Beyond Programming Languages

Terry Winograd
Stanford University

As computer technology matures, our growing ability to create large systems is leading to basic changes in the nature of programming. Current programming language concepts will not be adequate for building and maintaining systems of the complexity called for by the tasks we attempt. Just as high level languages enabled the programmer to escape from the intricacies of a machine's order code, higher level programming systems can provide the means to understand and manipulate complex systems and components. In order to develop such systems, we need to shift our attention away from the detailed specification of algorithms, towards the description of the properties of the packages and objects with which we build. This paper analyzes some of the shortcomings of programming languages as they now exist, and lays out some possible directions for future research.

Key Words and Phrases: programming, programming languages, programming systems, systems development

CR Categories: 4.0, 4.20, 4.22, 4.40

Introduction

Computer programming today is in a state of crisis (or, more optimistically, a state of creative ferment). There is a growing recognition that the available programming languages are not adequate for building computer systems. Of course, as any first year student of computation theory knows, they are logically sufficient. But they do not deal adequately with the problems we face in the day-to-day work of programming. We become swamped by the complexity of large systems, lost in code written by others, and mystified by the behavior of our almost debugged systems. When we look to the integrated multiprocessor systems that will soon dominate computing, the situation is even worse.

This crisis in software production is far greater (in overall magnitude at least) than the situation of the early 50's that led to the development of high level languages to relieve the burden of coding. The problems are harder to solve, and the costs of not solving them are in the hundreds of millions. "The symptoms appear in the form of software that is nonresponsive to user needs, unreliable, excessively expensive, untimely, inflexible, difficult to maintain, and not reusable." [3, p. 26.] There are many ways to improve things a little, and they are being tried. But to achieve a fundamental jump in our programming capacity, we need to rethink what we are doing from the beginning.

The Problem

I believe that the problem lies in an obsolete view of programming and programming languages. A widely accepted view can be paraphrased: The programmer's job is to design an algorithm (or a class of computations) for carrying out a task, and to write it down as a complete and precise set of instructions for a computer to follow. High level programming languages simplify the writing of these instructions by providing basic building blocks for stating instructions (both control and data structures) that are at a higher level of the logical structure of the algorithm than those of the basic machine.

Author's address: T. Winograd, Computer Science Dept., Stanford University, Stanford, CA 94305.

This view has guided the development of many programming languages and systems. It served well in the early days of computing, but in today's computational environment, it is misleading and stultifying. It focuses attention on the wrong issues and gives the most important aspects of programming a second-class status. It is irrelevant in the same sense that binary arithmetic is irrelevant—the things it deals with are a necessary part of computing, but should play a subsidiary rather than central role in our understanding.

As computer technology (for both software and hardware) matures, our growing ability to create complex systems has led to three basic changes in the nature of programming:

1. Computers are not primarily used for solving well-structured mathematical problems or data processing, but instead are components in complex systems.

According to Department of Defense studies [3], more than half of DoD software costs are associated with "embedded computer systems." An embedded computer system is "one that is logically incorporated in a larger system—e.g. an electromechanical device, a tactical system, a ship, an aircraft, or a communications system—whose primary function is not computation." Of course, embedded computer systems are not unique to the DoD. Many computer scientists spend the majority of their time dealing with embedded computer systems such as message systems and text editing and formatting systems. The example discussed below is a large embedded system in a university context. As the microcomputer revolution continues, this change will become even more extreme. There will be computers embedded in every conceivable kind of electrical and mechanical system, and applications like text editing and message processing will become widespread on the scale of today's telephone network. As Fisher [3] notes:

> Embedded computer software often exhibits characteristics that are strikingly different from those of other computer applications.... Many embedded computer applications require software that will continue to operate in the presence of faults.... For example, the applications may require the monitoring of sensors, control of equipment display, or operator input processing. They must interface special peripheral equipment.... Software must sometimes be able to respond at periodic (real time) intervals, to service interrupts within limited times, and to predict computation times.... In many applications... it is necessary to access, manipulate and display large quantities of data. Much of this data is symbolic or textual rather than numeric, and must be organized in an orderly and accessible fashion.

2. The building blocks out of which systems are built are not at the level of programming language constructs. They are "subsystems" or "packages," each of which is an integrated collection of data structures, programs, and protocols.

By making use of existing modules, a programmer can deal with design at a higher level, creating an integrated system with capacities far greater than a program that could be built with the same effort "from scratch." Components for general tasks (such as memory management, user interface, file management, and network communication) can be designed once, rather than reconstructed for each system that needs the capability. Unfortunately, in current programming practice this is more of an ideal than a reality. The difficulties of using existing packages often make it easier to replicate their function than to integrate them into a system. The only way such packages are generally used is by building programs within an "operating system" that provides facilities within a uniform environment. Only those packages that are needed by the majority of users find their way into operating systems, and the facilities for using them are complex and ad hoc relative to modern programming languages. We need better ways to deal with the problems of "programming in the large."

As noted in an IBM report on large software systems [1, p. 6], "... The understanding of how programs work individually and in cooperation with each other... remains very difficult to generalize, teach, communicate, or even preserve, due to lack of easy 'externalization,' i.e. representation, of ideas." Once we begin to deal with networks of processors, it will become even more important to deal explicity with properties of systems which integrate many independent components.

3. The main activity of programming is not the origination of new independent programs, but in the integration, modification, and explanation of existing ones.

This third change grows from the first two. As we are able to build more complex programs, we develop systems that grow to fit an environment of needs, rather than carrying out a single well-specified task. An embedded system (such as one for airline reservations or text preparation and formatting) evolves over many years, increasingly fitting the needs of those who use it, and incorporating new capacities as hardware advances make them practical. The DoD study [3, pp. 24–25] noted that, "The programs are frequently large (50,000 to 100,000 lines of code) and long-lived (10 to 15 years).... Change is continuous because of evolving system requirements—annual revisions are often of the same magnitude as the original development.... The majority of costs are incurred in software maintenance rather than development."

