San Jose Research Laboratory

January 21, 1974

Memorandum for: R. E. Gomory

Subject: Annual report and Research goals in programming

Attached is my annual report for 1973. In addition to reporting on progress, it argues that: (see pp 3-4)

> (1) Simple, whole-entity programming languages, as contrasted to conventional complex, word-at-a-time programming languages, have the potential for drastically reducing the cost of programming.

I further believe that:

> (2) Present efforts to clean up and extend conventional concepts of programming languages have no such potential; that indeed they will continue to add complexity to languages without dealing with the word-at-a-time problem, as they have for the past 15 years, and thereby actually increase the cost of programming and the expertise required for its practice. (Thus PL/I is perhaps 10 times as complex as Fortran, less economical in execution, and only 20-30% more powerful in expressiveness.)

If there is any truth to the two assertions above, then Research should ask itself whether it has fallen into a comfortable but mistaken orthodoxy with respect to programming languages. I believe it should re-evaluate its emphasis and goals in computer science and programming; at least IBM computer scientists should be aware that *reducing the cost of programming would do more to help IBM's growth than perhaps any other technical accomplishment.*

I do not question the quality of our computer science work, only that such an extremely small part of it seems to relate to the biggest issue.

John Backus

cc: P. S. Dauber, Yorktown
L. Y. Liu, San Jose
D. E. Rosenheim, San Jose

ANNUAL REPORT 1973

John Backus

During 1973 all of my time has been spent in the area of the foundations of programming, in particular, on the further study of closed applicative and Red languages described in last year's report.

There were the following developments:

(1) The work of last year on classification of programming languages was entirely revised, a "language" being distinguished now from its "realization". A "language" is characterized by its "semantic function", which maps expressions into their "meanings", whereas a language "realization" provides a transition function which must be applied many times to produce the meaning of an expression. By giving axioms which characterize the semantic function of a language, one discovers certain global properties of languages which are easily overlooked when focusing on the detailed behavior of a transition function. Heretofore only transition functions of language realizations have been studied.

(2) The Red languages of my 1972 paper "Reduction languages and variable-free programming" were considerably simplified and much simpler programming techniques were developed for them based on a higher-level set of primitive operators.

(3) Considerable effort was spent on trying to understand and specify the differences between closed applicative languages and other systems. This effort revealed a number of interesting properties of closed applicative languages which differ radically from most other languages: (a) idempotency of meaning; (b) the "anti-quote" property; (c) non-extensionality; (d) the extended Church-Rosser property;

(e) the reduction property; (f) all functions of one type; (g) single-level reduction rules.

(4) A paper describing the above work, "Programming language semantics and closed applicative languages", was published as a Research Report and was presented in an hour talk at the ACM Symposium on Principles of Programming Languages, Boston, October 1-3, 1973.

(5) A continuing effort was started to define a formal language for describing classes of Red expressions. It is hoped that such a language can be used to precisely describe the effect of Red operators and to make possible the mechanical analysis of the effect of many operators.

(6) A continuing study is underway of a very attractive property of Red languages: it is easy to prove some very general and useful operator identities (e.g., operator A represents the same function as operator B). These identities can then be used as algebraic rules for transforming an inefficient but simple operator into a more complicated but efficient one.

Potential technical and economic impact for IBM of Red languages and related research

Analysis of problems with current programming languages; the "von Neumann bottleneck"

The principal problem in the programming area is the complexity of the programmer's basic tools: the programming languages he uses. This complexity makes programming a very expensive and technical enterprise, accessible only to experts.

Despite their growing complexity, most programming languages constrain the programmer to think and program at the level of calculations with individual words. In an effort to closely mirror the programs of von Neumann computers, programming languages have acquired their word-at-a-time character as a reflection of what might be called the "von Neumann bottleneck" found in all computers. This is the word-at-a-time tube which connects the central processor and the store. After the CPU sends an address to the address register, a single word can pass through the tube from the store to the CPU; there it can be combined with the limited information in the CPU. Later, one of the words in the CPU will be sent back through the tube (after another address is supplied). This scenario is further complicated by the fact that a large percentage of the words that pass through this narrow tube are not really "problem" words, but words which are used to produce addresses for the address register. The real task of a program is to change the contents of the store in a major way; but in a von Neumann machine this can only be accomplished by millions of words passing through the von Neumann bottleneck.

Now it may be that the von Neumann bottleneck is a necessary feature of computers; the trouble is that programming languages are

based on it and consequently a programmer's thinking is severely restricted. As a result, the programmer is chiefly concerned with computing addresses (e.g., the i and j of A[i,j]), with computations on the single words specified by these addresses (e.g., A[i,j]+B[j,k]), and with the complex control of repetitions required so that these single-word actions give the desired result for larger entities. To contrast word-at-a-time versus whole-entity programming, compare any conventional program for n factorial with the APL program: $\times/\iota N$ where $\iota N = (1,2,\ldots,N)$ and $\times/(1,2,\ldots,N) = N!$

Although it may seem paradoxical, I strongly believe that most of the complexity of programming languages is a direct result of the word-at-a-time approach and the von Neumann bottleneck. In other words, I believe that the ability to deal constantly with large data entities also makes possible uncomplicated languages. If this is true, two large gains in power and economy can be achieved in a single language: linguistic simplicity plus new expressive power. I believe that LISP, Red languages and, to a much lesser degree, APL are examples to support this thesis. Each of these has certain problems and advantages. These are discussed below.

