The synthesis of algorithmic systems

by DR. A. J. PERLIS
Director, Computation Center
Carnegie Institute of Technology
Pittsburgh, Pennsylvania

INTRODUCTION

On what does the fame of Turing rest? That he proved a theorem showing that for a general computing device – later dubbed a "Turing machine" – there existed functions which it could not compute? I doubt it. More likely it is because of the model he invented and employed: his formal mechanism.

This model has captured the imagination and mobilized the thoughts of a generation of scientists. It has provided a base for arguments leading to theories. His model has proved so useful that its generated activity has been distributed not only in mathematics, but through several technologies as well. The arguments employed were not always formal and the consequent creations were not all abstract. Indeed a most fruitful consequence of the Turing machine has been with the creation, study, and computation of functions which are computable, i.e., in computer programming.

I am sure that all here will agree that this model has been enormously valuable. History will forgive me for not devoting any attention in this lecture to the affect which Turing had on the development of the general purpose digital computer which has further accelerated our involvement with the theory and practice of computation.

Since the appearance of Turing's model there have been others which have concerned and benefited us in computing. I think that only one has had an affect like Turing's: the formal mechanism called ALGOL. Many will immediately disagree, pointing out that too few of us have understood it or used it. While such has unhappily been the case, it is not the point. The impulse given to the development of research in computer science is relevant while the number of adherents is not. ALGOL has mobilized our thoughts and has provided us a base for our arguments.

I have long puzzled over why ALGOL has been such a useful model in our field. Perhaps some of the reasons are:

a. Its international sponsorship
b. The clarity of description in print of its syntax.
c. The natural way it combines important programmatic features of assembly and sub-routine programming.
d. The language is naturally decomposable so that one may suggest and define rather extensive modifications to parts of the language without destroying its impressive harmony of structure and notation. There is an appreciated substance to the phrase "ALGOL-like" which is often used in arguments about programming, languages, and computation. ALGOL appears to be a durable model, and even flourishes under surgery – be it explorative, plastic, or amputative.
e. It is tantalizingly incomplete.

Of one thing I am sure: ALGOL does not owe its magic to its process of birth: by committee. Thus we should not be disappointed when eggs, similarly fertilized, hatch duller models. These latter, while illuminating impressive improvements over ALGOL, bring on only a yawn from our collective imaginations. These may be improvements over ALGOL, but they are not successors as models.

Naturally we should put to good use the improvements they offer to rectify the incompleteness of ALGOL. And we should also ponder why they fail to stimulate our creative energies. Why, we should ask, will computer science research, even computer practice, worm but not leap forward under their influence? I do not pretend to know the whole answer, but I am sure that an important part of their dullness comes from focusing attention on the wrong omissions of ALGOL.

The synthesis of language and data structures

We know that we design a language to simplify the expression of an unbounded number of algorithms created by an important class of problems. The design should be performed only when the algorithms for this class imposes, or is likely to impose after some cultivation, considerable traffic on computers as well as considerable composition time by programmers using existing languages. The language, then, must reduce the

cost of a set of transactions to pay its cost of design, maintenance, and improvement.

Successor languages come into being from a variety of causes:

a. The correction of an error or omission or superfluity in a given language *exposes* a natural redesign which yields a superior language.

b. The correction of an error or omission or superfluity in a given language *requires* a redesign to produce a useful language.

c. From any two existing languages a third can usually be created which (i) contains the facilities of both in an integrated form; and (ii) requires a grammar and evaluation rules less complicated than the collective grammar and evaluation rules of both.

With the above in mind, where does one commence in synthesizing a successor model which will not only improve the commerce with machines but will focus our attention on important problems within computation itself?

I believe the natural starting point must be the organization and classifying of data. It is, to say the least, difficult to create an algorithm without knowing the nature of its data. When we attempt to represent an algorithm in a programming language, we must know the representation of the algorithm's data in that language before we can hope to do a useful computation.