As additional needs and possibilities arise, it should be possible to modify and combine existing well-tested systems rather than build new ones. In most cases, the needs for continuity in the use of the system (including "upward compatibility" for existing data and user programs) make it impractical to start from scratch. Using current programming techniques, systems often reach a point at which the accretion of changes makes their structure so baroque and opaque that further changes are impossible, and the performance of the system is irreversibly degraded. The situation is further complicated by the fact that modifications are often done not by the original builders, but by new programmers with

an incomplete or inaccurate understanding of the system. As Wulf [4] points out, "Another component of the software crises is less commonly recognized, but, in fact, is often more costly... namely, the extreme difficulty encountered in attempting to modify an existing program.... The cost of such evolution is almost never measured, but, in at least one case, it exceeded the original development cost by a factor of 100."

The difficulties in building and modifying large systems have long been recognized and lamented. They have led to various schools of "structured programming" and to the emphasis on restriction and discipline in the design and use of programming languages. There is a large body of lore shared by practicing programmers, providing ways to recognize the problems and guidelines for avoiding the most obvious of them. These include bodies of standards and conventions designed to avoid misunderstanding and miscommunication. But ultimately the solution lies not in greater discipline but in more adequate tools.

Towards a Solution

Just as high level languages enabled the programmer to escape from the intricacies of a machine's order code, *higher level programming systems* can provide help in understanding and manipulating complex systems and components. We need to shift our attention away from the detailed specification of algorithms, towards the description of the properties of the packages and objects with which we build. A new generation of programming tools will be based on the attitude that what we say in a programming system should be primarily *declarative*, not *imperative*: The fundamental use of a programming system is not in creating sequences of *instructions* for accomplishing tasks (or carrying out algorithms), but in expressing and manipulating *descriptions* of computational processes and the objects on which they are carried out.

To some extent, this attitude coincides with current work on specification languages, structured programming formalisms, and denotational theories of programming semantics. All of these emphasize the description of the results of computations, rather than instructions for carrying them out. Dijkstra [19], for example, describes a methodology for understanding programs in terms of predicate transformers from an initial to final state. A predicate transformer is "a rule telling us how to derive for any post-condition R the corresponding weakest precondition... for the initial state such that the attempted activation will lead to a properly terminating activity that leaves the system in a final state satisfying R." He argues that "a program written in a well-defined programming language can be regarded as a mechanism that we know sufficiently well, provided we know the corresponding predicate transformer." Languages such as Lucid [16] carry this philosophy directly into the programming formalism. Lucid is a strictly denotational language, and the statements of a Lucid program can be interpreted as true mathematical assertions about the results and effects of the program.

There is a critical difference, though, which is lost if we look only at the distinction between imperative and declarative. In stating that a programming system helps us to manipulate "descriptions of computational processes," we are saying something quite different from "assertions about the results and effects." In order to clarify this, it is useful to distinguish three types of specification:

(1) *Program specification.* A formal structure which can be interpreted as a set of instructions for a given machine. This is the imperative style of traditional programming languages.
(2) *Result specification.* A process-independent specification of the relationships between the inputs (or initial state), internal variables, and outputs (or resulting state) of the program. This is the specification style advocated by Dijkstra and in work on program verification and transformation.
(3) *Behavior specification.* A formal description of the time-course of activity of a machine. Any such description selects certain features of the machine state and action (e.g. input and output activities, use of memory resources, communication events among processes), without specifying in full detail the mechanisms which generate these.

A behavior specification is like a result specification in that it characterizes what will be done, rather than giving commands for how to do it. It is different in that it is explicitly concerned with issues of sequence, and potentially with real-time measures as well. In practice, result specifications for systems of significant size factor the specification, often using sequence as a dimension of factorization. In a behavior specification, the time-course description is an essential part of the description of what the system as a whole does, not a convenience for factoring it into result-producing modules.

Programming in the future will depend more and more on specifying behavior. The systems we build will carry out real-time interactions with users, other computers, and physical systems (e.g. for process control). In understanding the interaction among independent components, we will be concerned with detailed aspects of their temporal behavior. The machine must be thought of as a mechanism with which we interact, not a mathematical abstraction which can be fully characterized in terms of its results.

Current languages provide only scattered specialized mechanisms for description of either results or process. Declarations are a ubiquitous form of low level description, and assertions about the state of a computation are occasionally included. But if we look at what a programmer would say about a program to a colleague who wanted to work on it or use it, very little of the description appears anywhere in the "code." If (either because of

idealism or coercion) the programmer has included comments, they can provide useful but local description. If further (almost always through coercion) the program has been documented, there may be more global descriptions. In large systems, documentation will include a careful specification of protocols and conventions not belonging to any one program, but vital to the system as a whole. It may also include process descriptions along with the result descriptions. But these various pieces of description are scattered, and for the most part not accessible in any systematic fashion.

I want to turn the situation on its head. The main goal of a programming system should be to provide a uniform framework for the information that now appears (if at all) in the declarations, assertions, and documentation. The detailed specification of executable instructions is a secondary activity, and the language should not be distorted to emphasize it. The system should provide a set of tools of generating, manipulating, and integrating descriptions of both results and processes. The activity that we think of as "writing a program" is only one part of the overall activity that the system must support, and emphasis should be given to understanding rather than creating programs.

The rest of this paper explores some of the consequences of this view, and makes some suggestions as to what a higher level programming system might look like. It is an attempt to lay out the problems, not to solve them. It will take many years of research before these speculations can be backed up with concrete evidence.

A Motivating Example

One of the best ways to understand a general view of computing is to look at the examples used by those who hold it. Both Knuth's opus, *The Art of Computer Programming* [21], and Dijkstra's *A Discipline of Programming* [19] begin by discussing the Euclidean algorithm. In a clear and simple way it exemplifies the basic notions of algorithm and program as a mathematical abstraction. The Lisp 1.5 Manual [55] includes the Lisp interpreter written in Lisp, illustrating the manipulation of symbolic structures and the ability to treat the programs themselves as symbolic data. The view proposed in this paper is best illustrated by an example at a very different level.

Imagine that you have come to work as a system designer and programmer for a large university. You are given the following system development task.

