New Approaches to Programming *

Clayton Lewis

CU-CS-429-89

University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

* This research is supported by grants and contributions from the National Science Foundation, NASA, and the Air Force Human Resource Laboratory.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **1989**	2. REPORT TYPE	3. DATES COVERED **00-00-1989 to 00-00-1989**
4. TITLE AND SUBTITLE **New Approaches to Programming**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of Colorado at Boulder,Department of Computer Science,Boulder,Co,80309**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES		
14. ABSTRACT		
15. SUBJECT TERMS		

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**20**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

New Approaches to Programming

Clayton Lewis

CU-CS-429-89 March 1989

Department of Computer Science
Campus Box 430
University of Colorado,
Boulder, Colorado, 80309

This research is supported by grants and contributions from the National Science Foundation, NASA, and the Air Force Human Resource Laboratory.

New Approaches to Programming

Clayton Lewis
Department of Computer Science and
Institute of Cognitive Science
Campus Box 430
Boulder CO 80309 USA
(303) 492 6657
clayton@sigi.colorado.edu

March 16, 1989

Abstract: The difficulty of programming limits the usefulness of computers to many non-specialist users. This paper surveys the programming language and environment scene, looking for good ideas that might be exploited to make programming easier than it is today. It includes brief descriptions of two current projects inspired by this goal, and ends with some recommendations for research.

The problem.

Programming is difficult and hard to learn. The evidence is all around us: application programming backlogs that grow, not shrink; scientists who must work through intermediaries to analyze data or automate experiments; engineering students spending months or years to develop needed programming skills. Despite the proliferation of languages since Fortran, and many undeniably useful innovations, it is hard to point to any development within the programming mainstream that has had decisive impact in making programming not just a little easier, but much easier.

If we are to make progress with this pressing problem it seems likely that we must divide and conquer. We cannot expect ideas that improve the lot of professional systems programmers to help the non-programmer scientist, since the professional programmer can afford a much bigger investment in understanding esoteric concepts and methods than the scientist who wishes to devote his or her effort to science, not programming.

In this paper I focus on the non-programmer, the person who wants to use the computer flexibly with minimum study of it. I will review ideas in programming technology that seem to me to have promise in making programming much easier to learn and do for the non-specialist. Because of growing interest in user interfaces, including construction of visual representations of scientific data or of complex systems under analysis (and because of my own research interests in these topics) I will pay special attention to ideas that can make the handling of interactive graphics and animation easier than it is now.

My review will be subjective and incomplete. My aim in presenting it is to interest readers in this problem, and to learn from them about aspects of the problem, or pertinent research, that I have overlooked.

Programming system: Environment, Model of Computation, Notation.

It is now universally recognized, though still not always acted upon, that efficient and effective programming depends not just upon the language used but also upon the *environment*: the supporting tools and facilities that determine how separately-written programs can be combined, how debugging is done, what is required to make a change in a program and examine the effects of the change, and the like. It is also useful to distinguish two aspects of the language itself, the *model of computation* on which the language relies, and the *notation* used to express programs within this model.

The model of computation could also be called the abstract machine which executes programs in the language. It is the mechanism which must be understood to predict the course and outcome of computations. I will not be making fine technical distinctions among models here. For example, I will consider the model of computation in Fortran and Ada to be substantially the same, even though the Ada model is, when considered in detail, much more complex. Broadly, both languages assume a combination of expression evaluation and sequential execution of statements. Functional programming, to choose a contrasting model, uses expression evaluation extended to include function-valued functions, but not sequential execution.

I will have less to say about notations, which are the conventions used in writing programs. I will do little more than suggest ways in which attention to the demands of *comprehending* programs may influence the choice of notation.

I must warn the reader not to expect a clear and principled sorting out of notation from model, and model from environment. These aspects of a programming system are tightly intertwined, and while I think it useful to try to distinguish them in organizing a discussion I am not sure that rigid logical distinctions among them are possible, or useful.

Models of computation.