Since our successor is to be a general programming language, it should possess general data structures. Depending on how you look at it this is neither as hard nor as easy as you might think. How should this possession be arranged? Let us see what has been done in the languages we already have. There the approach has been to:

a. Define into the language a few "primitive" data structures, e.g., integers, reals, arrays homogeneous in type, lists, strings, and files.

b. On these structures provide a "sufficient" set of operations, e.g., arithmetic, logical, extractive, assignment, and combinational.

c. Any other data structure is considered to be nonprimitive and must be represented in terms of primitive ones. The inherent organization in the non-primitive structures is explicitly provided for by operations over the primitive data, e.g., the relationship between the real and imaginary parts of a complex number by real arithmetic.

d. The "sufficient" set of operations for these non-primitive data structures are organized as procedures.

This process of extension cannot be faulted. Every programming language must permit its facile use for ultimately it is always required. However, if this process of extension is too extensively used algorithms often fail to exhibit a clarity of structure which they really possess. Even worse, they tend to execute slower than necessary. The former weakness arises because the language was defind the wrong way for the algorithm, while the latter is because the language forces over-organization in the data and requires administration during execution that could have been done once prior to execution of the algorithm. In both cases, variables have been bound at the wrong time, the syntax and the evaluation rules.

I think that all of us are aware that our languages have not had enough data types. Certainly, in our successor model we should not attempt to remedy this shortcoming by adding a few more, e.g., a limited number of new types and a general catch-all structure.

Our experience with the definition of functions should have told us what to do: not to concentrate on a complete set of defined functions at the level of general use, but to provide within the language the structures and control from which the efficient definition and use of functions within programs would follow.

Consequently, we should focus our attention in our successor model on providing the means for defining data structures. But this is not of itself enough. The "sufficient" set of accompanying operations, the contexts in which they occur, and their evaluation rules must also then be given within the program for which the data structures are specified.

We might list some of the capabilities that must be provided for data structures:

a. structure definition.

b. assignment of a structure to an identifier, i.e., giving the identifier information cells.

c. rules for naming the parts, given the structure.

d. assignment of values to the cells attached to an identifier.

e. rules for referencing the identifier's attached cells.

f. rules of combination, copy, and erasure both of structure and cell contents.

These capabilities are certainly now provided in limited form in most languages, but usually in too fixed a way within the syntax and evaluation rules of the language.

We know that the designers of a language cannot fix how much information will reside in structure and how much in the data carried within a structure. Each program must be permitted its natural choice to achieve a balance between time and storage. We know there is no single way to represent arrays or list structures or

strings or files or combinations of them. The choice depends on

a. The frequency of access.
b. The frequency of structure changes in which given data is embedded, e.g., appending to a file new record structures or bordering arrays.
c. The cost of unnecessary bulk in computer storage requirements.
d. The cost of unnecessary time in accessing data.
e. The importance of an algorithmic representation capable of orderly growth so that clarity of structure always exists.

These choices, goodness knows, are difficult for a programmer to make. They are certainly impossible to make at the design level.

Data structures cannot be created out of thin air. Indeed the method we customarily employ is the use of a background machine with fixed, primitive data structures. These structures are those identified with real computers, though the background machine might be more abstract as far as the defining of data structures are concerned. Once the background machine is chosen, additional structure as required by our definitions, must be represented as data, i.e., as a name or pointer to a structure. Not all pointers reference the same kind of structure. Since segments of program are themselves structures, pointers such as procedure identifier contents of (x) establish a class of variables whose values are procedure names.

Constants and variables

Truly, the flexibility of a language is measured by that which programmers may be permitted to vary, either in composition or in execution. The systematic development of variability in language is a central problem in programming and hence in the design of our successor. Always our experience presents us with special cases from which we establish the definition of new variables. Each new experience focuses our attention on the need for more generality. Time sharing is one of our new experiences that is likely to become a habit. Time sharing focuses our attention on the management of our systems and the management by programmers of their texts both before and during execution. Interaction with program will become increasingly flexible, and our successor must not make this difficult to achieve. The vision we have of conversational programming takes in much more than rapid turn around time and convenient debugging aids: our most interesting programs are never wrong and never final. As programmers we must isolate that which is new with conversational programming before we can hope to provide an appropriate language model for it. I contend that what is new is the requirement to make variable in our languages what we previously had taken as fixed. I do not refer to new data classes now, but to variables whose values are programs or parts of programs, syntax or parts of syntax, and regimes of control.