The current situation. The univeristy administration has a computer system for scheduling and planning room use. Users at several sites on campus access the system through interactive graphics terminals that display building floorplans as well as text and other graphic data. There is a minicomputer running each cluster of terminals, connected to the central campus facility by a communication network. Each cluster has a device capable of printing out floorplans and graphic data like that displayed on the terminals. Large-scale data storage is at the central campus computer facility.

The system keeps track of the scheduled use of all rooms, such as long-term lab and office assignments, regular class schedules, and special events. It is able to answer questions about current scheduled use and availability. Querying is done from the terminals, through structured graphic interaction (menus, standardized forms, pointing, etc.) and a limited natural language interface. The system does not make complex abstract deductions, but can combine information in the database to answer questions like "Is there a conference room for 40 people with a projection screen available near the education building from 3 to 5 on the 27th?" Users with appropriate authorization enter new information, including the scheduling of room use and changes to the facilities (including the interactive drawing of new or modified floorplans). In addition to the current assignments, the system keeps a history of use for analysis. Standard statistical information and data representations (such as tables, bar charts, graphs, etc.) are produced on demand for use in long-range planning.

Your assignment. The dean wants the system to provide more help in making up the quarterly classroom assignments. It should be possible to give it a description of the courses scheduled for a future quarter and have it generate a proposed room assignment for all courses. In deciding on assignments, the system should consider factors such as expected enrollment (using past data and whatever new information is available on estimated enrollments), the proximity of rooms to the departments and teachers involved, the preference for keeping the same location over time, and the nature of any special equipment needed. It should print out notices summarizing the relevant parts of the plan for each teacher, department, dean, and building supervisor. Any of these people should be able to use the normal querying system to find out more about the plan, including the motivations for specific decisions. Properly authorized representatives of the dean's office should be able to request changes in the plan through an interactive dialog with the system in which alternatives can be proposed and compared. When a change is made, the system should readjust whatever is necessary and produce new notifications for the people affected.

A system like this is just at the edge of our programming powers today. It would take many programmer-years of effort to build and would be successful only if the project were managed extraordinarily well, even by the standards of the most advanced programming laboratories. But it is not hard because of the intrinsic difficulty of the tasks the system must carry out. It combines hardware and software facilities that have been demonstrated in various combinations a number of times. Even the question-answering and assignment-proposing components are within the bounds of techniques now considered standard in artificial intelligence.

The problem lies in the difficulties of organizing

complex systems. The integration of all of the components of the "initial system" would be a major achievement, calling on our best design tools and methodologies. The idea that a new programmer could come in to such a system and make changes widespread enough to handle the "assignment" is enough to make an experienced programmer shudder. It would be hard enough to add new types of questions (such as explanations for decisions), new information (such as distances and estimated enrollments), and new output forms (such as schedule summaries for departments). But even more, we are trying to integrate a new kind of data (projected plans) into a system that was originally built to handle only a single current set of room assignments and a record of their history. These projected plans must be integrated well enough for all of the existing facilities (including floorplan drawing, question answering, statistics gathering, etc.) to operate on them just as they do with the initial database.

Three Domains of Description

A system of this complexity can be viewed in each of three different "domains," *subject*, *interaction*, and *implementation*. Each viewpoint is appropriate (and necessary) for understanding some aspects of the system and inappropriate for others. We will look at the example from each of these viewpoints in turn, then discuss how they might be embodied in a programming system.

The subject domain. This system, like every practical system, is about some subject. There is a world of rooms and classes, times and schedules, that exists completely apart from the computer system that is understood as referring to them. The room or the course cannot be in the computer—only a pattern of bits which we interpret as a description of it. One of the primary tasks in programming is to develop a set of descriptions that are adequate for talking about the objects being represented. There are descriptions for things we think of as objects (e.g. buildings, rooms, courses, departments) and also for processes (e.g. the scheduling of events). These descriptions are relative to the goals of the system as a whole and embody a basic view of the problem. For example, what it takes to represent a room would be different for this system and for a system used by contractors in building construction.

All too often the development of descriptions in this domain is confused with the specification of data structures (which are in the domain of implementation). In deciding whether we want a course to be associated with a single teacher, or to leave open the potential representation of team-teaching, we are not making a data structure decision. The association of a teacher (or teachers) with a course may be represented in many different data structures in many different components of the system. One of the most common problems in integrating systems is that the components are based on different decisions in the subject domain, and therefore there is no effective way to translate the data structures.

Current work on data structure abstraction [33, 34, 36, and 38] is a step towards the systematic separation of subject domain decisions from their implementation. Representation languages in artificial intelligence provide additional structure for stating the relationships among abstractions [5, 6, 9, and 10]. A programming system needs to provide a powerful set of mechanisms for building up and maintaining "world views"—coherent sets of description structures in the subject domain that are independent of any implementation. Each component can then implement part or all of this in a way that will be consistent with both the structure of that component and the assumptions made in other components.

A complex system like the one described above will need hundreds of different categories of objects. Some of these (such as classrooms and courses) will be unique to the system. Others (such as times and dates, schedules, physical layouts) will be shared across a wide range of systems. They will be related into hierarchies of abstraction. We can think of a *seminar*, a *course*, and a *special lecture* as examples of a more general class of *event*, all having a time, a place, etc. For some purposes, this is the right level of generality. For other purposes, we need to distinguish carefully and use special information associated with each. A major part of building up systems will be the systematic development of descriptions that provide a uniform medium for describing objects and their interactions.

The domain of interaction. Every functioning system can be viewed as carrying on an interaction with its environment. As we will discuss in a moment, the choice of "system" and "environment" is relative to a specific viewpoint, but for the moment let us consider the system as viewed by the users. In this domain, the relevant objects are those that take part in the system's interactions: users, files, questions and answers, forms, maps, statistical summaries and notifications to departments. The processes to be described are those like querying the system, decribing a new event to be scheduled, and proposing a schedule for a new quarter.

