Report of a Workshop on Future Directions in Programming Languages and Compilers*

May 31, 1994

Purpose

In January, 1993, a panel of experts in the area of programming languages and compilers met in a one and a half day workshop to discuss the future of research in that area. This paper is the report of their findings. Its purposes are to explain the need for, and benefits of, research in this field—both basic and applied; to broadly survey the various parts of the field and indicate its general research directions; and to propose an initiative aimed at moving basic research results into wider use.

The panel consisted of:

Robert Cartwright	Rice University
Charles Consel	Oregon Graduate Institute
Charles N. Fischer	University of Wisconsin
Susan L. Graham	University of California, Berkeley
Gilles Kahn	INRIA, Sophia Antipolis
Bernard Lang	INRIA, Rocquencourt
James McGraw	Lawrence Livermore National Lab
Steven Muchnick[1]	SUN Microsystems
Tom Reps	University of Wisconsin
John C. Reynolds	Carnegie-Mellon University
David W. Wall	Digital Equipment Corp.
David S. Warren	State University of New York, Stony Brook
Peter Wegner	Brown University
Jeannette M. Wing	Carnegie-Mellon University
David S. Wise	Indiana University

The discussion was led by Samuel Kamin and Eric Golin, both of the University of Illinois. Also present were Forbes Lewis and John Cherniavsky of the National Science Foundation.

The workshop was supported by NSF under grant CCR–9304990.

*This report was compiled by Samuel Kamin, University of Illinois at Urbana-Champaign, kamin@cs.uiuc.edu.

[1] Steven Muchnick was unable to attend the meeting, but participated in its preparation and in the preparation of this report.

Executive summary

Programming languages are the machine tools of the computer age. The best of them are carefully designed and built to facilitate the construction of high-quality programs—the programs that underlie virtually all modern technology. Furthermore, some of the most widely-used programs in the world—programs such as LOTUS 1-2-3–achieve a large measure of their power from integrated programming languages. Languages for a countless variety of tasks, such as design specifications, database constraints, page layout, and test-case generation, are designed and implemented continually.

The field of **programming languages and compilers** is the study of these languages and the software that implements them. Its goal is to gain basic knowledge about the design and implementation of languages that promote reliable and efficient programs and improve portability and programmer productivity.

This report discusses the **impact** the field has had, the **scientific knowledge** it has produced, and the likely **directions** of future research in the area. It goes on to propose an **initiative** which would have the effect of accelerating the transfer of basic knowledge about languages and compilers into other areas of science and industry.

We summarize our findings as follows:

- The **impact** of our research has been, and will continue to be, strongly felt throughout society. When well designed and implemented languages are used, our society gains in productivity and software quality. Our research is attempting to provide superior alternatives to the poorly designed languages too often used. A prime example is the rapidly growing use of object-oriented languages for the graphical user interfaces that have given such a huge impetus to the personal computer industry; these languages are the results of research going back to the mid–1960's.

  We take particular note of the fact that full-scale, general-purpose programming languages like FORTRAN, COBOL, and C are not by any means the only languages of importance. Many others are in use—for example, database programming languages like DBASE, text-processing languages like TEX, and hardware description languages like VHDL–and their quality is equally important. Indeed, we view the development of such special-purpose languages as a major area of future impact for our field.

- The **science and engineering** of programming languages and compilers can be divided into three broad areas:

  - **Design**—the study of fundamental principles of language design.
  - **Implementation**—the study of programming techniques to aid the development of compilers across the spectra of languages and architectures.
  - **Programmer interfaces**—the study of new tools that can aid the programmer in using a programming language.

- We chart some general **directions** of our research in each of the above areas, as well as addressing specifically the issue of technology transfer.

- We propose that an **initiative** be begun to promote research aimed at developing a *language design and implementation workbench.* This suite of tools would aid the language designer in writing a *high-level language description* and deriving, from that description, language processors such as *compilers*, *language-based editors*, and *debuggers*. These tools would support the construction of production-quality compilers, aid equally in the development of general- and special-purpose languages, and admit a wide range of source languages, including graphical ones. The goal of this initiative is to give our clients tools that will *save them time* and produce *higher-quality languages* for all applications.

1 Introduction

Programming languages are the machine tools of the computer age. The best of them are carefully designed and built to facilitate the construction of high-quality programs—the programs that underlie virtually all modern technology. Furthermore, some of the most widely-used programs in the world—programs such as LOTUS 1-2-3–achieve a large measure of their power from integrated programming languages. Languages for a countless variety of tasks, such as design specifications, database constraints, page layout, and test-case generation, are designed and implemented continually.

Language design and implementation technology therefore falls squarely within the definition of an *enabling technology*, one whose mastery is vital to future economic and technological achievements. The cost of failing to achieve that mastery is the dissipation of thousands of person-hours of work, using ungainly languages to produce unsafe and inefficient applications. The benefits of succeeding are effective and increasingly leveraged use of the most important resource, human ingenuity.

The field of **programming languages and compilers** is the study of these languages and the software that implements them. Its goal is to gain basic knowledge about the design and implementation of languages that promote reliable and efficient programs and improve productivity and portability.