APL

First, APL is a complex language; this is because it has basically accepted the principle of the von Neumann bottleneck, but has enlarged the bottleneck to pass arrays while keeping all the word-at-a-time facilities of more conventional languages (although it is weak in the control facilities needed for word-at-a-time operations). APL offers a number of useful operations on arrays, but a fundamental weakness in definitional power usually means reverting to word-at-a-time operations to define new operations on arrays. One's ability to define new operations on arrays is further hampered

because either every member of an array is a scalar (in the original system), or, in the new system, one must deal with very complex operations on much-too-generalized arrays. However when suitable array operations are available for a given task, the APL program for it can be quite elegant.

LISP

LISP, in its pure form, is the most powerful and elegant language available today. One is not dealing with the von Neumann bottleneck when thinking in LISP. But the pure LISP interpreter, which accepts only recursive function definitions, severely limits one's ability to manage the use of storage and to write more conventional, do-this-then-this programs. Since each "instruction" undergoes a complex treatment by the interpreter, it is difficult for the user to know what actions will be performed as a result. These difficulties have required the addition of conventional programming features until LISP 1.5 has become as complex as conventional languages.

Red languages

It is perhaps fair to say that Red languages have both simpler syntax and simpler semantics than any programming language; and that, compared to conventional languages like Algol, the degree of simplification is almost two orders of magnitude. As in LISP there is no von Neumann bottleneck problem. Furthermore, the definitional power of Red languages is much greater than that of conventional ones; and they have other properties which may make it possible to write programs at a much higher level of sophistication than we do now.

Red languages avoid the von Neumann bottleneck but retain the ability to manage the store. Each operation has the entire store for its operand and its result is again a new value for the entire store. "Addresses" may appear in an operator, but since its result is a

complete new store, a subsequent use of the same "address" will yield a result entirely unrelated to that of its first use. Hence all resemblance to classical addressing is absent.

Since Red programs do not normally match the operation of von Neumann machines (although they may be constrained to do so), it is difficult for a user to know if his program is efficient for them. There are two possible solutions to this problem, and the properties of Red languages favor both: (1) Automatic optimization. Since Red programs seem to be unusually analyzable and transformable into equivalent ones (see items 5 and 6 in the progress report above), it may be possible to produce the required degree of efficiency on von Neumann machines by automatic optimization. The second possible solution is (2) to design an efficient computer without the von Neumann bottleneck. Heretofore programming languages have either incorporated the bottleneck approach (i.e., conventional languages), or have avoided any notion of operations on storage (i.e., LISP). Thus there has been no general-purpose conceptual basis for a computer design without the von Neumann bottleneck. Red languages provide such a basis.

Another problem, which is even greater for Red languages than for LISP, is the unfamiliar Red programming style. The answer here is that if that style carries with it sufficient power and economy, people will ultimately become familiar with it.

Plans for 1974

I plan to continue work on items (5) and (6) of the above progress report, and if successful, to write a paper on Red programming, program analysis, and algebraic transformations of programs. I plan to continue occasional lecturing, especially at the University of California at Santa Cruz where I expect to be named Adjunct Professor . If time and assistance permit, I hope to begin work on an optimizing interpreter for Red languages.

Lectures/talks given in 1973

1) Red study group, a group of about 15 people in the Los Angeles area organized by Val Schorre, SDC, three talks.
2) UCLA
3) Univ of Washington, Seattle
4) Washington State Univ, Pullman
5) Watson Research Center
6) ACM Symposium on Principles of Programming Languages, Boston, Oct 1-3, one hour talk.
7) Univ of Massachussets, Amherst
8) Carnegie-Mellon Univ, Immigration Lectures
9) Univ of San Francisco
10) Lawrence Livermore Laboratory
11) Univ of California, Santa Cruz, two two-hour lectures.

Abstract of annual report, 1973, John Backus

Reports progress in (1) classifying programming languages; (2) simplifying Red languages; (3) understanding radical properties of Red languages; (4) paper published; (5) effort to mechanically analyze Red programs; (6) algebraic transformation rules for Red programs.

Analyzes problems with current programming languages. Argues that (a) complexity and the word-at-a-time approach of most languages are the principal reasons they are costly to use and accessible only to experts; (b) word-at-a-time languages result from mirroring the behavior of computers and their "von Neumann bottleneck", the word-at-a-time connection between the CPU and the store; (c) whole-entity programming makes possible remarkably uncomplicated as well as powerful languages; (d) LISP, Red languages, and, to a lesser extent, APL are examples of such languages.

Discusses the problems and advantages of three whole-entity languages: APL: weak definitionally, relies too much on word-at-a-time philosophy, too complex, but elegant when whole-entity programming possible. LISP: pure LISP: definitionally powerful, most elegant language available, but storage management and do-this-then-this programming impossible. LISP 1.5: too complex. Red languages: simplest, definitionally powerful, storage management possible. Difficult to tell if program efficient for von Neumann machine. Possible solutions: (i) automatic optimization using algebraic transformations , or (ii) design a whole-entity computer without the von Neumann bottleneck. (Red languages form conceptual basis for such design.)

Activities: fourteen talks given.