The domain of interaction is concerned with descriptions that are largely orthogonal to those in the subject domain. We can talk about a question as having certain characteristics (e.g. looking for a yes-no answer) independently of whether it is about a room or a lecture. We can talk about the filling out of a form without reference to its specific contents. It is also (and more importantly) independent of the domain of implementation. From the traditional viewpoint this independence is a bit more difficult to see than the independence of interaction and subject matter. Whereas a subject domain object (like classroom) clearly cuts across large parts of the system, an interaction object (like a question or the process of filling out a form) is typically handled by a single component and described in terms of its implementation. But

for the same reasons that we want to keep subject domain descriptions independent of their implementations, we must do the same with interaction descriptions.

This view can be applied recursively to subparts and components of the system. In looking at one part (such as the on-site processor and its cluster of terminals) we can view it as an independent system interacting with an environment including both the users and the other parts of the overall system, such as the centralized data store. Even within a single implementation module (e.g. the question-answerer), we often want to describe what is happening as an interaction between several conceptual subsystems (the parser, the semantic analyzer, etc.). As with the larger system, it is vital to keep in mind the distinction between the interaction and implementation domains. In order to usefully view a system as made up of two distinct subsystems, they need not be implemented on physically different machines, or even in different pieces of the code. In general, any one viewpoint of a component includes a specification of a boundary. Behavior across the boundary is seen in the domain of interactions, and behavior within the boundary is in the domain of implementation. That implementation can in turn be viewed as interaction between subcomponents.

It is in the domain of interaction that there is currently the most to be gained from developing bodies of descriptive structures to be shared by system builders. There are already many pieces that can be incorporated, including protocols (e.g. network communication protocols, graphics representation conventions), standardized interaction facilities (e.g. ASKUSER and DLISP (Display Lisp) in Interlisp [57, 58], the Smalltalk display programs [35]), front-end query packages (in various artificial intelligence programs), database standards, and so forth. Currently each of these is in a world and formalism of its own. Given a sufficiently flexible tool for describing and integrating interaction packages, this level of description will be one of the basic building blocks for all systems.

The domain of implementation. Every computer system operates on a set of physical devices with hardwired mechanisms for storing and manipulating data. It is in this domain that we normally think of programming. The detailed choice of algorithms, data structures, and configuration is determined by properties of the hardware and descriptive languages we have available for specifying its behavior. However, it would be a mistake to equate this domain with our current notions of programming. The objects in this domain include everything from individual memory bits and subroutines to subsystems (e.g. the file system, the memory management system, the operating system), running processes, hardware devices, and code segments. They include those things we talk about in programs and the debugger, and those found in machine and hardware manuals. In this domain, as in the others, a uniform system for description is needed, which is not primarily a language for specifying a set of instructions.

In addition to all those things that are directly derivable from the code, the manuals, and the state of the machine, there is also a body of process description. For example, it may be a property of a specific memory-management package that it periodically undergoes a garbage collection period of up to 5 seconds during which no new memory allocations can be made. Such "performance" characteristics may be vital for understanding the interactions of a component with the rest of the system, but are not explicit in its code. Other descriptions bring into focus things that may be implicit in the code. A file system may delete a file if its creation process is interrupted in certain ways that would leave it in danger of being inconsistent. The code that does this may be distributed through various checks and actions, but for the programmer attempting to understand the program it is necessary to have a coherent overview of what is happening.

Similarly, many of the things we think of as properties of data structures are actually conventions spread through the code that manipulates them. Much of the work in structured programming has been to isolate these conventions into access functions that go into a "module" with the data structure [33, 36, and 38]. The object-oriented style of programming encouraged by Smalltalk [35] is another attempt to provide this kind of modularity. A system for implementation description would provide for stating these in a more general way along with those things that now appear only in the comments. As with procedures, data structures can also have implicit properties (e.g. the expected maximal size of variable length fields) that need to be stated explicitly in order for a person to understand how they will interact with other components.

The boundary between hardware and software has become increasingly blurred in the past few years through developments such as microcode, uniform logic arrays, and the extensive use of virtual machine architecture. A programming system based on description would go further in unifying our approach to different levels of architecture. The emphasis is on an overall description of a component rather than the instructions needed to cause some piece of hardware to run it. A piece of software and a piece of hardware designed to achieve the same purpose would have descriptions that differed in details (e.g. timing), and in the specific aspect that described the code (or logic circuits) used to carry out the steps. They might be identical in the domain of interaction, and even to a large degree in the domain of implementation (for example, in the logical description of their data structures).

A Sketch of a Higher Level Programming System

So far we have been talking in a general way about the different domains of description and the kinds of things that might be said about a component or system

from each of their viewpoints. This does not yet provide a coherent picture of what a program will be. What do we see on the printed page or the screen? How is it organized? How do we do anything with it?

Once again it is important not to let our preconceptions get in the way. Notions such as code, listing, file, and compilation are based on the idea of a program as a set of instructions. There need not be any directly corresponding objects in a higher level system. Instead, it should be based on something much more like what we now think of as an artificial intelligence system with its "knowledge base" of assertions and procedures. There will be a set of interrelated descriptions, stored in a form that makes it possible to retrieve, manipulate, and display them. These will include *prototypes* for categories of objects and processes (like *classroom* and *filling out a form*) and *instances*, which correspond to individual objects and processes in one of the domains (such as the course CS 365 in Winter 1978, the contents of the third page of next quarter's schedule, and the process currently running in the database server). Instances can be described by more than one prototype, and prototypes are related into hierarchies with different degrees of abstraction. The details of all this are still a matter for extensive research. One set of possibilities is being explored in KRL [6, 7], but the basic idea of higher level systems could be implemented using other descriptive representations, such as semantic networks [8, 10, 14, and 15] and other frame-based systems [9, 11].

In addition to the basic description system, there will be a wide range of commonly useful prototypes and instances. Some of them (e.g. graphics formats, dates and times, communication protocols) will be in the subject and interaction domains. Others (such as the data structures used in a particular database) will be in the implementation domain. Some (such as descriptions of specific pieces of hardware and software that are being used) will be specific to the programming system. Others (such as those for abstract objects like sets and sequences, and for process structures) will be very general. In approaching a problem, a programmer will make use of this vocabulary of concepts and descriptive categories, both for interfacing with existing components and for organizing new ones.