Programming languages and compilers is among the oldest research areas in Computer Science, and has many accomplishments to its credit, for example:

- The development of high-level languages—from FORTRAN and COBOL through ADA, C++, ML, and others—has had an enormous impact on programmer productivity.
- The portability obtained from using high-level languages to code systems software has allowed vendors and users to upgrade processors without overwhelming software development costs. The economic benefit is enormous.
- Development of the graphical user interfaces that have become a staple of modern workstations and opened computing to non-experts is directly tied to the invention of object-oriented languages.
- Compilation techniques developed over many years are allowing for fuller exploitation of advanced architectures, from the ubiquitous RISC machines to the emerging massively-parallel "multi-computers."

Yet, the field is not well-understood or appreciated. Many programmers feel that language design is just common sense (despite the number of poorly designed languages they are forced to use every day). Others simply do not believe that basic research in language design can have any impact on practice; but they forget that the object-oriented languages of the 1990's have their roots in research of the mid–60's, and were still considered visionary and impractical in the early 80's.

As representatives of the field, the workshop participants believe it is vital that such basic research continue to be supported. Aside from the direct economic and scientific payoff from results such as those just cited, we note that language development is an integral part of computer use, and that the goals of the field—reliability, productivity, portability, efficiency—will remain relevant as long as computers are in use. The purpose of this report is to make this case, and to suggest ways in which the field might achieve greater impact and higher visibility.

The rest of this report is organized as follows:

Section 2: Impact. Our field has contributed greatly to the development of the software industry, one of the outstanding growth industries of the past decade. Furthermore, it *could contribute more*, if its goals and its results were better understood. In this section, we make the case for the field, discussing how it has shaped the world of computing.

Section 3: Science and engineering. Research in programming languages—including highly theoretical research—can be understood in terms of the goals outlined above. This is the heart of the report, in which the field is surveyed and various research efforts put into perspective.

Section 4: Directions. Going beyond the survey of section 3, we attempt to predict what areas of the field will open up and have the most impact in the near future (within ten years).

Section 5: Initiative. Considering its importance as an enabling technology, long-term, fundamental research is essential to maintaining the capability to build the technologies of the future. In this section, we identify a specific research goal in which increased funding could generate excellent results in a short-term (approximately five year) time horizon, given modest additional resources, and at the same time support long-term research goals.

2 Impact

Like machine tools, languages and compilers are largely invisible to the consumer, but have an impact felt broadly across everyday life. Languages are the critical link between programmer and computer, and no other single factor has as much impact on the affordability and quality of software.

This section is divided into subsections outlining what we see as the impact of our field on society as a whole, and on industry and science in general, and computer science in particular.

2.1 Society

Computers and software touch nearly all aspects of our lives. Though most people may not even be aware they are using software, let alone think about what language was used to program it, the effects of programming languages on society can be seen in many ways:

- *Programming languages are an enabling technology—they make new applications feasible.* Society is increasingly dependent on software—to fly its airplanes, produce its newspapers, facilitate the design of its products, maintain its banking and financial infrastructure, and enhance its recreational opportunities. The complexity and diversity of the software needed to support these applications both require and exploit new language paradigms and implementation strategies.

- *Standardized, high-level programming languages allow software to be ported.* The ability to upgrade hardware without incurring huge software development costs has been a key factor in harvesting the ever-increasing power/cost ratio of computers. By allowing the same applications to run on a variety of platforms, portability lessens the specialized skills required to operate a computer and helps the computer become another household appliance.

- *Programming by end-users is likely to increase in importance as use of computers increases.* Though they may not consider themselves programmers, many users write "macros" for their spreadsheets and databases, "batch files" in their operating system command language, and a variety of other small procedures. It seems likely that an increasingly sophisticated user community will demand further customizability—which is just another word for programmability—and programming language design will affect all those users.

> TEX is a type-setting system designed for high-quality technical publishing, which includes an extension language. The latter has been used to adapt TEX for some very difficult type-setting tasks, with programs of up to several thousand lines having been written. Anecdotal evidence suggests that some of the best TEX programmers are not programmers by trade, but rather TEX users who have learned to use its extension language to solve their own type-setting problems.

2.2 Industry

The quantity of software being developed, the cost of its development, and its importance in critical applications have increased dramatically in the past twenty years. Indeed, talk of a "software crisis" is commonplace. With the rising role of automation in manufacturing and the growth in service industries, the reliance on software will continue to climb. No one sees a single "solution" for this crisis, but developments in language design and implementation are certainly a major part of the answer. We see two areas where research in programming languages is impacting industry:

- *Adopting new general purpose programming languages.* In the data processing world, a new class of languages is coming into use, the fourth-generation languages [17], which add considerable power over COBOL, the traditional language in that field. Microcomputer software developers are switching to object-oriented languages (especially C++) with surprising speed.

  In each case, adoption of these new languages, though initially painful, has had a demonstrably positive affect on the cost and quality of software. For example, object-oriented languages have the potential to reduce costs by allowing for more *code re-use* than do conventional languages.

- *Developing specialized languages.* Virtually every major company has languages, often designed in-house, that are specialized for tasks such as test-case generation, system configuration description, signal processing, project tracking, and so on. Many commercial software products come with integral languages; industry spends millions of dollars each year programming applications in the PARADOX database system, to name just one. Such special-purpose languages are indispensable, and tools for effectively designing and implementing them are essential.