The model of computation in a language for non-specialists must meet two criteria. First, it must be easy to grasp, so that only limited time and effort suffices to permit the new user to anticipate the course of computations and their result. Second, it must be easy to map the questions or behaviors the user is interested in in his or her application domain into computations in the model. How do the models associated with various classes of language measure up?

Ordinary procedural languages. As already mentioned, the most familiar languages, including Fortran, Cobol, Algol, Basic, Pascal, C, Ada, and many others share a model of computation in which sequential execution of statements is the basic process. All of these add a facility for the evaluation of expressions, so that the separate arithmetic operations required in evaluating a formula, for example, need not be explicitly ordered.

In its simplest form such a model is not hard to understand, but unfortunately it is also very awkward to use unless some of a bewildering collection of extensions are added, bringing in iteration, recursion, subroutine linkages (and parameter passing), and such other notions that make programming a time-consuming subject of study. Even Basic, designed expressly to support the non-specialist user, has been relentlessly extended. Indeed, one can argue that the structured programming movement, which is surely a good thing from the point of view of the specialist user, requires the replacement of the naive computational model in which statements are executed in order with a more complex and abstract variant in which groups of statements are managed somehow.

The sequential model fits some problems well and others poorly. Though it is surprising to programmers, non-programmers find it quite unnatural, and difficult, to write out the steps required to find the average of a collection of numbers. In the user interface domain, it is difficult to specify the simultaneous movement of two or more graphical objects, because of the need to distribute the flow of control over the objects being moved.

Object-oriented languages. The model of computation here is of a collection of objects which exchange messages, that is, requests for particular behaviors. But just under the covers there is another computational model in which the behaviors associated with messages are carried out. In Smalltalk, this is a sequential model; in C++ it is the model for C; in object-oriented extensions to Lisp it is the model for Lisp, and so on. Thus object-oriented languages do not provide a model of computation which can be used *instead of* other models, but rather one that can be used *on top of* another model.

The consequence for the non-specialist learner is obvious. The object-oriented model must be understood *in addition* to some other model, not instead of it. As the name suggests, the C++ learner learns C and then some, not a set of concepts that can supplant the complexity of C.

On the positive side, the object-oriented model does appear to fit many tasks better than the sequential model. Modern user interfaces, in particular, can be viewed quite naturally as collections of intercommunicating objects. But because of the dependence on an underlying model for producing behaviors, usually a sequential model, some problems remain. Animating multiple objects simultaneously is hard in Smalltalk, for example, just as it is in a sequential language. DiSessa's Boxer (diSessa 1985, diSessa and Abelson 1986), a language intended for non-specialists, had to extend and complicate the computational model to include graphical objects with their own independent control streams in order to solve this problem.

Functional languages. Functional languages (for example ML or pure Lisp; see Sethi 1989 for an introduction to this and many other topics discussed here) dispense with sequential execution and offer only expression evaluation as a computational model, with the significant extension that functions can return functions as values and accept them as operands. This is an elegant model of great flexibility, permitting striking economy of expression for many complex computations. But indications so far are that it is not easy or natural for non-specialists to understand.

This problem points up sharply the inadequacy of our intuitions, and our theoretical notions, in the area of conceptual complexity. Why should this elegant and easily-described model be hard to understand? The notion of function is already a difficult abstraction for many people, and the notion of functional, or function which produces or acts upon functions, is that much farther removed from concrete experience, it appears. However, we may want to reserve some skepticism about this. It may be that *the right way of explaining* functional programming has not been found, and that the difficulties people have now are not unavoidable. By contrast, the sequential model (with the extensions needed to make it useful) seems inherently complex, that is, not admitting a simple description at any level of abstraction.

Like the sequential model, the functional model is a good fit to some tasks and a bad one for others. Problems involving the control of ongoing processes, including animation, do not seem to have natural treatments. Handling user actions on a diverse collection of things on the screen, which is natural in the object-oriented model, seems very unnatural in functional terms. Even if one uses stream objects as inputs to and values of functions, so that inputs and values can be extended in time, it does not seem natural to think of a step in a complex interaction as a piece of a function evaluation. But as with the functional model itself it may be that ingenuity will provide a way to make this appear natural, and make it easy to think about.

