard *Mom2* as an ordinary ULU function with a parameter *M*, so that

Mom2(A, 0)

is the running second moment, and

Mom2(A, Avg(A))

is the running variance. *(Mom2(A, Avg(A))* can be viewed as a possibly infinite set of simultaneously running coroutines, one for each different value of the running average of *A* . The value of *Mom2(A, Avg(A))* at time t is the value, at time t , of the coroutine corresponding to the running average of *A* at time t .)

7. CONCLUSION

The operational interpretations may be of help in visualising the "running" of Lucid programs, but an actual implementation may work completely differently. In fact the Lucid interpreter works in a "demand driven" manner. The interpreting algorithm handles all Lucid programs but is quite involved. We feel that it is useful, when writing Lucid programs, to have simple and intuitive operational ideas, even if each such operational view is limited to describing a particular subset of Lucid programs.

8. REFERENCES

[1] Ashcroft, E.A. and Wadge, W.W., "R_x for Semantics", CS-79-37, Computer Science Department, University of Waterloo, December 1979.

[2] Landin, P.J., "A Correspondence Between Algol 60 and Church's Lambda Notation", CACM 8, pp. 89-101, 158-165.

[3] Landin, P.J., "The Next 700 Programming Languages", CACM 9, No. 3, pp. 157-166.

[4] McCarthy, J., "Towards a Mathematical Theory of Computation", Proceedings IFIP, 1962.

[5] Ostrum, C.B., "Luthid 0.0 Preliminary Reference Manual and Report", CS-81-24, Computer Science Department, University of Waterloo.

[6] Scott, D. and Strachey, C., "Towards a Mathematical Semantics for Computer Languages", Proc. Symp. Computers and Automata (J. Fox, ed.), Polytechnic Institute of Brooklyn Press, New York, 1976.