AT&T's largest telephone switch, the 5ESS, employs an internal database to track its status. The database must satisfy certain constraints, called "population rules," whenever an item is entered or removed. These rules are published in a document written largely in English, and their maintenance is so important that the 5ESS development team devotes approximately 250 programmers to interpreting and implementing this document. AT&T is now designing a language, called PRL5 [15], in which population rules can be formally defined. Rules written in PRL5 can be compiled, so that much of the work of those 250 programmers will be done mechanically. By designing this new language, AT&T expects to save a great deal of money, while improving the quality of the 5ESS switch.

The cost to industry of poorly designed, poorly implemented, and non-portable languages—whether vendor-supplied, off-the-shelf, or developed in-house—is impossible to calculate, but unquestionably huge. The accelerating movement toward newer languages shows that industry recognizes this.

2.3 Science and technology

The past decade has seen enormous growth in the use of the computer as a tool for scientists and engineers. In addition to traditional roles such as data collection and analysis and computer-aided design, computers are used increasingly for modeling, simulation, and visualization. Developments in programming languages and compilers are important in several areas:

- *High-performance computing requires effective compilation techniques and language design.* The most visible areas of scientific computer usage, commonly known as Computational Science and Engineering, are those that employ commercially available supercomputers of various kinds, programmed primarily in FORTRAN. They have seen great benefits from improved compilers, and are continuing to see developments in FORTRAN itself. Even greater productivity improvements will accrue from the development of languages to exploit newer, massively parallel computers.

  Many observers have argued that fully exploiting parallel architectures requires a *functional language*, but such languages are difficult to compile efficiently, for reasons not directly related to parallelism. SISAL is an example of a *dataflow language*, a class of languages closely related to functional languages. Recent results [5] on a broad range of problems showed significant improvements in parallel speed-up for SISAL over FORTRAN codes compiled with vendor-supplied parallelizing FORTRAN compilers.

- *Special purpose languages are used by scientists to communicate models to computers.* Scientists and engineers design and use many special-purpose languages; examples include hardware description languages, simulation languages, visualization languages, and others. Also noteworthy is the increasing use of symbolic mathematics programs like MATHEMATICA [24] and MAPLE[6], that have underlying programming languages which all but the most casual users must learn.

Thus, new languages eventually find their way into scientific and engineering projects. FORTRAN is here to stay, but its importance should not be exaggerated. Furthermore, we predict that as scientists become familiar with alternative styles of languages, they will not continue to tolerate the shortcomings of FORTRAN.

Computational fluid dynamics refers to the use of numerical computer simulations to help design devices that involve fluid flows, such as airplane wings and boat propellers. A production-quality computational fluid dynamics software package, such as those developed at the national laboratories and in industry, consumes 25–100 man-years in development and requires continuing use of the equivalent of 3–6 programmers for "maintenance." For example, in one implementation of an adaptive mesh refinement (AMR) algorithm for inviscid compressible gas dynamics, a few pages of mathematical description translated to 10,000 lines of FORTRAN.

FIDIL [11] is a language designed specifically for programming these applications. It includes the particular high-level data structures and operations needed for computational fluid dynamics. The 10,000 lines of FORTRAN for the AMR example cited above become a few hundred lines of FIDIL code. Furthermore the resulting program is re-usable: the changes needed to modify this FIDIL version of AMR for other, substantially different, applications, are manageably localized, while the same changes in the FORTRAN program require modifications throughout the code.

2.4 Computer Science

Programming languages and compilers is one of the central disciplines of Computer Science, and interacts closely with other disciplines.

- *Software engineering.* The extent to which the programming language used affects the success of a programming project is a controversial subject, but it can be substantial. It is for this reason that so many software development managers are considering switching to object-oriented languages [14]. (It is interesting to note that, while much of the work in software engineering is language-independent, changing language *paradigms* has a very significant impact; thus, a popular textbook on software engineering states, "Object-oriented design representations are more prone than other to have a programming language dependency" [19, p. 425].)

- *Computer architecture.* The fields of computer architecture and programming languages have a naturally symbiotic relationship. The C language was designed to take advantage of the minicomputers of the 1970's and has become the standard language for the microcomputers of the 80's and 90's. On the other hand, the RISC concepts now routinely seen in microprocessors were designed to exploit the capabilities of high-level-language compilers. The newer "multicomputer" architectures, and other architectures with high-level parallelism, present a difficult challenge to language designers and compiler writers, and many have taken it up.

  At the same time, improvements in cost and performance on the hardware side have allowed language designers more freedom. Advanced languages like LISP, ML, PROLOG, and SMALLTALK are no longer confined to research labs. That reality will no doubt affect programmer's language choices over time, with consequent feedback to the architects; this loop will continue to be a fact of computer research and development as far into the future as we can see.

- *Operating systems.* Languages, compilers, and operating systems have been closely related since very early in the history of computers. Some of the earliest time-sharing systems were built to support interactive programming environments, and a major area of language design has been that of *systems programming languages.* At a lower level, the close interaction of these areas is attested to by the papers in the biennial ASPLOS ("Architectural Support for Programming Languages and Operating Systems") conferences.

2.5 Summary

Research in programming languages and compilers includes theoretical work on language design paradigms and principles, applied research on compilation of conventional languages, and everything in between. What is essential to understand is this: language design and language processor construction are not problems that can be solved once and for all, but are as much an inherent part of computer usage as assembly-line design is to manufacturing; and that therefore whatever we can do to improve these designs and processors will have a payoff over both the short-term and long-term.