Earlier in the paper, we discussed the need for programming systems based on the description of both results and processes. In the light of the above discussion, we can see that this applies not only to the implementation domain, but to the others as well. By shifting emphasis away from programs made up of instructions (which are necessarily in the implementation domain) to a description of processes and results, it becomes possible to integrate descriptions in all three domains within a single formalism.

A programmer's use of a higher level system will be highly interactive. Since the understanding of a component comes from having multiple viewpoints, no single organization of the information on a printed page will be adequate. The programmer needs to be able to reorganize the information dynamically, looking from one view and then another, going from great generality down to specific detail, and maneuvering around in the space of descriptions to view the interconnections. This will require an interface that is more sophisticated than the question-answering interfaces now used on artificial intelligence knowledge bases. It seems likely that pictorial representations, interactive graphics, and "intelligent summarizing" will play a large role. Some current software development techniques (such as the Structured Analysis Design Technique [18]) emphasize the importance of multiple viewpoints of analysis in documentation and design.

Of course, there always remains the task of providing a description of each component that is detailed enough to allow the system to run it. This will be part of providing a broader description, and may be done in stages. A very abstract specification of what a component does will be sufficient for a kind of "high level debugging" in which its interactions with other components can be analyzed and described without carrying out the steps at the lowest level. There is a whole range of operations that are now thought of as "automatic programming," which will enable the programmer to let the system fill in details once the overall behavior of the component has been specified. Some of these will be based on standardized defaults, others on automated analysis of performance characteristics. It will be all based on the availability of descriptions in the implementation domain of the various machines and subsystems being used.

As systems become more complex, the level of desirable invisibility will rise. Current high level programming languages do not give the user the opportunity to decide just how the hardware registers of the machine will be used to store variables, since there is much more to be gained by a uniform integrated approach to their use by the compiler. Similarly, much of what we now think of as data structure and algorithm specification will be handled by programs that can take into account much more complex efficiency considerations than are practical for a human programmer.

In summary, a programmer will use a programming system that contains a base of knowledge about the system he or she is working on, and of other potential components and concepts of programming that might be of use. The programmer will modify the descriptions of previously described components, and create high level descriptions of new ones to be added. These modifications and additions may be from the viewpoints of all of the different domains, and will be carried out in cooperation with the system, which checks for consistency, for consequences of new information, etc. Checking and debugging will be done at a variety of levels as the description becomes more detailed. The programming system will attempt to fill in details that are needed to completely specify the implementation, so the working

system as a whole can be run. The debugging process will make use of sophisticated reasoning processes that can use all of the different domains of description in analyzing and reporting what is happening.

What Needs to Be Done

The notion of higher level programming systems is not new. Integrated programming environments have been one of the major areas of development in Lisp systems [56], and one of the major reasons for the DoD common language effort [3] is to encourage the development of programming tools which are too complex and costly to be justified in a single project, or even a single specialized language. These systems, however, have evolved within the standard view of programming, and although they contain many useful ideas, they are far from achieving the goals discussed above. We need further research in three distinct but interrelated areas: the development of an effective descriptive calculus; the creation of a body of descriptive concepts for computational processes; and the building of a complex integrated system which uses them.

A Descriptive Calculus

The main thrust of these ideas depends on the ability to create and manipulate descriptions in an effective, understandable way. There are existing formalisms for description (for example, the predicate calculus) which are clear and well-understood, but lack the richness typical in descriptions which people find useful. They can serve as a universal basis for description but only in the same sense that a Turing machine can express any computation. They lack the higher level structuring which makes it possible to manipulate descriptions at an appropriate level of detail. As Dijkstra [2, p.5] observes:

> That the first-order predicate calculus was the most suitable candidate for the characterization of machine states was assumed right at the start; early experiences, however, were not too encouraging, because it only seemed practicable in the simplest cases, and we discovered the second consequence: the large number of variables combined with the likely irregularity of the subsets to be characterized very quickly made most of the formal expressions to be manipulated unmanageably long.

The requirements for a descriptive language of the kind I propose are quite different from those used in the mathematical foundations of computation or program verification. Work in these areas (see, for example the collection of papers in [22]) emphasizes the use of descriptive languages in rigorous proofs of the properties of programs. A higher level programming system must instead emphasize the use of descriptive languages for communication. The concentration must be on those aspects which aid in giving a person a clear overall understanding (at variable levels of detail and from multiple points of view), rather than on those aspects which increase the mathematical tractability of the descriptions.

Wulf [4] describes the motivations for the programming language Alphard in similar terms:

> The "software crisis" is the result of our human limitations in dealing with complexity. To "solve" the problem we must reduce the "apparent complexity" of programs, and this reduction must occur in the program text.... We know something about the way humans have traditionally dealt with understanding complex problems... and we can try to mold the expression of a program so that it facilitates these techniques.

If we look at the ways in which people have "traditionally dealt with understanding complex problems," we find many features of natural language which serve to reduce complexity by allowing imprecision when precision is not required. This is not an excuse for avoiding all precision, or a justification for "natural language programming." We need to understand the deep psychological properties of how people understand language, not mimic its superficial forms. The justification is not that natural language is "better" in some abstract sense, but that it is what we as people know how to use. As discussed above, the most essential feature of a programming formalism is its understandability by programmers. We cannot turn programmers into native speakers of abstract mathematics, but we can turn our programming formalisms in the direction of natural descriptive forms.

There are artificial intelligence formalisms (such as semantic networks [8, 10, 14, and 15] and KRL [6, 7]) with increased dimensions of expressiveness, but these are not yet at a level of precision which would make them sufficiently understandable to be used in a system of the required complexity. The characteristics which they explore (and which will need to be part of a formalism to be used in a higher level programming system) include:

Prototype hierarchies. The nouns and verbs of a natural language can be organized into hierarchies (or taxonomies) which capture much of the logical structure of what they describe. We know that a *dog* is an *animal*, and the answers to questions about dogs will often be derived through general properties of animals. Systems such as semantic networks treat these deductions specially, rather than dealing with them uniformly as a set of universally quantified implications. This leads to greater efficiency for the common calculations, and provides a structure which makes it much easier for a programmer to organize a knowledge base. These hierarchies contain information which could be thought of as a set of independent facts, but has additional structure in the same sense that a structured program structures a set of steps and jumps.