Most of the attention is now paid to the development of systems for managing files which improves the administration of the over-all system. Relatively little is focused on improving the management of a computation. Whereas the former can be done outside the languages in which we write our programs, for the latter we must improve our control over variability within the programming language we use to solve our problems.

In the processing of a program text an occurrence of a segment of texts may appear in the test once but be executed more than once. This raises the need to identify both constancy and variability. We generally take that which has the form of being variable and make it constant by a process of initialization; and we often permit this process itself to be subject to replication. This process of initialization is a fundamental one and our successor must have a methodical way of treating it.

Let us consider some instances of initialization and variability in ALGOL:

a. Entry to a block. On entry to a block declarations make initializations, but only about some properties of identifiers. Thus, ***integer*** x initializes the property of being an integer but it is not possible to initialize the value of x as something that will not change during the scope of the block. The declaration ***procedure*** P(. . .); . . . ; emphatically initializes the identifier P but it is not possible to change the value attached to the identifier P from within the block. ***array*** A [1:n, 1:m] is assigned an initial structure. It is not possible to initialize the values of its cells, or to vary the structure attached to the identifier A.
b. *for* statement. The expressions, which I will call the step and until elements cannot be initialized.
c. The procedure declaration is an initialization of the procedure identifier. On a procedure call, its formal parameters are initialized as procedure identifiers are, and they may even be initialized as identifiers are, and they may even be initialized as to value. However different calls establish different initializations of the formal parameter identifiers but not different initialization patterns of the values.

The choice permitted in ALGOL in the binding of form and value to identifiers has been considered to be adequate. However if we look at the operations of assignment of form, evaluation of form, and initialization as important functions to be rationally specified in a language, we might find ALGOL to be limited and arbitrary in its available choices. We should expect the

successor to be far less arbitrary and limited.

Let me give a trivial example. In the ***for*** statement the use of a construct like ***value*** E, where E is an expression, as a step element would signal the initialization of the expression E. ***value*** is a kind of operator that controls the binding of value to a form. There is a natural scope attached to each application of the operator.

I have mentioned that procedure identifiers are initialized through declaration. Then the attachment of procedure to identifier can be changed by assignment. I have already mentioned how this can be done by means of pointers. There are of course other ways. The simplest is not to change the identifier at all, but rather a selection index that picks a procedure out of a set. The initialization now defines an array of forms, e.g., ***procedure array*** P [1:k] (f_1,f_2, ... f_j); ... begin ... ***end***; ... ; ***begin*** ... ***end***; The call P [i] (a_1, a_2, ... a_j) would select the ith procedure body for execution. Or one could define a ***procedure switch*** P := A,B,C and procedure designational expressions so that the above call would select the ith procedure designational expression for execution. The above approaches are too static for some applications and they lack an important property of assignment: the ability to determine when an assigned form is no longer accessible so that its storage may be otherwise used. A possible application for such procedures, i.e., ones that are dynamically assigned, is as generators. Suppose we have a procedure for computing (a) $\sum_{k=0}^{N} C_K(N) \times^K$ as an approximation to some function f(x), when the integer N is specified. Now once having found the $c_k(N)$ we are merely interested in evaluating (a) for different values of x. We might then wish to define a procedure which prepares (a) from (b). This procedure, on its initial execution, assigns, either to itself, or to some other identifier, the procedure which computes (a). Subsequent calls on that identifier, will only yield this created computation. Such dynamic assignment raises a number of attractive possibilities:

- Some of the storage for the program can be released as a consequence of the second assignment.
- Data storage can be assigned as the ***own*** of the procedure identifier whose declaration or definition is created.
- The initial call can modify the resulting definition, e.g., call by name or call by value of a formal parameter in the initial call will affect the kind of definition obtained.