3 The science and engineering of programming languages

In the future, more people will program more computers for more applications, using more languages, than we can possibly imagine now. The quality of these languages and their implementations will profoundly affect the productivity of our society. Research in this field is aimed at discovering the principles that should guide language development, by both theoretical and experimental means. In this section, we survey the field in three categories: **design**, **implementation**, and **interfaces**.

3.1 Design

The central activity in this field is the study of programming language design, a broad research agenda that seeks answers to questions such as: What basic principles do all computer languages share? What features make for greater reliability and ease of analysis for each application area? How is a human problem translated to a computer solution?

This study has historical roots in mathematics and engineering, and has yielded deep scientific and technological results. It is perhaps easiest to understand where the field is now by considering the problems that have driven it:

High-level programming. The difficulty of programming computers at the machine level gave the first impetus to the development of high-level languages. Some of the basic issues in language design and implementation were first raised in the early 1960's.

Large-scale programming. Through the 1960's, it became clear that "programming-in-the-large" required new program-structuring capabilities. The need to support modular programming has been a driving force in language design ever since. For example, object-oriented languages are based on a powerful modularity construct called a *class*.

Symbolic programming. Functional languages like LISP and SCHEME were developed within the Artificial Intelligence community beginning in the late 1950's. This is the first of many examples of new application areas forcing the development of radically new language designs.

Another example from the world of non-numerical computing is the idea of logic programming, originally developed for writing natural language-processing programs.

Reliability for critical applications. As more and more critical software applications in areas like avionics came on line, reliability became a paramount concern. Starting in the early 1970's, major effort was devoted to the problem of formally specifying, and guaranteeing the correctness of, programs.

Portability. Less of a concern in the days when most software was either vendor-supplied or developed in-house, portability has become a key goal today, when software is widely disseminated across geographical networks and software houses must supply customers who deploy a variety of platforms.

Distributed and real-time programming. Some of the most difficult programs to write, while at the same time being the most critical, are those that involve controlling devices, such as airplanes or power plants. These programs are characterized by the need to synchronize operations between computers and other devices, while meeting timing constraints imposed by the application. ADA is one example of a language designed with these kinds of programs in mind.

End-user programming. The growing ubiquity of powerful desktop computers is driving a need for computer languages that can be used by non-experts. Still in its infancy, two examples of languages developed to fill this need are MATHEMATICA–whose programs are intended to look like mathematical assertions—and Graqula [12], a visual language for constructing queries in a geographical information system.

Despite the variety of motivations for their development, these languages have much in common in their underlying design—far more than is superficially apparent. The ultimate goal of the field of language design is to create a "theory of language design" that will facilitate the development of high-quality languages for each application.

3.2 Implementation

The study of techniques for compiler development has been an active one since the late 1950's. The results of that study are a variety of language development techniques and tools, including some, like UNIX's **yacc**, that are now routinely used in compiler projects, and some that are more experimental. These techniques and tools can be used in developing implementations for any computer language, not just traditional programming languages.

ELI [9] is compiler construction system "shell"—in other words, a system that incorporates a variety of compiler construction tools and is designed primarily to facilitate interactions among those tools. The tools include parser and lexer generators and attribute evaluators, among others. In a recent course, the following languages were among those implemented by the students using ELI: The *Carrier Configuration Language*, in which constraints on the placement of components within telephone carriers are expressed; when these constraints are "compiled," a program is produced which accepts as input a customer request and produces as output a legal carrier configuration to satisfy that request. The *Zebra Code Generator*, in which instructions to a custom label-printing machine are given at a high level; the output from the processing of these instructions is a lengthy list of instructions in the machine's native page layout language. The *TG* language for expressing test cases to be presented to C programs; the output is a UNIX shell script containing the appropriate program invocations with appropriate test data; test data can be user-supplied or randomly generated.

What is specific to *programming* languages is the problem of producing efficient machine language programs from high-level language programs. Each generation of computer architectures presents new challenges to the compiler writer. Modern uniprocessor architectures, characterized by the presence of caches and various forms of low-level parallelism, have preoccupied compiler writers recently.

On the languages side, new programming paradigms require new compilation techniques. Even compiling C programs is considerably more difficult than compiling FORTRAN. Languages exploiting new paradigms involve radically different compilation techniques.

Code generation for new architectures and new languages place a premium on the *static analysis* of programs, in which the compiler attempts to gain a deep understanding of some property of the program, such as its use of memory.

3.3 Interfaces

Enhancing the programmer's interactions with the language processor is another way to help achieve the goals of the field. A famous example is the development of interactive programming systems in the 1960's, which facilitated program entry and debugging. Code browsers are an innovation of the 1980's to facilitate code reuse. Several current compiler generator projects emphasize the construction of entire "programming environments," including language-based editors, debuggers, and other tools [2, 4].

End-user programming has begun to place an even stronger emphasis on this aspect of programming language research. At the same time, the computing power now available to the average programmer creates great opportunities. Current research includes tools for "semantic browsing" (searching through a large collection of existing programs for ones that satisfy certain properties), for program visualization, and for combining modules graphically.

4 Directions

The fundamental factors driving work in this field are a mixture of basic issues that have faced programmers throughout the history of computing, and new issues and opportunities opened up by advances in technology, both hardware and software.

- **Demands on software.** Software carries an increasing burden of the capabilities and quality improvements of all goods and services. Increased pressure for reliability, programming productivity, and portability are the inevitable results. This is the real meaning of the "software crisis."