Equational languages. The model underlying these languages, also called rewriting systems, is the replacement of patterns; see O'Donnell (1985) for a description. An equational program for simplifying algebraic expressions might include a rule indicating that instances of the pattern A*1 should be replaced by the pattern A, along with many other such rules. An expression would be simplified by searching in it for occurrences of to-be-replaced patterns, and replacing them, until no to-be-replaced pattern matches the resulting expression. The model is conceptually simple, like that of functional programming, but much more concrete. The abstract notion of variable, possibly with some restrictions on possible values, is used in specifying the patterns, but no other abstractions are involved in understanding how execution proceeds. I know of no evidence indicating whether or not this apparently simple and clear model is simple and clear in practice, to non-specialist users.

The pattern replacement model is a natural fit for many symbol-manipulation problems, though ingenuity is sometimes needed in devising replacement rules that perform the desired operations while satisfying the technical requirements needed to make the execution process tractible. For example, it may be required that no two patterns can ever match the same expression.

Beyond this sort of difficulty, the pattern replacement model does not seem to fit situations in which the task is to control a process, or to produce a sequence of actions. O'Donnell (1985) notes this kind of exception, aiming to apply equational programming only in situations in which computation can be conceived as a process that generates answers for questions. Graphical user interfaces of the kind already discussed do not have this character.

Logic programming. The model of computation here is theorem proving (Kowalski 1979). A program is a collection of axioms and rules of inference, and it is executed by asking a theorem-prover, acting as interpreter, to attempt to prove a given assertion. The common logic programming language, Prolog (see for example Clocksin and Mellish 1987), fills in this model by guaranteeing that the inference process will go forward in a predictable order, and allowing some of the steps in the inference process to have side effects, such as displaying a value. These features have the effect of creating a sequential model of computation which coexists with the logic model.

Without these side effects, logic programming has much the character, and limitations, of equational programming. With side effects it gains much of the power of the ordinary sequential model, and Prolog systems exist that support drawing, dragging, and such operations via side effects. Such systems can be used to create graphical user interfaces of the familiar kind.

Ennals and coworkers report success in teaching the basics of logic programming to school children. See Ennals, Briggs and Brough (1984) for a discussion especially pertinent to this paper. Experience does not suggest that the more complex model that includes side effects is especially easy to learn, though it may be no more difficult than the sequential model. Even the basic model appears to be more difficult to describe than the functional model, or the pattern replacement model, though as we have seen abstract simplicity and concrete understandability cannot be equated.

Constraint languages. Sussman and Steele (1980) describe a programming system in which the computational model is the maintenance of constraints asserted to hold among related quantities. A typical constraint might specify that the voltage drop across a simulated electrical resistance must equal the product of the value of the resistor and the current flowing through it, as dictated by Ohm's law. In fact the idea of constraint languages grew out of the work of Sussman and colleagues in modelling electronic circuits.

There has been too little experience with constraint languages to offer much evidence about the understandability of the ideas involved. There is some complexity inherent in the fact that most constraints admit of more than one way of responding to changes, and the user must in some way indicate what ways are acceptable. In the above example, if E is constrained to be the product of I and R, and E changes, then the constraint could be satisfied by changing I, changing R, or changing both in combination. In this case the user would wish to indicate that R is constant, so that only I would be changed in response to the change in E.

Borning and colleagues (Borning 1981, 1986; Duisberg 1986) have used closely related ideas to manage interactive graphical displays and animation. In this latter work one can attach the end of one line segment to the midpoint of another, so that as the second segment is interactively moved or stretched the attached segment is moved as needed. The model appears to provide a natural fit for a variety of tasks involving interrelated quantities or objects, including highly interactive interfaces for physical simulations.