It is easy to see that the point I am getting at is the necessity of attaching a uniform approach to initialization and the variation of form and value attached to identifiers. This is a requirement of the computation process. As such our successor language must possess a general way of commanding the actions of initialization and variation for its classes of identifiers.

One of the actions we wish to perform in conversational programming is the systematic, or controlled, modification of values of data and text, as distinguished from the unsystematic modification which occurs in debugging. The performance of such actions clearly implies that certain pieces of a text are understood to be variable. Again we accomplish this by declaration, by initialization, and by assignment. Thus we may write, in a block heading, the declarations

real x,s;

arithmetic expression t,u;

In the accompanying text the occurrence of s:= x + t; causes the value of the arithmetic expression assigned to t, e.g., by input, to be added to that of x and the result assigned as the value of s. We observe that t may have been entered and stored as a form. The operation + can then only be accomplished after a suitable transfer function shall have been applied. The fact that a partial translation of the expression is all that can be done at the classical "translate time" should not deter us. It is time that we began to face the problems of partial translation in a systematic way. The natural pieces of text which can be variable are those identified by the syntactic units of the language.

It is somewhat more difficult to arrange for unpremeditated variation of programs. Here the major problems are the identification of the text to be varied in the original text and how to find its correspondent under the translation process in the text actually being evaluated. It is easy to say: execute the original text interpretively. But it is through intermediate solutions lying between translation and interpretation that the satisfactory balance of costs is to be found. I should like to express a point of view in the next section which may shed some light on achieving this balance as each program requires it.

Data structure and syntax

Even though list structures and recursive control will not play a central role in our successor language, it will owe a great deal to LISP. This language induces humorous arguments among programmers, often being damned and praised for the same feature. I should only like to point out here that its description consciously reveals the proper components of language definition with more clarity than any language I know of. The description of LISP includes not only its syntax, but the representation of its syntax as a data structure of the language, and the representation of the environment data structure also as a data structure of the language. Actually the description hedges somewhat on the latter description, but not in any fundamental way. From the

above descriptions it becomes possible to give a description of the evaluation process as a LISP program using a few primitive functions. While this completeness of description is possible with other languages, it is not generally thought of as part of the defining description of those languages.

An examination of ALGOL shows that its data structures are not appropriate for representing ALGOL texts, at least not in a way appropriate for descriptions of the language's evaluation scheme. The same remark may be made about its inappropriateness for describing the environmental data structure.

I regard it as critical that our successor language achieve the balance of possessing the data structures appropriate to representing syntax and environment so that the evaluation process can be clearly stated in the language.

Why is it so important to give such a description? Is it merely to attach to the language the elegant property of "closure" so that bootstrapping can be organized? Hardly. It is the key to the systematic construction of programming systems capable of conversational computing.

A programming language has a syntax and a set of evaluation rules. They are connected through the representation of programs as data to which the evaluation rules apply. This data structure is the internal or evaluation directed syntax of the language. We compose programs in the external syntax which, for the purposes of human communication, we fix. The internal syntax is generally assumed to be so translator and machine dependent that it is almost never described in the literature. Usually there is a translation process which takes text from an external to an internal syntax representation. Actually the variation in the internal description is more fundamentally associated with the evaluation rules than the machine on which it is to be executed. The choice of evaluation rules depends in a critical way on the binding time of the variables of the language.

This points out an approach to the organization of evaluation useful in the case of texts which change. Since the internal data structure reflects the variability of the text being processed, let the translation process choose the appropriate internal representation of the syntax, and a general evaluator select specific evaluation rules on the basis of the syntax structure chosen. Thus we must give clues in the external syntax which indicate the variable. For example, the occurrence of ***arithmetic expression*** t; ***real*** u,v; and the statement u := v/3 * t; indicates the possibility of a different internal syntax for v/3 and the value of t. It should be pointed out that t behaves very much like an ALGOL formal parameter. However the control over assignment is less regimented. I think this merely points out that formal-actual assignments are independent of the closed sub-routine concept and that they have been united in the procedure construct as a way of specifying the scope of an initialization.