  At the same time, programmers are being asked to program increasingly difficult applications. Parallel computers allow for the representation of more and more complex models of natural phenomena, but this complexity is overtaking the ability of traditional languages to support. And non-scientific ("symbolic") applications, such as intelligent user interfaces, continue to gain in importance, as they have historically.

- **Parallelism.** Computers will never be fast enough. As the limits of technology-based efficiency gains is reached, various forms of parallelism will become the norm. The demand for efficiency and portability will be no less urgent, but far more difficult to achieve.

- **End-user programming.** The line between *using* and *programming* computers will become increasingly fuzzy. Languages that are *easy to learn* and *reliable to use* are needed. Indeed, the very notion of what a program is may need updating.

- **New applications.** There is no end to the variety of circumstances in which an artificial, machine-processable language is useful. Defining a document layout, directing a robot, or describing the statistical analysis of a data set are just three of many applications for such languages.

There are also changes in the research environment that will affect how we work:

- **Realistic examples.** There is a growing recognition that in computers, as in the physical sciences, there is a big difference between the laboratory and the real world. Language designers and implementors are increasingly called upon to demonstrate their results in real-life applications and environments.

- **Standard tools.** The Internet enormously simplifies the sharing of research results. Language designers routinely post implementations of their new languages. However, experimentation with these implementations is often frustrating. There is a growing feeling that more use should be made of standardized tools, and better attention paid to code quality and user-friendliness, even of "prototype" implementations.

These are the large-scale forces that will drive our field for many years to come. Both short- and long-term research is indicated by this list, and we feel strongly that the natural tendency to short-change long-term research is ultimately self-defeating. However, we can only make predictions about the shape our field in the near-term future.

4.1 Language design

Basic research in the area of language design is highly abstract, because its goal is the discovery of design principles underlying *all* computer languages—it is the creation, in other words, of a science of language design. Basic principles that have emerged include the importance of *semantics*—the proposition that programs have meanings and that unifying concepts are to be found by studying those meanings; the ubiquity of *types* as an organizing principle of programs and languages; and the central role of *abstraction* in language design. Only by combining this developing theory with experimentation—namely, designing languages for a wide variety of applications—will the goals of the field be attained.

Specific trends we see over the remainder of the decade are listed here.

- **Design for composability.** A deep trend in language design throughout its history has been toward languages in which components can be *combined* more flexibly. John Backus emphasized this in his famous Turing Award lecture: "Perhaps the most important element in providing powerful changeable parts in a program is the availability of combining forms that can be generally used to build new procedures from old ones" [1, page 627]. The motivation for this emphasis is that composability leads to **reusability**, the one sure-fire way to increase programmer productivity and program quality.

- **New designs for new applications.** The central activity of the field is, and has always been, the design of new languages for new applications.

  - **Concurrent and parallel computing.** Robin Milner, in his recent Turing Award address [18], discussed the need for new design concepts in this area, stating, "Concurrency requires a fresh approach, not merely an extension of the repertoire of entities and constructions which explain sequential computing."

    > Most concurrent programming languages are based on an *asynchronous* communication model, in which programs must respond to requests at unpredictable times. This model is very general, but very complex. However, many applications are substantially *synchronous*—service events occur at predictable intervals. The ESTEREL language [3] was designed for such applications. Its features are quite different from those of conventional sequential languages or the more general asynchronous programming languages. An example of its use is the specification of the very complex behaviors and interactions of menus, scrollbars, and so on, in a graphical user interface [7].

  - **End-user programming.** Just as the needs of the Artificial Intelligence community sparked intensive language design creativity in the 1960's, and the problems of large-scale programming did in the 1970's, so the needs of unsophisticated users will drive that creativity in the 1990's. Examples of new approaches to programming are:

    * **Visual programming.** Since many modern computer applications are graphical, the needs of end-user programmers dictate a visual programming approach [22], in which complex processes are described pictorially.
    * **Example-based programming.** The central activity of programming is *generalization*—from a process that works for one case to one that will work for all of them. These systems rely on the user's ability to describe that one case, from which the language processor can generalize to a complete program. (Since end users often view their problems in graphical terms, these languages are closely related to visual languages [20].)
    * **Constraint languages.** In these languages, "programming" consists of defining the basic parameters of the solution to a problem, and having the computer search for a satisfactory solution, instead of telling the computer exactly how to go about constructing a solution [16].

- **Old designs for new applications.** We are well past the time when languages were designed from scratch, often taking an unfortunate syntax-first approach. Much is now known about language design at a deep, semantic level. We will increasingly be called upon to bring this knowledge to bear on the design of new languages, especially special-purpose languages.

> Music composition languages are used to describe the "construction" of music from primitive tonal components. Used to program synthesizers and other electronic instruments (usually using the MIDI interface), these languages are a major improvement over simple note entry; an example is a drumbeat, which can be programmed as a fairly simple repetition of a single note. PLA [21] and CANON [8] are two well-known examples. What is unmistakable in reviewing the evolution of these languages is that it follows a trajectory similar to that of programming languages in general, with increasing emphasis on composability, i.e. the flexible interconnection of components.