Spreadsheets. The spreadsheet computational model can be seen as a simple specialization of the constraint model in which constraints are strictly unidirectional. A given quantity in the spreadsheet can be related to other quantities by a formula, so that when any of these other quantities change the dependent quantity is recalculated. Unlike the general constraint model there is no provision for changes in the calculated quantity to be reflected back on the quantities on which it depends. Thus if E is calculated from I and R it would be possible to change I or R but not to change E.

It can be argued that the spreadsheet has done more to make computing accessible than any other development since Fortran. Clearly the computational model it rests on is easily understood by a wide spectrum of users, and its applications have spread far beyong the sphere of simple financial calculations for which it was designed. However, in its ordinary form the model does not support interactive graphics or the control of sequences of actions, in common with functional programming and pattern replacement. Familiar implementations of the model do not support symbolic operations, but there appears to be no obstacle to doing this; for example see van Emden, Ohki, and Takeuchi (1986) on the use of the model to support logic programming.

Taking stock of this discussion, it appears that only the spreadsheet model clearly meets the test of ready comprehensibility, but evidence one way or the other about some of the other models is scarce. As might be expected, none of the models handles the whole spectrum of computing tasks in a natural manner. The conceptually simplest models do not provide a good fit for user interface operations, in particular.

Notational issues.

Apart from the requirement to grasp the underlying computational model of a language, the user must cope with the notational conventions used in programs. I wish to bring up just two points in this connection here, both of which suggest dissenting views on some common language design practices.

First, virtually all languages make heavy use of variables. Backus' radical functional programming proposal (Backus 1978) attracted much attention by eliminating variables, but the proposed alternative notation was clumsy and subsequent work in functional programming has reintroduced them, at least in the limited guise of parameters.

Backus objected mainly to the complexities involved in reasoning formally about programs with variables. Our non-specialist users will probably not attempt such formal analysis, but one can argue that even informal reasoning and even simple comprehension of programs may be interfered with by the use of variables.

What's the problem? In almost all cases, an occurrence of a variable can be understood only by reference to some information not available in the immediate local context. Thus a comprehender must integrate information from two separate sources in arriving at a meaning. Consider for example the Prolog statement

```
grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
```

In reading this rule it cannot be determined that Z is existentially quantified in its occurrences without using the fact that it does not occur on the left. Similarly, the significance of the second Z is crucially determined by the occurrence of the first Z. Notice that the functional equation

grandparent = parent $\circ$ parent,

where $\circ$ denotes functional composition, avoids these complications (though it does not express as much as the Prolog example.) Uses of variables to express simple "wild cards" in pattern matching, where the variable is used only once, can be understood locally, but all other uses seem to involve some kind of non-local reference.

A somewhat similar problem is posed by the variety of devices used to modularize the presentation of programs. A feature of block structured languages is the freedom they provide to reuse names in different parts of a program, with some control over whether the names refer to the same or different things. From the point of view of the program comprehender, this useful freedom is a real nuisance. The comprehender cannot know what object a name in a block refers to without reading the declarations in that block, and, in general, several other blocks. This is the same demand for integration of information from distributed sources that is posed by variables.

The very useful inheritance feature of object-oriented languages creates this same problem for comprehenders. Since an object can inherit its response to a message from any of its enclosing classes, working out the behavior of an object involves examination of an often extensive body of code. Indeed, object-oriented programming environments often provide some kind of browsing tool to help readers find the definitions they need.

Environment issues.

Modern programming environments provide a wide range of tools for manipulating code, as well as helping the user to examine and manipulate the state of computations. I want to mention two points here that are especially important for supporting non-specialist users. First, we need to move away from our traditional view of programs as things built bit-by-bit from scratch. Rather, we should strive to support programming by combining and modifying existing examples of working programs. This approach has been widely advocated for specialist programmers in the software engineering literature, for example by Winograd (1979). But it may not be apparent that the same idea has critical advantages for the non-specialist. Perhaps the key point is that working from examples lessens the requirement to understand everything about what one is doing. One only has to understand enough about an example to know what to change and how to change it. Lewis and Olson (1987) developed this and related points further in a precursor of the present paper.