In the case of unpremeditated change a knowledge of the internal syntax structure makes possible the least amount of retranslation and alteration of the evaluation rules when text is varied.

Since one has to examine and construct the data structures and evaluation rules entirely in some language, it seems reasonable that it be in the source language itself. One may define as the target of translation an internal syntax whose character strings are a sub-set of those permitted in the source language. Such a syntax, if chosen to be close to machine code, can then be evaluated by rules which are very much like those of a machine.

While I have spoken glibly about variability attached to the identifiers of the language, I have said nothing about the variability of control. We do not really have a way of describing control, so we cannot declare its regimes. We should expect our successor to have the kinds of control that ALGOL has – and more. Parallel operation is one kind of control about which a good deal of study is being done. Another one, about which very little is being done, is the distributed control which I will call monitoring. Process A continuously monitors process B so that when B attains a state A intervenes to control the future activity of the process. The control within A could be written ***when*** P ***then*** S; P is a predicate which is always, within some defining scope, under test. Whenever P is ***true***, the computation under surveillance is interrupted and S is executed. We wish to mechanize this construct by testing P whenever an action has been performed which could possibly make P ***true***, but not otherwise. We must then, in defining the language, the environment, and the evaluation rules, include the states which can be monitored during execution. From these primitive states others can be constructed by programming. Knowing these primitive states, arrangement for splicing in testing at possible points can be done even before the specific predicates are defined within a program. We may then trouble shoot our programs without disturbing the programs themselves.

Variation of the syntax

Within the confines of a single language an astonishing amount of variability is attainable. Still all experience tells us that our changing needs will place increasing pressure on the language itself to change. The precise nature of these changes can not be anticipated by designers since they are the consequence of programs yet to be written for problems not yet solved. Ironically it is the most useful and successful languages that are most subject to this pressure for change. Fortunately,

the early kind of variation to be expected is somewhat predictable. Thus, in scientific computing the representation and arithmetic of numbers will vary, but the nature of expressions will not change except through its operands and operators. The variation in syntax from these sources is quite easily taken care of. In effect the syntax and evaluation rules of arithmetic expression is left undefined in the language. Instead syntax and evaluation rules are provided in the language for programming the definition of arithmetic expression, and to set the scope of such definitions.

The only real difficulty in this one-night-stand language definition game is the specification of the evaluation rules. They must be given with care. For example, in introducing this way the arithmetic of matrices, the evaluation of matrix expressions should be careful of the use of temporary storage and not perform unnecessary iterations.

A natural technique to employ in the use of definitions is to start with a language X, consider the definitions as enlarging the syntax to that of a language x′, and give the evaluation rules as a reduction process which reduces any text in x′ to an equivalent one in X.

It should be remarked that the variation of the syntax requires a representation of the syntax, preferably as a data structure of X itself.

CONCLUSION

Programming languages are built around the variable, its operations, control, and data structures. Since these are common concepts in all programming, general language must focus on their orderly development. While we owe great debt to Turing for his simple model, which also focused on the important concepts, we do not hesitate to operate with more sophisticated machines and data than he found necessary. Programmers should never be satisfied with languages which permit them to program everything, but to program nothing of interest easily. Our progress, then, is measured by the balance we achieve between effectiveness and generality. As the nature of our involvement with computation changes – and it does – the appropriate description of language changes, our emphasis shifts. I feel that our successor model will show such a change. Computer science is a restless infant and its progress depends as much on shifts in point of view as the orderly development of our current concepts.

None of the ideas presented here are new, they are just forgotten from time to time.

I wish to thank the Association for the privilege of delivering this first Turing lecture. And what better way is there to end this lecture than to say that if Turing were here today he would say things differently.