- **Timeless designs.** The long-term goal of language designers is to achieve an understanding of computer languages so deep that languages can be built for any application by combining known methods. Languages for all applications will continue to benefit from research in basic design issues, especially:
  - **Semantics.** This is the subfield that asks the most fundamental question about programs: what do they mean? We foresee the techniques of this area being applied to new language designs, such as those mentioned above.
  - **Combining paradigms.** Some approaches to language design have emerged as major organizing paradigms. Chief among these are the *imperative paradigm*—typical of conventional languages like FORTRAN and C–and the *functional paradigm*—including LISP and ML, for example; also important are the *logic programming* and *object-oriented* paradigms. An ongoing effort is to understand how to combine paradigms within a single language.
  - **Types.** Types are the programmer's static picture of a program's structure. Their application to functional languages was considered a major advance in language design in the 1980's, and recent research has expanded their usefulness in object-oriented and logic programming. Still, we feel that only the tip of the iceberg has been seen. The very notion of "type" will expand to allow for the description of a wider range of program properties. These more expressive types will improve program reliability, while permitting a greater degree of program integration and reuse and more efficient compilation.

- **Design for analyzability.** Programs for critical applications must be carefully analyzed for correctness and efficiency. Languages whose programs have a clear and simple semantics can obviously ease this job. Designing and working out the mathematics of such languages is a major thrust of the long-term research program in this field; the work by Milner cited above is an example. Challenges that are new or will receive increasing emphasis include:
  - Formal specification of concurrent and distributed programs. As mentioned earlier, there are a number of semantics models of concurrent and distributed systems. What is needed now are specification languages that the "run-of-the-mill" software designer can easily use—the equivalent, for concurrency, of languages like Z [23] and LARCH [10].

- The specification and prediction of run-time resource utilization—including, but not limited to, CPU time. This is essential, of course, for real-time applications. It is also becoming important to the community of programmers using advanced languages, which often do not map simply to existing computer architectures, and to those writing parallel programs.
- Relating specifications and programs. Systems that help to formally verify that programs satisfy their specifications have been limited historically by the power of available theorem-provers. We expect to see more modular and flexible theorem-provers, allowing domain-specific customization of verifiers.

4.2 Implementation

A major challenge for language implementors is the exploitation of parallel computers. We see the nature of this challenge changing somewhat, however, and coming to center more on the problem of *portability*, as parallel computers of varying designs proliferate.

This new emphasis will militate strongly for higher-level languages. They will compensate for their apparent disadvantage in absolute speeds by their greater adaptability across platforms.

Portability will also show up in another area, at the junction between languages and operating systems: memory management. Modern languages are heavily dependent upon dynamic memory allocation, but operating systems support tends to be minimal. Moving memory management toward the system level can promote portability, while improving system performance and facilitating the sharing of data among applications.

Compiler optimization will continue to be a growth area, especially in the realm of aggressive, "whole program" optimization, in which the compiler is able to reorganize an entire program rather than small "compilation units." Another promising area is that of table-driven static analyzers and code generators, which can make a major contribution to the goal of program portability by easing the development of compilers for various platforms.

4.3 Interfaces

Being the youngest field, and the one whose fundamental design boundaries have changed the most in recent years, this is also the field least amenable to crystal-ball gazing. Still, some trends in this area seem sure to continue:

- **Interoperability.** Viewed at the system level, "composability" refers to our ability to combine system components, regardless of their source language. Differing data representations and memory management requirements make such interoperability difficult to achieve, but it is nonetheless vital. One aspect of interoperability is *persistence*, which refers to the ability of objects created by a running program to survive beyond the expiration of the program. Implementing interoperability and persistence require that some responsibilities, such as memory management, devolve from the language processor to the operating system. Though it is always difficult to implement "high level" facilities at the system level, due to potential inefficiency and inflexibility, growing demand for integrated environments seems to make it inevitable.

- **Program visualization.** We have already mentioned the new area of visual languages. It is likely that various forms of visualization and graphical interaction, within the context of textual languages, will also grow in importance. Visual languages for expressing program and system configurations are an example. For debugging—especially, performance debugging—the ability to visualize the running program can be very helpful.

4.4 Technology transfer

The field of programming languages and compilers is, at its core, an applied science, and technology transfer has always played a substantial role in it. To continue providing benefits to science and industry from our research, we must be able to energize language design, analysis, and implementation expertise into action. There are a number of ways to do this:

Demonstrate increased productivity and product quality. Ultimately, these are the properties that a software manager looks for in a language, and are the most basic goals of our field. However, many other factors impinge on the the manager's choice of language, principally the availability of processors for the language, and the ability to train programmers to use it. The weight of these and other constraints have often proven greater than the promised benefits of a new language, so that migration to new languages tends to occur very slowly. However, it does occur, and we should by no means accept today's predominant languages as the end-point in general-purpose language development. Just as the rise of microcomputers led directly to the adoption of C, and the rise of graphical applications is leading to the use of object-oriented languages, so future computer architectures (e.g. parallel computers) and applications (e.g. multi-media) are likely to call again for new languages.

Extend popular applications. On the other hand, most new programming languages find their way into use as *extension languages* for popular applications. For example, perhaps the largest group of functional language programmers are the users of symbolic mathematics systems like MAPLE and MATHEMATICA.

> Stephen Wolfram is the developer of MATHEMATICA, a widely-used system for symbolic mathematics. Though MATHEMATICA is best known for its capabilities in symbolic algebra and graphics, Professor Wolfram himself holds a different view: "In the past, I have described MATHEMATICA as a Trojan Horse... In the long term, the most useful and significant way to view MATHEMATICA is as a computer language" [25].