Second, typical programming environments maintain a *barrier* between users and programs. Input data provided by the user passes behind this barrier, where it cannot be seen or manipulated. Only those computed results that the program chooses to reveal are passed back through the barrier. If one of many input items is wrong, typically all data must be resupplied and the entire computation repeated. Draper (1986) pointed out this problem, and called for a different style of environment in which user and program share a common data area, represented on the screen. Both parties can examine and modify the shared data as needed. Input values and computed results are all posted in the shared data space. Draper also pointed out that the spreadsheet provides just such an environment, and that the ability to manipulate input data and view results in a natural and convenient way probably contributes much to the popularity of the spreadsheet with users who have trouble imagining what is going on behind the barrier in conventional environments.

Current work: NoPumpG and ChemTrains.

My colleagues and I have been working over the last two years to realize systems that can capitalize on some of these observations about existing programming systems in providing superior support for non-specialist programmers. Our emphasis in this work, as in some of the discussion here, has been on making interactive graphics accessible to these non-specialist users.

NoPumpG (Lewis 1987) extends the spreadsheet computational model and environment to control graphical interactions and animation. The key notion is that graphical objects, such as line segments, are associated with cells in a spreadsheet in such a way that the coordinates that specify the position of an object are held in these cells. If the object is dragged by the user, the associated cells are updated to reflect the new position. If the value of one of these cells is changed by a calculation in the spreadsheet the corresponding object is moved. Thus input to the spreadsheet can be accomplished by moving graphical objects (which might take the form of slide controls, if desired), and output of results from the spreadsheet can be reflected