The centrality of defaults. Most logical calculi are optimized for handling generalizations which are either true or false. They do not provide means for stating generalizations that are not completely universal, but are "usual" or "normal" or "expected." In natural descrip-

tions of any kind, people draw heavily on the ability to use information of a less formal sort and a kind of *ceteris paribus* reasoning in which a standard fact is assumed true unless there is an explicit reason to believe the contrary. One of the major directions in artificial intelligence representation research is the attempt to provide this capability in a formally clear system. The notion of a *default* value is familiar to every programmer, but its place in a formal calculus needs to be carefully worked out.

The suppression of exceptional details. One of the major reasons for using precise formalizations is that they make everything explicit. For some purposes this is good, but there are times when understanding can come only through the suppression of detail. If we are trying to formally describe a program which normally involves simple input-output behavior (e.g. one that copies data from one place to another), we want to describe its behavior in a way which highlights that simplicity. If there are exceptional cases (e.g. when the storage allocator fails to find a sufficient block), these need to be described, but in a secondary place. This basic description of an object cannot be cluttered up with all of the details needed for handling the contingencies. Formalisms used in denotational semantics for programs demonstrate this problem well. In order to deal with a special escape at all, they demand that even the simplest programs be described as operating on continuations, environments, etc., and this description permeates every level of what is said.

Multiple levels of abstractions and instances. In dealing with programs and processes, we run into complexities involving the *instantiation* of general classes. For example, a specific algorithm (such as Euclid's algorithm) can be thought of as an instance of a general class (numerical algorithms), or as a class whose instances are programs implementing the algorithm. Each of those programs is in turn both an instance (of the class of formal objects known as programs) and a description of a class of process instances, each of which is carrying it out. If we look at the finer structure of programs, such as the instantiation of variables or pieces of code within loops similar phenomena arise. Higher order and typed logics deal in certain ways with the notion of a class (predicate) which is also an instance, but their austerity makes them inadequate for capturing the rich set of ways in which people interleave levels of abstraction. Artificial intelligence formalisms have not yet dealt adequately with these issues, which are currently a topic of active investigation.

A Basis for Describing Processes

Given a formalism for descriptions in general, we need prototypes for describing those things which are common to all of our programs (e.g. processes, programs, data structures, communication acts). This is a necessary kind of *library*, just as a library of standard data structures and statistical routines is a necessary part of a system for statistical manipulations. It need not be fixed once and for all, but a good deal of it must be in place before the system is usable, and it must be held relatively uniform if the system is to be extendable.

The domain which I believe is currently ripest for exploration is the description of processes. Traditional mathematics provides us with a broad variety of concepts for what I have called "result specifications," and they are being applied to programming (for example, [19]). In describing processes, we are on shakier ground. There are many promising ideas floating around that need to be captured in a more precise form. The success of higher level programming systems will depend on having a coherent body of descriptive categories which can capture a variety of modes for process description. There is a beginning of comprehensive work in this area, such as the development of the Delta language for system description [20], but most of the work so far has dealt with one or another aspect in isolation.

Modularity and structured procedures. There has been a good deal of attention in recent years to the higher level structure of control constructs. In addition, languages based on data abstractions (such as CLU [36], Alphard [38], and Mesa [33, 34]) provide for larger *modules* which encapsulate collections of data structures and procedures. Beginning from a different point of view, *structured system description languages* [18, 20] provide conceptual tools for describing the overall structure of large systems. We need a consistent way of talking about modularization and interaction between semi-independent modules which can be applied to system structure at all different levels of detail.

Structured data objects. Work on programming language constructs often emphasizes the structure of the sequence of operations, in terms of loops, recursive calls, etc. [19]. A related notion in describing processes is the ability to hide detail by allowing the combination of objects into a larger "structured object," and to define unitary operations on this higher level object which invoke collections of operations on the components. This has been explored for simple mathematical objects (e.g. lists in Lisp's MAP functions [55], vectors and arrays in APL [25], sets in SETL [30] and VERS [24]), and seems applicable to more specialized semantic objects (in all of the three domains) as well. In many cases, much of what is now thought of as control structure can be implicit in the data structure, leading to notions of "nonprocedural" or "procedureless" languages [26, 28]. The interaction between control and data structure needs to be put into a theoretical framework.

Program factoring—objects and procedures. In viewing a process as a structured sequence of individual steps, there are different ways to think about what each of those steps is. Most programming languages lead the programmer to think in terms of *operations* (either primitive or programmer-defined) to be carried out on *arguments.* Some (such as Simula [31], Smalltalk [35], and

Plasma [51]) think of typed *objects* which receive and interpret *messages.* Instead of organizing code around a single procedure (e.g. *print* or *plus*) which selects its action according to data type, they define *classes* (such as integer, list, etc), which select what to do on the basis of the message. Artificial intelligence languages take a more general approach in using *pattern directed invocation* [5, 32]. Each step specifies a pattern (or *goal*) to be achieved. A database of *pattern-action pairs* is accessed to decide what steps to carry out. Each of these viewpoints is best for some aspects of programming, and we need to understand how to integrate them into a larger framework.

States and transitions. There are two complementary ways of looking at a computational process—as a sequence of operations or a sequence of states. This duality has been exploited in the mathematical theory of computation, but has not really been integrated into programming languages. Transition nets and Petri nets [42, 44] are state-oriented, rather than operation-oriented. Production systems [47] are state-oriented, with each production specifying a partial state description and an appropriate transition function (not the name of the new state, but a set of operations which produce the new state). Languages which provide ways of specifying actions to be taken on special conditions [41] are really mixing state-transition description with the normal operation sequence. As with the operation/object distinction above, the goal is to find a synthesis which allows a process to be described using a mixture of the conceptual viewpoints, and to be run on the basis of that description.

