San Jose Research Laboratory

March 19, 1974

Memorandum for: P. S. Dauber
P. C. Goldberg

Subject: Pat Goldberg's memo of 2/15/74

Pat's memo suggests that Research should not study programming languages per se but rather problem areas and language "features" [e.g., data flow, function-valued functions]. I suggest that there are more fundamental issues involved in reducing the cost of programming. These issues require the best mathematical and logical skills which are peculiar to Research rather than those more widely available in IBM development labs such as those needed in non-fundamental problem-area studies and insertion of new features into languages.

The issues I refer to are (1) the basic syntactic and semantic framework of a language [i.e., the fixed notions of the language, the basic interpretation rules, etc.] and (2) the definitional capability which that framework allows. The framework of a language is all-important because it determines the basic properties of the language far more than the primitives which are then inserted. For one thing it determines what primitives can be inserted. For this and other reasons it determines the definitional power of the language.

Most languages have built most of their features into the framework rather than into their primitives, definable operators and/or statements because their basic framework philosophy cannot accept the necessary features except as additions to the framework; they cannot be inserted or defined. Thus pointers could not be added to Algol 60 without changing its framework. With this approach languages are either baroque or suffer from permanent limitations.

For example, Pat's memo suggests that function-valued functions and aggregate operations can be inserted into a conventional language framework [if she has a new kind of framework in mind, that would be quite different]. I cannot see how this language would avoid being labeled Algol 78 or PL/IV or LISP 4.5. For is it not essentially an effort to superimpose the features of LISP and/or APL on those of PL/I? The interaction of the new features with the old is unpleasant to contemplate. In addition to the usual assignment-type variables, there are now free and bound variables, substitution rules, binding issues for partially evaluated functions and their environments, questions about whether and when to evaluate an expression, about α-conversion and/or the handling of a-lists, about gotos in recursive function definitions, about whether functions will be unary and Schönfinkled, unary and applied to lists, or both, or n-ary. And there are questions about what functions can be applied to what aggregates, what kinds of variables can appear in which aggregates, about how many kinds of aggregates there are, about whether any kind of aggregate can belong to any other, and if not, what combinations are allowed. And in addition to problems of interactions, how can such a language avoid the issues, problems and features which have accumulated in PL/I, Algol 68, and LISP 1.x?

965 - IBM - 04

To summarize the preceding paragraph: I believe Pat's proposal to design a language with function-valued functions and aggregate operations is doomed to make Algol 68 look simple, unless Computer

Science has some basically new ideas about the framework for a language. And I am not aware of any research on such ideas in Computer Science.

I propose that some bright people in Computer Science should be persuaded to study programming languages, their frameworks and properties; that they should carefully define "programming language", "state", "semantics", "definitional power" and classify them by their properties; that they should look for variations of these definitions and for new kinds of languages and properties instead of formalizing and embellishing the old. I believe that these issues are important if we are to find a way to construct languages which are powerful and flexible without being grotesquely complex.

Certainly it is important to understand the ideas of the lambda calculus and Curry's combinators and proving programs and deductive systems and data-flow diagrams, but if we want to use the capabilities they offer, they must be embedded in some language framework. The lambda calculus, for instance, is itself a totally inadequate framework for a language: it admits only variables. The frameworks for it which have been advanced so far [e.g., pure LISP, LISP 1.5, the SECD machine] have grave difficulties; they are either inadequate or overly complex, in all of them the interpreter of recursive function definitions and its automatic handling of the a-list or of S,E,C, and D are barriers to programmer control. Thus the study of the lambda calculus is no substitute for studying language frameworks and basic semantic principles. [Although I am not fond of the SECD framework, I might cite here some work on it by Leavenworth: "Transition functions" RC3459. Although this work concerns a fundamental question, it seems to have been largely ignored by C.S.]

To turn a sentence of Pat's around: language development which concentrates on debugging, problem-oriented issues, data bases, and terminals and ignores the basic study of language frameworks and semantics is misguided. And we are ignoring that basic study.

People in Computer Science may prefer the more conventional programming language frameworks to those of Elgot and Eilenberg or to those of Landin or of Scott or of Reynold's Gedanken or to the simpler ones I have proposed [complete languages with strict or non-strict realizations, applicative or closed applicative languages]. Fine. But at least let's then try to understand the fundamental properties of conventional frameworks. If we roughly regard a conventional framework as comprising a set of program-state pairs with a transition function on such pairs, we can investigate the consequences of certain restrictions on the structure of programs and of states. Such a study might reveal, for instance, that there are quite powerful frameworks with relatively simple program and state structures and relatively simple transition functions which separate basic semantic rules from the meanings of individual primitives. In contrast, the conventional but trivial example language of the Vienna group, EPL, requires states which are quintuples of quite complex objects. The transition function for EPL is irretrievably entwined with all of the features of EPL. Its partial description occupies four pages of formal expressions. As we all know, the similar treatment of real conventional languages requires thick to very thick volumes; the PL/I state is a 19-tuple of complex objects. And in all of them, aside from a few items such as arithmetic operators which can be inserted into the framework, the framework is the language.

965-IBM-04

The framework for Turing machines, while hopeless for practical

languages, at least does not contain most of the features of each individual Turing machine. While Turing machines differ greatly one from another, they all have a common framework and the semantics of any one is simply described by the basic semantics plus that of its set of quadruples.

Surely we can find better conventional-like frameworks with simpler states and transition functions which separate some fundamental semantic properties from the individual properties of operators and statements. Surely we can enlarge and better understand the universe in which fundamental choices about the framework of a language are made. As things now stand in this area, I believe our effort in programming languages is indeed an orthodoxy: a framework is chosen largely in the belief that only one kind is available. Just as geometers used to believe there was only one valid set of axioms, so our Computer Science people seem to believe in only one kind of language framework. But just as geometry became a more fruitful subject when it was discovered that there are many kinds of geometries, so will the study of programming languages when it enlarges its concepts of the class of objects in its purview, and when it finds simpler objects therein and defines their properties more precisely.

The chief dogma of conventional language orthodoxy centers around the notion of assignment and its associated variables. Few would disagree that assignment and its variables are often useful notions, nor would I; but I do object to the orthodoxy with which the subject is treated; this prevents a deeper study of this complex subject and its many unexplored dimensions. One main advantage of the radical frameworks I have proposed is that assignment is not a primitive notion in them, and that various forms of assignment can be defined and used in experimenting with new programming styles. Moreover, these frameworks tend to show that many complex whole-entity programs can be more simply expressed without the use of assignment at all. Assignment carries with it a host of other complex dogmas [e.g., closures, scope rules, declarations, subscripts, use/mention rules, problems about call-by-name vs call-by-value, etc.] and these get embedded in, and complicate, the frameworks of conventional languages. Therefore a reasonable way to approach the problem of finding simpler frameworks is to look for [and at] non-orthodox ones which <u>can</u> function without assignment and in which assignment can be defined in different ways.

The purpose of this memo and its predecessor is to encourage the study of an important area of programming languages whose existence no one seems to recognize. Unfortunately, in the first one I failed to be specific about the nature of that area. This memo is an effort to remedy that. I hope its sometimes contentious tone will not offend. But the point it tries to make is one I believe to be important but widely ignored, a point which, if it continues to be ignored, may cause Research and IBM to miss an opportunity to radically reduce the cost of programming. Finally, as one 18th century letter writer apologized to a friend, I regret that I had not the leisure to be brief.

965 - IBM - 04

John Backus

cc: Marc A. Auslander