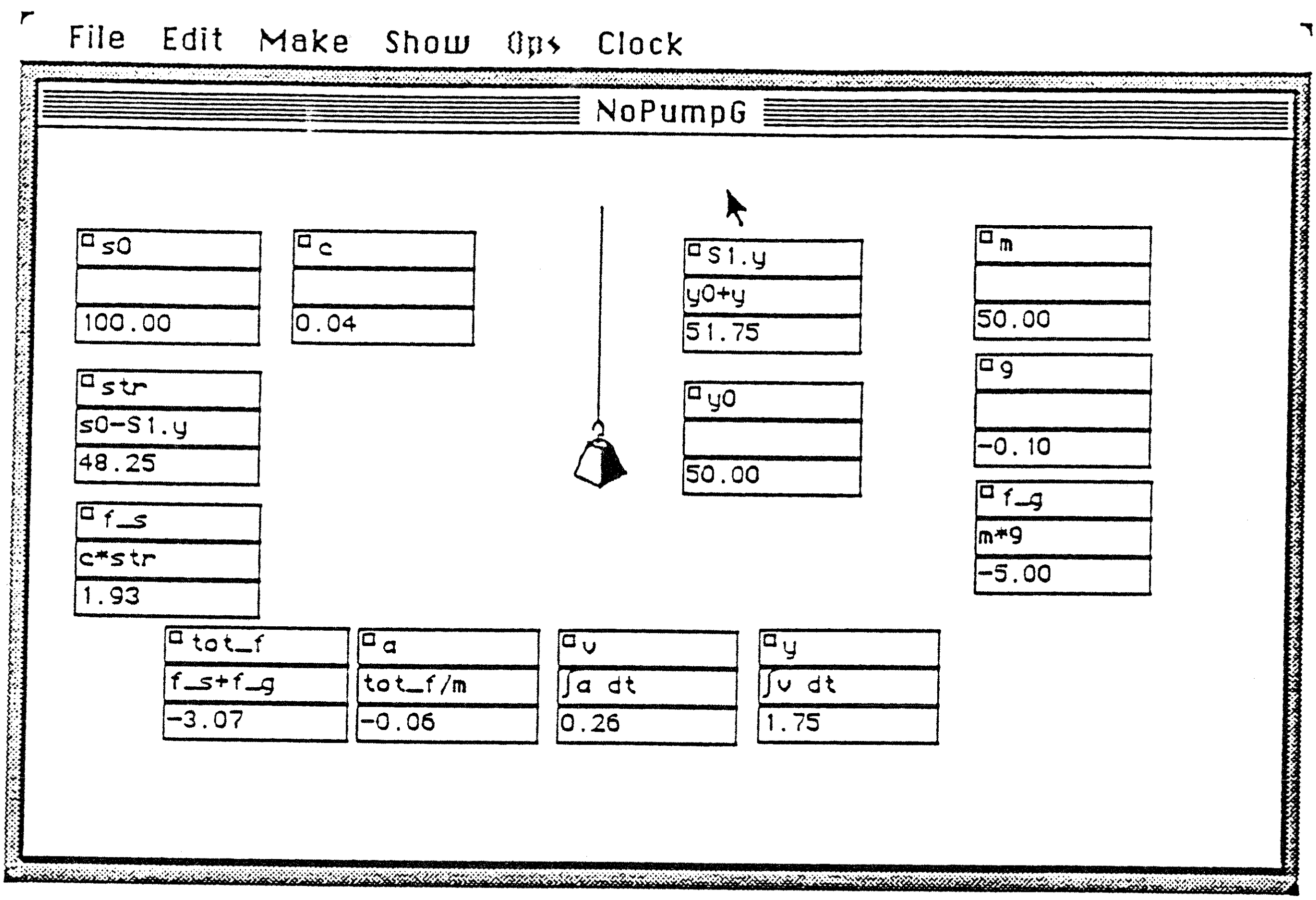

Figure 1: A NoPumpG simulation of a mass suspended from an elastic cord. The formulae in the cells show how the position of the mass is specified from basic physical principles, with the force due to the cord calculated on the left, the force due to gravity on the far right, and the change in position due to these forces at the bottom. The cell S1.y, which determines the vertical position of the mass, contains a value obtained by adding the change in position to an initial height of the mass. When the clock runs the mass oscillates up and down because of the time integrals in some of the formulae.