Interacting processes and communication. The notions above deal primarily with a single process. The most significant direction in computing over the coming years will be towards multiple processes, both virtual (e.g. organizing a speech system as a series of separate processes, even if it runs on a single PDP-10 [49, 54]) and actual (e.g. networks of computers cooperating on a single task). There are a number of issues which have been dealt with by system designers at lower levels (like operating systems) which have not found their way into higher level languages. There is also a wealth of metaphor provided by thinking of a computation as being carried on by a collection of independent individuals who must communicate by exchanging messages in a common language. We can draw many analogies from human communication. What language do they talk? Which subsystems need to be multilingual? What are the discourse rules for establishing and controlling message flow? Is it possible to learn a second language? How can two processes make use of shared knowledge to increase the efficiency of communication? How can one process make use of an internal model of another process in order to facilitate communication and cooperation? Some of these issues are being explored in the multiprocessor programming language PLITS [50] and are the basis for theoretical formalisms such as that of Hoare [52].

A Complex System

The kind of higher level programming system discussed in this paper is itself a massive and complex system. Its subject matter is not an external one, like room-scheduling, but the reflective one—the subject of programming. Much of the other work mentioned in this paper provides a starting point. Integrated programming systems, languages based on data abstraction, and representation languages are examples of work which can be incorporated. It is unlikely that a "crash program" to produce a unified higher level programming system would succeed today. There will have to be a careful program of bootstrapping to get from today's languages and systems to the one I have described. The reason for writing a paper of this sort (rather than setting out to build a system) is the recognition that the relevant ideas need more development, and the hope that people will turn their attention to them.

Acknowledgments. In a paper of this kind, it is impossible to properly credit the sources of the ideas. It grew out of ongoing interactions with people who hold similar ideas in different forms, and it is really just an expression of the current state of my intellectual environment. My joint work with Danny Bobrow and Brian Smith on the theoretical foundations of KRL has been a primary source of ideas, and the rest of the Stanford/Xerox KRL research group (David Levy, Mitch Model, Richard Fikes, Don Norman, and Henry Thompson) have been involved in all stages of our thinking. Stanford computer science students in the CS365 seminar in 1977 pushed and probed on many of the ideas about procedures, which in turn came from the authors of the papers we read there (included in the list of references). The Xerox PARC environment has been a context in which the problems of "programming in the large" are well-understood, and has provided a wealth of ideas and examples, including the work of Alan Kay and his group on Smalltalk, the implementation of the Mesa programming language, the development of programming environments by Warren Teitelman and Larry Masinter, Bob Sproull's understanding of graphics systems and protocols, and Peter Deutsch's views on system organization and programming environments. In addition, the cybernetic notions of Humberto Maturana as introduced to me by Fernando Flores have led to subtle but very important shifts of perspective in the way I look at systems of all kinds. I am also grateful to Alan Perlis, Peter Deutsch, Jim Horning, and the referees for extensive and insightful comments on earlier drafts of this paper.

Received January 1979; revised March 1979

References

The programming of complex systems

1. Belady, L.A. Large software systems. Res. Rep. RC 6966 (≠29862), IBM Thomas J. Watson Res. Ctr., Yorktown Heights, N.Y., Jan. 1978.

2. Dijkstra, E.W. On the interplay between mathematics and programming. Unpublished lecture, EWD641, 1977.
3. Fisher, D.A. DoD's common programming language effort. Computer (March 1978), 25–33.
4. Wulf, W.A. Some thoughts on the next generation of programming languages. In *Perspectives on Computer Science*, A.K. Jones, Ed., New York, Academic Press, 1977.

Representation formalisms
5. Bobrow, D., and Raphael, B. New programming languages for AI research, *Computing Surveys 6*, 3 (Sept. 1974), 153–174.
6. Bobrow, D., and Winograd, T. An overview of KRL, a knowledge representation language, *Cognitive Science 1*, 1 (Jan. 1977), 3–46.
7. Bobrow, D., Winograd, T., and the KRL research group. Experience with KRL-0: One cycle of a knowledge representation language, Fifth Int. Joint Conf. on Artif. Intell., pp. 223–227.
8. Brachman, R. What's in a concept: Structural foundations for semantic networks, *Int. J. Man-Machine Studies 9* (Sept. 1977), 127–152.
9. Davis, R. Knowledge about representations as a basis for system construction and maintenance. In *Pattern Directed Inference Systems*, D.A. Waterman, Ed., Academic Press, New York, 1978.
10. Fikes R., and Hendrix, G. A network-based knowledge representation and its natural deduction system. Fifth Int. Joint Conf. on Artif. Intell., pp. 235–246.
11. Goldstein I., and Roberts, B. Nudge, a knowledge-based scheduling program. MIT AI-Memo 405, M.I.T., Cambridge, Mass., Feb. 1977.
12. Hayes, P. Some problems and non-problems in representation theory. AISB Conf. 1974, pp. 63–79.
13. Hayes, P. In Defense of Logic. Fifth Int. Joint Conf. on Artif. Intell., pp. 559–565.
14. Levesque, H. A procedural approach to semantic networks. TR-105 Dept. of Comptr. Sci., U. of Toronto, Canada, 1977.
15. Woods, B. What's in a link? In *Representation and Understanding*, Bobrow and Collins, Eds., 1975.

Formalisms for specifying programs and describing systems
16. Ashcroft, E.A., and Wadge, W.W. Lucid, a non-procedural language with iteration. *Comm. ACM 20*, 7 (July 1977), 519–526.
17. Burstall, R.M., and Goguen, J.A. Putting specifications together. Fifth Int. Joint Conf. on Artif. Intell., 1977.
18. Dickover, M.E., McGowan, C.L., and Ross, D.T. Software design using SADT. Proc. 1977 ACM Nat. Conf., Seattle, pp. 125–133.
19. Dijkstra, E.W. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, New Jersey, 1976.
20. Holback-Hanssen, E., Handlykken, P., and Nygaard, K. System description and the delta language. Delta Proj. Rep. #4, Norwegian Comptg. Ctr. Pub. #523, Oslo, Sept. 1975.
21. Knuth, D. *The Art of Computer Programming. Vol. 1, Fundamental Algorithms.* Addison-Wesley, Reading, Mass., 1968.
22. Neuhold E.J., Ed. *Formal Description of Programming Languages*. North-Holland Pub. Co., Amsterdam, 1978.
23. Tennent, R.D. The denotational semantics of programming languages. *Comm. ACM 19*, 8 (Aug. 1976), 437–453.