Provide powerful libraries. By allowing the relatively easy interconnection of components, modern languages allow for the delivery of powerful libraries of routines, which go far beyond the traditional mathematical libraries of FORTRAN. Indeed, this has been an important selling point for object-oriented languages like SMALLTALK and C++. It is no coincidence that these languages have become popular for developing applications for graphical user interfaces (GUI's): such applications must necessarily be built on top of a large infrastructure of code supporting the GUI, and with object-oriented languages it is easier for an application to take advantage of this infrastructure.

5 Initiative

Researchers in this field know a lot about language design and implementation. The underlying goal of this initiative is to enable that knowledge to be disseminated and used more widely. We see the initiative as leading to the following benefits:

- Providing for the design of better languages and language processors.
- Promoting portability of languages across implementation platforms, including parallel processors.
- Creating an "infra-structure" of language design tools, which are widely known among sophisticated computer users.
- Encouraging the development of programming language theory that contributes to the design and implementation of practical languages.

5.1 Language design and implementation workbenches

We propose that an initiative be funded to promote research aimed at developing *language design and implementation workbenches.* These suites of tools would aid the language designer in writing a *high-level language description* and developing, from that description, language processors such as compilers, language-based editors, and debuggers. They would support the construction of production-quality compilers, aid equally in the development of general- and special-purpose languages, and admit a wide range of source languages, including graphical ones.

The goal of this initiative is to give our clients the tools that will save them time and produce higher-quality languages and language processors for all applications.

A model for this undertaking is the success of parser generators, like UNIX's **yacc**. These tools help in creating programs that understand the syntax of a language; starting from a high-level description of the grammar, they produce a program to parse it. These tools have been successful in two ways:

- They save time and effort, because it is easier to write a grammar description than a parser.
- They encourage well-designed grammars. Languages with "hand-coded" parsers tend to have awkward and non-uniform syntax. Such languages can be processed using a parser-generator only with extra effort; the tools "prefer" languages with simpler grammars. This effect of parser generators is less widely appreciated, but ultimately the higher quality of the languages being processed is a greater benefit than the lowered cost of their processors.

In part because of the success of these tools, parsing is one of the easiest aspects of language processing. One goal of the proposed initiative is to provide a significantly larger collection of such tools.

5.2 Research Goals

Attempts have been made in the past to build tools for language implementation and considerable success has been achieved. Yet these traditional tools are still far from the kind of capabilities we envision. They are limited in the class of languages they handle, they support only some of the language processing tasks that are needed, and they rarely achieve production quality in terms of efficiency or usability.

The workbench we envision would go far beyond the traditional approaches in several areas:

Design and description aids. Language theorists understand such basic concepts as "scope" and "type" and "function" quite well, yet those notions cannot be encapsulated and specified at a high level in language definitions. The result is that well-designed versions of those features and their implementations are not easily reused in new languages.

Portability across architectures. A primary goal of such a workbench would be to facilitate the development of portable implementations. The goal is that a single program run on a wide variety of architectures (including parallel architectures), on different local configurations, and across parallel programming models (global memory vs. message-passing; differing interconnection structures), efficiently exploiting each particular architecture or paradigm.

Support for many applications. Much new language development is for applications other than traditional programming. Examples of such non-traditional language applications include:

- Languages for describing data layouts in distributed-memory multi-processors. One such language is to be incorporated in the new HP-FORTRAN language.
- "Scripting languages," which are used to aid communication among different applications in a computer. Several major software manufacturers (e.g., Microsoft, Apple and Lotus) have or are developing such languages.
- Languages for providing extensible "macro" features in applications such as spreadsheets and word processors.

A workbench should support such application languages. Indeed, it is far more important to support a wide variety of applications than to support the development of conventional, general-purpose programming languages, for the simple reason that many more application-specific languages are being developed.

Extending the concept of "language". Current computer systems support a rich variety of input and output forms, such as gestures, graphics and multimedia. Programming languages are, by and large, still focused on the use of linear ASCII representations. The workbench should support both nontraditional representations (e.g., visual languages), and a more general notion of what constitutes programming (e.g., interaction as programming).

Language support tools. The term "language processor" includes far more than just "compiler" — indeed, for many languages it doesn't include "compiler" at all. Everything from pretty-printers to debuggers to profilers and more falls under this heading, and the workbench needs to provide for it.

Interoperability among languages. Compilers consist of a sequence of phases that progressively transform program text into machine code by passing through several intermediate representations. A workbench will expose and standardize these intermediate representations providing a framework for constructing programs from modules written in a variety of languages.

Code sharing and reuse in language processors. Production compilers for different languages can share the same code generator if they use the same backend intermediate language. Since a workbench will establish standard interfaces for all phases of the compilation process, it will encourage sharing modules among compilers—reducing the investment required to produce language processors for new languages. The definition of standard interfaces will also facilitate code sharing between different language developers.

Production-quality language processors. Last, but certainly not least, it must be possible to *use* the processors constructed in this way. Efficiency constraints are lower nowadays for most applications then they were in the past, because there are more fast computers being used.

Nonetheless, efficiency is one of the battlefields on which languages compete, and cannot be ignored.

Each of these goals require both **basic** and **applied research** to be fully realized. A clearer understanding of fundamental language design principles is needed — an understanding clear enough to implement. Indeed, it is likely that some new applications will require new language concepts, so that the problem will never be fully "solved." Similarly, the goal of "greater efficiency" is, by its nature, never completely achieved.