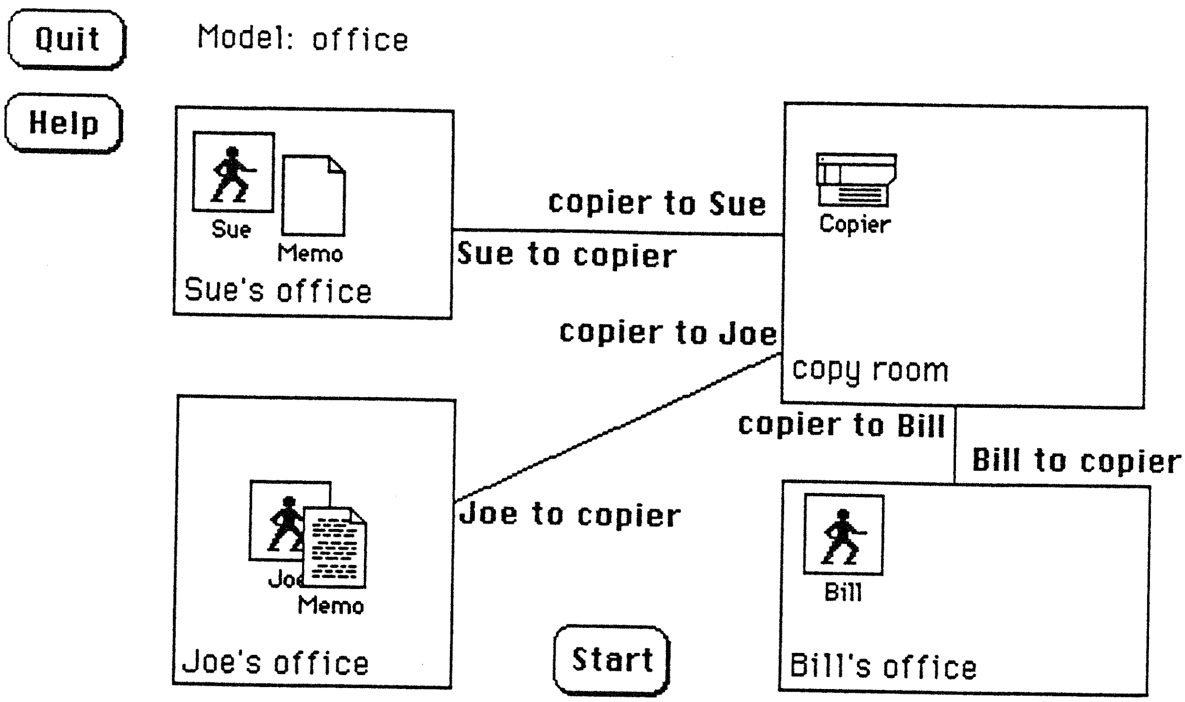

Figure 2: A simple ChemTrains model.

graphically in the positions of indicators (say) or the heights of bars. Animation is accomplished simply by placing a clock in one of the spreadsheet cells. If a cell containing a coordinate is made to depend directly or indirectly on the clock, the associated object will move as the clock runs. Many objects can be simultaneously and independently animated in this way, without confronting the difficult problem of sharing a sequence of update operations among different objects. Figure 1 shows a simple physical simulation in NoPumpG.

A prototype implementation of NoPumpG has been running for the last two years, but it is too limited in function to provide useful support for realistic programming tasks. We are currently designing and implementing an extended version intended to support the creation and modification of robot control interfaces.

ChemTrains, a system now being designed and prototyped, is intended to explore a different portion of the design space from NoPumpG, and to support graphical depiction of behavior driven by qualitative, not quantitative models. The analytic geometry which is at the heart of NoPumpG's treatment of graphics is clumsy for modelling motions which the user can easily visualize or draw, but for which he or she has no mathematical or physical specification. Similarly, while an object represented in NoPumpG can change its state and appearance, such changes must be controlled by numerical computations rather than qualitative logic.

ChemTrains is intended to permit non-specialists to create and manipulate models like that shown in Figure 2. The objects in the model behave according to rules provided by the user; for example, memos flow along the paths in the diagram, and a memo which arrives at the copying machine will be replicated. The figure is taken from our Hypercard prototype, which is only partly functional as of now.

Objects in ChemTrains are bundles of attributes. Objects are created and destroyed, and their attributes changed, by a process resembling a chemical reaction. Objects which find themselves in the same *place* in a model may react, according to reaction rules provided by the user. The reaction process is intended to have a character similar to the pattern replacement model of computation discussed above, and in particular to be equally concrete and simple.

Pictures are associated with an object in a manner dependent upon its attributes; thus a reaction can change the way in which an object is displayed on the screen.

Places are connected by *paths*. Each end of a path has an associated filter, which passes or rejects objects according to their attributes. An object in a place will leave the place and move along a path if its attributes satisfy the path's filter.

The specification of reaction rules in ChemTrains is influenced by the arguments about comprehensibility of notation developed above. In contrast to rules in equational programs, they contain no variables. This entails some loss of expressive power, in that some complex patterns cannot be specified, and experience must show whether any gain in comprehensibility balances this deficit.

Reaction rules are *global* in ChemTrains, contrary to one's language design instincts that would permit behaviors to be specified within an object, or possibly within a place. The intent is to reduce the integration of information from different sources that is required to understand what behavior has been specified. If rules of behavior are specified once and for all in one place, then one never need check to see if new rules are in effect in understanding how some new part of the model will behave.

Research agenda.

I hope that this discussion has made clear that the world of possible programming languages, and indeed computational models, is a large one, and that there is much territory to explore in seeking language ideas that may extend the power of programming to new users and new applications. I think it also makes clear that the theoretical support for this search is weak, and should be improved. As designers we must make judgements about "comprehensibility" and "conceptual clarity" without adequate models of the mental processes which determine what structures have or lack these characters. My colleagues and I are embarked on work to strengthen these foundations (see e.g. Kintsch 1988, Mannes and Kintsch 1988), but there is far to go.