Structured objects and structured procedures
24. Earley, J. High level operations in automatic programming. SIGPLAN Notices *9*, 4 (1974), pp. 34–42.
25. Falkoff, A.D., and Iverson, K.E. The design of APL. *IBM J. Res Develop.* (1973), 324–334.
26. Goldsmith, C. The design of a procedureless programming language. SIGPLAN Notices *9*, 4 (1974), pp. 13–24.
27. Kowalski, R. Predicate calculus as a programming language. Information Processing 75, North-Holland Pub. Co., Amsterdam, 1975.
28. Leavenworth, B. and Sammet, J. An overview of nonprocedural languages. SIGPLAN Notices *9*, 4 (1974), pp. 1–12.
29. Reynolds, J. GEDANKEN—A simple typeless language based on the principles of completeness and the reference concept. *Comm. ACM 13*, 5 (May 1970), 308–319.
30. Schwartz, J. On programming: An interim report on the SETL project. Installment I: Generalities. N.Y.U. Courant Inst., New York, Feb. 1973.

Program factoring—modules, objects, and procedures
31. Birtwistle, Dahl, Myhrhaug, and Nygaard. *SIMULA BEGIN*, Auerbach, Philadelphia, Pa., 1973.
32. Davis, R. Generalized procedure calling and content directed invocation. Proc. ACM Conf. on AI and Programming Languages, Aug. 1977.
33. Geschke, C.M., Morris Jr., J.H., Satterthwaite, E.H. Early experience with Mesa. *Comm. ACM 20*, 8 (Aug. 1977), 540–552.
34. Geschke, C.M., and Mitchell, J.G. On the problem of uniform references to data structures. *IEEE Trans. on Software Eng.* (June 1975), 207–219.
35. Ingalls, Dan. The Smalltalk-76 programming system: Design and implementation. Conf. Rec. of the Fifth Annual ACM Symp. on Principles of Programming Languages, Tucson, Arizona, Jan. 1978. pp. 9–16.
36. Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C. Abstraction mechanisms in CLU. *Comm. ACM 20*, 8 (Aug. 1977), 564–576.
37. Pratt, V. The competence/performance dichotomy in programming. Fourth ACM Symp. on the Principles of Programming Languages, 1977, pp. 194–200.
38. Shaw, M., Wulf, W.A., and London, R.L. Abstraction and verification in Alphard: Defining and specifying iteration and generators. *Comm. ACM 20*, 8 (Aug. 1977), 553–562.
39. Steele, G. LAMBDA, the ultimate imperative. MIT–AI Memo 353, M.I.T., Cambridge, Mass., March 1976.
40. Steele, G. LAMBDA, the ultimate declarative. MIT–AI Memo 379, M.I.T., Cambridge, Mass., Nov. 1976.

States and transitions
41. Goodenough, J. Exception handling: issues and a proposed notation. *Comm. ACM 18*, 12 (Dec. 1975), 683–696.
42. Holt, A. Introduction to occurrence systems. In *Associative Information Techniques*, Jacks, Ed., Elsevier, 1971, pp. 175–203.
43. Humby, E. *Programs from Decision Tables.* McDonald/Elsevier, 1973.
44. Lauer, P.E., and Campbell, R.H. A description of path expressions by Petri nets. Second ACM Symp. on Principles of Programming Languages, 1975, pp. 95–105.
45. Morgan, H.L. Event sequenced programming. Tech. Rep. 119 Dept. of Operations Research, Cornell U., Ithaca, N.Y., July 1970.
46. Reiger, C. The commonsense algorithm as a basis for computer models of human memory, inference, belief and contextual language comprehension. In *Theoretical Issues in Natural Language Processing*, Shank and Nash-Webber, Eds., 1976, pp. 180–195.
47. Rychener, M. Production systems: A case for simplicity in AI control structures. Draft of paper submitted to ACM 1977 Nat. Conf.
48. Sacerdoti, E. The non-linear nature of plans. Fourth Int. Joint Conf. on Artif. Intell., pp. 206–214.

Interacting processes and communication
49. Barnett, J. Module linkage and communication in large systems. In *Speech Recognition*, D.R. Reddy, Ed., pp. 500–520.
50. Feldman, J.A. High level programming for distributed computing. *Comm. ACM 22*, 6 (June 1979), 353–368.
51. Hewitt, C., and Smith, B. Towards a programming apprentice. *IEEE Trans. Software Eng. SE–1* (March 1976), 26–45.
52. Hoare, C.A.R. Communicating sequential processes. *Comm. ACM. 21*, 8 (Aug. 1978), 666–677.
53. Lampson, Mitchell, and Satterthwaite. On the transfer of control between processes. Proc. of Programming Symposium, Paris, April 1974; Lecture Notes in Computer Science 19, Springer–Verlag, 1974, pp. 181–203.
54. Lesser, V. Parallel procesing in speech understanding systems: A survey of design problems. In *Speech Recognition*, D.R. Reddy, Ed., 1975, pp. 481–499.

Lisp
55. Levin, M., et al. The Lisp 1.5 Programmer's Manual, M.I.T., Cambridge, Mass. 1965.
56. Sandewall, E. Programming in an interactive environment: The "Lisp" experience. *Computing Surveys 10*, 1 (March 1978), 35–71.
57. Teitelman, W. A display oriented programmer's assistant. Fifth Int. Joint Conf. on Artif. Intell., 1977, pp. 905–915.
58. Teitelman, W., et al. Interlisp Reference Manual, Xerox PARC, 1978.