5.3 Specific research objectives

We expect that significant progress can be made in the short term toward achieving the goals stated above. Here are some examples of the kinds of research that might be funded under this initiative:

- Developing better mechanisms for formally defining high-level languages. A long-term goal of language theorists has been to understand concepts like "static scope" and "block structure" so well, and define them so precisely, that they can be made a part of the formal definition of a language.

- Building a suite of tools for developing language processors based on those definition mechanisms.

- Building tools to support development of visual languages, and languages with graphical input.

- Studying languages oriented toward particular application areas, with the ultimate goal of better understanding the needs of such languages. This might involve, for example, designing languages to deal with physical device control and asynchronous parallelism; or it might involve dealing with notations that are foreign to ordinary programming languages. (Researchers should definitely be encouraged to submit proposals for such cross-disciplinary work.)

- Facilitating the development of production-quality language processors by establishing standard intermediate representations and investigating new approaches to static analysis, program optimization, run-time services including garbage collection, and other translator technologies.

- Basic research on semantic theories for dealing with such language features as state-oriented computation and parallelism. These have been the subject of intensive research of late. Progress in better understanding them from an abstract point of view will lead directly to application in the desired workbenches.

5.4 Conclusion

Our society pays an enormous cost for the widespread use of old-fashioned "machine tools" for software development — that is, poorly designed and documented, frustratingly limited, unfriendly, and inefficient languages and language processors.

What we are proposing is a set of tools that would make it easier to design better languages, and to make those languages widely available for use. It is our belief that such a programming language design and implementation workbench would speed the adoption of higher quality languages, and thus have great benefits to society.

References

[1] John Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," Comm. ACM 21(8), August 1978, 612–641.

[2] R. Ballance, S. Graham, M.L. Van De Vanter, "The PAN language-based editing system for integrated development environments," in **ACM SIGSOFT '90: Fourth Symposium on Software Development Environments**, R. Taylor (ed.), 1990, 77–93.

[3] Gérard Berry, Georges Gonthier, "The ESTEREL synchronous programming language: design, semantics, implementation," Science of Computer Programming 19, 1992, 87–152.

[4] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, V. Pascual, "CENTAUR: the system," in **ACM SIGSOFT '88: Third Symposium on Software Development Environments**, P. Henderson (ed.), 1988, 14–24.

[5] David Cann, "Retire FORTRAN? A debate rekindled," Comm. ACM 35(8), August 1992, 81–89.

[6] B.W. Char, K.O. Geddes, G.H. Gonnet, B.L. Leong, M.B. Monagan, S.W. Watt, **Maple V Language Reference Manual**, Springer-Verlag, New York, 1991.

[7] Dominique Clément, Janet Incerpi, "Specifying the behavior of graphical objects using ESTEREL," INRIA Rapports de Recherche No. 836, April 1988.

[8] Roger Dannenberg, "The CANON score language," Computer Music J. 13(1), Spring 1989, 47–56.

[9] R.W. Gray, V.P. Heuring, S.P. Levi, A.M. Sloane, W.M. Waite, "Eli: A complete, flexible compiler construction system," Comm. ACM 35(2), Feb. 1992, 121–131.

[10] J.V. Guttag, J.J. Horning, "Report on the LARCH Shared Language," Science of Computer Programming 6(2), March 1986, 103–157.

[11] P. Hilfinger, P. Colella, "FIDIL: A language for scientific programming," in **Symbolic Computation: Applications to Scientific Computing**, R. Grossman (ed.), 1989, 97–138.

[12] E. Jungert, "Graqula—A visual information-flow query language for a geographical information system," J. Visual Prog. 4, 1993, 383-401.

[13] Brian W. Kernighan, Dennis M. Ritchie, **The C Programming Language (Second Edition)**, Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[14] Mitch Kramer, "Developers find gains outweigh o-o learning curve," Software Magazine 13(7) ("Client/server Computing"), Nov. 1993, 23–33.

[15] David A. Ladd, J. C. Ramming, "Switch software and software research," Intl. Conf. on Communications Technology, Beijiing, 1992.

[16] Wm Leler, **Constraint Programming Languages, Their Specification and Generation**, Addison-Wesley, Reading, Mass., 1988.

[17] James Martin, "Fourth-generation Languages, Volumes I, II, and III," Prentice-Hall, Englewood Cliffs, NJ, 1985.

[18] Robin Milner, "Elements of Interaction," 1992 Turing Award Lecture, Comm. ACM 36(1), Jan. 1993, 78–89.

[19] Roger Pressman, **Software Engineering, A Practitioner's Approach (3rd Ed.)**, McGraw-Hill, New York, 1992.

[20] R.V. Rubin, E.J. Golin, S.P. Reiss, "ThinkPad: A graphical system for programming by demonstration," IEEE Software 2(2), 73–79, 1985.

[21] Bill Schottstaedt, "PLA: A composer's idea of a language," Computer Music J. 7(1), Spring 1983, 11–20.

[22] Nan C. Shu, **Visual Programming**, Van Nostrand Reinhold, 1988.

[23] J.M. Spivey, **Understanding Z: A Specification Language and its Formal Semantics**, Cambridge Univ. Press, 1988.

[24] Stephen Wolfram, **Mathematica (2nd ed.)**, Addison-Wesley, Reading, Mass., 1991.

[25] "Use of 'MATHEMATICA' program spreads," **The Chronicle of Higher Education**, Nov. 20, 1991.