An avenue we are not exploring yet in our work is that of programming by modification. Following the argument sketched above I suspect that no approach that neglects this idea can be fully successful.

I think a special urgency and excitement attaches to this broad area of research when we consider the advent of massively parallel machine architectures. As computer scientists we may not all share an interest in the support of non-specialist programmers, which is the problem I have used to structure this discussion. But we can all recognize that the challenge of exploiting these radically new machines will force us to discover new ways to think about computing, and new languages to express these thoughts.

Acknowledgements: I thank the National Science Foundation, NASA, and the Air Force Human Resources Laboratory for research support. Nicholas Wilde, Victor Schoenberg, and Elizabeth Richards contributed to the development of these ideas, as did Peter Polson, Walter Kintsch, Stephanie Doane, and the participants in a 1986 workshop on cognitive issues in programming: Gary Olson, Robert Balzer, Gerhard Fischer, Thomas Green, and Donald Norman. Ada is a trademark of the Ada Joint Program Office, Department of Defense, United States Government.

References.

Backus, J.W. (1978) Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, **21**, 613-641.

Borning, A. (1981) The programming language aspects of ThingLab, a constraint oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, **3**, 353.

Borning, A. (1986) Defining constraints graphically.*Proc. CHI'86 Human Factors in Computing Systems*, New York: ACM, 137.

Clocksin, W.F. and Mellish, C.S. (1987) *Programming in Prolog*. New York: Springer.

diSessa,A.A. (1985) A principled design for an integrated computational environment. *Human-Computer Interaction*, **1**, 1.

diSessa, A.A. and Abelson, H. (1986) Boxer: A reconstructible computational medium. *Communications of the ACM*, **29**, 859.

Draper, S.W. (1986) Display managers as the basis for user-machine communication. In D.A. Norman and S.W. Draper (Eds.) *User Centered System Design: New perspectives on human-computer interaction*. Hillsdale, NJ: Erlbaum, 339.

Duisberg, R.A. (1986) Animated graphical interfaces. *Proc. CHI'86 Human Factors in Computing Systems*. NewYork: ACM, 131.

Ennals, R., Briggs, J. , and Brough, D. (1984) What the naive user wants from Prolog. In J.A. Campbell (Ed.) *Implementations of Prolog*. Chichester, UK: Ellis Horwood, 376-386.

Goldberg, A. and Robson, D. (1983) *Smalltalk-80: The language and its implementation*. Reading, MA: Addison-Wesley.

Kintsch, W. (1988) The role of knowledge in discourse comprehension: A construction-integration model. *Psychological Review*, **95**, 163-182.

Kowalski, R.A. (1979) *Logic for problem-solving*. New York: Elsevier-North Holland.

Lewis, C. (1987) NoPumpG: Creating interactive graphics with spreadsheet machinery. Technical Report CUCS-372-87, Department of Computer Science, University of Colorado, Boulder, CO.

Lewis, C. and Olson, G.M. (1987) Can principles of cognition lower the barriers to programming? In G.M. Olson, S. Sheppard, and E. Soloway (Eds.) *Empirical studies of programmers: Second workshop*. Norwood, NJ: Ablex, 248-263.

Mannes, S.M. and Kintsch, W. (1988) Action planning: Routine computing tasks. In *Proc. 10th Annual Meeting of the Cognitive Science Society*. Hillsdale, NJ: Erlbaum, 97-103.

O'Donnell, M.J. (1985) *Equational logic as a programming language*. Cambridge, MA: MIT Press.

Sethi, R. (1989) *Programming languages*. Reading, MA: Addison-Wesley.

Sussman, G.J. and Steele, G.L. Jr. (1980) CONSTRAINTS-- A language for expressing almost-hierarchical descriptions. *Artificial Intelligence*, **14**, 1-39.

van Emden, M.H., Ohki, M., and Takeuchi, A. (1986) Spreadsheets with incremental queries as a user interface for logic programming. *New Generation Computing*, **4**, 287.

Winograd, T. (1979) Beyond programming languages. *Communications of the ACM*, **22**, 391-401.